DERS KİTABI

Mikroişlemciler

Sercan Külcü

Giresun Üniversitesi · Bilgisayar Mühendisliği Bölümü

Mikroişlemciler

Sercan Külcü
Giresun Üniversitesi, Bilgisayar Mühendisliği Bölümü
Giresun, Türkiye

Bu kitap, şu adreste yayımlanan ders notlarının basılı hâlidir:
`sercankulcu.github.io/teaching/microprocessors/`

Emülatör, işlemci simülatörü, sunumlar ve çalışma soruları o sayfadadır.

© Sercan Külcü. Tüm hakları saklıdır.

Notların 14 Kasım 2026 tarihli hâlinden üretilmiştir.

Önsöz

Mikroişlemciler komutları yürütür, veriyi işler ve bir bilgisayar sisteminin bütün işlemlerini düzenler. Bu kitap işlemci düzenini, yazmaçları, bellek erişimini, komutların yürütülmesini ve düşük düzeyli programlamayı 8086 mimarisi üzerinden inceler. Bugünkü x86-64 işlemciler çok daha karmaşık olsa da 8086; yazmaçları, bölütlemeyi, adresleme kiplerini, kesmeleri ve assembly programlamayı tanıtmak için hâlâ iyi bir temeldir. Kitap, Giresun Üniversitesi Bilgisayar Mühendisliği Bölümü'nde verilen Mikroişlemciler dersinin notlarından oluşur ve bölümleri dersteki sırayla izler.

Kitap dört kısımdır. Kısımlar sırasıyla şunlardır: **Temeller** (Bölüm 1–3), **Bellek ve Sistem Hizmetleri** (Bölüm 4–5), **Komutlar ve Programlama** (Bölüm 6–9) ve **Donanım ve Komut Kodlama** (Bölüm 10–11).

Bazı bölümler birden çok ders notundan oluşur. Bu bölümler kısa bir açılış sayfasıyla başlar; notlar ardından 4.1, 4.2 gibi numaralı alt bölümler olarak gelir.

Her bölüm aynı düzeni izler. Başta öğrenme çıktıları yer alır. Biçimsel tanımlar, teoremler ve çözülmüş örnekler adlandırılmış kutulardadır. *Sık yapılan hata* kutuları, öğrencilerin en sık düştüğü yanılgıları gösterir. Kod izlemeleri, bir programın ya da algoritmanın adım adım nasıl çalıştığını gösterir. Bölüm; bir özet, bir kavram listesi, cevaplarıyla birlikte *Kendini Dene* soruları, ileri okuma önerileri ve kaynakçayla kapanır. Kitabın sonundaki terim sözlüğü, her İngilizce terim için notlarda kullanılan tek Türkçe karşılığı verir.

Örnekler emu8086 ile çalıştırılabilir. Emülatör, işlemci simülatörü, sunumlar ve çalışma soruları ders sayfasındadır.

Sercan Külcü
Giresun

İçindekiler

Dersin Haritası 1

KISIM I · TEMELLER

1 Mikroişlemcilere Giriş ve Sayı Sistemleri 4

1.1 Giriş 5

1.2 Sayı Sistemleri 58

2 8086 Mimarisi ve Yazmaçlar 111

2.1 8086 Mimarisi 112

2.2 8086 Yazmaçları 157

3 Assembly Dili 202

3.1 Programlama Dilleri 203

3.2 Assembly 258

KISIM II · BELLEK VE SİSTEM HİZMETLERİ

4 Bellek Erişimi ve Değişkenler 297

4.1 Bellek Erişimi 298

4.2 Değişkenler 350

5 Kesmeler ve BIOS/DOS Hizmetleri 398

5.1 Kesmeler 399

5.2 Kesme Fonksiyonları 443

KISIM III · KOMUTLAR VE PROGRAMLAMA

6 Aritmetik, Mantık ve Kaydırma Komutları 504

6.1 Bayraklar ve işlemin sonucu 505

6.2 Komut grupları ve işlenen kuralları 510

6.3 Toplama, çıkarma ve karşılaştırma 514

6.4 Mantıksal işlemler 517

6.5 Çarpma ve bölme 521

6.6 Tek işlenenli komutlar 532

6.7 Kaydırma ve döndürme 534

6.8 Örnek: dizi toplama 540

6.9 BCD aritmetiği 543

6.10 Örnek: sıcaklık dönüşümü 549

6.11 Hangi komut hangi bayrağı etkiler? 551

6.12 Sık yapılan hatalar 552

6.13 Alıştırmalar ve çözümleri 552

6.14 Özet 554

6.15 Kavram listesi 555

6.16 Kendini Dene 556
6.17 İleri okuma 558
6.18 Kaynaklar 558

7 Akış Kontrolü 559

7.1 Sıralı yürütme ve akış kontrolü 560
7.2 Koşulsuz atlama: JMP 561
7.3 Koşullu atlamalar 568
7.4 Karşılaştırma: CMP 573
7.5 Yüksek düzey yapıların karşılıkları 582
7.6 Döngü komutları 587
7.7 Örnek programlar 595
7.8 Sık yapılan hatalar 602
7.9 Alıştırmalar ve çözümleri 603
7.10 Özet 604
7.11 Kavram listesi 605
7.12 Kendini Dene 605
7.13 İleri okuma 608
7.14 Kaynaklar 608

8 Yığın ve Prosedürler 609

8.1 Yığın 610
8.2 Prosedürler 647

9 Makrolar ve emu8086.inc Kütüphanesi 702

9.1 Makrolar 703
9.2 emu8086.inc Kütüphanesi 754

KISIM IV · DONANIM VE KOMUT KODLAMA

10 8086 Donanımı ve Aygıt Kontrolü 799

10.1 8086 Pinleri 800
10.2 Aygıt Kontrolü 854

11 Komut Kodlama ve Dizgi Komutları 909

11.1 Komut kümesine genel bakış 910
11.2 Komut biçimi 912
11.3 Komut kodlama adım adım 917
11.4 Makine kodunu çözme 923
11.5 Veri taşıma komutları 927
11.6 Aritmetik, mantık ve kaydırma komutları: özet 933
11.7 Dizgi komutları 934
11.8 Yineleme önekleri 939
11.9 Kontrol aktarma: JMP'nin kodlaması 944
11.10 İşlemci kontrolü komutları 947
11.11 Bayrak etkileri 950
11.12 Sık yapılan hatalar 952

11.13 Alıştırmalar ve çözümleri 952
11.14 Özet 954
11.15 Kavram listesi 955
11.16 Kendini Dene 956
11.17 İleri okuma 958
11.18 Kaynaklar 958

EKLER

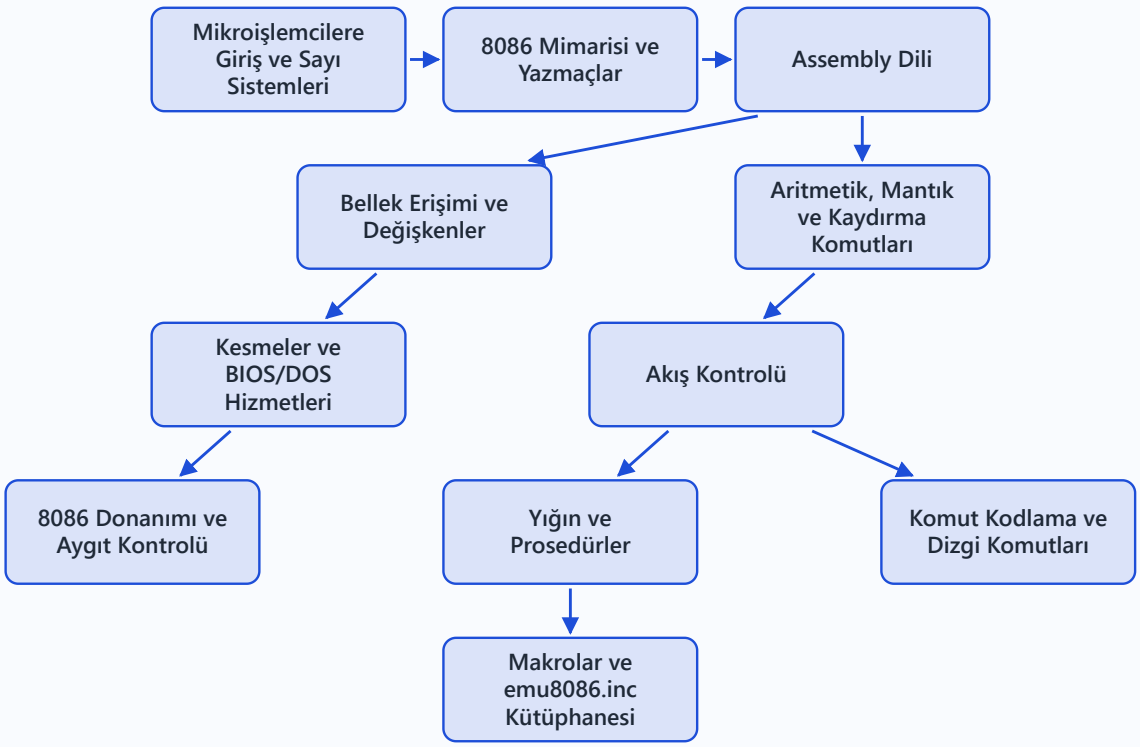
Ek A Terim Sözlüğü 959

Ek B Mikroişlemciler İçin Ön Bilgiler 961

Ek C Mikroişlemcilerde Önemli İsimler 969

Dersin Haritası

Bu sayfa dersin bütün bölümlerini tek bir haritada birleştirir. Her bölüm bir düğümdür; bir ok, sonraki konunun önceki konunun hangi fikrine dayandığını gösterir. Derse başlamadan önce okunursa konuların nereye gittiğini, ders sırasında okunursa bulunulan yeri gösterir.



Şekil 1: Dersin haritası. Oklar, bir konunun hangi konunun üzerine kurulduğunu gösterir. Mimariden assembly diline, komutlara, sistem hizmetlerine ve donanıma geçilir.

Şekil 1 konular arasındaki bağları gösterir. Her düğüm bir bölümdür; bir ok, sonraki konunun önceki-nin hangi fikrine dayandığını belirtir. Haritada yalnızca en yakın bağlar çizilmiştir; bir bölümün dayandığı bütün bölümler Tablo 1'dedir.

Konu (bölüm)	Dayandığı konu	Bağ
Mikroişlemcilere Giriş ve Sayı Sistemleri (1)	—	Getir, çöz, yürüt döngüsü; ikili ve onaltılık sayılar, ikiye tümleyen gösterim.
8086 Mimarisi ve Yazmaçlar (2)	Mikroişlemcilere Giriş ve Sayı Sistemleri	8086'nın iç yapısı: BIU ve EU, yazmaçlar, bayraklar ve kesim:bağıl konum adresi.

Konu (bölüm)	Dayandığı konu	Bağ
Assembly Dili (3)	8086 Mimarisi ve Yazmaçlar	Makine dilinden assembly'ye: assembler, bağlayıcı ve ilk program.
Bellek Erişimi ve Değişkenler (4)	8086 Mimarisi ve Yazmaçlar, Assembly Dili	Adresleme modları, etkin ve fiziksel adres; DB, DW, DUP ile değişkenler.
Kesmeler ve BIOS/DOS Hizmetleri (5)	8086 Mimarisi ve Yazmaçlar, Bellek Erişimi ve Değişkenler	INT sırasında FLAGS, CS ve IP yığına yazılır; kesme vektör tablosu bellektedir.
Aritmetik, Mantık ve Kaydırma Komutları (6)	Mikroişlemcilere Giriş ve Sayı Sistemleri, 8086 Mimarisi ve Yazmaçlar, Assembly Dili	Hesabın sonucu bayrakları kurar: CF, ZF, SF, OF; MUL, DIV ve kaydırmalar.
Akış Kontrolü (7)	Aritmetik, Mantık ve Kaydırma Komutları	Bayrakları okuyan koşullu atlamalar; CMP sonrası karar ve LOOP ile döngü.
Yığın ve Prosedürler (8)	Bellek Erişimi ve Değişkenler, Kesmeler ve BIOS/DOS Hizmetleri, Akış Kontrolü	SS:SP ile yığın; CALL ve RET dönüş adresini yığında saklar; BP ile yığın çerçevesi.
Makrolar ve emu8086.inc Kütüphanesi (9)	Yığın ve Prosedürler	Prosedürün metinsel karşılığı: MACRO ile tanım ve açılım, emu8086.inc yordamları.
8086 Donanımı ve Aygıt Kontrolü (10)	8086 Mimarisi ve Yazmaçlar, Kesmeler ve BIOS/DOS Hizmetleri	40 pinli 8086, yol döngüleri ve 8259A; IN ve OUT ile aygıt kontrolü.
Komut Kodlama ve Dizgi Komutları (11)	Bellek Erişimi ve Değişkenler, Aritmetik, Mantık ve Kaydırma Komutları, Akış Kontrolü	Komutların makine kodu: işlem kodu, D ve W bitleri, ModR/M; REP önekli dizgi komutları.

Tablo 1: Haritadaki bağlar.

Haritanın ana omurgası şöyle okunabilir: **sayı → mimari → assembly → komut → donanım**. İlk üç bölüm işlemciyi ve onu programlamanın dilini kurar. Buradan iki dal çıkar: bellek dalı adresleme ve değişkenlerle başlar, kesmelere ve aygıt kontrolüne varır; komut dalı aritmetik ve bayraklarla başlar, akış kontrolüne ve prosedürlere varır. Yığın iki dalın buluştuğu yerdir: hem CALL ile RET hem INT ile IRET onu kullanır. Komut kodlama bölümü öğrenilen komutların makine kodunda nasıl göründüğünü gösterir.

KISIM I

Temeller

Bölüm 1–3

KISIM I · BÖLÜM 1

Mikroişlemcilere Giriş ve Sayı Sistemleri

Mikroişlemci, komutları bellekten getirip çözen ve yürüten bir bilgisayar sisteminin merkezidir. Bölüm mikroişlemci ile mikrodenetleyici ayrımı, getir–çöz–yürüt döngüsü, işlemcinin iç yapısı ve saat hızıyla başlar; CISC, RISC ve EPIC tasarımları, özel amaçlı işlemciler ve 4004'ten günümüze Intel x86 ailesinin gelişimi ele alınır. Sayı sistemleri kısmında onluk, ikili ve onaltılık tabanlar, taban dönüşümleri, birler tümleyeni ve ikiye tümleyen ile işaretli sayılar ve ikili aritmetik işlenir. IEEE 754 kayan nokta biçimi, BCD, Gray kodu ve bayt sıralaması gibi kodlamalarla bölüm tamamlanır.

Kod (8086): `github.com/sercankulcu/microprocessors-8086/Bolum01`

BU BÖLÜMÜN NOTLARI

1.1 Giriş

1.2 Sayı Sistemleri

KISIM I · BÖLÜM 1.1

Giriş

Bir çamaşır makinesinin kartında, bir otomobilin motor denetiminde ya da bir dizüstü bilgisayarda aynı düşünce çalışır: bellekteki komutları sırayla okuyup uygulayan, programlanabilir bir yonga. Bu not o yongayı, mikroişlemciyi, dışarıdan içeriye doğru tanıtır. Önce mikroişlemcinin bir bilgisayar sistemindeki yerini, onu belleğe ve giriş/çıkış birimlerine bağlayan yolları ve mikrodenetleyiciden farkını ele alır. Ardından her komut için yinelenen getir–çöz–yürüt döngüsünü 8086 makine kodu üzerinde adım adım izler. İşlemcinin iç yapısı, saat sinyali ve bir programın süresini belirleyen üç büyüklük (komut sayısı, komut başına tur, saat frekansı) sayısal örneklerle verilir. Sonraki kesimler komut kümesi mimarilerini (CISC, RISC, EPIC), özel amaçlı işlemci türlerini ve Intel x86 ailesinin 4004'ten Core i7'ye uzanan gelişimini anlatır. Not, modern işlemcilerde performansı belirleyen önbellek, çok çekirdek ve güç yönetimi gibi özelliklerle biter. Kod örneklerinin hepsi 8086 assembly ile yazılmıştır ve her birinde yazmaçların, bayrakların ve belleğin nasıl değiştiği tablo hâlinde izlenir.

ÖĞRENME ÇIKTILARI

Temel kavramları tanımlamak
Mikroişlemciyi, mikrodenetleyiciyi ve bir bilgisayar sisteminin adres, veri ve kontrol yollarını tanımlamak; *n* hatlı bir adres yolunun kaç bayt adreslediğini hesaplamak.

Komut döngüsünü izlemek
Getir–çöz–yürüt döngüsünü makine kodu üzerinde adım adım izlemek; komut işaretçisinin (IP) sıralı akışta ve atlamada nasıl değiştiğini göstermek.

Performansı hesaplamak
Saat frekansı, periyot, yol döngüsü, komut başına tur (CPI) ve çalışma süresi arasındaki ilişkiyi kullanmak; küçük bir 8086 döngüsünün kaç saat turu sürdüğünü saymak.

Mimari türlerini karşılaştırmak
CISC, RISC ve EPIC yaklaşımlarını ve vektör, SIMD, DSP, GPU, sembolik işlemci türlerini temel fikirleri, güçlü yanları ve örnekleriyle karşılaştırmak; boru hattının bitiş turunu hesaplamak.

x86 ailesini açıklamak
4004'ten Core i7'ye Intel işlemcilerinin veri ve adres yolu genişliklerini, getirdikleri yenilikleri ve 8086'nın bu ailedeki yerini açıklamak.

Modern özellikleri tanımak
Önbellek ve bellek hiyerarşisi, çok çekirdek, sanallaştırma ve güç yönetimi gibi kavramları tanımak; saat hızının tek başına performans ölçüsü olmadığını sayısal örnekle göstermek.

1.1.1 Mikroişlemci ve bilgisayar sistemi

Mikroişlemciyi anlamanın en kısa yolu, onun bir bilgisayarda hangi işi üstlendiğine bakmaktır. Bu kesim mikroişlemciyi tanımlar, onu bellek ve giriş/çıkış birimleriyle birleştiren yolları açıklar ve aynı işlemcinin bir mikrodenetleyicide nasıl başka bir biçim aldığını gösterir.

1.1.1.1 Mikroişlemci nedir?

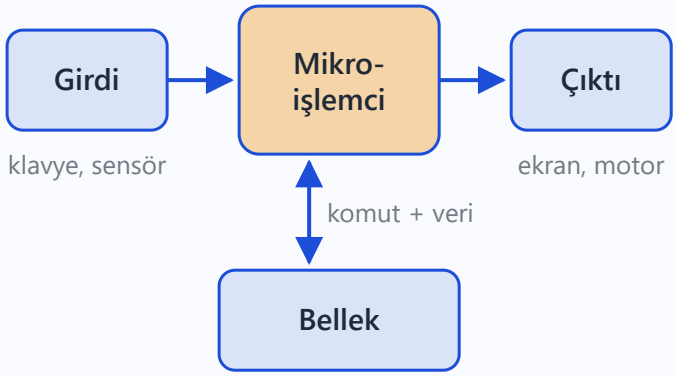
Her bilgisayarın merkezinde komutları yürüten bir birim bulunur. Bu birime **merkezî işlem birimi** (*central processing unit*, CPU) denir. İlk bilgisayarlarda CPU dolaplar dolusu lamba, röle ya da ayrık transistörden kurulurdu. Tümleşik devre teknolojisi geliştikçe aynı devreler önce birkaç kart üzerine, sonra tek bir silisyum parçası üzerine sığdı. Mikroişlemci bu son adımın ürünüdür.

Tanım 1.1.1

Mikroişlemci (*microprocessor*), bir bilgisayarın merkezî işlem biriminin tek bir **yonga** (*chip*) üzerinde gerçeklenmiş hâlidir. Programlanabilir bir aygıttır: ne yapacağını bellekte duran **makine dili** komutları belirler. Bu komutları sırayla okur ve uygular.

Tanımdaki “programlanabilir” sözcüğü mikroişlemciyi sayısal tasarım derslerinde kurulan devrelerden ayırır. Bir toplayıcı devresi yalnız toplar; bir sayıcı yalnız sayar. Devrenin ne yaptığı bağlantılarla sabitlenmiştir ve işlevi değiştirmek için devreyi yeniden kurmak gerekir. Mikroişlemcinin bağlantıları ise hiç değişmez. Değişen, bellekteki komutlardır. Aynı yonga bir programla metin düzenler, başka bir programla motor sürer. Bu esneklik mikroişlemciyi bugün hemen her elektronik ürünün parçası yapmıştır.

Mikroişlemcinin iş akışı üç adımla özetlenir. İşlemci önce **girdi** alır: klavyeden basılan bir tuş, bir sensörün ölçtüğü sıcaklık ya da diskten okunan bir dosya. Sonra bu veri üzerinde **aritmetik ve mantıksal işlemler** yapar: toplar, çıkarır, karşılaştırır, bitleri maskeler. En sonunda istenen **çıktıyı** üretir: ekrana bir karakter yazar, bir motoru döndürür ya da bir LED'i yakar. Bu üç adımın arasında işlemci sürekli bellekle konuşur. Hem yürüteceği komutları hem de üzerinde çalışacağı verileri bellekten okur, ara sonuçları da belleğe yazar. Şekil 1.1.1 bu ilişkiyi gösterir.



Şekil 1.1.1: Mikroişlemcinin iş akışı. Girdi (klavye, sensör) işlemciye girer, işlenir ve çıktı (ekran, motor) olarak dışarı verilir. İşlemci komutları ve verileri bellekten alır, sonuçları belleğe yazar.

Basit bir hesap makinesi bu akışın gündelik bir örneğidir. Tuşlar girdidir; toplama işlemi işlemdir; ekranda beliren sonuç çıktıdır. Hesap makinesinin içindeki küçük işlemci de tuşları okuyan, sayıları toplayan ve rakamları ekrana yazan bir programı yürütür. Bu programı değiştirmek, aynı yongaya başka bir iş yaptırmak demektir.

Mikroişlemcinin anladığı tek dil **makine dilidir**. Makine dilinde her komut bir bayt dizisidir. Örneğin 8086'da `B0 05` baytları "AL yazmacına 5 yükle" anlamına gelir. İnsanın bu sayıları ezberlemesi zor olduğu için programlar çoğunlukla **assembly** diliyle yazılır. Assembly, makine komutlarına `MOV`, `ADD`, `JMP` gibi kısa adlar, yani **anımsatıcılar** (*mnemonic*) verir. Bir çevirici program, assembler, bu metni makine koduna çevirir. Bu notta komutlar assembly ile yazılır; gerektiğinde karşılık gelen makine kodu baytları da verilir.

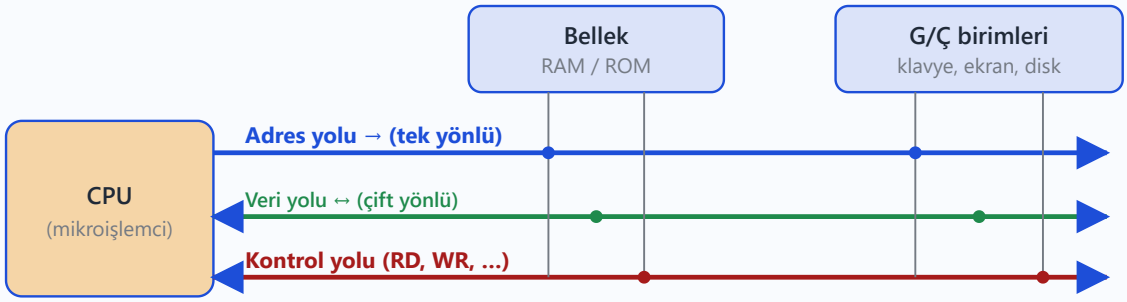
1.1.1.2 Bilgisayar sistemi ve yollar

Mikroişlemci tek başına çalışamaz. Komutların ve verilerin durduğu bir **belleğe** ve dış dünyayla bağlantı kuran **giriş/çıkış** (G/Ç) birimlerine gerek duyar. Bu üç parça, yani CPU, bellek ve G/Ç birimleri, ortak bir kablo demetiyle birbirine bağlanır. Bu demete **yol** (*bus*) denir. Sistemin bütün parçalarını bağlayan yol ise **sistem yolu** adını alır.

Tanım 1.1.2

Yol, birimler arasında aynı türden bilgiyi taşıyan paralel iletken hatların topluluğudur. Bir bilgisayar sisteminde üç yol bulunur:

- **Adres yolu** (*address bus*): işlemcinin hangi bellek gözüne ya da hangi G/Ç portuna erişeceğini seçer. Tek yönlüdür; adresi yalnız işlemci üretir.
- **Veri yolu** (*data bus*): okunan ya da yazılan veriyi taşır. Çift yönlüdür; veri okumada bellekten işlemciye, yazmada işlemciden belleğe akar.
- **Kontrol yolu** (*control bus*): okuma, yazma, bellek ya da G/Ç seçimi, saat ve kesme gibi zamanlama ve kontrol sinyallerini taşır.



Şekil 1.1.2: CPU, bellek ve G/Ç birimleri üç yolla birbirine bağlıdır. Adres yolu tek yönlüdür (CPU'dan dışarı), veri yolu çift yönlüdür, kontrol yolu RD (okuma) ve WR (yazma) gibi sinyalleri taşır.

Bir bellek okumasının nasıl yürüdüğü üç yolun görev paylaşımını açıkça gösterir. İşlemci okumak istediği gözün adresini adres yoluna koyar. Ardından kontrol yolundaki okuma sinyalini (8086'da `RD` ucunu) etkin yapar. Bellek, adres hatlarındaki deseni çözer, ilgili gözü seçer ve içindeki veriyi veri yoluna sürer. İşlemci veri yolundaki değeri yakalar ve okuma sinyalini bırakır. Yazma da benzer biçimde yürür; yalnız bu kez veriyi veri yoluna işlemci sürer ve kontrol yolunda yazma sinyali (`WR`) etkin olur. Bu adımların her biri saat sinyaline bağlı olarak belirli bir sırayla olur. Bir okuma ya da yazmanın baştan sona sürdüğü zamana **yol döngüsü** (*bus cycle*) denir.

Yolların genişliği, yani kaç hattan oluştuğu, sistemin sınırlarını belirler. Adres yolunda her hat 0 ya da 1 taşır. n hatla 2^n farklı desen kurulur ve her desen bir bellek gözünü seçer. Bu yüzden n hatlı bir adres yolu 2^n bayt belleği adresler. 8086'nın adres yolu 20 hattır: $2^{20} = 1.048.576$ bayt, yani 1 MB bellek. Veri yolunun genişliği ise bir yol döngüsünde aktarılan bit sayısıdır. 8086'nın veri yolu 16 bittir; bir yol döngüsünde 2 bayt, yani bir **sözcük** (*word*) taşır.

Örnek 1.1.1

16, 20 ve 24 hatlı adres yolları kaç bayt bellek adresler?

- 16 hat: $2^{16} = 65.536$ bayt = 64 KB (1 KB = 1024 bayt; 65.536 / 1024 = 64). 8080 ve 8085 böyledir.
- 20 hat: $2^{20} = 1.048.576$ bayt = 1024 KB = 1 MB. 8086 ve 8088 böyledir.
- 24 hat: $2^{24} = 16.777.216$ bayt = 16 MB. 80286 böyledir.

Tersten bakıldığında 1 MB'ın her baytını ayrı ayrı seçmek için $log_2(2^{20}) = 20$ hat gerekir. Her ek hat adreslenebilir belleği iki katına çıkarır.

8086'nın yapısında ilginç bir ayrıntı vardır: işlemci 16 bitliktir, ama adres yolu 20 bittir. 16 bitlik yazmaçlarla 20 bitlik adres doğrudan tutulamaz. 8086 bu sorunu iki 16 bitlik değeri, **kesim** (*segment*) ve **bağıl konum** (*offset*) değerlerini birleştirerek çözer. Fiziksel adres, kesim değerinin 16 ile çarpılıp bağıl konuma eklenmesiyle bulunur. Bu not boyunca adresler çoğunlukla yalnız bağıl konumla, örneğin `0100h` diye yazılır; kesim değeri sabit kabul edilir.

Sık yapılan hata

Veri yolunun genişliği ile adreslenebilir bellek karıştırılır. “8086 16 bitlik olduğu için 64 KB bellek adresler” sonucu yanlıştır. Adreslenebilir belleği veri yolu değil, adres yolu belirler: 8086'nın 20 adres hattı 1 MB'a ulaşır. 16 bitlik yazmaçlar yalnız bir kesim içindeki 64 KB'lık pencereyi doğrudan gösterebilir; kesim yazmacı bu pencereyi 1 MB içinde kaydırır.

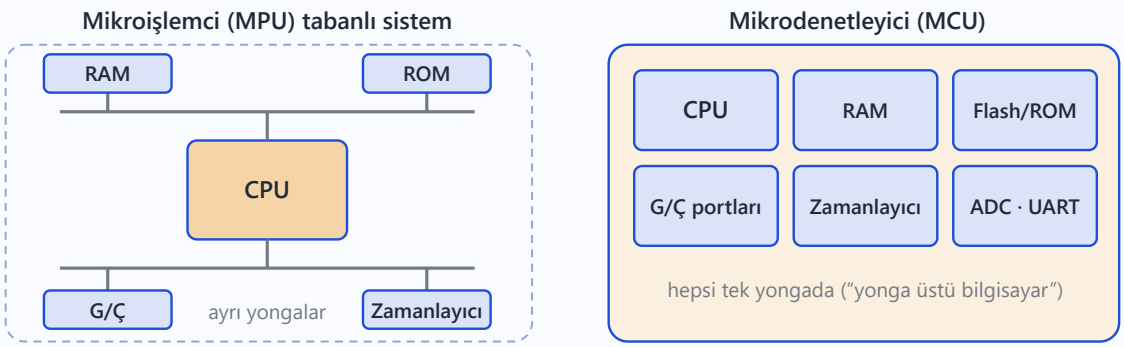
1.1.1.3 Mikroişlemci ve mikrodenetleyici

Bir mikroişlemciyle kurulan bilgisayarda bellek ve G/Ç birimleri ayrı yongalardır. 8086'lı bir kartta işlemcinin yanında RAM yongaları, programın durduğu ROM yongası, G/Ç için bir arayüz yongası ve zamanlayıcı yongası yan yana dizilir. Yongalar kart üzerindeki yollarla birbirine bağlanır. Bu düzen esnektir: bellek miktarı ve çevre birimleri ihtiyaca göre seçilir. Bedeli ise kart alanı, maliyet ve güç tüketimidir.

Pek çok uygulama bu esnekliğe gerek duymaz. Bir çamaşır makinesinin kartı yıllarca tek bir programı çalıştırır, birkaç kilobayt bellekle yetinir ve birkaç düğmeyle birkaç motoru yönetir. Bu tür işler için CPU, bellek ve çevre birimlerini tek yongada toplayan aygıtlar geliştirilmiştir.

Tanım 1.1.3

Mikrodenetleyici (*microcontroller*, MCU), CPU ile birlikte RAM, Flash/ROM program belleği, G/Ç portları, zamanlayıcılar ve analog-sayısal çevirici (ADC) gibi çevre birimlerini aynı yongada toplar. Bu yüzden “yonga üstü bilgisayar” diye de anılır. Mikroişlemci (MPU) ise yalnız CPU'yu içerir; bellek ve çevre birimleri dışarıdan eklenir.



Şekil 1.1.3: Solda mikroişlemci tabanlı sistem: CPU, RAM, ROM, G/Ç ve zamanlayıcı ayrı yongalardır. Sağda mikrodenetleyici: CPU, RAM, Flash/ROM, G/Ç portları, zamanlayıcı, ADC ve UART tek yongadadır.

Mikroişlemci örnekleri arasında 8086, Intel Core i7 ve AMD Ryzen sayılabilir. Mikrodenetleyici örnekleri ise 8051, PIC, Arduino kartlarındaki AVR ve STM32 ailesidir. Tablo 1.1.1 iki türü sekiz ölçütle karşılaştırır.

Ölçüt	Mikroişlemci (MPU)	Mikrodenetleyici (MCU)
İçerik	CPU (ALU, kontrol birimi, yazmaçlar)	CPU + RAM + Flash + G/Ç + zamanlayıcı + ADC…
Bellek	dışarıda, büyük (GB)	yonga içinde, küçük (KB–MB)
Saat hızı	yüksek (GHz)	düşük–orta (MHz)
Güç	yüksek (W–100 W)	çok düşük (mW, uyku modunda µW)
Maliyet	yüksek; ek yongalar gerekir	düşük; tek yonga yeterli
Yazılım	genelde işletim sistemi + çok sayıda uygulama	tek, sabit bir program (gömülü yazılım)
Kullanım	PC, sunucu, dizüstü	çamaşır makinesi, kumanda, otomobil ECU, IoT

Tablo 1.1.1: Mikroişlemci ile mikrodenetleyicinin karşılaştırılması.

Tablodaki farkların çoğu tek bir tasarım kararından doğar: belleğin nerede durduğu. MPU'da bellek dışarıda olduğu için gigabaytlarca RAM bağlanabilir. Bu da işletim sistemini ve aynı anda çalışan pek çok uygulamayı mümkün kılar. MCU'da bellek yonganın içindedir ve yonga alanı sınırlıdır. Kilobaytlarla ölçülen bellek, sabit ve küçük bir programa yeter. Saat hızı ve güç de bu ayrımı izler. Yüksek frekansta çalışan bir MPU onlarca watt harcar ve soğutucu ister. Düşük frekanslı bir MCU miliwatt düzeyinde kalır ve uyku modunda mikrowatta iner; bir pil onu yıllarca besleyebilir.

Örnek 1.1.2

Bir çamaşır makinesi kontrol kartı ve bir dizüstü bilgisayar için hangi tür seçilir?

Çamaşır makinesinin işi tektir ve değişmez: düğmeleri okumak, su seviyesini ve sıcaklığı izlemek, motoru ve vanaları sürmek. Program küçüktür; maliyet ve güç önemlidir. Bu iş için mikrodenetleyici seçilir. Dizüstü bilgisayar ise işletim sistemi çalıştırır, gigabaytlarca bellek ister

ve kullanıcı her an yeni bir uygulama açabilir. Yüksek performans ve büyük bellek gerektiği için mikroişlemci seçilir.

Mikroişlemci ile mikrodenetleyici arasındaki fark yongaya nelerin sığdırıldığıyla ilgilidir; içerideki CPU'nun çalışma ilkesi aynıdır. Her ikisi de komutları bellekten getirir, çözer ve yürütür. Her ikisinde de yazmaçlar, bir aritmetik-mantık birimi ve bayraklar bulunur. Bu ders 8086 mikroişlemcisini inceler. 8086'nın yolları, yazmaçları ve komutları üzerinde öğrenilen kavramlar bir mikrodenetleyiciye geçildiğinde de aynı mantıkla kullanılır.

1.1.2 Komut döngüsü

Mikroişlemcinin yaptığı her şey tek bir döngünün tekrarından ibarettir. İşlemci bir komutu bellekten alır, ne istendiğini anlar, istenen işi yapar ve sonucu yerine koyar. Sonra aynı işi bir sonraki komut için yineler. Bu kesim döngünün aşamalarını tanımlar, döngüyü üç komutluk bir 8086 programının makine kodu üzerinde izler ve program sayacının akışı nasıl yönettiğini gösterir.

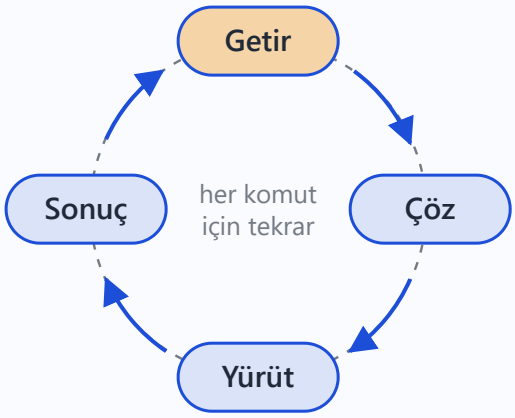
1.1.2.1 Getir, çöz, yürüt

Tanım 1.1.4

Komut döngüsü (*instruction cycle*), işlemcinin her komut için yinelediği aşamalar dizisidir:

1. **Getir** (*fetch*): komutu, program sayacının gösterdiği adresten bellekten çeker.
2. **Çöz** (*decode*): **işlem kodunu** (*opcode*) çözer; hangi işlemin hangi **işlenenlerle** (*operand*) yapılacağını belirler.
3. **Yürüt** (*execute*): çözülen komutu uygular; ALU'da hesap yapar, veri taşır ya da atlar.
4. **Sonuç** (*result*): sonucu yazmaca ya da belleğe yazar ve bayrakları günceller.

Bazı kitaplar dördüncü aşamayı yürütmenin içinde sayar ve döngüye kısaca “getir–çöz–yürüt” der. Hangi adlandırma seçilirse seçilsin, sıra aynıdır. İşlemci açık olduğu sürece bu döngü hiç durmadan döner. Şekildeki ok, son aşamadan ilk aşamaya geri döner; döngünün bittiği tek durum işlemcinin durdurulmasıdır.



Şekil 1.1.4: Getir, çöz, yürüt ve sonuç aşamaları bir döngü oluşturur. Döngü her komut için yeniden başlar.

Her aşamayı işlemcinin farklı bir parçası üstlenir. Getirmeyi yol arayüzü yapar: program sayacındaki adresi adres yoluna koyar, okuma sinyalini verir ve veri yolundan gelen baytları alır. Çözmeyi kontrol birimi yapar: işlem kodu baytını bir tabloya bakar gibi yorumlar ve hangi birimlerin çalışacağını belirleyen kontrol sinyallerini üretir. Yürütmede ALU devreye girer ya da veri bir yazmaçtan başkasına aktarılır. Sonuç aşamasında yazmaç dosyasına veya belleğe yazılır.

Aşamalar sırayla yürüdüğü için birim zamanda yürütülen komut sayısı sınırlıdır: getirme sürerken ALU boş bekler, yürütme sürerken yol boş bekler. Hızlı işlemciler bu boşluğu aşamaları üst üste bindirerek kapatır. Bir komut yürütülürken bir sonraki çözülür, ondan sonraki de getirilir. Bu düzene **boru hattı** (*pipeline*) denir; RISC işlemcilerin temel tekniğidir. 8086 bu fikrin basit bir biçimini kullanır: yol arayüz birimi, yürütme birimi çalışırken sıradaki komut baytlarını önceden getirir ve 6 baytlık bir kuyrukta bekletir.

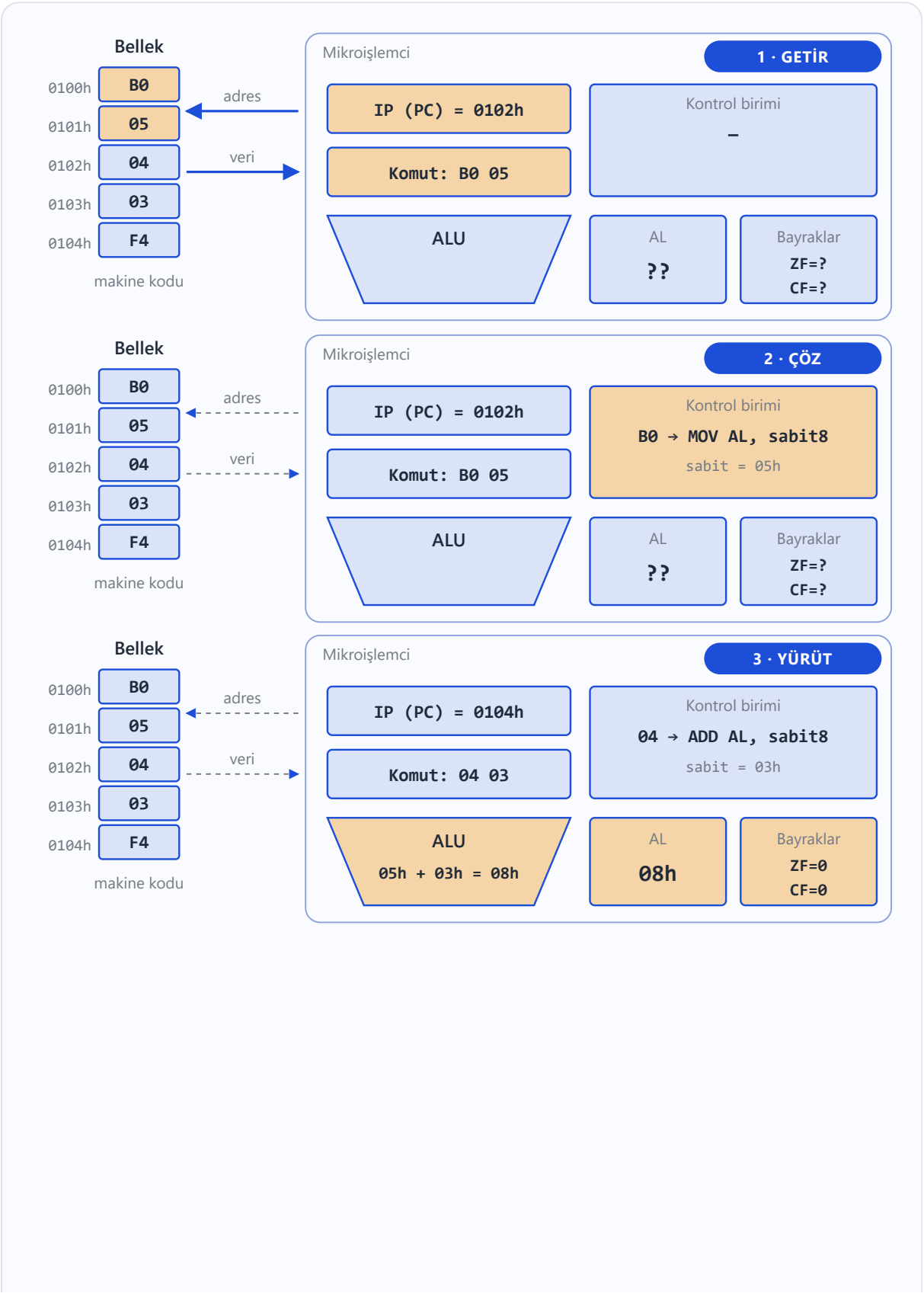
1.1.2.2 Makine kodu üzerinde izleme

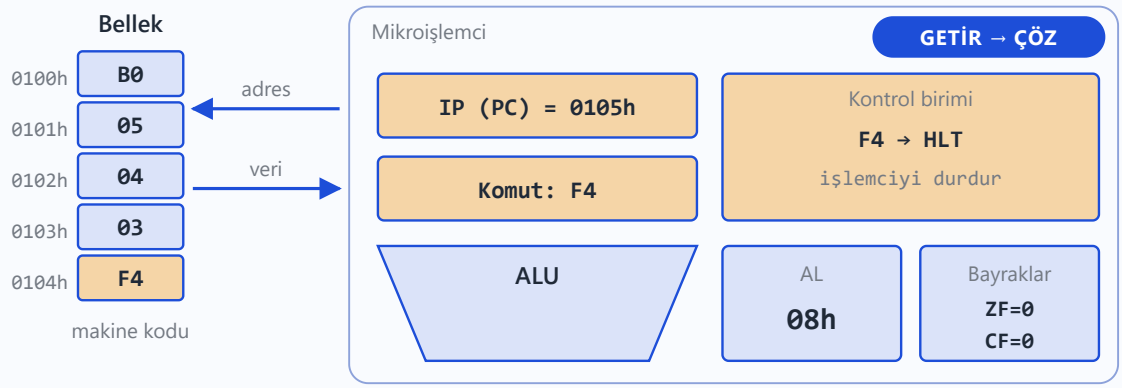
Döngüyü somut görmek için üç komutluk bir program ele alalım:

```
mov al, 5     ; B0 05
add al, 3     ; 04 03
hlt           ; F4
```

Assembler bu metni beş baytlık makine koduna çevirir. Program belleğe `0100h` adresinden başlayarak yüklenmiştir: `0100h`'de `B0`, `0101h`'de `05`, `0102h`'de `04`, `0103h`'te `03`, `0104h`'te `F4` durur. İlk bayt işlem kodudur. `B0`, "AL yazmacına 8 bitlik sabit yükle" demektir ve sabit, işlem kodunun hemen arkasındaki baytta bulunur. Böyle komutun içine gömülü sabite **anlık değer** (*immediate value*) denir. `04` işlem kodu "AL'ye 8 bitlik anlık değer ekle" anlamına gelir. `F4` ise `HLT` komutudur ve işlenen almaz; tek bayttır.

8086'da program sayacının adı **komut işaretçisidir** (*instruction pointer*, IP). IP her zaman bir sonraki getirilecek baytın bağıl konumunu tutar. Başlangıçta IP = `0100h`'dir. Şekil 1.1.5 döngünün dört anını gösterir.





Şekil 1.1.5: `MOV AL,5 · ADD AL,3 · HLT` programında komut döngüsü. Yukarıdan aşağı: ilk komut getirilmiş (IP = 0102h); `B0` çözülmüş; ikinci komut yürütülmüş (AL = 08h, ZF = 0, CF = 0); `F4` getirilip çözülmüş (IP = 0105h).

İzlemenin adımları şöyledir:

1. **Getir.** IP'deki `0100h` adres yoluna konur. Bellek `B0` baytını döndürür. Kontrol birimi işlem kodundan komutun 2 bayt olduğunu anlar ve `05` baytını da getirir. IP komut uzunluğu kadar artar: `0100h + 2 = 0102h`.
2. **Çöz.** Kontrol birimi `B0`'ı "AL'ye 8 bitlik sabit yükle" diye çözer; sabit `05h`'dir.
3. **Yürüt ve sonuç.** `05h`, AL yazmacına yazılır. ALU kullanılmaz. `MOV` komutu hiçbir bayrağı değiştirmez; bayraklar önceki değerlerinde kalır.
4. **Getir.** Döngü yeniden başlar. `0102h`'deki `04 03` getirilir; IP = `0104h` olur.
5. **Çöz.** `04` = "AL'ye 8 bitlik sabit ekle". İşlenenler AL'deki `05h` ve sabit `03h`'tir.
6. **Yürüt ve sonuç.** ALU `05h + 03h = 08h` hesaplar ve sonucu AL'ye yazar. Sonuç sıfır olmadığı için sıfır bayrağı ZF = 0'dır. En üst bitten taşan elde olmadığı için elde bayrağı CF = 0'dır.
7. **Getir ve çöz.** `0104h`'teki `F4` getirilir; IP = `0105h` olur. `F4`, `HLT` diye çözülür ve işlemci bir kesme ya da sıfırlama gelene kadar durur.

Adres	Makine kodu	Komut	IP (sonra)	AL	ZF	CF
—	—	başlangıç	0100h	??	?	?
0100h	`B0 05`	`mov al, 5`	0102h	05h	?	?
0102h	`04 03`	`add al, 3`	0104h	08h	0	0
0104h	`F4`	`hlt`	0105h	08h	0	0

Tablo 1.1.2: `MOV AL,5 · ADD AL,3 · HLT` programının izlemesi. Her satır o komutun yürütülmesinden sonraki durumu verir.

`ADD` yalnız ZF ile CF'yi değil, 8086'nın altı durum bayrağının hepsini günceller. `08h = 0000 1000b` sonucunda en üst bit 0 olduğu için işaret bayrağı SF = 0'dır. İki pozitif sayının toplamı pozitif kaldığı için taşma bayrağı OF = 0'dır. Düşük dört bitte `5 + 3 = 8` 4 bite sığdığı için yardımcı elde bayrağı AF

= 0'dır. Sonucun düşük baytında tek sayıda (bir tane) 1 biti olduğu için eşlik bayrağı PF = 0'dır. Tabloda yalnız ZF ve CF gösterilmiştir.

Bu izlemede iki ayrıntı önemlidir. Birincisi, IP komut getirilir getirilmez artar; komut yürütülürken IP zaten bir sonraki komutu gösterir. Atlama komutları bu değeri değiştirerek akışı yönlendirir. İkincisi, işlemci baytların "komut" mu "veri" mi olduğunu bilmez. `0101h`'deki `05` işlem kodu olarak değil sabit olarak okunur, çünkü kontrol birimi `B0`'ı çözdüğünde arkasından bir sabit geleceğini bilir. IP yanlışlıkla `0101h`'e gitseydi işlemci `05`'i işlem kodu sayar ve bambaşka bir komut yürütürdü.

Gerçek 8086'da getirme bu anlatımdan biraz farklı yürür. Yol arayüz birimi (BIU), yürütme birimi (EU) bir komutla uğraşırken sıradaki baytları önceden getirir ve 6 baytlık önden getirme kuyruğuna yazar. EU bir komutu bitirdiğinde sıradaki komutun baytları çoğunlukla kuyrukta hazırdır; getirme için belleği beklemek gerekmez. Programcı açısından sonuç değişmez: komutlar yine sırayla ve aynı etkilerle yürür.

> **Sık yapılan hata**
>
> IP'nin her komutta 1 arttığı sanılır. IP komut sayısını değil bayt adresini tutar ve komut uzunluğu kadar artar. 8086 komutları 1 ile 6 bayt arasındadır: `hlt` 1, `mov al, 5` 2, `mov cx, 3` 3 bayttır. Bir komutun adresini bulmak için önceki komutların uzunlukları toplanır.

1.1.2.3 Üç temel işlev

Bir işlemcinin komut kümesi yüzlerce komut içerse de bu komutlar üç işlevden birine hizmet eder.

Aritmetik ve mantıksal işlemler. Toplama, çıkarma, karşılaştırma, AND, OR, XOR ve kaydırma bu gruptadır. Bu işlemleri **aritmetik-mantık birimi** (*arithmetic logic unit*, ALU) yapar. Yeni işlemciler kayan noktalı sayılarla da çalışır; bu iş için ayrı bir **kayan nokta birimi** (*floating-point unit*, FPU) bulunur. 8086'da örnekler: `ADD AX, BX` (AX'e BX'i ekle), `CMP AL, 10` (AL'yi 10 ile karşılaştır), `AND AL, 0Fh` (AL'nin üst dört bitini sıfırla).

Veri taşıma. Veri yazmaçlar, bellek ve G/Ç arasında bir yerden başka bir yere kopyalanır. Programlarda en sık görülen işlem budur; derleyicilerin ürettiği kodun büyük kısmı taşıma komutlarıdır. Örnekler: `MOV AX, [1234h]` (bellekteki sözcüğü AX'e oku), `MOV [BX], AL` (AL'yi BX'in gösterdiği adrese yaz), `IN AL, 60h` (60h numaralı G/Ç portundan bir bayt oku).

Program sayacını değiştirme. Program sayacı sıradaki komutun adresini tutar. Ona yeni bir değer yazmak, akışı başka bir yere atlatmak demektir. Koşullu atlamalar, bayraklara bakarak atlayıp atlamamaya karar verir; programdaki her `if` ve her döngü bu yolla kurulur. Örnekler: `JMP etiket` (koşulsuz atla), `JZ sifirsa` (ZF = 1 ise atla), `CALL alt_program` (dönüş adresini saklayıp alt programa atla).

Her program bu üç işlevin, yani hesaplamanın, taşımanın ve sırayı değiştirmenin birleşimidir. Bir sıralama programı sayıları okur (taşıma), karşılaştırır (hesap), sonuca göre yer değiştirir ya da geçer (sırayı değiştirme).

1.1.2.4 Program sayacı ve döngü

Sıralı akışta IP kendiliğinden artar. Döngü kurmak için IP'ye geriye doğru bir adres yazılması gerekir. Aşağıdaki program 3 + 2 + 1 toplamını hesaplar. Soldaki sayılar komutların bellekteki bağıl konumlarıdır (onaltılık).

```
0100  mov  cx, 3     ; sayaç = 3
0103  mov  ax, 0     ; toplam = 0
0106  dongu: add ax, cx
0108  loop dongu     ; CX--, CX≠0 ise atla
010A  hlt            ; dur
```

`mov cx, 3` CX'e sayaç değerini koyar; 16 bitlik yazmaca 16 bitlik sabit yükleyen bu komut 3 bayttır (`B9 03 00`). `mov ax, 0` toplamı sıfırlar; o da 3 bayttır. `add ax, cx` iki yazmacı toplar ve 2 bayttır. `loop dongu` önce CX'i 1 azaltır; CX sıfır değilse `dongu` etiketine atlar, sıfırsa sıradaki komuta geçer. `loop` 2 bayttır ve hiçbir bayrağı değiştirmez. `hlt` işlemciyi durdurur.

Adım	Yürütülen	IP	CX	AX	Açıklama
1	`mov cx, 3`	0103h	0003h	????	3 bayt ilerler
2	`mov ax, 0`	0106h	0003h	0000h	toplam sıfırlandı
3	`add ax, cx`	0108h	0003h	0003h	0 + 3
4	`loop dongu`	0106h	0002h	0003h	CX = 2 ≠ 0, atla
5	`add ax, cx`	0108h	0002h	0005h	3 + 2
6	`loop dongu`	0106h	0001h	0005h	CX = 1 ≠ 0, atla
7	`add ax, cx`	0108h	0001h	0006h	5 + 1
8	`loop dongu`	010Ah	0000h	0006h	CX = 0, atlama yok
9	`hlt`	010Bh	0000h	0006h	işlemci durur

Tablo 1.1.3: IP izlemesi. Her satır o komutun yürütülmesinden sonraki durumu verir.

Tablonun dördüncü satırı sıralı akışın nasıl bozulduğunu gösterir. `loop` getirildiğinde IP `010Ah` olmuştu. CX sıfır olmadığı için `loop` IP'ye `0106h` yazar ve bir sonraki getirme `add ax, cx`'ten yapılır. Makine kodunda hedef adres doğrudan yazılmaz. `loop` komutunun ikinci baytı, sıradaki komutun adresine eklenecek işaretli bir uzaklıktır: `0106h − 010Ah = −4`, yani `FCh`. Böylece komut `E2 FC` diye kodlanır ve program bellekte başka bir adrese taşınsa da doğru çalışır.

Döngü gövdesi üç kez çalışır, çünkü CX başta 3'tür ve her `loop` onu bir azaltır. Toplam dokuz komut yürütülür ve sonuç AX = 3 + 2 + 1 = 6 olur. Döngü sayacı olarak CX'in seçilmesi bir alışkanlık değil zorunluluktur: `loop` komutu sayaç olarak her zaman CX'i kullanır.

Sık yapılan hata

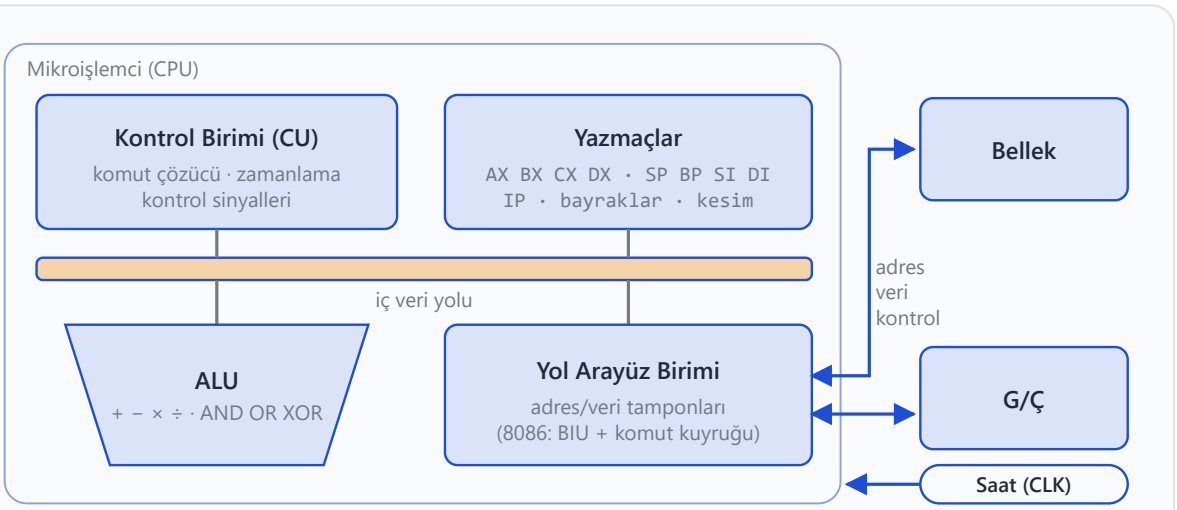
`loop` komutunun önce CX'i sınayıp sonra azalttığı sanılır. Sıra terstir: önce CX bir azalır, sonra yeni değer sınanır. Bu yüzden CX = 0 iken döngüye girilirse ilk `loop` CX'i `FFFFh` yapar ve gövde 65.536 kez çalışır.

1.1.3 Mikroişlemcinin yapısı

Komut döngüsünün aşamalarını işlemcinin farklı birimleri üstlenir. Bu kesim bu birimleri tanıtır, hepsini aynı adımda yürüten saat sinyalini açıklar ve bir programın süresini hesaplamayı gösterir. Kesim, işlemcinin bit genişliğinin ne anlama geldiğiyle ve 16 bitlik bir işlemcinin 32 bitlik sayıları nasıl topladığıyla biter.

1.1.3.1 Yapı bileşenleri

Bir mikroişlemcinin içi dört ana birimden oluşur: kontrol birimi, yazmaçlar, ALU ve yol arayüzü. Bu birimler yonganın içindeki bir **iç veri yoluyla** birbirine bağlıdır. Yol arayüzü ise iç dünyayı dış yollara, yani belleğe ve G/Ç birimlerine bağlar. Şekil 1.1.6 bu düzeni gösterir.



Şekil 1.1.6: Mikroişlemcinin iç yapısı. Kontrol birimi, yazmaçlar, ALU ve yol arayüz birimi iç veri yoluyla bağlıdır. Kontrol birimi öbür birimlere kontrol sinyalleri gönderir. Yol arayüzü dış adres, veri ve kontrol yollarını sürer. Bütün birimler saat sinyaliyle (CLK) eşzamanlı çalışır.

Aritmetik-mantık birimi (ALU). İkili sayılar üzerinde aritmetik (+, −) ve mantıksal (AND, OR, NOT, XOR) işlemleri, karşılaştırmayı ve kaydırmayı yapar. Sayısal tasarımda kurulan tam toplayıcılar, çoklayıcılar ve kaydırıcılar ALU'nun yapı taşlarıdır. ALU iki girişini alır, kontrol biriminden gelen işlem seçme sinyallerine göre bir sonuç üretir ve sonucun özelliklerini **bayraklara** (*flag*) yazar. Sonuç sıfır-

sa sıfır bayrağı (ZF) 1 olur. En üst bitten elde taştıysa elde bayrağı (CF), sonucun en üst biti 1 ise işaret bayrağı (SF) 1 olur. İşaretli sonuç sığmadıysa taşma bayrağı (OF) 1 olur. Koşullu atlamalar bu bayraklara bakar.

Kontrol birimi (*control unit*, CU). İşlemcinin orkestra şefidir. Komutu çözer; hangi yazmacın iç yola bağlanacağını, ALU'nun hangi işlemi yapacağını ve bellek okuma/yazma sinyallerinin ne zaman verileceğini belirler. Kontrol birimi iki yolla kurulabilir. Donanımsal (*hardwired*) kontrol biriminde her komutun sinyalleri doğrudan mantık kapılarıyla üretilir; bu yol hızlıdır ama karmaşık komut kümelerinde tasarımı zorlaşır. Mikroprogramlı kontrol biriminde her komut, yonga içindeki küçük bir ROM'da duran adımlar dizisine, yani **mikrokoda** (*microcode*), çevrilir. 8086 mikroprogramlı bir işlemcidir.

Yazmaçlar (*register*). İşlemci içindeki küçük ve çok hızlı bellek gözleridir. Her yazmaç flip-floplardan kurulmuş bir kaydedicidir ve bir saat turunda okunup yazılabilir. Belleğe gitmek ise bir yol döngüsü, yani birkaç saat turu ister. Bu yüzden programlar sık kullandıkları değerleri yazmaçlarda tutar. Yazmaçlar görevlerine göre gruplanır: genel amaçlı (veri) yazmaçları, adres ve işaretçi yazmaçları, durum (bayrak) yazmacı ve program sayacı. 8086'da genel amaçlı yazmaçlar AX, BX, CX ve DX'tir. SP, BP, SI ve DI işaretçi ve indis yazmaçlarıdır. IP program sayacı, FLAGS bayrak yazmacıdır; CS, DS, SS ve ES de kesim yazmaçlarıdır.

Yol arayüzü. Adres, veri ve kontrol yollarını sürer. Bellekten okunacak ya da belleğe yazılacak her bayt bu birimden geçer. 8086'da bu birimin adı **yol arayüz birimidir** (*bus interface unit*, BIU). BIU fiziksel adresi hesaplar, yol döngülerini yürütür ve komut baytlarını önden getirip 6 baytlık bir kuyrukta bekletir. Komutları çözüp yürüten bölüm ise **yürütme birimidir** (*execution unit*, EU). İki birim aynı anda çalıştığı için getirme ile yürütme üst üste biner.

Saat ve zamanlama. İşlemcinin bütün birimleri saat sinyalinin vuruşlarıyla eşzamanlı ilerler. Bir yazmaç ancak saat kenarında yeni değerini alır; bir yol döngüsünün her adımı belirli bir saat turuna düşer. Bir vuruşun süresi $T = 1/f$ dir.

Önbellek, FPU ve çekirdekler. 8086'da bu birimler yonganın içinde yoktur. Modern işlemcilerde ise aynı yongada önbellek, kayan nokta birimi, birden çok çekirdek ve grafik birimi de bulunur. Bu birimler, 8086'dan bugüne işlemcilere eklenen özelliklerin başında gelir.

Örnek 1.1.3

`add ax, bx` komutu yürütülürken hangi birim ne yapar?

BIU, IP'nin gösterdiği adresten `03 C3` baytlarını getirip kuyruğa koyar (büyük olasılıkla önceki komut yürütülürken). EU kuyruktan baytları alır; kontrol birimi `03` işlem kodunu "bir yazmaca sözcük ekle" diye, `C3` baytını da "hedef AX, kaynak BX" diye çözer. Kontrol birimi AX'i ve BX'i iç yol üzerinden ALU'nun girişlerine bağlar ve ALU'ya toplama işlemini seçtirir. ALU toplamı hesaplar; sonuç AX'e yazılır, bayraklar güncellenir. Bellek erişimi olmadığı için yol arayüzü bu sırada boştadır ve sıradaki komut baytlarını getirebilir.

1.1.3.2 Saat sinyali ve periyot

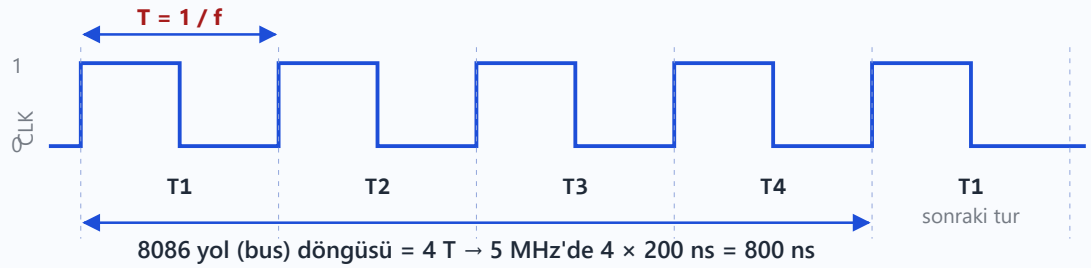
Saat sinyali, belirli bir frekansta 0 ile 1 arasında gidip gelen kare dalgadır. 8086'da bu sinyali 8284A saat üreteci yongası üretir ve işlemcinin CLK ucuna verir. Saat sinyali sayısal tasarımdaki eşzamanlı devrelerin saatiyle aynı işi görür: bütün flip-flopların aynı anda durum değiştirmesini sağlar.

Tanım 1.1.5

Saat frekansı *f*, saat sinyalinin saniyedeki vuruş sayısıdır ve hertz (Hz) ile ölçülür. Bir vuruşa **saat turu** (*clock cycle*) ya da kısaca **tur** denir. Turun süresi **periyottur**:

$$T = 1 / f$$

Bir komut çoğunlukla birkaç saat turu sürer. 8086'da bir bellek erişimi, yani bir yol döngüsü, en az dört saat turudur. Bu turlara **T durumu** (*T state*) denir ve T1, T2, T3, T4 diye adlandırılır. Kabaca T1'de adres yola konur ve T2'de okuma ya da yazma sinyali etkin olur. T3'te veri yolda geçerli hâle gelir, T4'te yol döngüsü kapanır. Bellek yavaşsa T3 ile T4 arasına **bekleme durumları** (*wait state*, Tw) eklenir.



Şekil 1.1.7: Kare dalga saat sinyali. Bir periyot *T*'dir; 8086'nın bir yol döngüsü T1–T4 olmak üzere dört T durumundan oluşur. 5 MHz'de bir yol döngüsü *4 × 200 ns = 800 ns* sürer.

f	*T = 1/f*	*4T*
5 MHz (8086)	200 ns	800 ns
10 MHz (8086-1)	100 ns	400 ns
3 GHz	0,333 ns	1,33 ns

Tablo 1.1.4: Üç saat frekansı için periyot ve dört turluk yol döngüsünün süresi.

Örnek 1.1.4

5 MHz'lik bir 8086'da saat periyodu ve dört turluk bir yol döngüsünün süresi nedir?

f = 5 MHz = 5 × 10⁶ Hz. Periyot *T = 1 / (5 × 10⁶) s = 0,2 × 10⁻⁶ s = 200 ns*. Yol döngüsü dört tur sürdüğü için *4 × 200 ns = 800 ns*'dir. Bu, 5 MHz'lik bir 8086'nın belleğe saniyede en çok *1 / 800 ns = 1,25* milyon kez erişebileceği anlamına gelir. 16 bitlik veri yoluyla her erişimde 2 bayt geldiği için bu, saniyede 2,5 MB'lık bir üst sınırdır.

1.1.3.3 Saat hızı, CPI ve çalışma süresi

Saat periyodu bir turun süresini verir, ama bir programın ne kadar süreceğini tek başına söylemez. Bunun için iki büyüklük daha gerekir: programın yürüttüğü komut sayısı *N* ve bir komutun ortalama kaç tur sürdüğü.

Tanım 1.1.6

Komut başına tur (*cycles per instruction*, CPI), bir programda komut başına harcanan ortalama saat turu sayısıdır. *N* komutluk bir programın **CPU süresi**:

$$\text{CPU süresi} = N \times \text{CPI} \times T = N \times \text{CPI} / f$$

Saniyede yürütülen komut sayısı *f / CPI*'dır. Bu sayı milyon olarak verildiğinde birimi **MIPS**'tir (*million instructions per second*, saniyede milyon komut).

Formülün üç çarpanı üç ayrı tasarım düzeyine karşılık gelir. *N*'yi program ve derleyici belirler. CPI işlemcinin mimarisine bağlıdır: boru hattı, önbellek ve paralel yürütme birimleri CPI'yı düşürür. *f* ise üretim teknolojisine ve devre tasarımına bağlıdır. Birini iyileştirmek çoğu zaman bir başkasını kötüleştirir. Örneğin komutları basitleştirmek CPI'yı düşürür ama *N*'yi artırır.

Sunumdaki etkileşimli hesaplayıcı dört tarihî işlemci için bu formülü uygular. CPI değerleri kabaca örnektir. 8086 komutları ortalama 10 tur civarında sürer. Modern işlemciler ise bir turda birden çok komut bitirebilir; CPI'ları 1'in altına iner. Tablo 1.1.5 1 milyon komutluk bir program için hesaplayıcının verdiği sonuçları toplar.

İşlemci	*f*	CPI	*T*	*4T*	MIPS	Süre
8086	5 MHz	10	200 ns	800 ns	0,5	2 s
80386	33 MHz	4	30,3 ns	121 ns	8,25	121 ms
Pentium	66 MHz	1,5	15,2 ns	60,6 ns	44	22,7 ms
Core i7	3,33 GHz	0,5	0,30 ns	1,20 ns	6660	0,15 ms

Tablo 1.1.5: *N* = 1 milyon komut için periyot, yol döngüsü, hız ve çalışma süresi. Değerler hesaplayıcının hazır ayarlarıyla bulunmuştur.

Tablonun son sütununda 8086 ile Core i7 arasında yaklaşık 13.000 katlık bir fark vardır. Bu farkın yalnız yaklaşık 670 katı saat frekansından gelir (*3330 / 5 = 666*). Geri kalan 20 kat CPI'daki düşüşün, yani mimarideki gelişmelerin ürünüdür (*10 / 0,5 = 20*). 4T sütunu yalnız karşılaştırma içindir; modern işlemcilerin yolları 8086'nınki gibi dört turluk döngülerle çalışmaz.

Örnek 1.1.5

$N = 2 \times 10^9$ komutluk bir program, CPI = 1,5 olan 3 GHz'lik bir işlemcide kaç saniye sürer?

$$süre = N \times CPI / f = 2 \times 10^9 \times 1{,}5 / (3 \times 10^9) = 3 \times 10^9 / (3 \times 10^9) = 1\ s$$

İşlemci saniyede $f / CPI = 3 \times 10^9 / 1{,}5 = 2 \times 10^9$ komut, yani 2000 MIPS yürütür; 2 milyar komut bu yüzden tam 1 saniye sürer.

1.1.3.4 Bir döngünün saat turu sayımı

CPI bir ortalamadır. Gerçek bir programın süresini bulmak için her komutun tur sayısı ayrı ayrı toplanır. Intel'in 8086 veri sayfası her komut için bir tur sayısı verir. Aşağıdaki döngü 4 + 3 + 2 + 1 toplamını hesaplar; açıklamalarda her komutun tur sayısı yazılıdır.

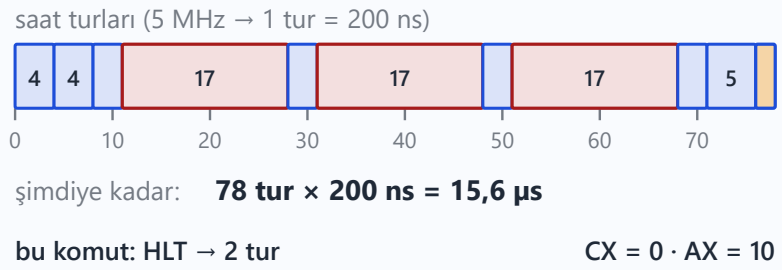
```
org 100h
mov cx, 4        ; 4 tur
mov ax, 0        ; 4 tur
dongu: add ax, cx ; 3 tur
loop dongu       ; 17 / 5 tur
hlt              ; 2 tur
```

Tur sayıları komutun türüne göre değişir. Yazmaca anlık değer yükleyen `mov` 4 tur sürer. Yazmaçtan yazmaca `add` 3 tur sürer; bellek erişimi olmadığı için kısadır. `loop` komutunun iki süresi vardır: atlayan `loop` 17, atlamayan `loop` 5 tur. Fark, atlamanın önden getirme kuyruğunu boşaltmasından gelir. Kuyrukta duran baytlar artık geçersizdir, çünkü akış başka bir adrese gitmiştir; BIU hedef adresten yeniden getirmeye başlamak zorundadır. `hlt` 2 tur sürer.

Adım	Komut	Tur	Toplam tur	Süre	CX	AX
1	`mov cx, 4`	4	4	0,8 µs	4	?
2	`mov ax, 0`	4	8	1,6 µs	4	0
3	`add ax, cx`	3	11	2,2 µs	4	4
4	`loop` (atlar)	17	28	5,6 µs	3	4
5	`add ax, cx`	3	31	6,2 µs	3	7
6	`loop` (atlar)	17	48	9,6 µs	2	7
7	`add ax, cx`	3	51	10,2 µs	2	9

Adım	Komut	Tur	Toplam tur	Süre	CX	AX
8	`loop` (atlar)	17	68	13,6 µs	1	9
9	`add ax, cx`	3	71	14,2 µs	1	10
10	`loop` (atlamaz)	5	76	15,2 µs	0	10
11	`hlt`	2	78	15,6 µs	0	10

Tablo 1.1.6: Döngünün saat turu sayımı (5 MHz, 1 tur = 200 ns).
Her satır o komuttan sonraki durumu verir.



Şekil 1.1.8: Döngünün zaman çubuğu. Her dikdörtgen bir komutun tur sayısıdır: 4, 4, sonra üç kez 3 + 17, en sonda 3 + 5 ve 2. Toplam 78 tur, 5 MHz'de 15,6 µs eder.

Toplam bir formülle de bulunabilir. Gövde dört kez çalışır; `loop` üç kez atlar, bir kez atlamaz:

$$4 + 4 + 4 \times 3 + 3 \times 17 + 5 + 2 = 8 + 12 + 51 + 5 + 2 = 78 \text{ tur}$$

$$78 \times 200 \text{ ns} = 15.600 \text{ ns} = 15{,}6 \text{ µs}$$

Sonuç dikkat çekicidir: 78 turun 51'i, yani yarıdan fazlası atlayan `loop` komutlarında harcanmıştır. Asıl iş olan toplama yalnız 12 tur sürer. Döngü yükü küçük gövdeli döngülerde süreyi belirler. Bu yüzden hız gereken yerlerde döngü gövdesi birkaç kez açılarak yazılır ve atlama sayısı azaltılır. Bu programın ortalama CPI'sı da hesaplanabilir: 11 komut 78 tur sürdüğü için *CPI = 78 / 11 ≈ 7,1*'dir.

Veri sayfasındaki tur sayıları en iyi durum değerleridir. Komut baytlarının kuyrukta hazır beklediği ve belleğin bekleme durumu eklemediği varsayılır. Gerçek sürede kuyruk boşken getirme beklemesi ve yavaş bellek için eklenen Tw durumları da yer alır; ölçülen süre bu yüzden biraz daha uzun çıkabilir. Süre her durumda *tur sayısı × T*'dir.

Sık yapılan hata

Döngü süresi hesaplanırken `loop` komutunun bütün geçişleri aynı süreyle sayılır. Atlayan ve atlamayan `loop` farklı sürer (17 ve 5 tur). *n* kez dönen bir döngüde `loop` *n − 1* kez atlar, son geçişte atlamaz.

1.1.3.5 Saat hızlarının gelişimi

Intel işlemcilerinin saat hızları kırk yılda megahertzden gigahertze çıkmıştır. 16 bitlik dönemde 8086 5, 8 ve 10 MHz sürümleriyle üretildi. 8088 5 ve 8 MHz'de çalıştı; IBM PC'deki 8088'in saat hızı ise 4,77 MHz'dir. 80186/80188 6 MHz'den, 80286 da 6, 8, 10 ve 12,5 MHz sürümleriyle başladı. 32 bitlik dönemde 80386 16–33 MHz, 80486 16–100 MHz aralığında çalıştı. 80486'nın DX2 ve DX4 sürümleri saat çarpanı kullanır: yonga içi saat dış yol saatinin iki ya da üç katıdır. Pentium 60 ve 66 MHz'le başladı. 64 bitlik dönemde Core 2 1,2–3 GHz aralığında çalışır. Birinci nesil Core i7 2,66–3,33 GHz, Core i5 2,4–3,6 GHz, Core i3 2,93–3,33 GHz aralığındadır.

5 MHz'den yaklaşık 3 GHz'e çıkış, otuz yılda yaklaşık 600 katlık bir artıştır. 2005 civarında bu artış durdu. Saat frekansı yükseldikçe her saniyede daha çok transistör anahtarlanır; dinamik güç frekansla ve besleme geriliminin karesiyle orantılı artar. Yonganın ürettiği ısıyı uzaklaştırmanın sınırına dayanıldı. O tarihten sonra performans artışı daha çok çekirdek sayısını artırarak ve bir turda daha fazla komut bitirerek, yani CPI'yı düşürerek sağlandı.

1.1.3.6 Bit genişliği

> **Tanım 1.1.7**
>
> Bir işlemcinin **bit genişliği** (*word length*), bir seferde işleyebildiği sözcüğün bit sayısıdır: yazmaçların ve ALU'nun boyutu, çoğunlukla da veri yolunun genişliği. “*n* bitlik işlemci” bu anlama gelir. *n* bitle işaretsiz olarak $0 \ldots 2^n - 1$ aralığı temsil edilir.

Sözcük	İşaretsiz aralık
8 bit	0 … 255
16 bit	0 … 65.535
32 bit	0 … 4.294.967.295
64 bit	0 … ≈ $1{,}8 \times 10^{19}$

Tablo 1.1.7: Sözcük genişliğine göre işaretsiz aralık.

Bit genişliği sayıların büyüklüğünü sınırlar ama işlemcinin daha büyük sayılarla çalışmasını engellemez. 8 bitlik bir işlemci de 32 bitlik sayıları toplayabilir; yalnız bunu birkaç adımda yapar. Genişlik arttıkça aynı iş daha az komutla biter.

Adres yolunun genişliği ise ayrı bir büyüklüktür ve erişilebilen bellek miktarını belirler: *n* hat 2^n bayt adresler. Tablo 1.1.8 Intel işlemcilerinden örnekler verir.

Adres yolu	Bellek	Örnek
16 bit	64 KB	8080, 8085
20 bit	1 MB	8086, 8088

Adres yolu	Bellek	Örnek
24 bit	16 MB	80286
32 bit	4 GB	80386, 80486
36 bit	64 GB	Pentium Pro (PAE)

Tablo 1.1.8: Adres yolu genişliği ve adreslenebilir bellek.

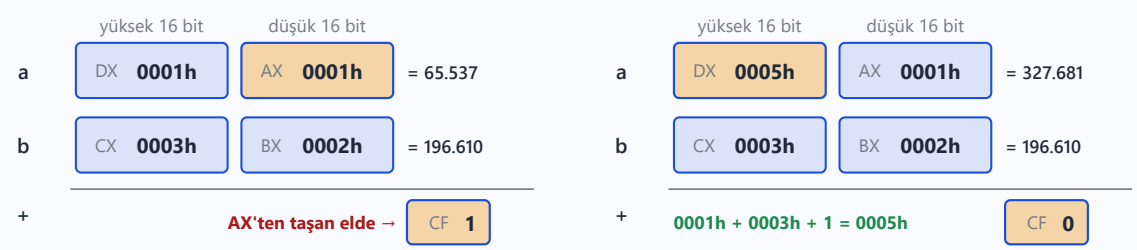
Tablodaki işlemcilerde veri genişliği ile adres genişliği her zaman eşit değildir. 8 bitlik 8080'in adres yolu 16 bittir. 16 bitlik 8086'nın adres yolu 20 bittir. 20 bitlik adres, iki 16 bitlik değerin, kesim ve bağıl konumun birleştirilmesiyle üretilir: *fiziksel adres = kesim × 16 + bağıl konum*. Örneğin kesim `1234h`, bağıl konum `0010h` ise fiziksel adres `12340h + 0010h = 12350h`'dir.

1.1.3.7 16 bitlik işlemcide 32 bitlik toplama

Bit genişliğinin pratikteki etkisini görmek için iki 32 bitlik sayıyı 16 bitlik 8086 ile toplayalım: *a = 0001FFFFh = 131.071* ve *b = 00030002h = 196.610*. Bu sayılar 16 bitlik bir yazmaca sığmaz. Her biri iki yarıya bölünür ve bir yazmaç çiftinde tutulur. *a* için yüksek yarı DX'e, düşük yarı AX'e konur; bu çift DX:AX diye yazılır. *b* de CX:BX çiftine konur.

```
org 100h
mov ax, 0FFFFh ; a: düşük yarı
mov dx, 0001h  ; a: yüksek yarı
mov bx, 0002h  ; b: düşük yarı
mov cx, 0003h  ; b: yüksek yarı
add ax, bx     ; düşük yarılar
adc dx, cx     ; yüksek + CF
ret
```

İlk dört komut sayıları yazmaçlara yükler. `0FFFFh` yazımındaki baştaki 0, assembler'ın sayıyı bir etiket adı sanmaması içindir; harfle başlayan onaltılık sayılar böyle yazılır. `add ax, bx` düşük yarıları toplar. `adc` (*add with carry*, eldeli topla) yüksek yarıları toplar ve önceki toplamadan gelen eldeyi, yani CF'yi de ekler. Kâğıt üzerinde basamak basamak toplama yapılırken eldenin bir sonraki basamağa eklenmesi neyse, `adc` da 16 bitlik "basamaklar" için odur.



Şekil 1.1.9: 32 bitlik toplamanın iki anı. Solda `add ax, bx`: `FFFFh + 0002h` toplamının 17. biti AX'e sığmaz; AX = `0001h`, CF = 1. Sağda `adc dx, cx`: `0001h + 0003h + 1 = 0005h`; sonuç DX:AX = `00050001h`.

Komut	DX	AX	CX	BX	CF	Açıklama
`mov ax, 0FFFFh`	????	FFFFh	????	????	0	*a*'nın düşük yarısı
`mov dx, 0001h`	0001h	FFFFh	????	????	0	DX:AX = 131.071
`mov bx, 0002h`	0001h	FFFFh	????	0002h	0	*b*'nin düşük yarısı
`mov cx, 0003h`	0001h	FFFFh	0003h	0002h	0	CX:BX = 196.610
`add ax, bx`	0001h	0001h	0003h	0002h	1	`FFFFh + 0002h = 1 0001h`
`adc dx, cx`	0005h	0001h	0003h	0002h	0	`0001h + 0003h + 1`

Tablo 1.1.9: 32 bitlik toplamanın izlemesi. Her satır o komuttan sonraki durumu verir.

Kritik adım `add ax, bx`'tir. `FFFFh + 0002h = 10001h` sonucu 17 bittir. En üstteki 1 biti 16 bitlik AX'e sığmaz: AX'te `0001h` kalır ve taşan bit CF'ye yazılır. Bu komut diğer bayrakları da günceller: sonuç sıfır olmadığı için ZF = 0, en üst bit 0 olduğu için SF = 0'dır. İşaretli yorumda `FFFFh` −1 olduğundan *−1 + 2 = 1* doğru bir sonuçtur ve OF = 0 kalır. Düşük dört bitte *Fh + 2h = 11h* taştığı için AF = 1'dir. `adc dx, cx` ise *0001h + 0003h + CF(1) = 0005h* hesaplar; yeni bir taşma olmadığı için CF = 0 olur. Sonuç DX:AX = `00050001h`'dir. Onluk sağlaması: *5 × 65.536 + 1 = 327.681 = 131.071 + 196.610*.

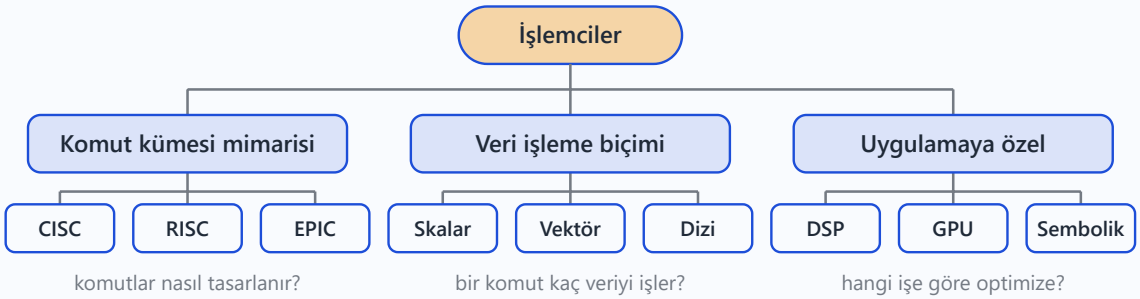
16 bitlik işlemci 32 bitlik toplamayı iki komutta yaptı. 32 bitlik bir işlemci, örneğin 80386, aynı işi tek bir `add` ile yapar: `add eax, ebx`. 64 bitlik toplama 8086'da dört komut (`add` ve üç `adc`) sürerken 80386'da iki, 64 bitlik bir işlemcide bir komuttur. Bit genişliğinin performansa etkisi budur.

> **Sık yapılan hata**
>
> Yüksek yarılar da `add` ile toplanır. Bu durumda düşük yarıdan gelen elde kaybolur ve sonuç DX:AX = `00040001h` = 262.145 çıkar; doğru sonuçtan tam 65.536 eksiktir. Ayrıca `add` ile `adc` arasına CF'yi değiştiren bir komut (örneğin bir `cmp`) konmamalıdır; `mov` ise bayrakları değiştirmediği için araya girebilir.

1.1.4 Komut kümesi mimarileri

İşlemciler birçok açıdan sınıflanabilir. Üç soru öne çıkar: Komutlar nasıl tasarlanır? Bir komut kaç veriyi işler? İşlemci hangi işe göre eniyilenmiştir? Bu kesim ilk soruyu, yani komut kümesi mimarisini ele alır: CISC, RISC ve EPIC. Bu mimarilerin dayandığı en önemli teknik olan boru hattı da adım adım izlenir.

1.1.4.1 İşlemci türlerine genel bakış



Şekil 1.1.10: İşlemcilerin üç ölçüte göre sınıflanması. Komut kümesi mimarisine göre CISC, RISC ve EPIC; veri işleme biçimine göre skalar, vektör ve dizi; uygulamaya göre DSP, GPU ve sembolik işlemciler.

Bir işlemcinin **komut kümesi** (*instruction set*), yürütebildiği komutların tamamıdır. **Komut kümesi mimarisi** (*instruction set architecture*, ISA) ise bu komutların yanında yazmaçları, adresleme modlarını, veri tiplerini ve komutların bellekte nasıl kodlandığını da kapsar. ISA, yazılım ile donanım arasındaki sözleşmedir: aynı ISA'yı uygulayan iki işlemci aynı makine kodunu çalıştırır, içerideki devreleri ne kadar farklı olursa olsun. Üç yaklaşımın adları şöyle açılır:

- **CISC**: *Complex Instruction Set Computer*, karmaşık komut kümeli bilgisayar.
- **RISC**: *Reduced Instruction Set Computer*, indirgenmiş (azaltılmış) komut kümeli bilgisayar.
- **EPIC**: *Explicitly Parallel Instruction Computing*, açıkça paralel komut işleme.

1.1.4.2 CISC

CISC yaklaşımı karmaşık işlemleri tek bir komutla yapmayı amaçlar. Bu yaklaşımın kökü 1960'lara ve 1970'lere uzanır. O yıllarda bellek pahalı ve yavaştı; programlar çoğunlukla assembly ile elle yazılırdı. Bir işi tek komutla yapmak hem bellekten tasarruf ettiriyor hem de programcının işini kolaylaştırıyordu.

> **Tanım 1.1.8**
>
> **CISC** mimarisinde bir komut, bellekten yükleme, aritmetik işlem ve belleğe kaydetme gibi birden çok düşük düzeyli işlemi içerebilir. Komutlarda birden çok adresleme modu ve doğrudan bellek işlenenleri bulunur. Komut uzunlukları değişkendir; yazmaç sayısı görece azdır; kontrol birimi çoğunlukla mikroprogramlıdır.

8086 tipik bir CISC işlemcisidir. Komutları 1 ile 6 bayt arasındadır; işlem kodundan sonra gelen baytların sayısı işlenenlere göre değişir. Bu yüzden kontrol birimi bir komutun kaç bayt olduğunu ancak ilk baytları çözdükten sonra öğrenir; çözme karmaşıktır. Genel amaçlı yazmaç sayısı dörttür (AX, BX, CX, DX). Pek çok komut belleği doğrudan işlenen olarak kullanabilir. İki tipik örnek:

```
ADD [BX], AX   ; bellek = bellek + AX
REP MOVSB      ; CX bayt kopyala
```

`ADD [BX], AX`, BX'in gösterdiği bellek sözcüğüne AX'i ekler: tek komutta belleği okur, toplar ve sonucu belleğe yazar. `REP MOVSB` ise CX bayt uzunluğundaki bir bellek bloğunu SI'nın gösterdiği yerden DI'nin gösterdiği yere kopyalar; tek bir komut, bir döngünün yerini tutar. CISC ailesinin başka örnekleri Intel 80386, 80486 ve Pentium serisi, Motorola 68000, 68020 ve 68040, DEC VAX ve IBM System/360'tır.

1.1.4.3 RISC

1970'lerin sonunda yapılan ölçümler ilginç bir sonuç verdi. Derleyicilerin ürettiği kod, CISC komut kümelerindeki karmaşık komutların çok küçük bir kısmını kullanıyordu. Programların büyük bölümü basit yükleme, saklama, toplama ve atlama komutlarından oluşuyordu. Karmaşık komutlar ise kontrol birimini büyütüyor ve basit komutları da yavaşlatıyordu. RISC yaklaşımı bu gözlemden doğdu.

> **Tanım 1.1.9**
>
> **RISC** mimarisinde az sayıda, basit ve sabit uzunluklu (örneğin hepsi 32 bit) komut bulunur. Belleğe yalnız yükle ve sakla (*load/store*) komutları erişir; aritmetik ve mantık işlemleri yalnız yazmaçlar arasında yapılır. Bellek erişimini azaltmak için çok sayıda (32 ve üzeri) yazmaç vardır. Kontrol birimi donanımsaldır ve komutlar boru hattıyla yürütülür. Hedef, ideal durumda her saat turunda bir komut tamamlamaktır (CPI ≈ 1).

Komutların sabit uzunlukta olması çözmeyi kolaylaştırır: her komut aynı yerden başlar, işlem kodu ve yazmaç numaraları hep aynı bit alanlarındadır. Belleğe yalnız yükle/sakla komutlarının erişmesi, komutların büyük çoğunluğunun aynı sürede bitmesini sağlar. Bu düzenlilik boru hattının ön koşuludur. Karmaşıklık ortadan kalkmaz, derleyiciye kayar: bir CISC komutunun yaptığı işi RISC'te birkaç komut yapar ve bu komutları uygun sırayla dizmek derleyicinin görevidir.

RISC örnekleri IBM RS/6000 (POWER), Motorola MC88100, DEC Alpha 21064, 21164 ve 21264, MIPS ve SPARC'tır. Bugün en yaygın RISC ailesi ARM'dir: telefonların ve tabletlerin neredeyse hepsinde ve Apple'ın M serisi bilgisayarlarında ARM çekirdekleri bulunur. RISC-V ise lisans ücreti gerektirmeyen açık bir komut kümesidir ve hem akademide hem gömülü sistemlerde hızla yayılıyor.

RISC'in “her turda bir komut” hedefi bir komutun tek turda bittiği anlamına gelmez. Bir komut yine birkaç aşamadan geçer. Boru hattı sayesinde her turda bir komut tamamlanır; komutun tek başına süresi (gecikmesi) değil, birim zamanda biten komut sayısı (**verim**, *throughput*) artar.

1.1.4.4 Boru hattı

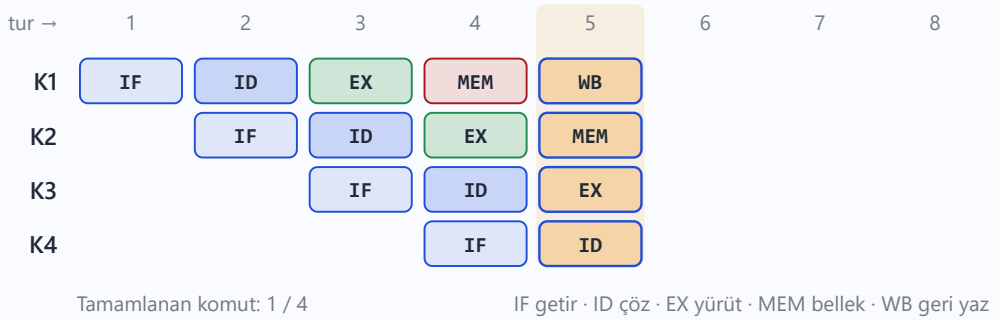
Tanım 1.1.10

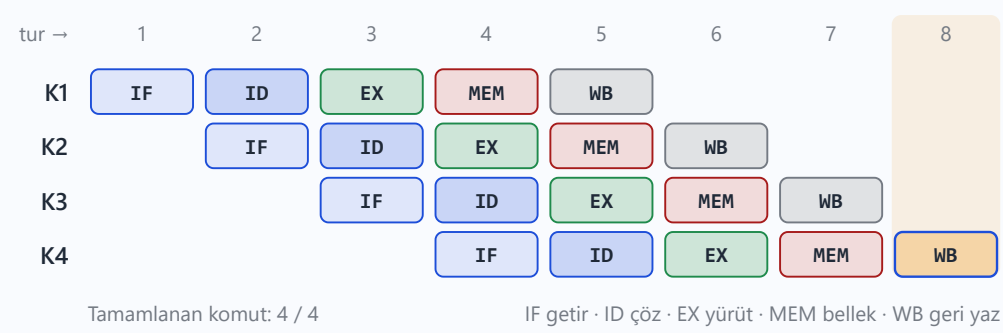
Boru hattı (*pipeline*), komut yürütmeyi ardışık aşamalara bölen ve ardışık komutların farklı aşamalarını aynı turda yürüten tekniktir. Her aşama ayrı bir donanım birimidir ve her turda işini bir sonraki aşamaya devreder.

Klasik RISC boru hattının beş aşaması vardır: IF (*instruction fetch*, getir), ID (*instruction decode*, çöz ve yazmaçları oku), EX (*execute*, yürüt), MEM (*memory*, belleğe eriş) ve WB (*write back*, sonucu yazmaca geri yaz). Fabrikadaki bir montaj bandı iyi bir benzetmedir. Bir otomobilin bütün montajı beş saat sürse de bant her saat bir otomobil çıkarır, çünkü beş istasyon aynı anda beş farklı otomobil üzerinde çalışır.

Şekil 1.1.11 dört bağımsız komutun (K1–K4) boru hattından geçişini gösterir. Turlar şöyle ilerler:

- **Tur 1:** K1 getirilir (IF). Öbür aşamalar boştur.
- **Tur 2:** K1 çözülürken (ID) K2 getirilir. İki komut aynı anda farklı aşamalardadır.
- **Tur 3:** K1 yürütülür, K2 çözülür, K3 getirilir: üç komut üç aşamada.
- **Tur 4:** dört komut birden boru hattındadır.
- **Tur 5:** K1 WB aşamasına gelir ve tamamlanır. Boru hattı dolmuştur; bundan sonra her turda bir komut biter.
- **Tur 6, 7, 8:** sırasıyla K2, K3 ve K4 tamamlanır.





Şekil 1.1.11: Beş aşamalı boru hattı. Üstte tur 5: boru hattı dolmuş, K1 tamamlanmıştır. Altta tur 8: dört komutun hepsi bitmiştir.

Teorem 1.1.1 (Boru hattında bitiş turu)

k aşamalı bir boru hattında, aralarında bağımlılık olmayan n komut

$$k + n - 1$$

turda biter. Boru hattı olmasaydı her komut k tur sürer ve toplam $k \times n$ tur gerekirdi.

Formülün gerekçesi basittir. İlk komut boru hattını k turda baştan sona geçer. Ondan sonra her turda bir komut daha biter; kalan $n - 1$ komut için $n - 1$ tur daha gerekir. Toplam $k + (n - 1)$'dir. Örnekte $k = 5$, $n = 4$: $5 + 4 - 1 = 8$ tur. Boru hattısız $4 \times 5 = 20$ tur gerekirdi.

Örnek 1.1.6

5 aşamalı bir boru hattında 100 komut kaç turda biter? Boru hattısız kaç tur gerekir?

Boru hattıyla $k + n - 1 = 5 + 100 - 1 = 104$ tur. Boru hattısız $5 \times 100 = 500$ tur. Hızlanma $500 / 104 \approx 4{,}8$ kattır. Komut sayısı büyüdükçe hızlanma $k\,n / (k + n - 1)$ ifadesi k'ya, yani aşama sayısına yaklaşır: 1000 komut için $5000 / 1004 \approx 4{,}98$.

Bu ideal tablo her zaman gerçekleşmez. Bir komut, kendinden hemen önceki komutun henüz yazmadığı bir sonuca gerek duyabilir (veri bağımlılığı). Bir koşullu atlamanın sonucu belli olana kadar hangi komutun getirileceği bilinmez (kontrol bağımlılığı). Bu durumlarda boru hattına boş turlar girer; buna **boru hattında bekleme** (*stall*) denir. Modern işlemciler beklemeleri azaltmak için sonuçları aşamalar arasında doğrudan iletir ve atlamaların sonucunu önceden tahmin eder.

8086'nın boru hattı bu beş aşamalı yapıdan çok daha basittir: yalnız iki aşama vardır, getirme (BIU) ve yürütme (EU). Atlayan `loop` komutunun 17 tur sürmesi de boru hattının bir bedelidir: atlama, önden getirilmiş baytları geçersiz kılar ve kuyruğun yeniden doldurulmasını gerektirir.

> **Sık yapılan hata**
>
> Boru hattının tek bir komutu hızlandırdığı düşünülür. Tek bir komutun süresi (gecikmesi) boru hattıyla kısalmaz, hatta aşamalar arası yazmaçlar yüzünden biraz uzar. Kısalan, çok sayıda komutun toplam süresidir; boru hattı verimi artırır.

1.1.4.5 Aynı iş, iki yaklaşım

İki yaklaşımın farkını tek bir görev üzerinde görmek öğreticidir: bellekteki bir sayıya bir yazmacın değerini eklemek, yani `bellek[adr] += x`. CISC işlemcide (8086) bu iş tek komuttur:

```
ADD [BX], AX     ; 1 komut
                 ; oku, topla, yaz
```

RISC işlemcide (ARM) ise üç komut gerekir. `r1` adresi, `r0` eklenecek değeri tutar; ARM assembly'sinde açıklamalar `@` işaretiyle başlar:

```
LDR r2, [r1]      @ yükle
ADD r2, r2, r0    @ topla
STR r2, [r1]      @ sakla
```

`LDR` (*load register*) bellekteki sözcüğü `r2`'ye yükler. `ADD r2, r2, r0` üç işlenenli bir toplamadır: `r2 ← r2 + r0`. `STR` (*store register*) sonucu belleğe geri yazar. CISC kodu kısadır ve bellekte az yer kaplar, ama komut birçok tur sürer. RISC kodunun üç komutu basittir ve her biri boru hattında akıcı ilerler. Toplanan değer yazmaçta kaldığı için, bu sayı üzerinde başka işlemler de yapılacaksa yeniden belleğe gitmek gerekmez.

İki yaklaşım bugün iç içe geçmiştir. Pentium Pro'dan bu yana x86 işlemciler karmaşık komutları içeride RISC benzeri basit **mikro-işlemlere** (*micro-operation*, µop) çevirir ve bu mikro-işlemleri boru hattında yürütür. Dışarıdan bakıldığında komut kümesi CISC'tir; içeride ise yürütme RISC gibidir.

1.1.4.6 CISC komutu içeride: kod izleme

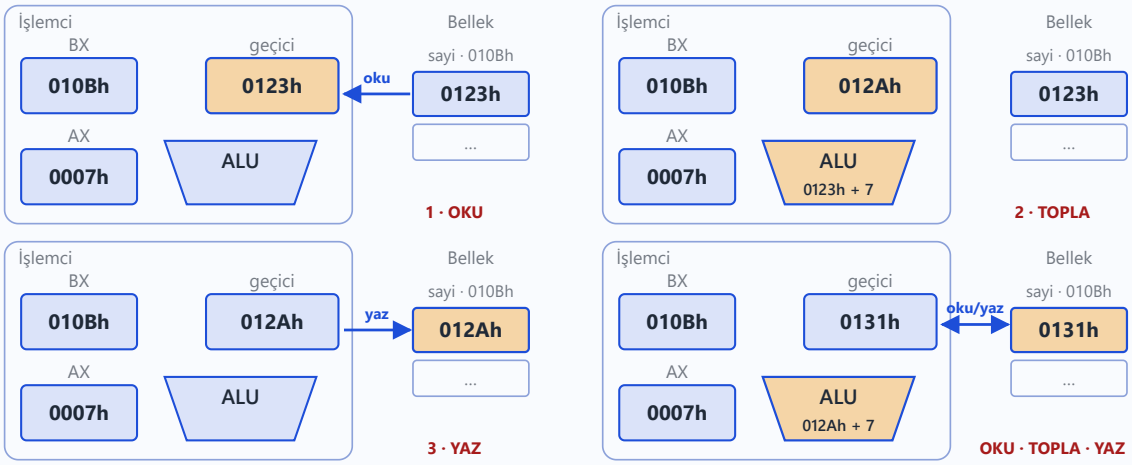
Tek bir CISC komutunun içeride kaç iş yaptığını görmek için aşağıdaki programı izleyelim. Program bellekteki `sayi` değişkenine iki kez 7 ekler.

```
org 100h
mov bx, offset sayi ; adres
mov ax, 7           ; değer
add [bx], ax        ; oku, topla, yaz
add [bx], ax        ; tekrar
ret
sayi dw 0123h
```

Program `org 100h` ile `0100h` adresinden başlar. Komut uzunlukları toplanınca `sayi` değişkeninin adresi bulunur: `mov bx, offset sayi` 3 bayt (0100h–0102h), `mov ax, 7` 3 bayt (0103h–0105h), iki `add [bx], ax` ikişer bayt (0106h–0109h), `ret` 1 bayt (010Ah). `sayi` bu yüzden `010Bh`'dedir. `offset` işleci assembler'a değişkenin bağıl konumunu, yani `010Bh`'yi yazdırır. `dw` (*define word*) bellekte bir sözcük ayırır ve ilk değerini `0123h` yapar.

`add [bx], ax` tek bir komuttur, ama içeride üç iş yapar:

1. **Oku.** BIU, BX'in gösterdiği `010Bh` adresindeki sözcüğü (`0123h`) bir bellek okuma yol döngüsüyle getirir ve işlemcinin içindeki geçici bir yazmaca koyar. Bu geçici yazmaç programcıya görünmez.
2. **Topla.** ALU `0123h + 0007h = 012Ah` hesaplar ve bayrakları günceller.
3. **Yaz.** Sonuç ikinci bir yol döngüsüyle belleğe geri yazılır: `sayi = 012Ah` .



Şekil 1.1.12: `add [bx], ax` komutunun içeride yaptığı üç iş. Üst sırada oku (geçici = 0123h) ve topla (0123h + 7 = 012Ah); alt sırada yaz (sayi = 012Ah) ve aynı komutun ikinci yürütülmesi (sayi = 0131h).

Komut	BX	AX	sayi (010Bh)	Yol döngüsü	Bayraklar
başlangıç	?	?	0123h	—	—
`mov bx, offset sayi`	010Bh	?	0123h	—	değişmez
`mov ax, 7`	010Bh	0007h	0123h	—	değişmez
`add [bx], ax`	010Bh	0007h	012Ah	oku + yaz	ZF=0 CF=0 SF=0 OF=0 AF=0

Komut	BX	AX	sayi (010Bh)	Yol döngüsü	Bayraklar
`add [bx], ax`	010Bh	0007h	0131h	oku + yaz	ZF=0 CF=0 SF=0 OF=0 AF=1

Tablo 1.1.10: `add [bx], ax` programının izlemesi. Her satır o komuttan sonraki durumu verir.

İkinci toplamada düşük dört bit *Ah + 7h = 11h* taştığı için AF = 1 olur; birincide *3h + 7h = Ah* dört bite sığdığı için AF = 0'dır. Her iki sonuçta da en üst bit 0'dır ve 16 bitten taşma yoktur.

Bu komutun bedeli tur sayısında görülür. 8086 veri sayfasına göre bellek hedefli `add` 16 tur artı etkin adres hesabı sürer; `[bx]` için etkin adres hesabı 5 tur olduğundan toplam 21 turdur. Bu 21 turun 8'i iki yol döngüsüdür (okuma ve yazma, dörder tur). Yazmaçtan yazmaca `add` ise 3 tur sürer. Bir CISC komutu böylece iki bellek erişimini ve bir toplamayı tek komutta gizler. RISC işlemcide aynı iş üç komuttur (`LDR` , `ADD` , `STR`) ve ara değer bir yazmaçta görünür olur.

1.1.4.7 RISC ve CISC karşılaştırması

Ölçüt	RISC	CISC
Komut kümesi	az ve basit	geniş ve karmaşık
Komut uzunluğu	sabit	değişken
Komut başına tur	az (≈ 1, boru hattıyla)	çok (değişken)
Bellek erişimi	yalnız yükle/sakla	çoğu komut belleğe erişebilir
Yazmaç sayısı	çok	az
Program boyutu	daha uzun	daha kısa
Güç tüketimi	düşük	görece yüksek
Esneklik	yük derleyicide	geniş komut yelpazesi, esnek kullanım

Tablo 1.1.11: RISC ve CISC işlemcilerin karşılaştırılması.

Tablonun özeti şudur: RISC az çeşit komutu hızlı çalıştırır ve yüksek performansı düşük güçle sunar. CISC çok çeşit komut sunar, ama tek tek komutları daha yavaştır. Karşılaştırmayı CPU süresi formülüyle de okumak mümkündür. CISC *N*'yi küçültür ama CPI'yı büyütür; RISC CPI'yı küçültür ama *N*'yi büyütür. Hangisinin kazanacağını çarpım belirler. Bugünkü yüksek performanslı işlemcilerde iki ailenin iç yapıları birbirine çok yaklaşmıştır; fark daha çok kod yoğunluğunda, çözücünün karmaşıklığında ve güç tüketiminde kalmıştır.

1.1.4.8 EPIC

Süperskalar işlemciler aynı turda birden çok komut başlatır. Hangi komutların birbirinden bağımsız olduğunu ve birlikte yürütülebileceğini ise yürütme sırasında donanım bulur. Bu donanım büyük ve güç tüketen bir devredir. EPIC yaklaşımı bu kararı donanımdan alıp derleyiciye verir.

Tanım 1.1.11

EPIC mimarisinde hangi komutların paralel çalışabileceğine donanım değil derleyici karar verir ve bunu koda açıkça yazar. Komutlar 128 bitlik **paketler** (*bundle*) hâlinde kodlanır: her pakette 41 bitlik üç komut ve 5 bitlik bir şablon bulunur.



Şekil 1.1.13: 128 bitlik EPIC paketi. Bit 127–87 komut 2, bit 86–46 komut 1, bit 45–5 komut 0, bit 4–0 şablondur. *3 × 41 + 5 = 128.*

Şablon alanı, paketteki üç yuvanın hangi tür yürütme birimine (tam sayı, bellek, kayan nokta, dallanma) gideceğini ve paralel yürütülebilecek bir komut grubunun nerede bittiğini söyler. İşlemci bu bilgiyi okuyup komutları doğrudan birimlere dağıtır; bağımlılık aramak için büyük bir devreye gerek kalmaz. Böylece yüksek saat frekansı gerekmeden çok sayıda komut aynı anda yürütülebilir. Bit sınırları da tutarlıdır: her komut 41 bit kapladığı için komut 0 bit 5'ten 45'e, komut 1 bit 46'dan 86'ya, komut 2 bit 87'den 127'ye uzanır.

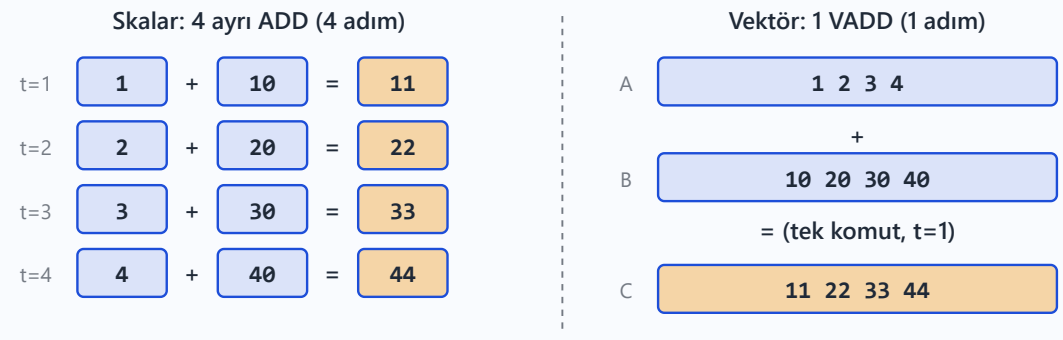
EPIC'in bilinen tek ticari örneği Intel ile HP'nin ortak mimarisi IA-64 ve bu mimariyi uygulayan Itanium işlemcileridir (2001–2021). Yaklaşım beklenen başarıyı göstermedi. Derleyicinin derleme anında bilmediği durumlar, örneğin bir bellek erişiminin önbellekte bulunup bulunmayacağı, paralelliği sınırladı; iyi kod üretmek derleyiciye çok yük bindirdi. Itanium mevcut x86 yazılımını yavaş çalıştırıyordu. Piyasada AMD'nin x86'yı 64 bite genişleten x86-64 mimarisi kazandı.

1.1.5 Veri işleme biçimine ve uygulamaya göre işlemciler

Komut kümesi mimarisi komutların nasıl tasarlandığını anlatır. İşlemcileri ayırmanın ikinci yolu, bir komutun kaç veri öğesini işlediğine bakmaktır: skalar işlemci bir, vektör ve dizi işlemcileri pek çok öğeyi işler. Üçüncü yol ise işlemcinin hangi işe göre eniyilendiğidir: sinyal işleme, sembolik hesap ya da grafik. Bu kesim bu türleri sırayla tanıtır ve iki küçük 8086 programıyla skalar işlemenin bedelini gösterir.

1.1.5.1 Skalar ve vektör işleme

İki dizinin öğe öğe toplanması, yani *C[i] = A[i] + B[i]*, bilimsel hesaplarda, ses ve görüntü işlemede sürekli karşılaşılan bir iştir. **Skalar** (*scalar*) bir işlemci bu işi her öğe için ayrı bir toplama komutuyla yapar. **Vektör** (*vector*) işlemci ise tek bir komutla dizinin bütün öğelerini toplar.



Şekil 1.1.14: Dört öğeli dizilerin toplanması. Skalar işlemci dört ayrı `ADD` komutuyla dört adımda, vektör işlemci tek bir `VADD` komutuyla tek adımda hesaplar: *C = (11, 22, 33, 44)*.

Vektör işlemenin kazancı yalnız toplamanın kendisinden gelmez. Skalar işlemcide her öğe için komut yeniden getirilir ve çözülür; döngü sayacı azaltılır ve sınanır; indis artırılır. Vektör işlemcide bu yük bir kez ödenir. Dört öğede kazanç küçük görünür, ama bin öğelik vektörlerde fark çok büyür.

1.1.5.2 Skalar döngü: kod izleme

8086 skalar bir işlemcidir. Aşağıdaki program *c = a + b* toplamını dört öğeli bayt dizileri için hesaplar.

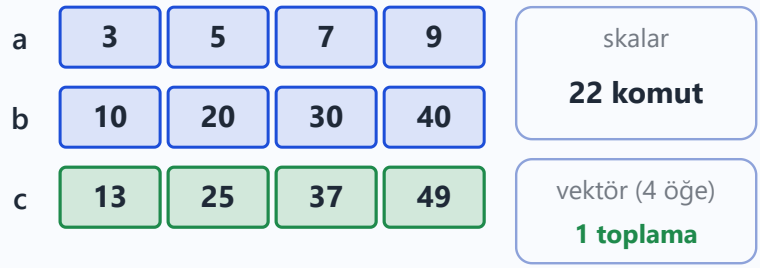
```
org 100h
mov si, 0
mov cx, 4
dongu: mov al, a[si]  ; bir öğe oku
add al, b[si]         ; bir toplama
mov c[si], al         ; bir öğe yaz
inc si
loop dongu
ret
a db 3, 5, 7, 9
b db 10, 20, 30, 40
c db 4 dup(0)
```

SI dizilerin ortak indisidir; CX döngü sayacıdır. `a[si]` yazımı, `a` dizisinin başlangıç adresine SI'nın eklenmesiyle bulunan adresteki baytı gösterir. Döngü gövdesinde üç komut asıl işi yapar: `mov al, a[si]` bir öğe okur, `add al, b[si]` bir toplama yapar, `mov c[si], al` sonucu yazar. Kalan iki komut, `inc si` ve `loop dongu`, döngü yüküdür. `4 dup(0)` dört baytlık sıfırlarla dolu bir alan ayırır.

Geçiş	İşlem	AL	c dizisi	SI	CX	Yürütülen komut (toplam)
başlangıç	—	?	0, 0, 0, 0	0	4	2
1	3 + 10	13	13, 0, 0, 0	1	3	7
2	5 + 20	25	13, 25, 0, 0	2	2	12
3	7 + 30	37	13, 25, 37, 0	3	1	17

Geçiş	İşlem	AL	c dizisi	SI	CX	Yürütülen komut (toplam)
4	9 + 40	49	13, 25, 37, 49	4	0	22

Tablo 1.1.12: Skalar döngünün izlemesi. Her satır bir döngü geçişinin sonundaki durumu verir.



Şekil 1.1.15: Döngü bittiğinde *c = (13, 25, 37, 49)*. Skalar işlemci 22 komut yürütmüştür; vektör işlemci aynı dört toplamayı tek bir vektör toplama komutuyla yapar.

Sayım şöyle yapılır: döngüden önce iki komut (`mov si, 0` ve `mov cx, 4`), her geçişte beş komut. Dört geçişte *2 + 4 × 5 = 22* komut eder. Bu 22 komutun yalnız dördü toplamadır. Sekizi okuma ve yazma, sekizi döngü yükü, ikisi hazırlıktır. Her öğe için üç komut getirilip çözülür ve döngü yükü her öğe için yeniden ödenir. Vektör işlemcinin ortadan kaldırdığı yük budur.

1.1.5.3 Vektör işlemcisi

Tanım 1.1.12

Vektör işlemcisi (*vector processor*), vektör adı verilen veri dizileri üzerinde tek komutla matematiksel işlem yapan işlemcidir. Birden çok veri öğesini aynı anda ya da derin bir boru hattında art arda işler. Uzun vektör yazmaçları ve vektör yükle/sakla birimleri bulunur.



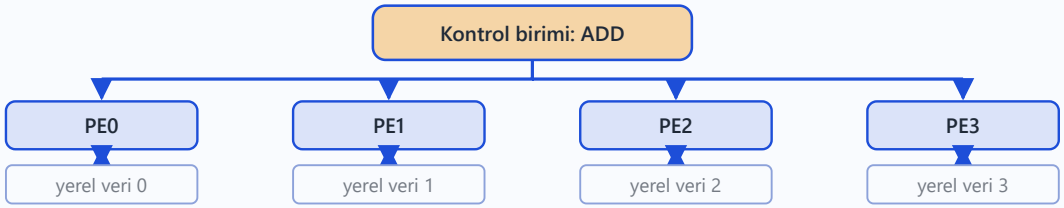
Şekil 1.1.16: Vektör toplama. V1 ve V2 vektör yazmaçlarındaki sekiz öğe tek `VADD V3, V1, V2` komutuyla toplanır ve V3 yazmacına yazılır.

Vektör işlemcisi, tek bir veri öğesiyle çalışan skalar işlemcilere göre bu tür işlerde çok daha hızlıdır. Hava tahmini, fizik benzetimleri ve yapay zekâ modelleri gibi büyük dizilerle çalışan uygulamalar bu yüzden vektör işlemeden yararlanır. Tarihteki ünlü örnek Cray-1 süper bilgisayarıdır (1976). Bugün vektör işleme genel amaçlı işlemcilere uzantı olarak girmiştir: x86'da SSE ve AVX, ARM'da NEON ve SVE, RISC-V'de V uzantısı. Örneğin AVX'in 256 bitlik bir yazmacı sekiz adet 32 bitlik sayıyı tutar ve tek komutla sekizini birden toplar.

1.1.5.4 Dizi işlemcisi ve Flynn sınıflandırması

Tanım 1.1.13

Dizi işlemcisi (*array processor*), tek bir kontrol biriminin aynı komutu çok sayıda **işlem elemanına** (*processing element*, PE) yayınladığı işlemcidir. Her PE kendi yerel belleğindeki veriyi işler. Bu düzene **SIMD** (*single instruction, multiple data*; tek komut, çok veri) de denir.



Şekil 1.1.17: Dizi işlemcisi. Kontrol birimi aynı `ADD` komutunu dört işlem elemanına (PE0–PE3) yayınlar; her işlem elemanı kendi yerel verisini işler.

Vektör işlemcisi ile dizi işlemcisi arasındaki fark daha çok donanımın düzenindedir. Vektör işlemcisi öğeleri tek bir derin boru hattında art arda işleyebilir. Dizi işlemcisinde ise her öğe için ayrı bir işlem elemanı vardır ve öğeler gerçekten aynı anda işlenir. Dizi işlemcileri paralel hesaplamalar için eniyilenmiştir. Süper bilgisayarlarda, sunucularda ve GPU'larda kullanılır; genel amaçlı hesaplamada nadirdir. Tarihteki örnek ILLIAC IV'tür.

Michael Flynn'in 1966'da önerdiği sınıflandırma, işlemcileri komut ve veri akışlarının sayısına göre dört gruba ayırır:

Sınıf	Anlamı	Örnek
SISD	tek komut, tek veri	klasik 8086
SIMD	tek komut, çok veri	dizi ve vektör işlemcileri, GPU
MISD	çok komut, tek veri	nadir; hata toleranslı sistemler
MIMD	çok komut, çok veri	çok çekirdekli işlemciler, kümeler

Tablo 1.1.13: Flynn sınıflandırması.

8086 her komutta tek bir veri öğesini işler ve tek bir komut akışını yürütür; SISD sınıfındadır. Çok çekirdekli bir işlemcide her çekirdek kendi programını kendi verisiyle yürütür; bu da MIMD'dir. Bugünkü bir masaüstü işlemcisi aynı anda birkaç sınıfa girer: her çekirdek AVX komutlarıyla SIMD işlem yapar, çekirdekler birlikte MIMD düzeninde çalışır.

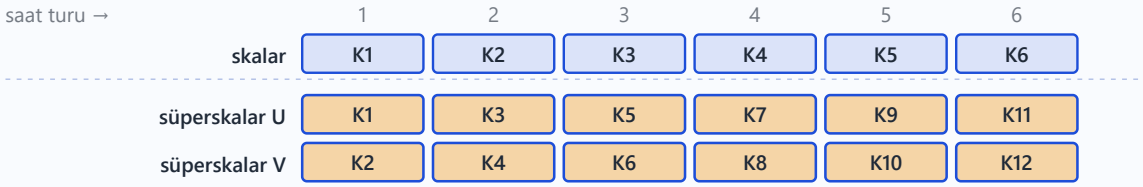
1.1.5.5 Sayıl (skalar) işlemci ve süperskalar

Tanım 1.1.14

Sayıl işlemci (*scalar processor*), sayıl, yani vektör olmayan veriyi, komut başına bir veri öğesi olarak işleyen işlemcidir.

Basit sayıl işlemciler yalnız tam sayılarla çalışır. Güçlü olanlar kayan noktalı sayıları da işler; bunlarda tam sayı birimi (ALU) ve kayan nokta birimi (FPU) aynı yongadadır. Sayıl bir işlemci CISC ya da RISC mimarisinde olabilir; bu iki sınıflama birbirinden bağımsızdır. Genel amaçlı hesaplamada en yaygın tür sayıl işlemcidir.

Sayıl bir işlemci de birden çok komutu aynı anda yürütebilir. **Süperskalar** (*superscalar*) işlemci, birden çok boru hattı ya da yürütme birimiyle bir saat turunda birden fazla komut başlatan sayıl işlemcidir. Her komut yine tek bir veri öğesiyle çalışır; paralellik veride değil komut akışındadır. x86'daki ilk süperskalar işlemci Pentium'dur: U ve V adlı iki boru hattı vardır ve uygun iki komut aynı turda birer hatta girer.



Şekil 1.1.18: Sayıl işlemcide her saat turunda bir komut başlar (6 turda K1–K6). İki boru hatlı süperskalar Pentium her turda U ve V hatlarına birer komut verir (6 turda K1–K12).

Şekildeki ideal durumda süperskalar işlemci aynı sürede iki kat komut başlatır; CPI 0,5'e iner. Gerçekte bu oran düşer. İki komut ancak birbirinden bağımsızsa aynı turda yürüyebilir: ikinci komut birincinin sonucunu kullanıyorsa beklemek zorundadır. Pentium'da V hattının yürütebildiği komutlar da sınırlıydı.

1.1.5.6 Sayısal sinyal işlemcisi (DSP)

Tanım 1.1.15

Sayısal sinyal işlemcisi (*digital signal processor*, DSP), sayısal sinyaller üzerinde işlem yapmak için eniyilenmiş işlemcidir. Sayısal sinyal, ses ve görüntü gibi duyusal bilgileri temsil eden bir **örnek** (*sample*) dizisidir. DSP'nin sinyal işlemeye özel komut kümesi ve donanımı vardır: tek turda çarp-topla (MAC), dairesel tampon adreslemesi ve sıfır yüklü döngü.

Sunumda "dijital sinyal işlemcisi" adı geçer; ders sözlüğündeki terim "sayısal sinyal işlemcisi"dir. İki ad aynı aygıtı anlatır. DSP'ler ses işleme ve süzgeçleme, görüntü işleme ve telekomünikasyon altyapısında (modemler, baz istasyonları) kullanılır. Tanınmış örnekler Texas Instruments TMS320 ve Analog Devices SHARC aileleridir.

Sinyal işlemenin en temel işlemi sayısal süzgeçtir. **FIR süzgeci** (*finite impulse response*, sonlu dürtü yanıtlı süzgeç) her çıktı örneğini son N girdi örneğinin ağırlıklı toplamı olarak hesaplar:

$$y[n] = \Sigma_{k=0}^{N-1} h[k] \cdot x[n-k]$$

Burada $h[k]$ süzgeç katsayıları, x girdi örnekleridir. Her çıktı örneği için N çarpma ve N toplama gerekir. 48 kHz'lik bir ses sinyalinde 100 katsayılı bir süzgeç saniyede 4,8 milyon çarpma ve 4,8 milyon toplama ister. DSP her terimi tek bir **MAC** (*multiply-accumulate*, çarp-topla) komutuyla tek turda yapar:

$$A \leftarrow A + h[k] \times x[n-k]$$

Burada A akümülatördür. Dairesel tampon adreslemesi, son N örneği tutan tamponun sonuna gelindiğinde adresi kendiliğinden başa sarar; böylece yazılımda sınır denetimi yapılmaz. Sıfır yüklü döngü ise döngü sayacını donanımda tutar ve döngü sonundaki atlamaya tur harcamaz.

1.1.5.7 Çarp-topla: kod izleme

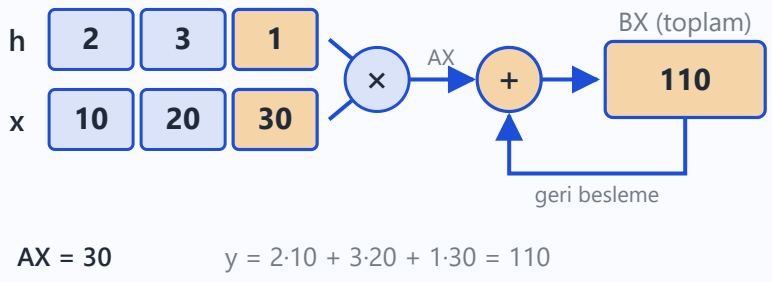
Aynı hesabın 8086'da nasıl yapıldığı, DSP'nin neden ayrı bir tür olduğunu açıkça gösterir. Aşağıdaki program üç katsayılı bir süzgeç toplamını hesaplar: $y = h[0]{\cdot}x[0] + h[1]{\cdot}x[1] + h[2]{\cdot}x[2]$.

```
org 100h
mov si, 0
mov bx, 0        ; toplam
mov cx, 3        ; terim sayısı
dongu: mov al, h[si]
mul x[si]        ; AX = h × x
add bx, ax       ; toplama ekle
inc si
loop dongu
ret
h db 2, 3, 1
x db 10, 20, 30
```

BX akümülatör görevini görür ve sıfırla başlar; CX terim sayısıdır. `mov al, h[si]` katsayıyı AL'ye alır. `mul x[si]` bayt çarpmasıdır: AL'yi bellekteki bayt ile çarpar ve 16 bitlik sonucu AX'e yazar. `mul`'un bir işleneni her zaman örtük olarak AL'dir (bayt çarpmasında); komutta yazılmaz. `add bx, ax` çarpımı toplama ekler.

Komut	SI	AL / AX	BX (toplam)	CX
hazırlık	0	?	0	3
`mov al, h[0]`, `mul x[0]`	0	AX = 2 × 10 = 20	0	3
`add bx, ax`, `inc si`, `loop`	1	AX = 20	20	2
`mov al, h[1]`, `mul x[1]`	1	AX = 3 × 20 = 60	20	2
`add bx, ax`, `inc si`, `loop`	2	AX = 60	80	1
`mov al, h[2]`, `mul x[2]`	2	AX = 1 × 30 = 30	80	1
`add bx, ax`, `inc si`, `loop`	3	AX = 30	110	0

Tablo 1.1.14: Çarp-topla döngüsünün izlemesi. Her satır o komuttan sonraki durumu verir.



Şekil 1.1.19: Döngü bittiğinde toplam BX = 110'dur: *2·10 + 3·20 + 1·30 = 110*. Her terim için ayrı bir `mul` ve `add` gerekmiştir.

Üç çarpımın hiçbiri 255'i aşmadığı için her `mul`'dan sonra AH = 0'dır; bu durumda 8086 CF ve OF bayraklarını 0 yapar. Sonuç bir yüksek bayta taşsaydı ikisi de 1 olurdu.

Süre farkı büyüktür. 8086'da bellekten bayt çarpan `mul` tek başına 76–83 tur artı etkin adres hesabı sürer; yani 70 turdan uzundur. Bir terim için `mov`, `mul`, `add`, `inc` ve `loop` birlikte yüz turu aşar. DSP aynı terimi tek MAC komutuyla, tek turda yapar ve sıfır yüklü döngüsü sayesinde döngü yükü de ödemez.

1.1.5.8 Sembolik işlemci

Tanım 1.1.16

Sembolik işlemci (*symbolic processor*), sayılar yerine semboller, listeler ve kurallar üzerinde hesap yapmak için tasarlanmış işlemcidir. PROLOG ya da LISP işlemcisi diye de adlandırılır.

Sembolik işlemciler anlamsal (semantik) hesaplama ve sembolik mantık kuralları üzerine yoğunlaşır. Matematiksel ifadeleri sembol olarak işler; karmaşık ifadeleri sadeleştirebilir ve türev alabilir. Uygulama alanları uzman sistemler, makine zekâsı ve kural tabanlı çıkarımdır. Aşağıdaki Prolog parçası bu tür hesaba bir örnektir:

```
baba(ali, veli).
dede(X, Z) :- baba(X, Y),
              baba(Y, Z).
```

İlk satır bir olgudur: Ali, Veli'nin babasıdır. Ardından gelen kural, X'in Z'nin dedesi olması için X'in bir Y'nin babası ve Y'nin de Z'nin babası olmasını şart koşar. Program sayı hesaplamaz; olgular ve kurallar arasında eşleme arayarak sonuç çıkarır. Bu işin donanımda ağırlıklı kısmı liste işleme, eşleme ve geri izlemedir.

1980'lerin LISP makineleri (örneğin Symbolics firmasının makineleri) ve Japonya'nın 5. Nesil Bilgisayar projesi bu türdendi. Genel amaçlı işlemciler hızlandıkça özel donanımın üstünlüğü kayboldu. Bugün bu işler genel amaçlı işlemcilerde yazılımla yapılır.

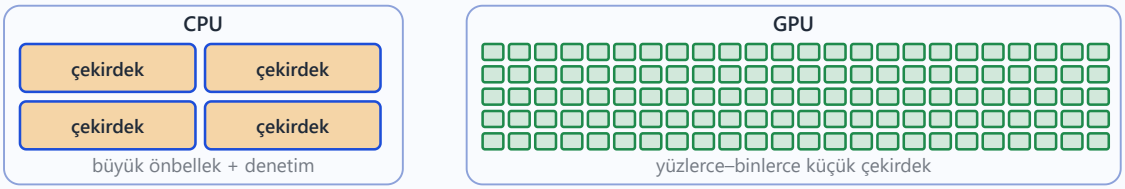
1.1.5.9 Grafik işlem birimi (GPU)

Tanım 1.1.17

Grafik işlem birimi (*graphics processing unit*, GPU), iki ve üç boyutlu grafiklerin hızlandırılması için tasarlanmış, çok sayıda basit çekirdekten oluşan işlemcidir.

Bir ekran karesi milyonlarca pikselden oluşur ve her pikselin rengi aynı formülle, yalnız farklı verilerle hesaplanır. GPU bu yapıyı kullanır: binlerce basit çekirdek aynı programı farklı piksellerde çalıştırır. Bu düzen SIMD'nin bir biçimidir; GPU üreticileri ona SIMT (*single instruction, multiple threads*; tek

komut, çok iş parçacığı) der. GPU gerçek zamanlı ve yüksek kaliteli grafik üretir; gölgelendirme (*shading*), dokulama (*texturing*) ve rasterleştirme gibi görüntü oluşturma görevlerinde etkindir. Grafik kartlarında ve hızlandırıcı kartlarda kullanılır.



Şekil 1.1.20: CPU birkaç büyük çekirdek ve geniş önbellek ile denetim devresinden, GPU ise yüzlerce ya da binlerce küçük çekirdekten oluşur.

CPU ile GPU farklı hedeflere göre tasarlanır. CPU'nun birkaç güçlü çekirdeği vardır; amaç tek bir iş parçacığının gecikmesini azaltmak ve dallanmalarla dolu karmaşık akışları hızlı yürütmektir. Yonga alanının büyük kısmı önbelleğe ve denetim devrelerine ayrılır. GPU'nun ise binlerce basit çekirdeği vardır; amaç tek bir işin süresini değil toplam verimi artırmaktır. Paralel veri üzerinde çalışan her iş bu yapıdan yararlanır. **GPGPU** (*general-purpose computing on GPU*), GPU'nun CUDA ya da OpenCL gibi programlama ortamlarıyla grafik dışı işlerde, örneğin bilimsel hesaplarda ve derin öğrenmede kullanılmasıdır.

1.1.5.10 Türlerin özeti

Tür	Temel fikir	Güçlü yanı	Örnek
CISC	zengin, karmaşık komutlar	kısa kod, uyumluluk	8086, Pentium, 68000
RISC	basit komut + boru hattı	hız, düşük güç	ARM, MIPS, RISC-V
EPIC	derleyici paralelliği belirler	komut düzeyi paralellik	IA-64 Itanium
Skalar	komut başına bir veri	genel amaçlı	çoğu CPU çekirdeği
Vektör	komut başına veri dizisi	bilimsel hesap	Cray-1, AVX
Dizi (SIMD)	çok PE, aynı komut	büyük paralellik	ILLIAC IV, GPU
DSP	MAC, özel adresleme	gerçek zamanlı sinyal	TMS320, SHARC
Sembolik	sembol/kural işleme	yapay zekâ (klasik)	LISP makineleri
GPU	binlerce basit çekirdek	grafik, derin öğrenme	NVIDIA, AMD Radeon

Tablo 1.1.15: İşlemci türlerinin özeti.

Tablodaki sınıflar birbirini dışlamaz. Bir ARM çekirdeği hem RISC hem sayıldır ve NEON uzantısıyla vektör işlemi de yapar. Bir Core i7 dışarıdan CISC, içeriden RISC benzeri, süperskalar, AVX ile vektörel ve çok çekirdekli olarak MIMD'dir. Sınıflar, bir işlemcinin hangi tasarım fikirlerini kullandığını anlatan etiketler gibi okunmalıdır.

1.1.6 Intel x86 ailesinin evrimi

8086 bir boşlukta doğmadı. Önünde 4 ve 8 bitlik işlemcilerden oluşan kısa ama hızlı bir tarih, arkasında da bugün hâlâ süren bir aile vardır. Bu kesim bu tarihi transistörden başlayarak izler, her işlemci kuşağının getirdiği yenilikleri sıralar ve x86 mimarisinin güçlü ve zayıf yanlarını tartışır.

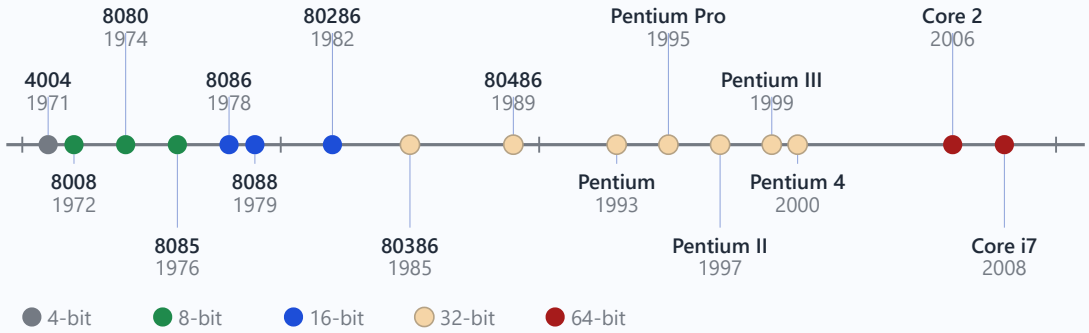
1.1.6.1 Transistörden mikroişlemciye

Mikroişlemcinin tarihi üç buluşla özetlenir.

- **1947, transistör.** Bell Laboratuvarları'nda John Bardeen, Walter Brattain ve William Shockley transistörü icat etti; buluş 1948'de duyuruldu. Transistör, lambanın yaptığı anahtarlama ve yükseltme işini çok daha küçük, dayanıklı ve az güç harcayan bir yarı iletkenle yaptı. Mikroelektronik devrimi böyle başladı.
- **1958, tümleşik devre.** Texas Instruments'ta Jack Kilby ilk **tümleşik devreyi** (*integrated circuit*, IC) gösterdi. Birçok transistör ve bağlantıları tek bir yarı iletken parça üzerinde birlikte üretilebildi. Böylece karmaşık devreler küçük ve ucuz hâle geldi.
- **1971, ilk mikroişlemci.** Intel 4004, bütün bir CPU'yu tek bir yongaya sığdırdı. Mikroişlemci artık bilgisayarın merkezî işlem birimi görevini tek başına üstlenebiliyordu.

Sonuç bugün her yerde görülür. Bilgisayarlar oda büyüklüğünden masaüstüne, oradan da cebe sığdı. Her adımda daha küçük, daha güçlü, daha hızlı ve daha ucuz oldular.

1.1.6.2 Zaman çizelgesi



Şekil 1.1.21: Intel mikroişlemcilerinin zaman çizelgesi: 1971'deki 4004'ten 2008'deki Core i7'ye. Renkler 4, 8, 16, 32 ve 64 bitlik kuşakları ayırır.

x86 adı, 8086'dan sonra gelen 80186, 80286, 80386 ve 80486 adlarının ortak "86" sonekinden gelir. Ailenin bütün üyeleri 8086 kodunu çalıştırabilir. Bu özelliğe **geriye uyumluluk** denir ve x86'nın kırk yılı aşan başarısının temelidir: her yeni işlemci, kullanıcıların elindeki yazılımı değiştirmeden çalıştı-

rır. Bunun bir sonucu da vardır: bugünkü bir Core i7 açıldığında 8086 gibi davranan **gerçek modda** (*real mode*) başlar ve 20 bitlik adreslerle çalışır. Daha geniş özelliklere ancak yazılım işlemciyi başka bir moda geçirdiğinde ulaşılır.

1.1.6.3 4 ve 8 bitlik işlemciler

İşlemci	Yıl	Saat	Transistör	Komut hızı / not
4004 (4 bit)	1971	740 kHz	2.300	≈ 60.000 komut/s; Ted Hoff, Federico Faggin, Stanley Mazor
4040 (4 bit)	1974	740 kHz	≈ 3.000	4004'ün geliştirilmiş sürümü (kesme, daha çok komut)
8008 (8 bit)	1972	500 kHz	3.500	≈ 50.000 komut/s; ilk 8 bitlik
8080 (8 bit)	1974	2 MHz	6.000	8008'in ≈ 10 katı hız; 16 bit adres (64 KB); Altair 8800
8085 (8 bit)	1976	3 MHz	6.500	tek +5 V besleme; 16 bit adres yolu; ≈ 769.000 komut/s

Tablo 1.1.16: 4 ve 8 bitlik Intel mikroişlemcileri.

4004 bir Japon hesap makinesi firması için tasarlanmıştı. 4 bitlik sözcüğü, ondalık rakamları BCD olarak işlemeye yetiyordu: her rakam 4 bite sığar. 8008 ile 8 bite geçildi; bir bayt artık bir karakteri tutabiliyordu. 8080 hem hızıyla hem 64 KB'lık adres alanıyla gerçek bir genel amaçlı işlemciydi ve ilk kişisel bilgisayarlardan Altair 8800'de kullanıldı. 8085 ise 8080'in birden çok besleme gerilimi isteyen tasarımını tek +5 V beslemeye indirdi. 8085'in en kısa komutu 3 MHz'de 1,3 µs sürer; bu da saniyede yaklaşık $1 / 1{,}3\ \mu s \approx 769.000$ komut eder.

1.1.6.4 16 bitlik işlemciler

İşlemci	Yıl	Saat	Transistör	Özellik
8086	1978	5 · 8 · 10 MHz	29.000	16 bit veri yolu, 20 bit adres yolu (1 MB); donanımsal çarpma/bölme; ≈ 0,33–0,75 MIPS
8088	1979	5 · 8 MHz	29.000	8086'nın 8 bit dış veri yollu ucuz sürümü; IBM PC (1981, 4,77 MHz)
80186 / 80188	1982	6 MHz'den	≈ 55.000	yonga üstünde kesme denetleyicisi, saat üreteci, yerel yol denetleyicisi, sayaçlar, DMA; 80188 ucuz (8 bit dış yol) sürüm
80286	1982	6–12,5 MHz	134.000	16 bit veri, 24 bit adres yolu (16 MB); korumalı mod; ≈ 0,9–2,7 MIPS

Tablo 1.1.17: 16 bitlik Intel mikroişlemcileri.

8086 komutları ortalama 10 turdan fazla sürer. Bu yüzden 5 MHz'de yaklaşık 0,33 MIPS elde edilir; bu da komut başına ortalama 15 tur demektir ($5 / 15 \approx 0{,}33$). 8088'in iç yapısı ve komut kümesi 8086 ile aynıdır, adres yolu da 20 bittir. Fark dış veri yolundadır: 8088 dışarıya 8 bitlik bir yolla bağlanır ve bir sözcüğü iki yol döngüsünde aktarır. Önden getirme kuyruğu da 6 değil 4 bayttır. IBM, ucuz 8 bitlik çevre yongalarını kullanabilmek için PC'de 8088'i seçti ve x86 bu kararla kişisel bilgisayarların standardı oldu.

80186, 8086'nın çevresinde ayrı yongalarla kurulan birimleri (kesme denetleyicisi, saat üreteci, zamanlayıcılar, DMA) yonganın içine aldı. Bu yönüyle mikrodenetleyiciye yaklaşan bir tasarımdır ve daha çok gömülü sistemlerde kullanıldı. 80286 ise iki yenilik getirdi: 24 bitlik adres yolu ve **korumalı mod** (*protected mode*).

1.1.6.5 32 bitlik işlemciler

İşlemci	Yıl	Saat	Transistör	Özellik
80386	1985	16–33 MHz	275.000	32 bit veri ve adres yolu (4 GB), sayfalama; sürümler: 386DX, 386SX (1988; 16 bit dış veri, 24 bit adres), 386SL (dizüstü)
80486	1989	16–100 MHz	1,2 milyon	yonga üstü 8 KB önbellek, tümleşik FPU, 5 aşamalı boru hattı; sürümler: 486DX, 486SX (FPU'suz), DX2, DX4
Pentium	1993	60 · 66 MHz (ilk)	3,1 milyon	süperskalar (2 boru hattı); ayrı 8 KB komut + 8 KB veri önbelleği; 64 bit dış veri yolu

Tablo 1.1.18: 32 bitlik Intel mikroişlemcileri.

Pentium'un dış veri yolu 64 bittir ama Pentium 32 bitlik bir işlemcidir: yazmaçları ve ALU'su 32 bittir. Geniş dış yol, önbelleği bellekten daha hızlı doldurmak içindir. Bu örnek, "*n* bitlik işlemci" tanımının yazmaç ve ALU genişliğine dayandığını bir kez daha gösterir.

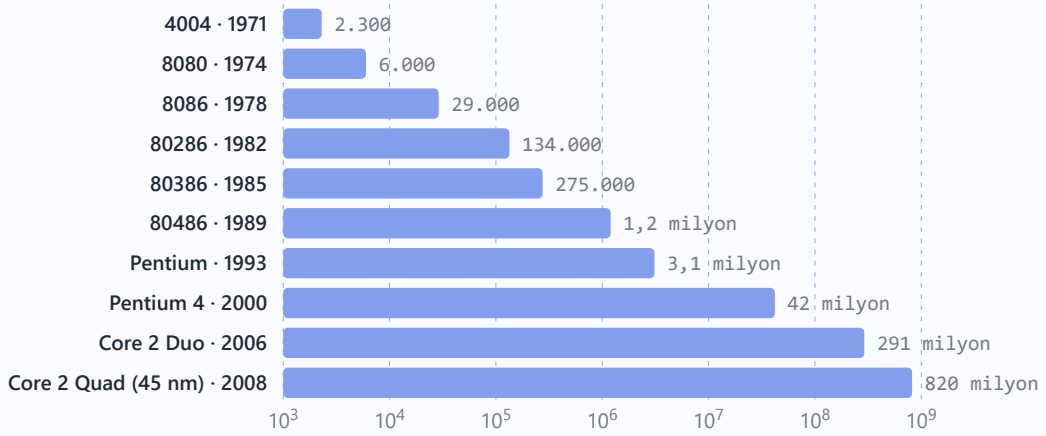
1.1.6.6 64 bitlik işlemciler

İşlemci	Yıl	Saat	Transistör	Özellik
Core 2	2006	1,2–3 GHz	291 milyon (Duo)	çekirdek başına 64 KB L1 (32 KB komut + 32 KB veri), 2–4 MB paylaşımlı L2; sürümler: Duo, Quad, Extreme
Core i7	2008	2,66–3,33 GHz	731 milyon	4 çekirdek, Hyper-Threading, tümleşik bellek denetleyicisi, 8 MB L3
Core i5	2009	2,4–3,6 GHz	—	orta sınıf; Turbo Boost
Core i3	2010	2,93–3,33 GHz	—	giriş sınıfı; 2 çekirdek, tümleşik grafik

Tablo 1.1.19: 64 bitlik Intel mikroişlemcileri. Saat aralıkları birinci nesil masaüstü ve mobil modellere göredir.

64 bitlik x86, yani x86-64, ilk kez AMD'nin AMD64 mimarisiyle (Opteron ve Athlon 64, 2003) ortaya çıktı. Intel bu mimariyi 2004'te Pentium 4 ile benimsedi. Bu, x86 tarihinde ailenin yönünü Intel'in değil rakibinin belirlediği nadir anlardandır; Intel'in kendi 64 bitlik yolu olan IA-64 (Itanium) piyasada tutunamamıştı.

1.1.6.7 Transistör sayısı ve Moore yasası



Şekil 1.1.22: Logaritmik ölçekte transistör sayıları: 4004'te 2.300, 8086'da 29.000, Pentium'da 3,1 milyon, Core 2 Quad'da (45 nm) 820 milyon. Logaritmik ölçekte düzgün artış, sabit oranlı (üstel) büyüme demektir.

Tanım 1.1.18

Moore yasası, Intel'in kurucularından Gordon Moore'un bir gözlemidir. Moore'un 1965'te ortaya koyup 1975'te düzelttiği biçimiyle yonga başına transistör sayısı yaklaşık her iki yılda bir iki katına çıkar.

Gözlem bir doğa yasası değil, yarı iletken endüstrisinin onlarca yıl tutturduğu bir hedeftir. Sağlaması kolaydır. 4004'ten (1971, 2.300 transistör) Core 2 Quad'a (2008, 820 milyon) 37 yıl geçmiştir. Oran $820 \times 10^6 / 2300 \approx 356.500$'dür ve $2^{18} = 262.144$ ile 2^{19} arasındadır; yani yaklaşık 18 kez iki katına çıkma. 37 yılda iki yılda bir iki katına çıkma $37 / 2 \approx 18{,}5$ kez eder. Gözlem bu aralıkta şaşırtıcı ölçüde tutmuştur. Transistör sayısındaki artış, önbellek, boru hattı, çok çekirdek gibi bu notta anlatılan bütün yeniliklerin hammaddesidir.

1.1.6.8 8080 ve 8086

Mikroişlemcilerin yapısı, örgütlenimi ve teknolojisi kısa sürede köklü biçimde değişti. Bu değişim ilk genel amaçlı mikroişlemci 8080'den 29.000 transistörlü 8086'ya, oradan 820 milyon transistörlü dört çekirdekli Core 2'ye uzanır. Bu dersin işlemcisi 8086'dır; onu öncülü 8080 ile karşılaştırmak, hangi adımın neden atıldığını gösterir.

8080 (1974) ilk genel amaçlı mikroişlemci olarak anılır. Veri yolu 8 bit, adres yolu 16 bittir; 64 KB bellek adresler. İlk kişisel bilgisayarlarda, örneğin Altair 8800'de kullanıldı.

8086 (1978) 16 bitlik bir işlemcidir: veri yolu 16 bittir; 8080'den daha çok sayıda 16 bitlik yazmacı vardır. Adres yolu 20 bittir ve 1 MB bellek adresler. Yalnız gerçek modda çalışır; bellek koruması yoktur. En önemli yeniliği, komutları çalıştırmadan önce **önden getirip** (*prefetch*) 6 baytlık bir komut

kuyruğunda bekletmesidir. BIU kuyruğu doldururken EU önceki komutu yürütür; getirme ile yürütme üst üste biner. 8086'da önbellek yoktur; hız kazancını bu kuyruk sağlar. Yonga üstü önbellek x86'ya ancak 80486 ile geldi.

Örnek 1.1.7

Kuyruğun kazancı kabaca nedir? 8086'da 2 baytlık bir komut, kuyruk olmasaydı önce bir yol döngüsüyle (4 tur) getirilip sonra yürütülürdü. `add ax, cx` gibi 3 turluk bir komut için bu *4 + 3 = 7* tur ederdi. Kuyruk doluyken getirme önceki komutların yürütülmesi sırasında yapılmış olur ve komut 3 turda biter. Veri sayfasındaki tur sayıları bu yüzden "kuyruk dolu" varsayımıyla verilir. Atlamalarda kuyruk boşaltıldığı için kazanç kaybolur; atlayan `loop` 'un 17 tur sürmesinin nedeni budur.

1.1.6.9 80286, 80386 ve 80486

80286 (1982) 24 bitlik adres yoluyla 16 MB bellek adresler. Gerçek modun yanında 16 bitlik korumalı mod sunar. Veri yolu ve yazmaçları yine 16 bittir.

80386 (1985) Intel'in ilk 32 bitlik mikroişlemcisidir. Çoklu görevi (*multitasking*) donanımla destekler. 32 bitlik korumalı mod, **sayfalama** (*paging*) ve sanal 8086 modu getirir; 4 GB bellek adresler. Sanal 8086 modu, korumalı modda çalışan bir işletim sisteminin altında eski 8086 programlarını kendi yalıtılmış ortamlarında çalıştırmayı sağlar.

80486 (1989) yonga üstü önbellek (8 KB) ve komut boru hattı getirir. Sayfa yazma korumasını (CR0 yazmacının WP biti) ekler. Karmaşık aritmetik için tümleşik bir matematik işlemcisi (FPU) içerir; bu birim önceden ayrı bir yonga, 80387'ydi.

Tanım 1.1.19

Korumalı mod, her programın belleğini öbürlerinden yalıtan çalışma modudur. Bir program kendisine ayrılmamış belleğe erişmeye çalışırsa işlemci bunu engeller ve işletim sistemine haber verir. Böylece bir programın hatası işletim sistemini ya da başka programları çökertemez. Modern işletim sistemlerinin temeli budur.

8086'nın gerçek modunda böyle bir koruma yoktur. Bir program 1 MB'ın herhangi bir yerine, örneğin işletim sisteminin kendi koduna ya da kesme vektör tablosuna yazabilir. DOS döneminde hatalı bir programın bütün bilgisayarı kilitlemesi bu yüzden sıradandı.

1.1.6.10 Pentium ailesi

Pentium (1993) süperskalardır; iki boru hattında (U ve V) iki komut aynı anda çalışabilir. x86'da ilk kez **dallanma tahmini** (*branch prediction*) kullanır: bir koşullu atlamanın sonucunu önceki davranışına bakarak tahmin eder ve boru hattını tahmin edilen yoldan doldurur. Sayfa boyutu uzantısı (PSE), 4 KB'lık sayfaların yanında 4 MB'lık sayfalar kullanarak sayfalamayı iyileştirir. Komut ve veri önbellekleri ayrıdır.

Pentium Pro (1995) üç teknikle yürütmeyi kökten değiştirir: gelişmiş dallanma tahmini, veri akış analiziyle **sıra dışı yürütme** (*out-of-order execution*) ve **spekülatif yürütme** (*speculative execution*). Sıra dışı yürütmede, işlenenleri hazır olan bir komut, önündeki ve bir sonucu bekleyen komutu beklemeden yürütülür; sonuçlar yine program sırasıyla kesinleştirilir. Spekülatif yürütmede, tahmin edilen dallanma yolundaki komutlar sonuç kesinleşmeden yürütülür; tahmin yanlış çıkarsa sonuçlar atılır. Intel bu üç tekniğin birlikte kullanımına "dinamik yürütme" (*dynamic execution*) adını verdi. Pentium Pro, düzey 2 (L2) önbelleği aynı paketin içine alır. PAE (*physical address extension*) ile 32 bitlik sanal adresler 36 bitlik fiziksel adreslere çevrilir ve 64 GB belleğe ulaşılır.

Pentium II (1997), multimedya için tam sayı SIMD komutları olan MMX teknolojisini Pentium Pro çekirdeğiyle birleştirdi. Video, ses ve grafik verisini etkili işler. MMX ilk kez 1997 başında "Pentium MMX" ile gelmişti; Pentium II onu devraldı.

Pentium III (1999) SSE'yi (*Streaming SIMD Extensions*) getirdi: 70 yeni komut ve kayan noktalı SIMD. Üç boyutlu grafik yazılımını destekler. En yüksek saat hızı 1,4 GHz'dir.

Pentium 4 (2000) multimedya için kayan nokta geliştirmeleri (önce SSE2, sonra SSE3) ekledi. NetBurst adlı çok derin bir boru hattıyla 3,8 GHz'e kadar çıktı ve Hyper-Threading'i getirdi. Hyper-Threading tek bir çekirdeğin iki iş parçacığını aynı anda yürütmesini sağlar; işletim sistemi bir çekirdeği iki mantıksal işlemci olarak görür. Çok derin boru hattı yüksek frekansa izin verdi, ama yanlış tahmin edilen her dallanmada boşaltılan aşama sayısı da arttı ve güç tüketimi yükseldi. Saat hızı yarışı bu kuşakla sınırına dayandı.

1.1.6.11 Core ve Core 2

Core (Core Duo, 2006) mobil kökenli, düşük güçlü bir tasarımdır. İki çekirdek aynı kalıpta (*die*) durur ve L2 önbelleğini paylaşır; aynı kalıptaki iki çekirdeğin önbelleği paylaştığı ilk Intel işlemcisidir. Sanallaştırma teknolojisi (VT-x) bu kuşakla eklendi.

Core 2 (2006) 64 bitlik mimariyi (Intel 64) kullanır: 64 bitlik yazmaç kümesi ve adresleme modları. Core 2 Quad tek pakette dört çekirdek sunar; paketin içinde iki çift çekirdekli kalıp vardır.



Şekil 1.1.23: Core 2 Duo'da tek kalıpta iki çekirdek ve paylaşılan L2 önbellek vardır. Core 2 Quad bir pakette iki çift çekirdekli kalıp taşır; her kalıbın kendi L2 önbelleği vardır.

Intel'in ilk çift çekirdekli masaüstü işlemcisi 2005'te çıkan Pentium D'dir. Bu işlemcide iki çekirdeğin L2 önbellekleri ayrıydı ve çekirdekler birbirleriyle dış yol üzerinden haberleşiyordu; 2006 sürümünde iki çekirdek iki ayrı kalıp olarak aynı pakete konmuştu. Önbelleği paylaşmak, çekirdeklerin ortak veriyi belleğe gitmeden kullanmasını sağlar.

Kalıp ile **paket** farklı şeylerdir. Kalıp, silisyum dilimden kesilen tek bir yonga parçasıdır. Paket ise kalıbı (ya da birden çok kalıbı) taşıyan, pinleri ve ısı yayıcısı olan dış kılıftır. Kullanıcının elindeki "işlemci" pakettir.

1.1.6.12 x86 mimarisinin güçlü ve zayıf yanları

x86'nın uzun ömrünün dört nedeni vardır.

- **Uyumluluk** (*compatibility*): eski yazılımlar değiştirilmeden yeni x86 işlemcilerde çalışır. 1978'de 8086 için yazılmış kod bugün de çalışabilir.
- **Performans** (*performance*): her yeni işlemci kuşağı öncekinden daha hızlı ve verimli oldu.
- **Çeşitlilik** (*versatility*): kişisel bilgisayarlardan sunuculara, gömülü sistemlerden mobil aygıtlara kadar geniş bir kullanım yelpazesi.
- **Geniş endüstri desteği**: büyük bir donanım ve yazılım üreticisi topluluğu x86'yı destekler; bu destek yeniliğin ve gelişmenin itici gücü oldu.

Aynı tarih, mimarinin zayıf yanlarını da açıklar.

- **Karmaşık komut kümesi**: binlerce komut ve değişken uzunluklu kodlama çözücü donanımını karmaşıklaştırır; kodu performans için eniyilemek zorlaşır. Geriye uyumluluk, kullanılmayan eski komutların bile korunmasını gerektirir.
- **Güç tüketimi**: x86'nın evrimi güç tüketiminde önemli bir artış getirdi. Pil ömrünün kritik olduğu aygıtlarda bu büyük bir sorundur.
- **Isı yayılımı**: işlemciler güçlendikçe daha çok ısınır; soğutma maliyeti ve gürültü artar.
- **Maliyet ve lisans**: x86 mimarisini yalnız lisans anlaşması olan firmalar (örneğin AMD) üretebilir; rekabet sınırlıdır.

Bu nedenlerle telefon ve tabletlerde ARM (RISC) baskındır; x86 ise masaüstü ve sunucuda güçlüdür. Son yıllarda ARM tabanlı işlemciler dizüstü bilgisayarlara ve sunuculara da girmiştir.

1.1.7 Modern işlemci özellikleri

8086 ile bugünkü bir işlemci arasındaki fark yalnız saat hızında değildir. Bu kesim önce saat hızının neden tek başına performans ölçüsü olmadığını sayısal bir örnekle gösterir. Sonra bugünkü işlemcilerin performansını belirleyen özellikleri sıralar ve bellek hiyerarşisini açıklar.

1.1.7.1 Saat hızı tek başına performans değildir

CPU süresi formülü bu kesimin temelidir:

$$\text{CPU süresi} = N \times CPI \times T = N \times CPI / f$$

Burada N çalışan komut sayısı, CPI komut başına ortalama saat turu, f saat frekansıdır ($T = 1/f$). Formül, saat frekansının üç çarpandan yalnız biri olduğunu açıkça gösterir.

Örnek 1.1.8

Aynı program ($N = 10^9$ komut) iki işlemcide çalışıyor. İşlemci A: $f = 2$ GHz, CPI = 2. İşlemci B: $f = 1{,}5$ GHz, CPI = 1. Hangisi daha hızlıdır?

$$A: 10^9 \times 2 / (2 \times 10^9) = 1 \text{ s}$$
$$B: 10^9 \times 1 / (1{,}5 \times 10^9) \approx 0{,}67 \text{ s}$$

Saati düşük olan B, $1 / 0{,}67 = 1{,}5$ kat, yani %50 daha hızlıdır. A saniyede $2 \times 10^9 / 2 = 10^9$, B ise $1{,}5 \times 10^9 / 1 = 1{,}5 \times 10^9$ komut yürütür.

Saat hızı “saniyedeki tur sayısıdır”, “saniyedeki komut sayısı” değildir. Saniyedeki komut sayısı f / CPI'dır ve mimari (boru hattı, önbellek, süperskalar yapı) CPI'yı düşürür. Pentium 4'ten Core kuşağına geçiş bunun tarihteki örneğidir. Core işlemcilerin saat hızı daha düşüktü, ama CPI'ları da düşüktü. Pentium 4'ten hem daha hızlı çalıştılar hem daha az güç harcadılar.

Sık yapılan hata

İki işlemci yalnız saat hızlarına bakılarak karşılaştırılır. Saat hızı yalnız aynı mimarideki ve aynı CPI'lı iki işlemciyi karşılaştırmaya yeter. Farklı mimarilerde CPI, hatta aynı program için N bile farklı olabilir; doğru karşılaştırma, aynı programın iki işlemcideki süresidir.

1.1.7.2 Performansı belirleyen özellikler

Saat hızı (*clock speed*): saniyedeki saat turu sayısıdır (GHz). Komut hızı ayrıca CPI'ya bağlıdır.

Komut kümesi mimarisi (ISA): CISC ve RISC gibi mimariler verimliliği ve karmaşıklığı etkiler. ISA, yazılım ile donanım arasındaki sözleşmedir.

Önbellek (*cache*): sık kullanılan veriyi işlemcinin yanında tutan küçük ve çok hızlı bellektir. Düzeyleri L1, L2 ve L3 diye adlandırılır.

Çok çekirdek (*multi-core*): birden çok görevin gerçekten aynı anda yürütülmesini sağlar ve toplam performansı artırır. Tek bir programın hızlanması için ise programın iş parçacıklarına bölünmüş olması gerekir.

Sanallaştırma (*virtualization*): aynı donanım üzerinde birden çok işletim sisteminin, yani sanal makinenin, birbirinden yalıtılmış olarak çalışmasını sağlar. Donanım desteği Intel'de VT-x, AMD'de AMD-V adını taşır.

Güç yönetimi (*power management*): yük azaldığında saat frekansı ve besleme gerilimi düşürülür; buna DVFS (*dynamic voltage and frequency scaling*, dinamik gerilim ve frekans ölçekleme) denir. Boştaki çekirdekler uyku durumlarına (C-durumları) alınır. Intel'in SpeedStep ve Turbo Boost teknolojileri bu ilkeyle çalışır: Turbo Boost, ısı ve güç sınırı izin verdiğinde bazı çekirdeklerin frekansını kısa süre için yükseltir. Dinamik güç kabaca $C\,V^2\,f$ ile orantılı olduğu için gerilimi düşürmek gücü karesiyle azaltır.

Grafik işleme: modern işlemciler tümleşik bir grafik işlem birimi (iGPU) içerir; grafik yoğun görevler ayrı bir ekran kartı olmadan daha hızlı ve verimli işlenir.

Güvenlik özellikleri: donanım düzeyinde şifreleme (AES-NI komutları), güvenli önyükleme (*secure boot*) ve yalıtılmış güvenli bölgeler (TPM, SGX, ARM TrustZone).

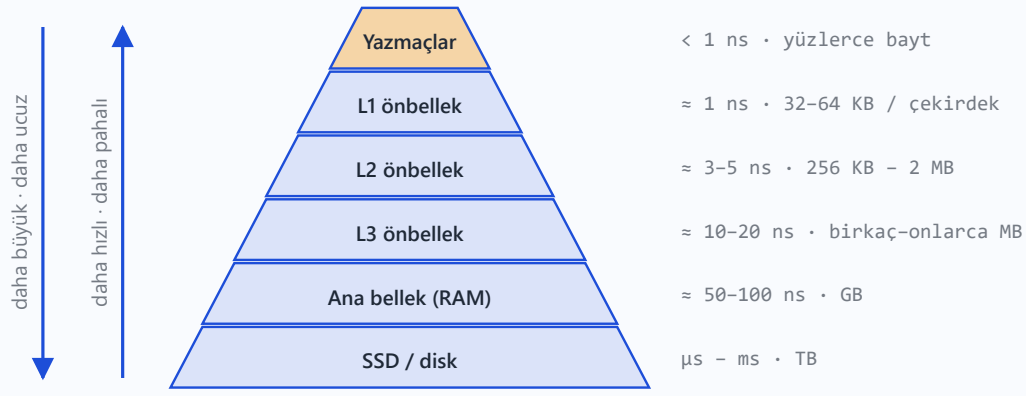
Bağlantı (*connectivity*): özellikle mobil ve gömülü yongalarda Wi-Fi, Bluetooth ve Ethernet gibi ağ birimleri yonganın içindedir.

Makine öğrenmesi: yapay zekâ için özel birimler eklenmiştir: sinir ağı işlem birimleri (NPU), matris hızlandırıcıları (Intel AMX) ve vektör nokta çarpımı komutları.

Yonga üstü sistem (*system on chip*, SoC): CPU, GPU, NPU, bellek denetleyicisi ve G/Ç'nin tek yongada birleşmesidir. Bu eğilim, mikrodenetleyicinin “her şey tek yongada” fikrine daha büyük ölçekte bir geri dönüştür. Telefonlardaki işlemciler ve Apple'ın M serisi birer SoC'dir.

1.1.7.3 Önbellek ve bellek hiyerarşisi

İşlemci hızı yıllar içinde bellek hızından çok daha hızlı arttı. Bugünkü bir işlemci bir turda birkaç komut bitirirken ana bellekten bir okuma 50–100 ns, yani yüzlerce tur sürer. Her erişimde ana belleğe gidilseydi işlemci zamanının çoğunu beklemekle geçirirdi. Çözüm, farklı hız ve boyutlardaki bellekleri bir hiyerarşi içinde birlikte kullanmaktır.



Şekil 1.1.24: Bellek hiyerarşisi: yazmaçlar, L1, L2 ve L3 önbellek, ana bellek ve disk. Yukarı çıktıkça hız artar ve bayt başına maliyet yükselir; aşağı indikçe kapasite büyür ve maliyet düşer. Süreler yaklaşıktır.

Tanım 1.1.20

Önbellek (*cache*), ana bellekteki verinin sık kullanılan bir kısmının kopyasını işlemcinin yanında tutan küçük ve hızlı bellektir. **Yerellik ilkesi** (*principle of locality*), programların aynı veriye kısa sürede yeniden (zamansal yerellik) ve yakın adreslerdeki veriye art arda (uzamsal yerellik) erişme eğilimidir.

Önbelleğin işe yaramasının nedeni yerellik ilkesidir. Bir döngünün komutları her geçişte yeniden okunur; bir dizinin öğeleri ardışık adreslerden okunur. Önbellek bu küçük çalışma kümesini işlemcinin yakınında tutar ve erişimlerin büyük çoğunluğu önbellekte bulunur. Bu yüzden ortalama erişim süresi RAM'inkinin çok altına iner.

Örnek 1.1.9

Önbelleğe erişim 1 ns, ana belleğe erişim 100 ns sürüyor. Erişimlerin %95'i önbellekte bulunuyorsa ortalama erişim süresi nedir?

Erişimin %95'i 1 ns sürer. Kalan %5'inde önce önbelleğe bakılır (1 ns), bulunamayınca ana belleğe gidilir (100 ns):

$$\text{ortalama} = 1 + 0{,}05 \times 100 = 1 + 5 = 6 \text{ ns}$$

Ana belleğin 100 ns'lik erişim süresi, ortalamada 6 ns'ye iner; yaklaşık 17 kat iyileşme. Bulunma oranı %99'a çıkarsa ortalama $1 + 0{,}01 \times 100 = 2$ ns olur.

8086'da önbellek yoktu; bellek erişimi zaten yalnız dört saat turu sürüyordu ve işlemciyle bellek hızları birbirine yakındı. Bu denge 1980'lerin sonunda bozuldu. 80486 yonga üstü önbelleği, Pentium ayrı komut ve veri önbelleklerini, Pentium Pro aynı paketteki L2'yi, Core i7 ise çekirdekler arasında paylaşılan L3'ü getirdi.

> **Sık yapılan hata**
>
> Önbelleğin RAM'i hızlandırdığı ya da veriyi sıkıştırdığı sanılır. Önbellek RAM'in kendisini hızlandırmaz ve veriyi sıkıştırmaz. Yalnız verinin sık kullanılan kopyasını yakında tutar; erişimlerin çoğu RAM'e hiç gitmeden karşılandığı için ortalama süre düşer. Yerelliği zayıf bir programda, örneğin belleğe rastgele dağılmış adreslere erişen bir programda, önbelleğin kazancı da küçüktür.

1.1.8 Sık yapılan hatalar

- **Veri yolu ile adres yolunu karıştırmak.** Adreslenebilir belleği adres yolu belirler (2^n bayt); veri yolu bir erişimde taşınan bit sayısını belirler. 16 bitlik 8086 1 MB adresler.
- **IP'nin her komutta 1 arttığını sanmak.** IP bayt adresi tutar ve komut uzunluğu kadar artar; 8086 komutları 1–6 bayttır.
- **Komut döngüsü, yol döngüsü ve saat turunu karıştırmak.** Saat turu en küçük zaman birimidir. Bir yol döngüsü 8086'da en az 4 turdur. Bir komut döngüsü bir ya da birkaç yol döngüsü içerebilir ve yol döngüsü olmayan turlar da kapsayabilir.
- **Atlayan ve atlamayan `loop`'u aynı süreyle saymak.** 8086'da atlayan `loop` 17, atlamayan 5 tur sürer; n geçişli bir döngüde $n - 1$ atlama vardır.
- **Çok kelimeli toplamada yüksek yarıyı `add` ile toplamak.** Düşük yarıdan gelen elde ancak `adc` ile eklenir.
- **Boru hattının tek komutu hızlandırdığını sanmak.** Boru hattı gecikmeyi değil verimi artırır; n komut $k + n - 1$ turda biter.
- **Saat hızını performansla eş tutmak.** Süre $N \times CPI / f$'dir; düşük saatli ama düşük CPI'lı bir işlemci daha hızlı olabilir.
- **"n bitlik işlemci" tanımını dış veri yoluna dayandırmak.** Tanım yazmaç ve ALU genişliğine dayanır: 8088'in dış yolu 8 bit ama işlemci 16 bitlik; Pentium'un dış yolu 64 bit ama işlemci 32 bitlik.
- **Önbelleğin RAM'i hızlandırdığını sanmak.** Önbellek yalnız sık kullanılan verinin kopyasını yakında tutar; kazancı yerellik ilkesinden gelir.

1.1.9 Alıştırmalar ve çözümleri

Bu alıştırmalar sunumun alıştırma slaytındaki soruların çözümlü hâlidir.

1.1.9.1 Saat periyodu ve yol döngüsü

Soru. 5 MHz'lik bir 8086'da saat periyodu ve 4 T'lik bir yol döngüsünün süresi nedir?

Çözüm. $T = 1 / f = 1 / (5 \times 10^6)\ s = 200\ ns$. Yol döngüsü dört turdur: $4 \times 200\ ns = 800\ ns$.

1.1.9.2 24 bitlik adres yolu

Soru. 24 bitlik bir adres yolu kaç bayt belleği adresleyebilir? Hangi işlemcide vardır?

Çözüm. $2^{24} = 16.777.216$ bayt. 1 MB = 2^{20} bayt olduğundan bu $2^4 = 16$ MB'tır. 24 bitlik adres yolu 80286'da vardır.

1.1.9.3 Beş aşamalı boru hattında 100 komut

Soru. 5 aşamalı bir boru hattında 100 komut kaç turda biter? Boru hattısız kaç tur gerekir?

Çözüm. $k + n - 1 = 5 + 100 - 1 = 104$ tur. Boru hattısız $5 \times 100 = 500$ tur gerekir. Hızlanma $500 / 104 \approx 4{,}8$ kattır.

1.1.9.4 Çalışma süresi

Soru. $N = 2 \times 10^9$ komutluk bir program, CPI = 1,5 olan 3 GHz'lik bir işlemcide kaç saniye sürer?

Çözüm. $2 \times 10^9 \times 1{,}5 / (3 \times 10^9) = 1\ s$.

1.1.9.5 Mikroişlemci mi, mikrodenetleyici mi?

Soru. Çamaşır makinesi kontrol kartı ve dizüstü bilgisayar için mikroişlemci mi mikrodenetleyici mi seçilir? Neden?

Çözüm. Çamaşır makinesi için mikrodenetleyici: ucuzdur, az güç harcar ve tek, sabit bir işi yapar; bellek ve G/Ç yonganın içindedir. Dizüstü bilgisayar için mikroişlemci: yüksek performans, gigabaytlarca bellek ve işletim sistemi gerekir.

1.1.9.6 EPIC paketi

Soru. EPIC paketinde 128 bit nasıl paylaştırılır?

Çözüm. 41 bitlik üç komut $3 \times 41 = 123$ bit tutar; kalan 5 bit şablondur: $123 + 5 = 128$.

1.1.10 Özet

- **Mikroişlemci**, bellekteki komutları getir–çöz–yürüt döngüsüyle çalıştıran programlanabilir CPU yongasıdır. **Mikrodenetleyici** bellek ve çevre birimlerini de aynı yongada taşır; ucuz, az güç harcayan ve tek işe adanmış sistemlerde kullanılır.
- CPU dış dünyayla üç yolla konuşur: **adres yolu** (tek yönlü; n hat 2^n bayt), **veri yolu** (çift yönlü) ve **kontrol yolu**. 8086'nın adres yolu 20 bit (1 MB), veri yolu 16 bittir.
- **Komut döngüsü** getir, çöz, yürüt ve sonuç aşamalarından oluşur. **IP** sıradaki komutun adresini tutar ve komut uzunluğu kadar artar; atlamalar IP'ye yeni değer yazar.
- CPU; **ALU**, **kontrol birimi**, **yazmaçlar** ve **yol arayüzünden** oluşur. 8086'da BIU komutları önden getirip 6 baytlık kuyrukta bekletir, EU onları yürütür.
- Periyot $T = 1/f$; 8086'da bir yol döngüsü en az 4 T'dir. *CPU süresi* $= N \times CPI / f$. Saat hızı tek başına performans ölçüsü değildir.
- Bir döngünün süresi komutların tur sayıları toplanarak bulunur; küçük gövdeli döngülerde süreyi atlamaların bedeli belirler.
- 16 bitlik işlemci 32 bitlik sayıları yazmaç çiftlerinde tutar ve `add` + `adc` ile toplar.
- **CISC** zengin ve değişken uzunluklu komutlar, **RISC** basit sabit uzunluklu komut + yükle/sakla + **boru hattı**, **EPIC** derleyici güdümlü paralellik sunar. k aşamalı boru hattında n komut $k + n - 1$ turda biter.
- **Vektör** ve **dizi (SIMD)** işlemcileri bir komutla çok veri işler; **DSP** tek turda çarp-topla yapar; **GPU** binlerce basit çekirdekle veri paralelliğinden yararlanır; **sembolik** işlemciler kural ve sembol işler.
- Intel 4004 (1971) → 8080 (8 bit) → 8086 (1978, 16 bit, 1 MB) → 80286 (korumalı mod) → 80386 (32 bit) → 80486 (önbellek, FPU) → Pentium (süperskalar) → Core 2 (64 bit, çok çekirdek). **Geriye uyumluluk** x86'nın en büyük gücüdür.
- Modern işlemciler **önbellek**, çok çekirdek, sanallaştırma, güç yönetimi ve özel hızlandırıcılarla performans kazanır. Önbelleğin kazancı **yerellik ilkesinden** gelir.

1.1.11 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

mikroişlemci *microprocessor* · mikrodenetleyici (MCU) *microcontroller* · merkezî işlem birimi (CPU) *central processing unit* · yonga *chip* · tümleşik devre (IC) *integrated circuit* · yol *bus* · adres yolu *address bus* · veri yolu *data bus* · kontrol yolu *control bus* · yol döngüsü *bus cycle* · T durumu *T state* · bekleme durumu *wait state* · komut döngüsü *instruction cycle* · getir / çöz / yürüt *fetch / decode / execute* · işlem kodu *opcode* · işlenen *operand* · anlık değer *immediate value* · komut işaretçisi (IP) *instruction pointer* · program sayacı (PC) *program counter*

aritmetik-mantık birimi (ALU) *arithmetic logic unit* · kontrol birimi *control unit* · yazmaç *register* · bayrak *flag* · yol arayüz birimi (BIU) *bus interface unit* · yürütme birimi (EU) *execution unit* · önden getirme kuyruğu *prefetch queue* · mikrokod *microcode* · saat sinyali *clock signal* · saat turu *clock cycle* · saat hızı *clock speed* · komut başına tur (CPI) *cycles per instruction* · bit genişliği *word length* · sözcük *word* · kesim *segment* · bağıl konum *offset* · elde bayrağı (CF) *carry flag* · komut kümesi *instruction set* · CISC *complex instruction set computer* · RISC *reduced instruction set computer* · boru hattı *pipeline* · boru hattında bekleme *pipeline stall* · sayıl (skalar) işlemci *scalar processor* · vektör işlemcisi *vector processor* · SIMD *single instruction, multiple data* · sayısal sinyal işlemcisi (DSP) *digital signal processor* · grafik işlem birimi (GPU) *graphics processing unit* · kayan nokta birimi (FPU) *floating-point unit* · gerçek mod *real mode* · korumalı mod *protected mode* · dallanma tahmini *branch prediction* · önbellek *cache* · bellek hiyerarşisi *memory hierarchy*

1.1.12 Kendini Dene

1. Saat turu sayımında incelenen döngü (`mov cx, 4` … `hlt`, 78 tur) 10 MHz'lik bir 8086'da kaç mikrosaniye sürer?

   Cevap

   10 MHz'de $T = 100\ ns$. Tur sayısı frekanstan bağımsızdır: $78 \times 100\ ns = 7{,}8\ \mu s$. Süre 5 MHz'deki 15,6 µs'nin yarısıdır.

2. Aynı döngüde `mov cx, 4` yerine `mov cx, 5` yazılırsa döngü kaç tur sürer ve AX ne olur?

   Cevap

   Gövde beş kez çalışır; `loop` dört kez atlar, bir kez atlamaz: $4 + 4 + 5 \times 3 + 4 \times 17 + 5 + 2 = 98$ tur. AX = 5 + 4 + 3 + 2 + 1 = 15'tir. 5 MHz'de süre $98 \times 200\ ns = 19{,}6\ \mu s$'dir.

3. IP izlemesindeki program `0100h` yerine `0200h` adresinden başlasaydı `loop dongu` komutunun makine kodu değişir miydi?

   Cevap

   Değişmezdi. `loop` hedef adresi değil, sıradaki komuta göre uzaklığı saklar. Etiket `0206h`'de, sıradaki komut `020Ah`'de olur; uzaklık yine -4 (`FCh`) ve kod yine `E2 FC`'dir.

4. DX:AX = `0000FFFFh` ve CX:BX = `00000001h` iken `add ax, bx` ve `adc dx, cx` yürütülüyor. Her

komuttan sonra AX, DX, CF ve ZF ne olur?

Cevap

`add ax, bx`: `FFFFh + 0001h = 10000h`; AX = `0000h`, CF = 1, ZF = 1. `adc dx, cx`: *0000h + 0000h + 1 = 0001h*; DX = `0001h`, CF = 0, ZF = 0. Sonuç DX:AX = `00010000h` = 65.536'dır.

5. AL = `05h` iken `add al, 0FBh` yürütülüyor. AL, ZF ve CF ne olur?

Cevap

`05h + FBh = 100h`. 9. bit AL'ye sığmaz: AL = `00h`, CF = 1. Sonuç sıfır olduğu için ZF = 1'dir. İşaretli yorumda *5 + (−5) = 0* doğru bir sonuçtur, OF = 0 kalır.

6. 6 aşamalı bir boru hattında bağımsız 50 komut kaç turda biter? Boru hattısız kaç tur gerekir?

Cevap

6 + 50 − 1 = 55 tur. Boru hattısız *6 × 50 = 300* tur. Hızlanma *300 / 55 ≈ 5,5* kattır.

7. Skalar döngü (dört öğeli *c = a + b*) 100 öğeli dizilere uygulansaydı kaç komut yürütülürdü? Bunların kaçı toplama olurdu?

Cevap

2 + 100 × 5 = 502 komut; bunların 100'ü toplamadır. Kalan 402 komut okuma, yazma, döngü yükü ve hazırlıktır.

8. Bir işlemci 2,4 GHz'de ve CPI = 0,8 ile çalışıyor. Saniyede kaç milyon komut yürütür? 6×10^9 komutluk bir program kaç saniye sürer?

Cevap

$f / CPI = 2{,}4 \times 10^9 / 0{,}8 = 3 \times 10^9$ komut/s, yani 3000 MIPS. Süre $6 \times 10^9 / (3 \times 10^9) = 2$ *s*.

9. L1 önbelleğe erişim 2 ns, ana belleğe erişim 80 ns sürüyor ve erişimlerin %90'ı önbellekte bulunuyor. Ortalama erişim süresi nedir?

Cevap

2 + 0,10 × 80 = 2 + 8 = 10 ns. Ana belleğe göre 8 kat iyileşme vardır.

10. 4 GB belleği adreslemek için kaç adres hattı gerekir? 8086'nın adres yoluna kaç hat eklenmesi ge-

rekirdi?

Cevap

4 GB = $2^2 \times 2^{30} = 2^{32}$ bayt, yani 32 hat. 8086'nın 20 hattına 12 hat eklenmesi gerekirdi; 80386'nın adres yolu 32 bittir.

1.1.13 İleri okuma

- B. B. Brey, *The Intel Microprocessors* (8. baskı), Bölüm 1–2: 8086'dan Pentium 4'e Intel ailesinin tarihini, bellek ve G/Ç düzenini ve yazmaç modelini 8086 ağırlıklı anlatır; bu dersin temel kaynağıdır.
- W. Stallings, *Computer Organization and Architecture* (10. baskı), Bölüm 2–3 ve 14–16: bilgisayarın evrimini, komut döngüsünü, yol yapılarını, boru hattını ve RISC ile süperskalar işlemcileri ayrıntılı ele alır.
- D. A. Patterson ve J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Bölüm 1 ve 4: CPU süresi formülünü ve performans karşılaştırmasını, beş aşamalı RISC boru hattını ve bağımlılıkları açık örneklerle verir.
- Intel Corporation, *8086 16-Bit HMOS Microprocessor* veri sayfası: 8086'nın pinlerini, yol döngüsü zamanlamasını (T1–T4) ve komutların tur sayılarını içerir; bu nottaki tur sayıları bu kaynaktandır.

1.1.14 Kaynaklar

1. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 3. baskı, Pearson.
2. B. B. Brey, *The Intel Microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium …*, 8. baskı, Pearson.
3. W. Stallings, *Computer Organization and Architecture*, 10. baskı, Pearson.
4. D. A. Patterson, J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann.
5. Intel Corporation, *8086 16-Bit HMOS Microprocessor* veri sayfası; *Intel 64 and IA-32 Architectures Software Developer's Manual*.

KISIM I · BÖLÜM 1.2

Sayı Sistemleri

Bir mikroişlemcinin içindeki her şey bitlerden oluşur: yazmaçlar, bellek hücreleri, adres ve veri yolundaki hatlar, bayraklar. Bir yazmaçta duran 1111 1011 deseni bağlama göre 251 olabilir, −5 olabilir, bir karakterin parçası ya da bir kayan noktalı sayının üs alanı olabilir. Desenin kendisi anlam taşımaz; anlamı programcının seçtiği gösterim verir. Bu not, 8086 programlamaya geçmeden önce gereken sayı gösterimlerini toplar. Konumsal sayı sistemleriyle başlar, tabanlar arası dönüşümü hem elle hem 8086 komutlarıyla yapar. Ardından negatif sayıların ikiye tümleyen gösterimini, ikili ve onaltılık aritmetiği, elde ile işaretli taşma arasındaki farkı inceler. Son kesimler bitin donanımdaki karşılığını (gerilim düzeyleri, flip-flop), IEEE 754 kayan nokta biçimini, BCD'yi, bayt sıralamasını ve Gray kodunu anlatır.

ÖĞRENME ÇIKTILARI

Konumsal sistemleri okumak
Onluk, ikili ve onaltılık bir sayının değerini basamak ağırlıklarıyla hesaplamak; bit, nibble, bayt ve sözcük birimlerinin aralığını söylemek; assembly'deki `b`, `h`, `o` soneklerini doğru kullanmak.

Tabanlar arasında dönüştürmek
Onluk sayıyı bölme-kalan yöntemiyle, kesirli kısmı 2 ile çarpma yöntemiyle çevirmek; ikili ile onaltılık arasında dörtlü gruplamayla geçmek; aynı işi `DIV` ve `SHR` komutlarıyla yapan programları izlemek.

İşaretli sayıları kullanmak
Negatif sayıları ikiye tümleyen gösterimiyle yazmak; birler tümleyeni ve işaret-büyüklük gösterimiyle karşılaştırmak; 8 ve 16 bitlik işaretli aralıkları, işaret genişletmeyi ve sabit noktalı kesirleri açıklamak.

Aritmetik yapmak ve bayrakları yorumlamak
İkili ve onaltılık toplama ve çıkarma yapmak; çıkarmayı tümleyenle toplamaya çevirmek; bir işlemden sonra CF ve OF bayraklarının değerini gerekçesiyle bulmak.

Kodlamaları çözmek
32 bitlik bir IEEE 754 desenini çözmek ve kodlamak; BCD toplamayı düzeltmeyle yapmak; çok baytlı bir değerin little-endian bellek düzenini yazmak; Gray kodunu üretmek ve geri çevirmek.

Biti donanıma bağlamak
Gerilim düzeylerinin bitlere nasıl karşılık geldiğini, flip-flop türlerinin davranışını ve bir yazmacın D flip-flop'lardan nasıl kurulduğunu açıklamak.

1.2.1 Sayıları temsil etmek

Sayı soyut bir kavramdır. Beş elma, beş parmak ve beş gün aynı niceliği paylaşır; ama bu niceliği kâğıda, taşa ya da bir devreye yazmak için bir gösterim gerekir. Gösterim seçimi, sayıyla yapılacak işlerin ne kadar kolay olacağını belirler. Bu kesim, konumsal sistemlerin neden bugünkü biçimini aldığını iki eski gösterim üzerinden açıklar.

1.2.1.1 Sopadan simgeye

İnsanlar saymak için önce somut nesneler kullandı: çakıl taşları, düğümler, sopalar. Sopalarla sayan biri beşi `| | | | |` diye gösterir. Bu gösterimin bir üstünlüğü vardır: değer, simge sayısına eşittir ve hiçbir kural bilmeden okunur. Sakıncası da açıktır. Yüz için yüz çizgi gerekir. İki sayıyı karşılaştırmak için çizgileri tek tek saymak gerekir.

Roma İmparatorluğu'nda bu soruna kısmi bir çözüm bulundu. Küçük sayılar hâlâ çizgiyle yazıldı (`III` üçtür), ama belirli öbekler için ayrı simgeler kullanıldı: V beş sopayı, X on sopayı temsil etti. Bugün kullandığımız gösterim ise bambaşka bir fikre dayanır: bir rakamın değeri yalnız rakamın kendisine değil, sayının içindeki **konumuna** da bağlıdır. 754'teki 7 yedi değil, yedi yüzdür. Şekil 1.2.1 üç gösterimi yan yana koyar.



Şekil 1.2.1: Aynı fikrin üç gösterimi. Solda beş sopa, ortada Romen simgeleri (III = 3, V = 5, X = 10), sağda konumsal onluk gösterim: 754 = 7·100 + 5·10 + 4.

1.2.1.2 Romen rakamları

Romen sisteminde yedi simge vardır. Tablo 1.2.1 bunları değerleriyle verir. Sayı, simgelerin değerleri toplanarak ve gerektiğinde çıkarılarak okunur.

Simge	I	V	X	L	C	D	M
Değer	1	5	10	50	100	500	1000

Tablo 1.2.1: Romen simgeleri ve değerleri.

Tanım 1.2.1

Romen rakamları toplama ve çıkarma kuralıyla okunur. Büyük bir simgeden sonra gelen küçük simge eklenir: VI = 5 + 1 = 6. Büyük bir simgeden önce gelen küçük simge çıkarılır: IV = 5 − 1 = 4. Çıkarma yalnız altı çiftte kullanılır: IV (4), IX (9), XL (40), XC (90), CD (400) ve CM (900).

Okumanın güvenli yolu, sayıyı soldan sağa değer öbeklerine ayırmaktır. Her öbek ya tek başına toplanan simgelerden ya da altı çıkarma çiftinden biridir. Öbeklerin değerleri bulunur ve toplanır.

Örnek 1.2.1

MCMLXXXVI sayısını okuyalım. Soldan sağa öbekler M, CM, LXXX ve VI'dır. M = 1000'dir. CM bir çıkarma çiftidir: C, M'den önce geldiği için 1000 − 100 = 900 eder. LXXX = 50 + 10 + 10 + 10 = 80'dir. VI = 5 + 1 = 6'dır. Toplam 1000 + 900 + 80 + 6 = **1986** olur.

MMVII için öbekler M, M ve VII'dir: 1000 + 1000 + 7 = **2007**. Bir eksik I ile yazılan MMVI ise 2006'dır.



Şekil 1.2.2: MCMLXXXVI'nın öbeklere ayrılması: M = 1000, CM = 900, LXXX = 80, VI = 6; toplam 1986.

Romen sisteminin iki eksiği vardır ve ikisi de aritmetiği zorlaştırır. Birincisi, **sıfır** için bir simge yoktur. İkincisi, **basamak** kavramı yoktur: X her yerde ondur. Bu yüzden iki Romen sayısını alt alta yazıp sütun sütun toplamak mümkün olmaz. MCMLXXXVI ile MMVII'yi toplamak için önce ikisini de değerlerine çevirmek gerekir. Çarpma ve bölme daha da zahmetlidir. Konumsal sistem bu iki eksiği birlikte giderir.

Sık yapılan hata

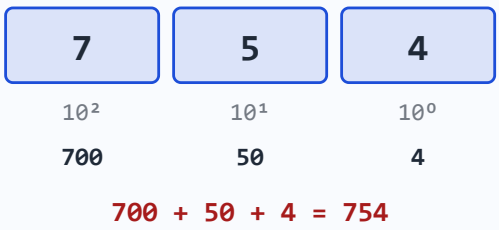
Çıkarma kuralını her küçük-büyük çiftine uygulamak. IC 99 değildir; 99, XCIX (90 + 9) diye yazılır. MCMXLIV okunurken XL'yi LX (60) gibi okumak da sık görülür: XL'de X, L'den önce geldiği için çıkarılır ve öbek 40 eder. Doğru okuma 1000 + 900 + 40 + 4 = 1944'tür.

1.2.2 Konumsal sayı sistemleri

Konumsal sistemde her basamağın bir **ağırlığı** vardır ve rakam bu ağırlıkla çarpılır. Ağırlıklar tabanın kuvvetleridir. Onluk sistemde ağırlıklar 1, 10, 100, 1000 diye gider; ikili sistemde 1, 2, 4, 8 diye. Bu kesim üç tabanı (10, 2, 16) aynı formülle ele alır, bilgisayar dünyasının veri birimlerini tanımlar ve assembly'de sayıların nasıl yazıldığını gösterir.

1.2.2.1 Onluk sistem ve genel formül

Gündelik hayatta kullandığımız sistem **onluk** (*decimal*) sistemdir. Tabanı 10'dur ve on rakamı vardır: 0, 1, 2, …, 9. 754 sayısında 4 birler basamağında, 5 onlar basamağında, 7 yüzler basamağındadır. Değer, her rakamın kendi basamak ağırlığıyla çarpımlarının toplamıdır: 7 × 100 + 5 × 10 + 4 × 1 = 754. Şekil 1.2.3 bu toplamı basamak basamak gösterir.



Şekil 1.2.3: 754'ün basamakları ve ağırlıkları. $7 \times 10^2 = 700$, $5 \times 10^1 = 50$, $4 \times 10^0 = 4$; toplam 754.

Tanım 1.2.2

Konumsal sayı sistemi (*positional number system*), rakamın değerinin bulunduğu basamağa göre belirlendiği sistemdir. b tabanında (*base, radix*) yazılmış n basamaklı $d_{n-1} \ldots d_1\, d_0$ sayısının değeri

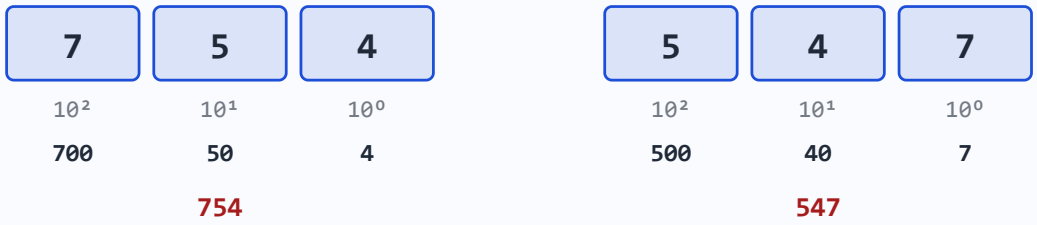
$$N = d_{n-1} \times b^{n-1} + \ldots + d_1 \times b^1 + d_0 \times b^0 = \Sigma_{i=0..n-1}\, d_i \times b^i$$

ifadesidir. Burada her d_i rakamı 0 ile $b - 1$ arasındadır; i basamağın konumudur ve sağdan 0'dan başlayarak sayılır.

Formül her taban için aynıdır; değişen yalnız b'dir. Bu yüzden ikili ve onaltılık sayılar için yeni bir kural öğrenmek gerekmez. Rakam kümesi tabana göre değişir: 10 tabanında on rakam, 2 tabanında iki rakam, 16 tabanında on altı rakam vardır. Bir rakamın tabana eşit ya da tabandan büyük olması mümkün değildir. İkili bir sayıda 2 rakamı, onluk bir sayıda “10” diye tek bir rakam yoktur.

1.2.2.2 Konumun önemi ve sıfır

754 ile 547 aynı üç rakamdan oluşur, ama değerleri çok farklıdır. 7 en sola konduğunda 700, en sağa konduğunda 7 değerini alır. Şekil 1.2.4 iki sayıyı yan yana gösterir. Romen sisteminde X her yerde on iken konumsal sistemde aynı rakam, konumuna göre 700, 70 ya da 7 değerini alır.



Şekil 1.2.4: Aynı rakamlar, farklı konumlar: 754 = 700 + 50 + 4, 547 = 500 + 40 + 7.

En sağdaki basamak her tabanda "birler" basamağıdır. Bunun nedeni, sıfırdan farklı her sayının sıfırıncı kuvvetinin 1 olmasıdır: $x^0 = 1$. Matematikte 0^0 belirsiz kabul edilir; sayı sistemlerinde ve polinomlarda ise uzlaşım olarak 1 alınır. Taban hiçbir zaman 0 olmadığı için bu ayrıntı pratikte bir sorun yaratmaz.

Konumsal sistemin işlemesi için **sıfır** rakamı zorunludur. 705 sayısında onlar basamağı boştur. Bu boşluk bir sıfırla doldurulmazsa 705 ile 75 ayırt edilemez. Sıfır burada "hiçbir şey yok" anlamında bir yer tutucudur. Romen sisteminde sıfıra gerek yoktu, çünkü simgelerin değeri konumdan bağımsızdı.

1.2.2.3 İkili sistem

Bilgisayarların onluk yerine ikili sistemi kullanmasının nedeni elektroniktir. Bir devre elemanını on farklı gerilim düzeyinden birinde tutmak ve bu düzeyleri gürültü altında birbirinden güvenle ayırmak zordur. İki durumlu bir eleman ise kolay yapılır. Bir transistör ya iletimdedir ya kesimdedir. Bir hat ya yüksek gerilimdedir ya düşük. Bir kondansatör ya yüklüdür ya boştur. Bu iki duruma 1 ve 0 adları verilir. Şekil 1.2.5 bir hattaki gerilimin zaman içinde 1 ve 0 değerlerini nasıl taşıdığını gösterir.



Şekil 1.2.5: İkili sinyal. Yüksek gerilim (5 V) 1'i, düşük gerilim (0 V) 0'ı temsil eder; çizilen dizi 1011 0010'dır.

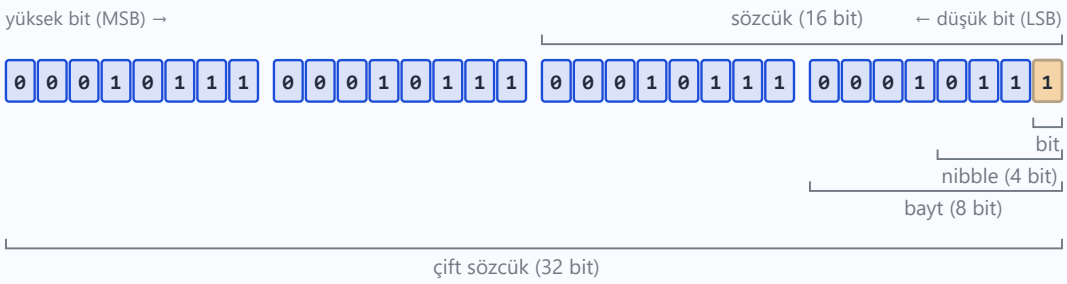
Tanım 1.2.3

İkili (*binary*) sistem, tabanı 2 olan konumsal sistemdir. Rakamları yalnız 0 ve 1'dir. Basamak ağırlıkları 2'nin kuvvetleridir: $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$, $2^5 = 32$, $2^6 = 64$, $2^7 = 128$, … İkili sayının her basamağına **bit** (*binary digit*) denir.

İkili sistem onluğa göre çok daha fazla basamak kullanır. 1000 sayısı onlukta dört basamaktır, ikilide on basamak gerektirir. Buna karşılık her basamağın yalnız iki değeri olduğu için aritmetik kuralları çok basittir. Toplama tablosu dört satırdan, çarpma tablosu da dört satırdan oluşur. Bu basitlik, bir toplayıcı devresini birkaç mantık kapısıyla kurmayı mümkün kılar.

1.2.2.4 Bit, nibble, bayt ve sözcük

Bitler tek tek değil, sabit genişlikte öbekler hâlinde işlenir. Her öbek genişliğinin bir adı vardır. Şekil 1.2.6 32 bitlik bir değeri bu öbeklere ayırır; Tablo 1.2.2 her birimin aralığını ve 8086'daki karşılığını verir.



Şekil 1.2.6: 32 bitlik bir değerin öbekleri: en sağdaki tek basamak bit, 4 bit nibble, 8 bit bayt, 16 bit sözcük, 32 bit çift sözcük. En soldaki bit en anlamlı bit (MSB), en sağdaki en az anlamlı bittir (LSB).

Birim	Bit	İşaretsiz aralık	8086'da
bit	1	0 … 1	bir bayrak (CF, ZF …)
nibble	4	0 … 15 (0h … Fh)	bir onaltılık rakam, bir BCD basamağı
bayt	8	0 … 255	AL, AH, BL …; `DB`
sözcük	16	0 … 65.535	AX, BX, CX, DX …; `DW`
çift sözcük	32	0 … 4.294.967.295	DX:AX çifti; `DD`

Tablo 1.2.2: Veri birimleri, işaretsiz aralıkları ve 8086'daki karşılıkları.

Tablonun üçüncü sütunu tek bir kuraldan çıkar: n bit ile 2^n farklı desen yazılabilir. Desenler 0'dan başlayarak sayıldığında en büyük değer $2^n - 1$ olur. 8 bit için $2^8 = 256$ desen vardır ve en büyük değer 255'tir. 16 bit için 65.536 desen vardır ve en büyük değer 65.535'tir. Bu sınırlar 8086 programlamada sürekli karşımıza çıkar. Bir bayta 256 yazılamaz; bir sözcüğe 70.000 yazılamaz.

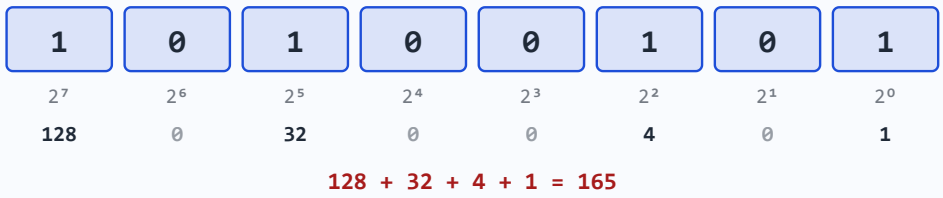
Tanım 1.2.4

Nibble 4 bitlik öbektir. **Bayt** (*byte*) 8 bitlik öbektir ve belleğin adreslenen en küçük birimidir. **Sözcük** (*word*) 8086'da 16 bitlik, **çift sözcük** (*double word*) 32 bitlik öbektir. Bir öbeğin en soldaki biti **en anlamlı bit** (*most significant bit, MSB*), en sağdaki biti **en az anlamlı bit** (*least significant bit, LSB*) adını alır.

Sözcük genişliği işlemciye bağlıdır. 8086 16 bitlik bir işlemcidir: genel amaçlı yazmaçları 16 bittir ve veri yolu bir yol döngüsünde 16 bit taşır. AX yazmacı iki bayttan oluşur. Yüksek baytı AH, düşük baytı AL'dir. AX = 1234h ise AH = 12h ve AL = 34h olur. 32 bitlik bir değer 8086'da tek bir yazmaca sığmaz; çarpma ve bölme komutları böyle değerleri DX:AX çiftinde tutar. Bu çiftte DX yüksek sözcüğü, AX düşük sözcüğü taşır.

1.2.2.5 İkili sayıları okumak

Assembly kaynak kodunda ikili sayılar `b` harfiyle biter: `101b` beştir. İkili bir sayının değerini bulmak için konumsal formül uygulanır. Rakamlar yalnız 0 ve 1 olduğu için çarpmaya gerek kalmaz. 0 olan basamaklar toplama hiçbir şey katmaz, 1 olan basamaklar ağırlıklarını katar. Kısa yol şudur: yalnız 1 olan basamakların ağırlıkları toplanır.



Şekil 1.2.7: 10100101b'nin değeri. 1 olan basamakların ağırlıkları 128, 32, 4 ve 1'dir; toplam 165.

8 bitlik ağırlıkları ezbere bilmek okumayı hızlandırır: soldan sağa 128, 64, 32, 16, 8, 4, 2, 1. Şekil 1.2.7 10100101b örneğini çözer: 128 + 32 + 4 + 1 = 165. Basamak sayısı 8'den az olan bir sayıda ağırlıklar yine sağdan başlar. Soldaki eksik basamaklar 0 kabul edilir.

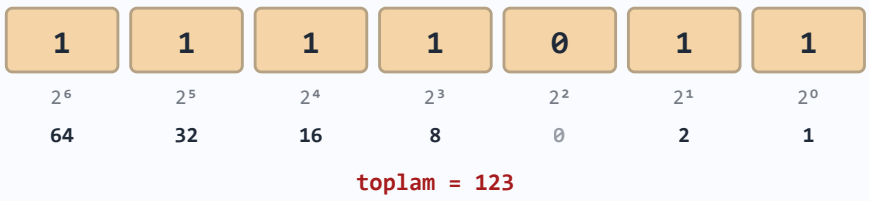
Örnek 1.2.2

1111011b'yi onluğa çevirelim. Sayının yedi basamağı vardır; ağırlıklar sağdan 2^0'dan 2^6'ya kadar gider. Basamakları sağdan sola tek tek ekleyip ara toplamı izleyelim:

1. Basamak 0: $1 \times 2^0 = 1$, ara toplam 1.
2. Basamak 1: $1 \times 2^1 = 2$, ara toplam 3.
3. Basamak 2: $0 \times 2^2 = 0$, ara toplam 3.
4. Basamak 3: $1 \times 2^3 = 8$, ara toplam 11.

5. Basamak 4: $1 \times 2^4 = 16$, ara toplam 27.
6. Basamak 5: $1 \times 2^5 = 32$, ara toplam 59.
7. Basamak 6: $1 \times 2^6 = 64$, ara toplam **123**.

Sonuç $(1111011)_2 = 1 + 2 + 0 + 8 + 16 + 32 + 64 = 123$'tür. Kısa yolla bakılırsa yedi bitin hepsi 1 olsaydı değer 127 olurdu; 2^2 basamağı 0 olduğu için 127 − 4 = 123 bulunur.



Şekil 1.2.8: 1111011b'nin basamak ağırlıkları ve son ara toplam: 64 + 32 + 16 + 8 + 0 + 2 + 1 = 123.

Örneğin son cümlesindeki hile genel bir kuraldır. *n* bitin hepsi 1 olan sayı $2^n - 1$'dir. Birkaç biti 0 olan bir sayının değeri, bu üst sınırdan 0 olan bitlerin ağırlıkları çıkarılarak bulunabilir. Çoğu biti 1 olan sayılarda bu yol daha hızlıdır.

1.2.2.6 Onaltılık sistem

İkili sayılar uzundur ve okunması yorucudur. 16 bitlik bir adresi 0 ve 1 dizisi olarak yazmak hem yer kaplar hem hata yapmayı kolaylaştırır. Onluk sistem kısa yazım sağlar, ama onluk ile ikili arasındaki dönüşüm bölme ve çarpma ister. **Onaltılık** sistem iki üstünlüğü birleştirir: kısadır ve ikiliye dönüşümü hesap gerektirmez.

Tanım 1.2.5

Onaltılık (*hexadecimal*) sistem, tabanı 16 olan konumsal sistemdir. On altı rakamı vardır: 0–9 ve 10–15 değerleri için A, B, C, D, E, F harfleri. Basamak ağırlıkları 16'nın kuvvetleridir: 1, 16, 256, 4096, 65.536, …

$16 = 2^4$ olduğu için her onaltılık rakam tam olarak dört bite, yani bir nibble'a karşılık gelir. Tablo 1.2.3 on altı nibble'ın onluk ve onaltılık karşılıklarını verir. Bu tablo, sayı sistemleriyle çalışırken ezbere bilinmesi gereken tek tablodur. Bir bayt iki onaltılık rakamla, bir sözcük dört onaltılık rakamla yazılır: 1010 0101b = A5h.

Onluk	İkili	Onaltılık	Onluk	İkili	Onaltılık
0	`0000`	0	8	`1000`	8
1	`0001`	1	9	`1001`	9
2	`0010`	2	10	`1010`	A

Onluk	İkili	Onaltılık	Onluk	İkili	Onaltılık
3	`0011`	3	11	`1011`	B
4	`0100`	4	12	`1100`	C
5	`0101`	5	13	`1101`	D
6	`0110`	6	14	`1110`	E
7	`0111`	7	15	`1111`	F

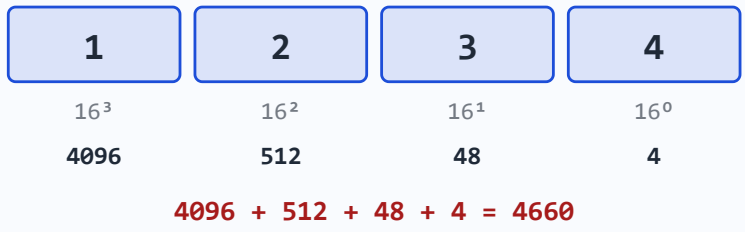
Tablo 1.2.3: On altı nibble ve onaltılık rakamlar.

Assembly'de onaltılık sayılar `h` harfiyle biter: `5Fh` = 5 × 16 + 15 = 95. Harfle başlayan onaltılık sayıların önüne bir **0** eklenir: `0E120h`. Bu sıfır sayının değerini değiştirmez, ama assembler'a bunun bir sayı olduğunu söyler. Sıfır konmazsa assembler `E120h`'yi bir değişkenin ya da etiketin adı sanır. Tablo 1.2.4 bu kuralın sonuçlarını özetler.

Yazım	Anlamı
`MOV AL, 0FFh`	AL = 255; doğru yazım
`MOV AL, FFh`	hata: `FFh` tanımsız bir ad olarak okunur
`MOV AX, 10`	onluk 10 (= 0Ah); sonek yoksa sayı onluktur

Tablo 1.2.4: Onaltılık sabitlerin yazımı.

Onaltılık bir sayının onluk değeri yine konumsal formülle bulunur. `1234h` için ağırlıklar 4096, 256, 16 ve 1'dir: 1 × 4096 + 2 × 256 + 3 × 16 + 4 × 1 = 4096 + 512 + 48 + 4 = 4660. Şekil 1.2.9 bu hesabı gösterir. Harfli rakamlar önce onluk değerlerine çevrilir, sonra çarpılır.



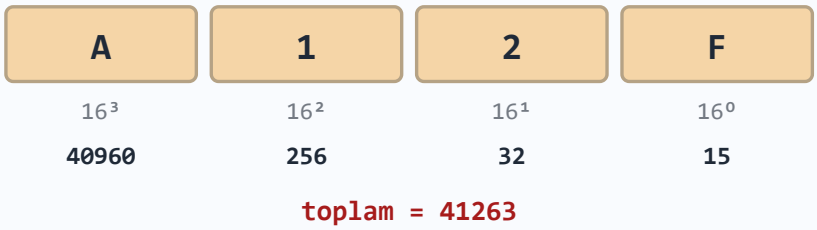
Şekil 1.2.9: 1234h = $1\cdot16^3 + 2\cdot16^2 + 3\cdot16^1 + 4\cdot16^0$ = 4096 + 512 + 48 + 4 = 4660.

Örnek 1.2.3

A12Fh'yi onluğa çevirelim. Önce harfler çözülür: A = 10, F = 15. Ağırlıklar 16^0 = 1, 16^1 = 16, 16^2 = 256 ve 16^3 = 4096'dır. Sağdan başlayarak:

1. F × 16^0 = 15 × 1 = 15; ara toplam 15.
2. 2 × 16^1 = 32; ara toplam 47.
3. 1 × 16^2 = 256; ara toplam 303.
4. A × 16^3 = 10 × 4096 = 40.960; ara toplam **41.263**.

Sonuç $(A12F)_{16}$ = 41.263'tür. Değer 65.535'ten küçük olduğu için 16 bitlik bir yazmaca sığar: `MOV AX, 0A12Fh` geçerli bir komuttur.



Şekil 1.2.10: A12Fh'nin basamak katkıları: 40.960 + 256 + 32 + 15 = 41.263.

1.2.2.7 İkili ile onaltılık arasında dörtlü gruplama

Her onaltılık rakam dört bite karşılık geldiği için ikili ile onaltılık arasındaki dönüşüm, bir sözlükten kelime bakmak kadar kolaydır. İkili sayı sağdan başlayarak dörtlü öbeklere ayrılır. En soldaki öbek dört bitten kısaysa soluna 0 eklenir. Her öbek, Tablo 1.2.3 kullanılarak tek bir onaltılık rakama çevrilir.

Örnek 1.2.4

1010 1011 0101 1100b'yi onaltılığa çevirelim. Sayı zaten dörtlü öbeklere ayrılmıştır. 1010 = 8 + 2 = 10, yani A'dır. 1011 = 8 + 2 + 1 = 11, yani B'dir. 0101 = 4 + 1 = 5'tir. 1100 = 8 + 4 = 12, yani C'dir. Sonuç AB5Ch'dir. A ile başladığı için assembly'de `0AB5Ch` diye yazılır.

Ters yön de aynı tabloyla yapılır: her onaltılık rakam dört bite açılır. `3Eh` → 0011 1110b. Bit sayısı dördün katı olmayan bir örnek: 1101011b yedi bittir. Sağdan ayrılınca öbekler 110 ve 1011 olur. Soldaki öbeğe bir 0 eklenir: 0110 1011 → 6Bh.



Şekil 1.2.11: 16 bitlik ikili sayının dörtlü öbekleri ve onaltılık rakamları: 1010 → A, 1011 → B, 0101 → 5, 1100 → C.

Programcıların ikili veriyi neredeyse her zaman onaltılık yazmasının nedeni budur. Bir bayt tam iki onaltılık rakamdır, bir sözcük tam dört. 8086 hata ayıklayıcıları ve emülatörleri yazmaç ve bellek içeriğini onaltılık gösterir. Bir yazmacın değeri onaltılık okunduğunda hangi bitlerin 1 olduğu bir bakışta görülür. Örneğin AL = 80h ise yalnız bit 7 birdir; AL = 0Fh ise düşük nibble'ın dört biti birdir.

Aynı yöntem **sekizli** (*octal*) sistem için de geçerlidir. $8 = 2^3$ olduğu için her sekizli rakam üç bite karşılık gelir ve ikili sayı üçlü öbeklere ayrılır. 0100 0001b üçlü öbeklerle 001 000 001 diye yazılır; bu da 101o eder. Sekizli sistem eski bilgisayarlarda yaygındı; 8086 programlamada nadiren kullanılır.

> **Sık yapılan hata**
>
> Dörtlü öbekleri soldan başlayarak ayırmak. 1101011b soldan ayrılırsa 1101 ve 011 öbekleri çıkar; sağdaki öbeğe sağdan 0 eklenirse 1101 0110 = D6h bulunur. Bu yanlıştır: sağa eklenen sıfır sayıyı iki katına çıkarır. Öbekler her zaman sağdan, yani birler basamağından başlar; eksik bitler yalnız sola eklenir.

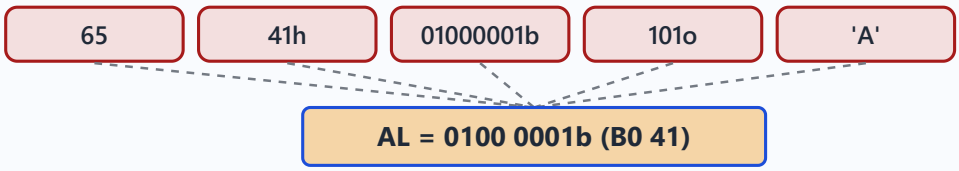
1.2.2.8 Assembly'de sayı gösterimleri

emu8086 assembler'ı sabitleri dört tabanda ve karakter olarak kabul eder. Tabanı sayının sonundaki ek, yani **sonek** (*suffix*) belirler. Tablo 1.2.5 AL'ye 65 yükleyen beş farklı yazımı ve bir negatif sabiti listeler.

Yazım	Taban	AL'ye giden değer
`MOV AL, 65`	onluk (sonek yok ya da `d`)	65 = 41h
`MOV AL, 41h`	onaltılık	65 = 41h
`MOV AL, 01000001b`	ikili	65 = 41h
`MOV AL, 101o`	sekizli (`o` ya da `q`)	65 = 41h
`MOV AL, 'A'`	ASCII karakter	65 = 41h
`MOV AL, -1`	onluk, işaretli	FFh

Tablo 1.2.5: Aynı değerin farklı yazımları.

İlk beş satırın hepsi aynı makine kodunu üretir: `B0 41`. Burada B0h, "AL'ye sabit bir bayt yükle" komutunun işlem kodudur; 41h ise sabitin kendisidir. Assembler kaynak koddaki sayıyı hangi tabanda bulursa bulsun ikiliye çevirir ve komutun içine gömer. İşlemci kaynak kodu hiç görmez; yalnız `1011 0000 0100 0001` bitlerini görür. Taban, yalnız insanlar için bir yazım kolaylığıdır.



Şekil 1.2.12: Beş farklı yazım aynı bayta dönüşür: 0100 0001b = 41h. Komutun makine kodu her durumda B0 41'dir.

Son satır ayrıca ilgi çekicidir: `MOV AL, -1` AL'ye FFh yükler. Assembler negatif sabiti ikiye tümleyen gösterimine çevirir: −1 için 256 − 1 = 255 = FFh. AL'ye giden FFh bit deseni, `MOV AL, 255` 'in yüklediği desenle aynıdır. İşlemci için ikisi arasında hiçbir fark yoktur.

Sık yapılan hata

Soneki unutmak. `MOV AL, 41` AL'ye 41h değil, onluk 41'i, yani 29h'yi yükler. Bellek dökümünden ya da bir tablodan onaltılık bir değer kopyalanırken `h` eki yazılmazsa program hata vermeden yanlış değerle çalışır. Assembler bunu fark edemez, çünkü 41 geçerli bir onluk sayıdır.

1.2.3 Taban dönüşümleri

Bir sayıyı başka bir tabandan onluğa çevirmek konumsal formülle yapılır: basamaklar ağırlıklarıyla çarpılıp toplanır. Ters yön, yani onluktan başka bir tabana geçiş, ayrı bir yöntem ister. Bu kesim tam sayı kısmı için bölme-kalan yöntemini, kesirli kısım için çarpma yöntemini anlatır. Her iki yöntemin neden çalıştığını gösterir ve aynı işi yapan iki küçük 8086 programını adım adım izler.

1.2.3.1 Bölme-kalan yöntemi

Tanım 1.2.6

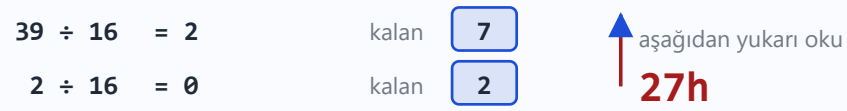
Bölme-kalan yöntemi, onluk bir tam sayıyı b tabanına çevirir. Sayı b'ye bölünür; bölüm saklanır, kalan bir basamak olarak yazılır. Bölüm 0 olana kadar bölme bölüm üzerinde sürdürülür. Kalanlar sondan başa (aşağıdan yukarı) okununca sayının b tabanındaki gösterimi elde edilir: ilk kalan en düşük basamaktır.

Yöntemin neden çalıştığı konumsal formülden görülür. $N = d_k b^k + \ldots + d_1 b + d_0$ yazalım. d_0 dışındaki bütün terimler b'nin katıdır. Bu yüzden N, b'ye bölündüğünde kalan tam olarak d_0'dır ve bölüm $d_k b^{k-1} + \ldots + d_1$ olur. Bölüm, sayının en sağdaki basamağı silinmiş hâlidir. Onu yeniden b'ye bölmek d_1'i verir. Her bölme bir basamak üretir ve sayıyı bir basamak kısaltır. Kalan her zaman 0 ile $b - 1$ arasında olduğu için her kalan geçerli bir rakamdır.

Onluk sistemde aynı yöntem kendiliğinden bilinir. 754'ü 10'a bölünce bölüm 75, kalan 4'tür; kalan son basamaktır. 75'i 10'a bölünce kalan 5, bölüm 7 olur. Yöntem, bu tanıdık işlemin başka bir tabanla yapılmasından ibarettir.

Örnek 1.2.5

39'u onaltılığa çevirelim. 39 ÷ 16 = 2, kalan 7'dir; 7 birler basamağıdır. 2 ÷ 16 = 0, kalan 2'dir. Bölüm 0 olduğu için durulur. Kalanlar aşağıdan yukarı okunur: **27h**. Sağlama: 2 × 16 + 7 = 39.



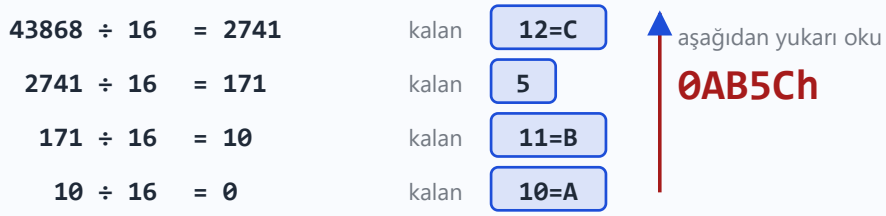
Şekil 1.2.13: 39'un 16'ya art arda bölünmesi. Kalanlar 7 ve 2'dir; aşağıdan yukarı okununca 27h.

Örnek 1.2.6

Daha büyük bir sayı: 43.868'i onaltılığa çevirelim.

1. 43.868 ÷ 16 = 2741, kalan 12 → C.
2. 2741 ÷ 16 = 171, kalan 5.
3. 171 ÷ 16 = 10, kalan 11 → B.
4. 10 ÷ 16 = 0, kalan 10 → A. Bölüm 0: dur.

Kalanlar aşağıdan yukarı okununca AB5Ch bulunur. A ile başladığı için assembly'de **0AB5Ch** yazılır. Sağlama: 10 × 4096 + 11 × 256 + 5 × 16 + 12 = 40.960 + 2816 + 80 + 12 = 43.868. Bu sayı, dörtlü gruplama örneğindeki 1010 1011 0101 1100b ile aynıdır.



Şekil 1.2.14: 43.868'in bölme merdiveni. Kalanlar C, 5, B, A sırasıyla çıkar; aşağıdan yukarı okununca 0AB5Ch.

Her bölme adımında kalanı hesaplamak için bölüm ile bölenin çarpımı sayıdan çıkarılır: 43.868 − 2741 × 16 = 43.868 − 43.856 = 12. Hesap makinesiyle çalışırken ondalık sonuç da kullanılabilir. 43.868 / 16 = 2741,75 çıkar. Tam kısım bölümdür, kesirli kısım 16 ile çarpılınca kalanı verir: 0,75 × 16 = 12.

1.2.3.2 Kod izleme: DIV ile onaltılık basamaklar

Bölme-kalan yöntemi bir döngüdür ve doğrudan 8086 koduna dönüşür. 8086'nın `DIV` komutu bölümü ve kalanı tek komutta verir. Aşağıdaki program 2989'u 16'ya art arda böler ve kalanları `kalan` adlı bir diziye yazar.

```
ORG 100h
MOV AX, 2989   ; sayı
MOV BX, 16     ; taban
MOV SI, 0      ; sayaç
bol: MOV DX, 0 ; DX:AX
DIV BX         ; bölüm, kalan
MOV kalan[SI], DL
INC SI
CMP AX, 0      ; bölüm 0 mı?
JNE bol
RET
kalan DB 4 DUP(0)
```

Program satır satır şunu yapar. `ORG 100h` bir komut değil, assembler'a verilen bir yönergedir (*directive*). Programın .COM dosyası olarak kesimin 100h bağıl konumundan başlayacağını bildirir. `MOV AX, 2989` bölünecek sayıyı AX'e koyar; 2989 onaltılık olarak 0BADh'dir. `MOV BX, 16` böleni, yani hedef tabanı BX'e koyar. BX'e 2 ya da 8 yazılsaydı aynı kod ikili ya da sekizli basamakları üretirdi. `MOV SI, 0` sıradaki kalanın dizideki yerini tutan indisi sıfırlar.

`bol` etiketli satır döngünün başıdır ve DX'i sıfırlar. Bunun nedeni `DIV`'in çalışma biçimidir. İşleneni 16 bitlik bir yazmaç olan `DIV BX`, 32 bitlik DX:AX çiftini BX'e böler. Bölümü AX'e, kalanı DX'e yazar. DX'te önceki turdan kalan bir değer bulunursa bölünen 65.536 × DX + AX olur ve sonuç tamamen yanlış çıkar. Bölüm 16 bite sığmazsa işlemci bir bölme hatası kesmesi de üretir. Bu yüzden her bölmeden önce DX = 0 yapılır.

`MOV kalan[SI], DL` kalanın düşük baytını dizinin SI. elemanına yazar. Kalan 16'dan küçük olduğu için DL'ye sığar ve DH her zaman 0'dır. `INC SI` indisi bir artırır. `CMP AX, 0` AX'ten 0'ı çıkarır, sonucu bir yere yazmaz ama bayrakları kurar: bölüm 0 ise ZF = 1, değilse ZF = 0 olur. `JNE bol` ("eşit değilse atla") ZF = 0 iken döngünün başına döner. Bölüm 0 olunca atlama yapılmaz ve `RET` programı bitirir. Son satır, dört baytlık ve başlangıçta sıfır olan `kalan` dizisini tanımlar.

Tablo 1.2.6 programın her önemli adımdan sonraki durumunu verir. Yazmaç değerleri onaltılık yazılmıştır.

Adım	AX	DX	SI	`kalan` (+0 … +3)	Açıklama
ilk üç MOV	0BADh	0000h	0	00 00 00 00	BX = 0010h
tur 1: DIV BX	00BAh	000Dh	0	00 00 00 00	2989 ÷ 16 = 186, kalan 13 (D)
MOV, INC, CMP	00BAh	000Dh	1	0D 00 00 00	AX ≠ 0 → ZF = 0, JNE atlar
tur 2: DIV BX	000Bh	000Ah	1	0D 00 00 00	186 ÷ 16 = 11, kalan 10 (A)
MOV, INC, CMP	000Bh	000Ah	2	0D 0A 00 00	AX ≠ 0 → JNE atlar
tur 3: DIV BX	0000h	000Bh	2	0D 0A 00 00	11 ÷ 16 = 0, kalan 11 (B)
MOV, INC, CMP	0000h	000Bh	3	0D 0A 0B 00	AX = 0 → ZF = 1, JNE atlamaz
RET	0000h	000Bh	3	0D 0A 0B 00	üç basamak bulundu

Tablo 1.2.6: DIV programının izlemesi (2989 = 0BADh, taban 16).

Döngü üç kez döner, çünkü 2989 onaltılık olarak üç basamaklıdır. SI'nin son değeri basamak sayısını verir. Bellekte kalanlar en düşük basamaktan başlayarak 0D 0A 0B sırasıyla durur. Sondan başa okununca B, A, D, yani **0BADh** elde edilir. Sağlama: 11 × 256 + 10 × 16 + 13 = 2816 + 160 + 13 = 2989. Şekil 1.2.15 son durumu gösterir.



Şekil 1.2.15: Döngü bittikten sonra yazmaçlar ve `kalan` dizisi. Üç bölmenin kalanları D, A ve B; sondan başa okununca 0BADh.

Programda dizinin boyutu dört bayttır. 16 bitlik bir sayı onaltılık olarak en çok dört basamaklıdır (FFFFh), bu yüzden dört bayt her durumda yeter. Taban 2 yapılsaydı 16 basamak gerekirdi ve dizinin `16 DUP(0)` diye tanımlanması gerekirdi. Kalanları ekrana yazdırmak için her biri bir ASCII karaktere çevrilmelidir: 0–9 için 30h, A–F için 37h eklenir (0Dh + 37h = 44h = 'D').

Programın hızı hakkında bir not düşmek gerekir. 8086'da 16 bitlik `DIV` en yavaş komutlardandır; yazmaç işlenenle yaklaşık 144–162 saat turu sürer. Aynı süre içinde onlarca `MOV` ya da `ADD` çalışabilir. Bölen 2'nin bir kuvveti olduğunda bölmeye hiç gerek yoktur: kaydırma komutları aynı işi çok daha hızlı yapar. 2'ye bölme bunun en sık görülen örneğidir.

1.2.3.3 İkili tabana bölme

Bölme-kalan yöntemi her tabanda çalışır. İkili taban için bölen 2'dir ve kalan her zaman 0 ya da 1'dir. Tek sayıların kalanı 1, çift sayıların kalanı 0'dır.

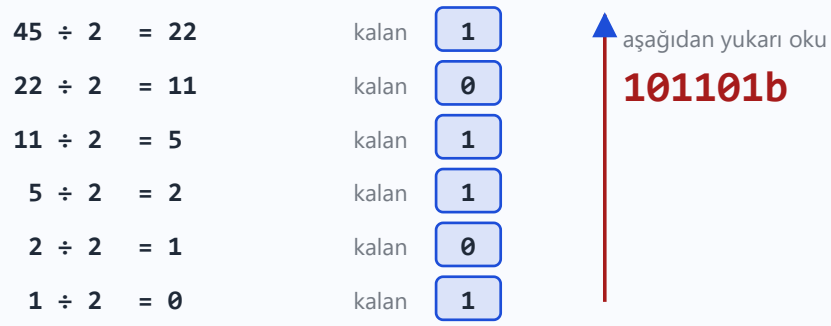
Örnek 1.2.7

45'i ikiliye çevirelim.

1. 45 ÷ 2 = 22, kalan 1.
2. 22 ÷ 2 = 11, kalan 0.
3. 11 ÷ 2 = 5, kalan 1.
4. 5 ÷ 2 = 2, kalan 1.
5. 2 ÷ 2 = 1, kalan 0.

6. 1 ÷ 2 = 0, kalan 1. Bölüm 0: dur.

Kalanlar aşağıdan yukarı okununca **101101b** bulunur. Sağlama: 32 + 8 + 4 + 1 = 45. 8 bitlik bir yazmaçta bu sayı 0010 1101b = 2Dh olarak durur.



Şekil 1.2.16: 45'in 2'ye art arda bölünmesi. Altı kalan aşağıdan yukarı okununca 101101b.

Büyük sayılarda 2'ye bölme uzun sürer: 43.868 gibi bir sayı on altı adım ister. Daha hızlı yol, sayıyı önce onaltılığa çevirmek ve sonra her rakamı dört bite açmaktır. 45 ÷ 16 = 2, kalan 13 (D) olduğu için 45 = 2Dh'dir. 2 = 0010 ve D = 1101 açılınca 0010 1101b elde edilir. Bu yol iki bölme ve bir tablo bakışıyla biter.

1.2.3.4 Kod izleme: kalanlar CF'den gelir

Bir ikili sayıyı bir bit sağa kaydırmak, onu 2'ye bölmektir. Onluk bir sayıda son basamağı silmek 10'a bölmek olduğu gibi, ikili bir sayıda son biti atmak da 2'ye bölmektir. Dışarı düşen bit ise bölmenin kalanıdır. 8086'nın `SHR` (*shift right*, sağa mantıksal kaydırma) komutu bu düşen biti elde bayrağına (CF) yazar. Aşağıdaki program bu gözlemi kullanır: `DIV` olmadan 45'in ikili basamaklarını sırayla CF'ye düşürür ve BL'de toplar.

```
ORG 100h
MOV AL, 45      ; bölünen
MOV CX, 8       ; 8 bit
bit: SHR AL, 1 ; kalan → CF
RCR BL, 1       ; CF → BL
LOOP bit
RET
```

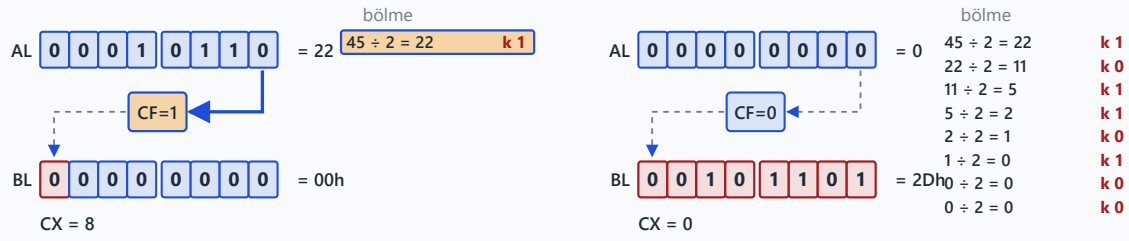
`MOV AL, 45` bölüneni AL'ye koyar: AL = 0010 1101b = 2Dh. `MOV CX, 8` döngü sayacını kurar; AL'nin sekiz bitinin her biri için bir tur dönülecektir. `SHR AL, 1` AL'nin bütün bitlerini bir sağa kaydırır. En soldaki bite 0 girer, en sağdaki bit CF'ye düşer. Sonuçta AL'de bölüm, CF'de kalan bulunur.

`RCR BL, 1` (*rotate through carry right*, elde üzerinden sağa döndürme) BL'yi CF ile birlikte 9 bitlik bir halka gibi döndürür. CF'deki bit BL'nin bit 7'sine girer, BL'nin bit 0'ı da CF'ye çıkar. Böylece SHR'nin ürettiği kalan BL'nin en soluna yerleşir. Her turda BL'deki eski kalanlar bir sağa kayar ve yenisi en sola girer. `LOOP bit` CX'i bir azaltır; CX sıfır değilse `bit` etiketine döner. Bu komut, "CX'i azalt, sıfır değilse atla" işini tek komutta yapar ve bayrakları değiştirmez.

Tur	Bölme	SHR kalanı	AL	BL (ikili)	BL	CX
başta	—	—	2Dh	`0000 0000`	00h	8
1	45 ÷ 2 = 22	1	16h	`1000 0000`	80h	7
2	22 ÷ 2 = 11	0	0Bh	`0100 0000`	40h	6
3	11 ÷ 2 = 5	1	05h	`1010 0000`	A0h	5
4	5 ÷ 2 = 2	1	02h	`1101 0000`	D0h	4
5	2 ÷ 2 = 1	0	01h	`0110 1000`	68h	3
6	1 ÷ 2 = 0	1	00h	`1011 0100`	B4h	2
7	0 ÷ 2 = 0	0	00h	`0101 1010`	5Ah	1
8	0 ÷ 2 = 0	0	00h	`0010 1101`	2Dh	0

Tablo 1.2.7: SHR/RCR programının izlemesi. Her satır o turun `LOOP` komutundan sonraki durumdur; "SHR kalanı" o turda CF'ye düşen bittir.

Tablo 1.2.7 sekiz turu verir. Program BL'yi sıfırlamaz; tabloda BL'nin başlangıçta 0 olduğu varsayılmıştır. Sonuç bu varsayıma bağlı değildir: sekiz turda BL'nin eski bitlerinin hepsi CF'ye çıkar ve bir sonraki SHR'nin kalanıyla ezilir. Her turda RCR, BL'nin bit 0'ını CF'ye çıkarır. Bu bit sekizinci tura kadar hep 0'dır, bu yüzden `LOOP` anında CF her turda 0 görünür. CF = 1 yalnız SHR ile RCR arasındaki kısa anda görülür: 1, 3, 4 ve 6. turlarda. İlk kalan sekiz turda sekiz kez sağa kayarak BL'nin bit 0'ına ulaşır. Son kalan ise BL'nin bit 7'sine yerleşir. Program bitince BL = 0010 1101b = 2Dh = 45'tir.



Şekil 1.2.17: Solda ilk `SHR AL, 1` : 45 ÷ 2 = 22 (AL = 16h), düşen bit CF = 1. Sağda döngü bitince: AL = 0, BL = 0010 1101b = 2Dh.

Sonuç ilk bakışta şaşırtıcı görünebilir: program 45'i alıp yine 45 üretmiştir. Aslında öğretici olan da budur. AL zaten ikili bir sayı tutar; onu "ikiliye çevirmek" için bitleri yeniden yazmaya gerek yoktur. Program, 2'ye bölme kalanlarının sayının bitlerinin ta kendisi olduğunu gösterir. Kalanlar sondan

başa dizilince sayının ikili gösterimi ortaya çıkar. Aynı fikir, bir sayının bitlerini tek tek işlemek için kullanılır: SHR ile düşürülen bit CF'den okunur ve `JC` ya da `JNC` ile karar verilir.

Sık yapılan hata

RCR yerine ROR kullanmak. `ROR BL, 1` BL'yi CF'den bağımsız döndürür: bit 0 bit 7'ye geçer ve CF'ye yalnız kopyalanır. SHR'nin CF'ye düşürdüğü kalan BL'ye hiç girmez; program BL'nin başlangıç değerini döndürmekle kalır. Kalanın taşınması için dönmenin elde üzerinden (RCR) ya da SHR'den sonra CF'yi okuyan ayrı bir komutla yapılması gerekir.

1.2.3.5 Kesirli kısım: 2 ile çarpma

Konumsal formül virgülün sağına da uzanır. Virgülden sonraki basamakların ağırlıkları tabanın negatif kuvvetleridir. Onlukta $0{,}375 = 3 \times 10^{-1} + 7 \times 10^{-2} + 5 \times 10^{-3}$'tür. İkilide ise virgülden sonraki ağırlıklar $2^{-1} = 0{,}5$, $2^{-2} = 0{,}25$, $2^{-3} = 0{,}125$, $2^{-4} = 0{,}0625$ diye gider. $0{,}101_b = 0{,}5 + 0{,}125 = 0{,}625$'tir.

Tanım 1.2.7

Çarpma yöntemi, 0 ile 1 arasındaki onluk bir kesri ikiliye çevirir. Kesir 2 ile çarpılır; çarpımın tam kısmı (0 ya da 1) sıradaki kesir biti olur. Çarpımın kesirli kısmıyla devam edilir. Kesir 0 olunca ya da istenen bit sayısına ulaşınca durulur. Bitler **yukarıdan aşağı**, yani üretildikleri sırayla okunur: ilk bit 2^{-1} basamağıdır.

Yöntemin gerekçesi bölme yönteminin aynasıdır. $x = 0{,}f_1f_2f_3\ldots$ (ikili) yazalım. $2x$, virgülü bir sağa kaydırır: $f_1{,}f_2f_3\ldots$. Çarpımın tam kısmı f_1'dir, kesirli kısmı ise kalan bitlerden oluşan sayıdır. Her çarpma bir biti virgülün soluna taşır. Bölmede ilk kalan en düşük basamaktı; çarpmada ilk tam kısım en yüksek kesir basamağıdır. Okuma yönünün tersine dönmesi bundan kaynaklanır.

Örnek 1.2.8

$0{,}625$'i ikiliye çevirelim.

1. $0{,}625 \times 2 = 1{,}25$: tam kısım **1** ($2^{-1}$ biti); kesir $0{,}25$ ile devam.
2. $0{,}25 \times 2 = 0{,}5$: tam kısım **0** (2^{-2} biti); kesir $0{,}5$.
3. $0{,}5 \times 2 = 1{,}0$: tam kısım **1** (2^{-3} biti); kesir 0, dur.

Bitler yukarıdan aşağı okununca **$0{,}101_b$** bulunur. Sağlama: $1/2 + 1/8 = 0{,}5 + 0{,}125 = 0{,}625$.

Hem tam hem kesirli kısmı olan bir sayıda iki kısım ayrı ayrı çevrilir ve birleştirilir. $25{,}625$ için $25 = 11001_b$ (bölme yöntemiyle) ve $0{,}625 = 0{,}101_b$ olduğundan $25{,}625 =$ **$11001{,}101_b$** olur.



Şekil 1.2.18: 0,625'in 2 ile art arda çarpılması. Tam kısımlar 1, 0, 1; yukarıdan aşağı okununca 0,101b.

Kesirli ikili sayılar onaltılığa da dörtlü gruplamayla çevrilir. Tek fark, öbeklerin virgülden başlayarak iki yana ayrılmasıdır. Tam kısım virgülden sola doğru, kesirli kısım virgülden sağa doğru öbeklenir. Kesirli kısımda eksik bitler sağa 0 olarak eklenir; virgülün sağına eklenen sıfır değeri değiştirmez. 1101,011b için öbekler 1101 ve 0110'dır, sonuç D,6h olur.

1.2.3.6 Tekrar eden kesirler: 0,1

Her onluk kesir ikilide sonlu yazılamaz. 1/3 onlukta 0,333… diye sonsuza dek sürer; benzer biçimde 0,1 ikilide sonsuza dek sürer. Çarpma yöntemi bunu açıkça gösterir.

Örnek 1.2.9

0,1'i ikiliye çevirelim.

1. 0,1 × 2 = 0,2 → bit 0; kesir 0,2.
2. 0,2 × 2 = 0,4 → bit 0; kesir 0,4.
3. 0,4 × 2 = 0,8 → bit 0; kesir 0,8.
4. 0,8 × 2 = 1,6 → bit 1; kesir 0,6.
5. 0,6 × 2 = 1,2 → bit 1; kesir **0,2**.
6. 0,2 × 2 = 0,4 → bit 0; ikinci adım tekrar ediyor.

Beşinci adımdan sonra kesir 0,2'ye döndü. 0,2 ikinci adımda görülmüştü; buradan sonra 2.–5. adımlar sonsuza dek tekrar eder ve 0011 bit deseni yinelenir. Sonuç 0,1 = 0,0 0011 0011 0011…b'dir. 8 kesir biti ile saklanırsa 0,0001 1001b = 0,097 656 25 elde edilir.



Şekil 1.2.19: 0,1'in ikiliye çevrilmesi. Kesir 0,2'ye geri dönünce 0011 deseni sonsuza dek tekrar eder.

Hangi kesirlerin sonlu yazılabileceğini tek bir kural belirler. Bir kesir en sade hâlinde p/q ise ikilide sonlu yazılması için q'nun 2'nin bir kuvveti olması gerekir. 0,625 = 5/8'dir ve $8 = 2^3$ olduğu için üç bitle tam yazılır. 0,1 = 1/10'dur ve 10'un 5 çarpanı vardır; bu yüzden sonlu değildir. Onluk sistemde de aynı kural geçerlidir: paydası yalnız 2 ve 5 çarpanları içeren kesirler sonludur, 1/3 sonsuzdur.

Bu kuralın pratikteki sonucu önemlidir. Bilgisayar sınırlı sayıda bit kullandığı için 0,1 gibi değerleri ancak yaklaşık saklayabilir; fazla bitler kesilir ya da yuvarlanır. IEEE 754 kayan nokta biçiminde de 0,1 tam değil, yaklaşık olarak saklanır. Pek çok programlama dilinde 0,1 + 0,2 işleminin 0,3'e tam eşit çıkmamasının nedeni budur. Para gibi onluk kesirlerin tam tutulması gereken uygulamalarda BCD ya da kuruş cinsinden tam sayı kullanılır.

Sık yapılan hata

Kesir bitlerini bölme yöntemindeki gibi aşağıdan yukarı okumak. 0,625 için bitler 1, 0, 1 sırasıyla çıkar ve simetrik olduğu için yanlış okuma fark edilmez. 0,375 için ise bitler 0, 1, 1'dir: doğru sonuç 0,011b = 0,375'tir; ters okuma 0,110b = 0,75 verir. Bölmede kalanlar ters, çarpmada tam kısımlar düz okunur.

1.2.3.7 Dönüşümleri birlikte kullanmak

Sunumdaki taban dönüştürücü, girilen bir sayıyı dört tabanda birden gösterir ve seçilen tabana bölme adımlarını listeler. Aynı işi elle yaparken izlenecek yol, kaynağa ve hedefe göre değişir. Tablo 1.2.8 en kısa yolları özetler.

Kaynak → hedef	Yöntem
ikili/onaltılık/sekizli → onluk	konumsal formül: basamak × ağırlık toplamı
onluk → onaltılık	16'ya bölme-kalan; kesir için 16 ile çarpma
onluk → ikili	önce onaltılığa, sonra her rakamı 4 bite açma; ya da 2'ye bölme
ikili ↔ onaltılık	sağdan dörtlü öbekler; kesirde virgülden iki yana
ikili ↔ sekizli	sağdan üçlü öbekler

Kaynak → hedef	Yöntem
onaltılık ↔ sekizli	ikili üzerinden: önce 4 bite aç, sonra 3'lü öbekle

Tablo 1.2.8: Tabanlar arası dönüşümde kısa yollar.

Örnek 1.2.10

2024'ü onaltılık, ikili ve sekizli yazalım. 2024 ÷ 16 = 126, kalan 8; 126 ÷ 16 = 7, kalan 14 (E); 7 ÷ 16 = 0, kalan 7. Sonuç 7E8h'dir. Her rakam dört bite açılır: 0111 1110 1000b. Sekizli için aynı bitler sağdan üçlü öbeklenir: 011 111 101 000 → 3750o. Sağlama: 7 × 256 + 14 × 16 + 8 = 1792 + 224 + 8 = 2024 ve 3 × 512 + 7 × 64 + 5 × 8 + 0 = 2024.

1.2.4 İşaretli sayılar

Buraya kadar bütün sayılar sıfır ya da pozitifti. Gerçek programlar ise sıcaklık farkları, borçlar, yön değişimleri gibi negatif değerlerle de çalışır. Kâğıt üzerinde negatif sayı önüne bir eksi işareti konarak yazılır. Bir yazmaçta ise eksi işareti için ayrı bir yer yoktur; yalnız bitler vardır. Bu kesim, negatif sayıların bitlerle nasıl gösterilebileceğini inceler ve 8086'nın kullandığı ikiye tümleyen gösterimini ayrıntılı ele alır.

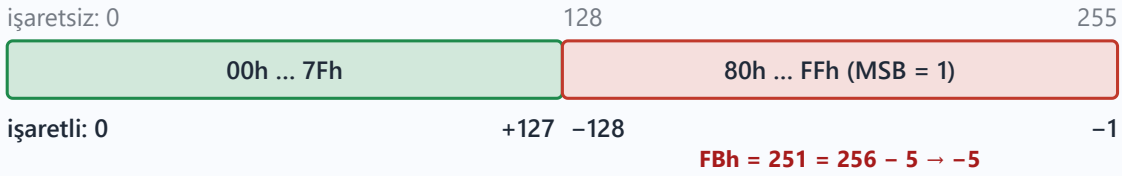
1.2.4.1 Desen ve yorum

`0FFh` pozitif mi, negatif mi? Bit desenine bakarak bu soruya cevap verilemez. 1111 1111 deseni işaretsiz okunursa 255'tir, ikiye tümleyen gösterimiyle okunursa −1'dir. İşlemci bu iki yorumdan hangisinin kastedildiğini bilmez ve bilmesi de gerekmez. Yorumu, deseni kullanan komut ya da programcı belirler.

Tanım 1.2.8

İşaretsiz (*unsigned*) yorumda n bitlik desenler 0 ile $2^n - 1$ arasındaki değerleri gösterir. **İşaretli** (*signed*) yorumda desenlerin yarısı negatif sayılara ayrılır. 8086'nın kullandığı **ikiye tümleyen** (*two's complement*) gösteriminde $-x$ negatif sayısı, $2^n - x$ işaretsiz değerinin deseniyle yazılır.

8 bit ile 256 farklı desen vardır. İkiye tümleyende bunların ilk 128'i (00h … 7Fh, yani 0 … 127) sıfır ve pozitif sayıları gösterir. Sonraki 128'i (80h … FFh, yani işaretsiz 128 … 255) negatif sayıları gösterir. −5'i yazmak için 5, desen sayısından çıkarılır: 256 − 5 = 251 = FBh. −1 için 256 − 1 = 255 = FFh, −128 için 256 − 128 = 128 = 80h bulunur. Şekil 1.2.20 256 desenin iki yorumunu aynı eksen üzerinde gösterir.

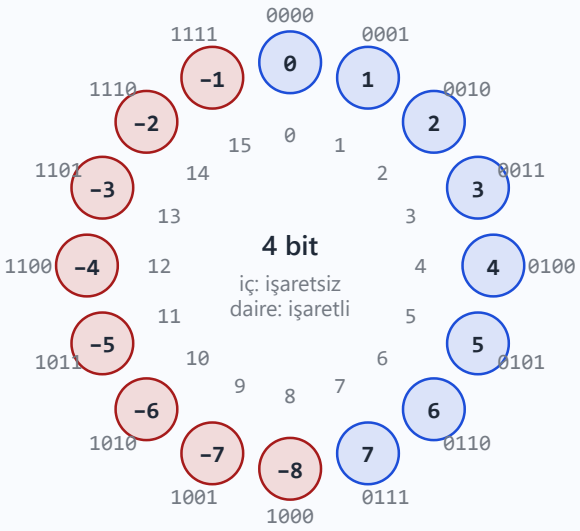


Şekil 1.2.20: 8 bitlik 256 desenin iki yorumu. 00h … 7Fh her iki yorumda 0 … 127'dir. 80h … FFh işaretsiz yorumda 128 … 255, işaretli yorumda −128 … −1'dir; bu desenlerin hepsinde MSB = 1'dir.

İki yorumun ortak bir özelliği vardır: negatif sayıların hepsinde en anlamlı bit 1'dir, pozitiflerin hepsinde 0'dır. Bu yüzden MSB'ye **işaret biti** (*sign bit*) denir. 8086 her aritmetik ve mantıksal işlemden sonra sonucun MSB'sini **işaret bayrağına** (SF) kopyalar. SF = 1 ise sonuç, işaretli yorumda negatiftir.

1.2.4.2 Sayı çemberi

Sabit genişlikli bir yazmaç bir doğru gibi değil, bir çember gibi davranır. 4 bitlik bir yazmacı düşünelim. 0000'dan başlayıp 1 ekledikçe 0001, 0010, … 1111'e ulaşılır. 1111'e 1 eklenince sonuç 10000'dır; beşinci bit yazmaca sığmaz ve yazmaç 0000'a döner. Şekil 1.2.21 on altı deseni bir çemberin üzerine dizer. Saat yönünde bir adım ilerlemek 1 eklemek, ters yönde bir adım 1 çıkarmaktır.



Şekil 1.2.21: 4 bitlik sayı çemberi. Her desenin yanında içte işaretsiz değeri, dışta ikiye tümleyen değeri yazılıdır. 0000–0111 iki yorumda da 0–7'dir; 1000–1111 işaretsiz 8–15, işaretli −8 … −1'dir.

Çember iki "kopukluk" noktasını açıkça gösterir. Birincisi 1111 ile 0000 arasıdır. İşaretsiz yorumda burada 15'ten 0'a atlanır. Bir toplama bu noktayı geçerse sonuç işaretsiz yoruma sığmamıştır ve 8086 bunu **elde bayrağıyla** (CF) bildirir. İkincisi 0111 ile 1000 arasıdır. İşaretli yorumda burada +7'den −8'e atlanır. Bir işlem bu noktayı geçerse sonuç işaretli yoruma sığmamıştır ve 8086 bunu **taşma bayrağıyla** (OF) bildirir.

Çemberin en önemli dersi şudur: toplama ve çıkarma devresi iki yorum için de **aynıdır**. Çemberde bir adım ilerlemek, desenin hangi yorumla okunduğundan bağımsızdır. 1011 + 0001 = 1100 işlemi işaretsiz yorumda 11 + 1 = 12, işaretli yorumda −5 + 1 = −4'tür; iki sonuç da doğrudur. İşlemci tek bir toplayıcı kullanır ve iki yorum için ayrı bayraklar kurar. Hangi bayrağa bakılacağına program karar verir.

1.2.4.3 Üç işaretli gösterim

Negatif sayıları göstermenin tek yolu ikiye tümleyen değildir. Tarihte üç gösterim kullanılmıştır. Tablo 1.2.9 4 bitlik on altı deseni dört yorumla karşılaştırır.

İkili	İşaretsiz	İşaret-büyüklük	Birler tümleyeni	İkiye tümleyen
`0000`	0	0	0	0
`0001`	1	1	1	1
`0010`	2	2	2	2
`0011`	3	3	3	3
`0100`	4	4	4	4
`0101`	5	5	5	5
`0110`	6	6	6	6
`0111`	7	7	7	7
`1000`	8	−0	−7	**−8**
`1001`	9	−1	−6	−7
`1010`	10	−2	−5	−6
`1011`	11	−3	−4	−5
`1100`	12	−4	−3	−4
`1101`	13	−5	−2	−3
`1110`	14	−6	−1	−2
`1111`	15	−7	−0	−1

Tablo 1.2.9: 4 bitlik desenlerin dört yorumu.

> **Tanım 1.2.9**
>
> **İşaret-büyüklük** (*sign-magnitude*) gösteriminde en üst bit işareti, kalan bitler sayının mutlak değerini gösterir. 4 bitte 0101 = +5, 1101 = −5'tir. Kâğıt üzerindeki eksi işaretinin doğrudan karşılığıdır.

İşaret-büyüklük gösterimi okuması en kolay olanıdır, ama iki ciddi sorunu vardır. Birincisi, sıfırın iki gösterimi vardır: 0000 = +0 ve 1000 = −0. Bir sayının sıfır olup olmadığını sınamak için iki deseni de denetlemek gerekir. İkincisi, toplama devresi karmaşıklaşır. +5 + (−3) işleminde 0101 ve 1011 doğrudan toplanırsa 1 0000 çıkar; bu, beklenen +2 değildir. Toplayıcının önce işaretleri karşılaştırması, sonra büyüklükleri toplaması ya da büyükten küçüğü çıkarması gerekir. Bu gösterim tam sayılarda bugün kullanılmaz. Kayan nokta sayılarının işaret biti ise aynı fikre dayanır.

1.2.4.4 Birler tümleyeni

Tanım 1.2.10

Birler tümleyeni (*one's complement*) gösteriminde bir sayının negatifi, sayının **bütün bitleri ters çevrilerek** (0 → 1, 1 → 0) elde edilir. *n* bitte *–x*'in deseni $(2^n - 1) - x$'tir. 8086'da `NOT` komutu bir işlenenin birler tümleyenini üretir.

8 bitlik bir örnek: +45 = 0010 1101b'dir. Bütün bitler ters çevrilince 1101 0010b = D2h elde edilir. Birler tümleyeni gösteriminde bu desen −45'tir. Aynı desen ikiye tümleyende −46'yı gösterir; iki gösterim birbirinden tam bir birim farklıdır. Şekil 1.2.22 işlemi bit bit gösterir.



Şekil 1.2.22: +45 = 2Dh'nin bütün bitleri ters çevrilince D2h çıkar. Birler tümleyeni gösteriminde D2h = −45'tir.

Birler tümleyeni, işaret-büyüklüğün toplama sorununu büyük ölçüde çözer, ama iki kusur bırakır. Sıfırın yine iki gösterimi vardır: 0000 0000 = +0 ve 1111 1111 = −0. Ayrıca toplama sonucunda en üst bitten bir elde çıkarsa bu eldenin sonucun en düşük bitine geri eklenmesi gerekir; bu işleme **çevrimsel elde** (*end-around carry*) denir. 4 bitte −2 + 5: 1101 + 0101 = 1 0010 olur. Eldeki 1 geri eklenince 0010 + 1 = 0011 = 3 bulunur. Eski bazı bilgisayarlar (CDC 6600, UNIVAC 1100) bu gösterimi kullandı. Bugün birler tümleyeni tek başına bir sayı gösterimi olarak değil, ikiye tümleyenin ara adımı ve bit düzeyinde DEĞİL işlemi olarak karşımıza çıkar.

1.2.4.5 İkiye tümleyen

İkiye tümleyen, birler tümleyeninin iki kusurunu tek bir hamleyle giderir: ters çevrilen desene 1 eklenir. Bu ekleme, −0 desenini ortadan kaldırır ve çevrimsel eldeye olan ihtiyacı yok eder.

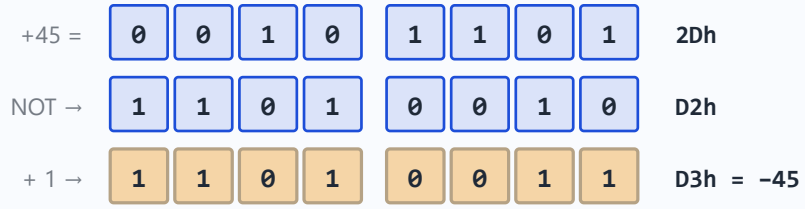
Tanım 1.2.11

Bir sayının **ikiye tümleyeni**, önce bütün bitleri ters çevrilerek, sonra sonuca 1 eklenerek bulunur: $-x = NOT\ x + 1$. *n* bitte bu desen $2^n - x$'tir. 8086'da `NEG` komutu işlenenin ikiye tümleyenini tek adımda üretir; işlemi $0 - x$ çıkarması olarak yapar.

İki formülün aynı deseni verdiği kolayca gösterilir. *n* bitin hepsi 1 olan desen $2^n - 1$'dir. Bir sayının bitlerini ters çevirmek, onu bu desenden çıkarmakla aynıdır, çünkü 1 − 0 = 1 ve 1 − 1 = 0'dır ve hiçbir basamakta ödünç gerekmez. Dolayısıyla $NOT\ x = (2^n - 1) - x$ ve $NOT\ x + 1 = 2^n - x$ olur.

Örnek 1.2.11

−45'in 8 bitlik ikiye tümleyen gösterimini bulalım. +45 = 0010 1101b'dir. Bütün bitler ters çevrilince 1101 0010b çıkar. Buna 1 eklenince 1101 0011b = **D3h** bulunur. Sağlama: D3h = 211 = 256 − 45. İkinci sağlama toplamayla yapılır: D3h + 2Dh = 100h; 8 bitte sonuç 00h'dir, yani −45 + 45 = 0.



Şekil 1.2.23: −45'in ikiye tümleyeni: 0010 1101 → NOT → 1101 0010 → +1 → 1101 0011 = D3h.

Elle hesapta daha hızlı bir kısa yol vardır. Sağdan başlayarak ilk 1'e kadar olan bitler (o 1 dahil) aynen yazılır, solda kalan bütün bitler ters çevrilir. 0010 1101 için en sağdaki bit zaten 1'dir; o aynen kalır, soldaki yedi bit ters çevrilir: 1101 001**1**. 0001 1000 (24) için sağdan ilk 1 bit 3'tedir: sağdaki 1000 aynen kalır, soldaki 0001 ters çevrilir: 1110 1000 = E8h = 256 − 24. Kısa yolun nedeni, 1 eklemenin sağdaki 1'ler dizisini sıfırlayıp ilk 0'ı 1 yapmasıdır.

İkiye tümleyenin iki önemli özelliği vardır. Birincisi, bir sayının ikiye tümleyeninin ikiye tümleyeni sayının kendisidir: $2^n - (2^n - x) = x$. D3h'nin ikiye tümleyeni yeniden 2Dh'dir. Bu yüzden negatif bir sayının büyüklüğü, aynı işlemle bulunur. İkincisi, sıfırın tek bir gösterimi vardır. 0000 0000'ın ikiye tümleyeni 1111 1111 + 1 = 1 0000 0000'dır; dokuzuncu bit atılınca yine 0000 0000 kalır. Birler tümleyenindeki −0 deseni (1111 1111) ikiye tümleyende −1'dir.

İkiye tümleyende bir desenin işaretli değeri doğrudan da hesaplanabilir. MSB'nin ağırlığı negatif alınır, diğer bitler her zamanki gibi toplanır. 8 bitte ağırlıklar −128, 64, 32, 16, 8, 4, 2, 1'dir. D3h = 1101 0011b için −128 + 64 + 16 + 2 + 1 = −45 bulunur. Bu yorum, sabit noktalı kesirlerde de işe yarar.

1.2.4.6 Kod izleme: NOT + 1 ile NEG

Aşağıdaki program aynı sayının ikiye tümleyenini iki yoldan hesaplar ve sonuçları karşılaştırır. Ardından −45 + 45 toplamını yapar ve aynı işlemi 16 bitte tekrarlar. Program, her komutun bayraklara etkisini görmek için iyi bir örnektir.

```
ORG 100h
MOV AL, 45       ; +45
MOV BL, AL
NOT BL           ; birler tümleyeni
INC BL           ; + 1
NEG AL           ; tek komutla
CMP AL, BL       ; aynı mı?
ADD AL, 45       ; -45 + 45
MOV AX, 45
NEG AX           ; 16 bitte -45
RET
```

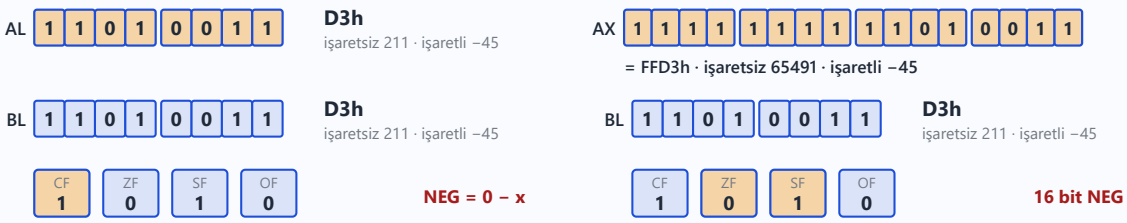
`MOV AL, 45` AL'ye 2Dh yükler; `MOV BL, AL` aynı değeri BL'ye kopyalar. MOV komutları bayrakları değiştirmez. `NOT BL` BL'nin bütün bitlerini ters çevirir: BL = D2h. NOT da hiçbir bayrağı etkilemez. `INC BL` BL'ye 1 ekler: BL = D3h. INC; ZF, SF, OF, AF ve PF bayraklarını kurar, ama CF'ye dokunmaz. Sonuç sıfır değildir (ZF = 0), MSB'si 1'dir (SF = 1).

`NEG AL` AL'yi 0 − AL olarak hesaplar ve aynı sonucu tek komutta verir: AL = D3h. NEG, işlenen sıfır değilse CF = 1 yapar; çünkü 0'dan sıfır olmayan bir sayı çıkarmak işaretsiz yorumda ödünç gerektirir. `CMP AL, BL` AL − BL farkını hesaplar ama sonucu yazmaz. D3h − D3h = 0 olduğu için ZF = 1, CF = 0, SF = 0 olur. ZF = 1, iki yolun aynı bit desenini ürettiğini gösterir.

`ADD AL, 45` −45 ile +45'i toplar. D3h + 2Dh = 100h'dir. Dokuzuncu bit AL'ye sığmaz ve CF'ye düşer: CF = 1. AL = 00h olur; ZF = 1. İşaretli yorumda sonuç doğrudur (−45 + 45 = 0), bu yüzden OF = 0'dır. Bu satır, ikiye tümleyende CF = 1'in bir hata anlamına gelmediğini gösterir. `MOV AX, 45` AX'e 002Dh yükler. `NEG AX` 16 bitlik ikiye tümleyeni hesaplar: 65.536 − 45 = 65.491 = **FFD3h**. Düşük bayt yine D3h'dir; üst bayt FFh olmuştur.

Komut	AX	BL	CF	ZF	SF	OF	Açıklama
`MOV AL, 45`	002Dh	00h	0	0	0	0	AL = +45
`MOV BL, AL`	002Dh	2Dh	0	0	0	0	kopya
`NOT BL`	002Dh	D2h	0	0	0	0	birler tümleyeni; bayrak değişmez
`INC BL`	002Dh	D3h	0	0	1	0	D3h = −45; CF'ye dokunulmaz
`NEG AL`	00D3h	D3h	1	0	1	0	0 − 2Dh; işlenen ≠ 0 → CF = 1
`CMP AL, BL`	00D3h	D3h	0	1	0	0	fark 0 → eşit
`ADD AL, 45`	0000h	D3h	1	1	0	0	D3h + 2Dh = 100h
`MOV AX, 45`	002Dh	D3h	1	1	0	0	MOV bayrağa dokunmaz
`NEG AX`	FFD3h	D3h	1	0	1	0	65.536 − 45 = FFD3h

Tablo 1.2.10: NOT/NEG programının izlemesi. Bayraklar her komuttan sonraki değerdir; ilk satırda başlangıç değeri 0 alınmıştır.



Şekil 1.2.24: Solda `NEG AL` sonrası: AL = D3h, BL = D3h, CF = 1, SF = 1. Sağda `NEG AX` sonrası: AX = FFD3h (işaretsiz 65.491, işaretli −45).

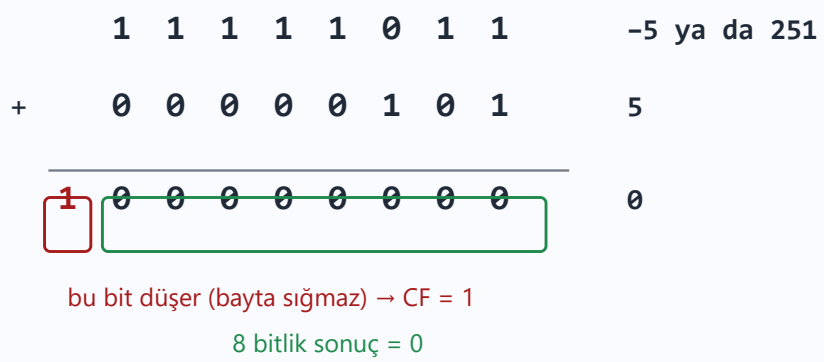
Son iki satır, aynı sayının farklı genişliklerde nasıl yazıldığını gösterir. −45, 8 bitte D3h, 16 bitte FFD3h'dir. 16 bitlik gösterimin düşük baytı 8 bitlik gösterimin aynısıdır; üst bayt yalnız işaret bitinin kopyalarından (FFh) oluşur. Bu gözlem, işaret genişletmenin temelidir.

Sık yapılan hata

NOT ile NEG'i karıştırmak. `NOT AL` birler tümleyenini verir ve sonuç gerçek negatiften 1 eksiktir: AL = 2Dh iken NOT AL = D2h = −46. Bir sayının işaretini değiştirmek için NEG kullanılmalı ya da NOT'tan sonra INC eklenmelidir. Bir başka yanılgı, NEG sonrası CF = 1'i hata sanmaktır; NEG, işlenen sıfır olmadıkça her zaman CF = 1 verir.

1.2.4.7 −5 + 5 = 0

Bir negatif sayı gösteriminin sağlaması gereken en temel kural, matematikteki *−x + x = 0* eşitliğidir. İkiye tümleyende −5 = FBh = 1111 1011b'dir. Buna 5 = 0000 0101b eklenince toplam 1 0000 0000b = 256 olur. Dokuzuncu bit bayta sığmaz ve düşer; kalan 8 bitin hepsi sıfırdır. Şekil 1.2.25 toplamayı sütun sütun gösterir.



Şekil 1.2.25: 1111 1011 (−5 ya da 251) + 0000 0101 (5) = 1 0000 0000. Dokuzuncu bit bayta sığmaz ve CF'ye düşer; 8 bitlik sonuç 0'dır.

Bu toplama, aynı toplayıcı devresinin iki yorum için de doğru sonuç verdiğinin bir örneğidir. İşaretsiz yorumda 251 + 5 = 256 sonucu 8 bite sığmamıştır; CF = 1 bunu bildirir. İşaretli yorumda −5 + 5 = 0 doğrudur ve OF = 0'dır. İkiye tümleyen bu yüzden seçilmiştir: negatif sayılar için ayrı bir devre gerekmez, toplayıcı desenlerin anlamını bilmeden çalışır. Bunun matematiksel nedeni, n bitlik toplamanın aslında 2^n modülünde yapılmasıdır. Bu modülde $2^n - x$ ile $-x$ aynı sınıftadır.

1.2.4.8 İşaretli aralıklar ve işaret genişletme

Tablo 1.2.11 8, 16 ve 32 bitlik sayıların işaretsiz ve işaretli aralıklarını verir. Her genişlikte desenlerin yarısı negatiftir.

Genişlik	Desen sayısı	İşaretsiz	İşaretli	Negatif desenler
8 bit	256	0 … 255	−128 … +127	80h … FFh
16 bit	65.536	0 … 65.535	−32.768 … +32.767	8000h … FFFFh
32 bit	4.294.967.296	0 … 4.294.967.295	−2.147.483.648 … +2.147.483.647	80000000h … FFFFFFFFh

Tablo 1.2.11: İşaretsiz ve işaretli (ikiye tümleyen) aralıklar.

Aralık simetrik değildir: negatif taraf pozitiften bir fazladır. Bunun nedeni sıfırın pozitif tarafta yer almasıdır. 8 bitte 128 negatif ve 128 sıfır-ya-da-pozitif desen vardır; pozitifler 1 … 127, negatifler −1 … −128'dir. Bu asimetrinin ilginç bir sonucu vardır: −128'in pozitif karşılığı 8 bite sığmaz. `NEG` ile 80h'nin ikiye tümleyeni alınırsa 0111 1111 + 1 = 1000 0000 = 80h çıkar; sonuç değişmez ve 8086 OF = 1 ile taşmayı bildirir.

Tanım 1.2.12

İşaret genişletme (*sign extension*), işaretli bir değeri daha geniş bir yazmaca değeri değiştirmeden taşır: yeni eklenen üst bitlerin hepsine eski işaret biti kopyalanır. **Sıfırla genişletme** ise üst bitleri 0 yapar ve işaretsiz değerler için doğrudur. 8086'da `CBW` AL'yi AX'e, `CWD` AX'i DX:AX'e işaretle genişletir.

Genişletmenin neden işaret bitiyle yapıldığı ikiye tümleyen formülünden görülür. −5, 8 bitte 256 − 5 = FBh, 16 bitte 65.536 − 5 = FFFBh'dir. İki desen arasındaki fark yalnız üst baytın FFh olmasıdır. Pozitif bir sayıda ise üst bayt 00h olur: +5, 05h ve 0005h'dir. Her iki durumda da üst bitler işaret bitinin kopyasıdır. FBh sıfırla genişletilseydi 00FBh = +251 elde edilirdi; −5 değeri kaybolurdu.

Örnek 1.2.12

AL = 9Ch (−100) iken `CBW` çalışıyor. AL'nin MSB'si 1'dir; bu yüzden AH = FFh olur ve AX = FF9Ch elde edilir. Sağlama: 65.536 − 100 = 65.436 = FF9Ch. AL = 64h (+100) olsaydı MSB 0 olduğundan AH = 00h ve AX = 0064h olurdu. `CWD` aynı işi bir üst genişlikte yapar: AX = FF9Ch ise DX = FFFFh olur.

1.2.4.9 Sabit noktalı işaretli kesirler

Kesirli sayılar için kayan nokta donanımı her zaman bulunmaz. 8086'nın kayan nokta komutu yoktur; birçok küçük mikrodenetleyicide de yoktur. Böyle sistemlerde kesirler tam sayı yazmaçlarında **sabit noktalı** olarak tutulur. Virgülün yeri önceden kararlaştırılır ve hiçbir yerde saklanmaz.

Tanım 1.2.13

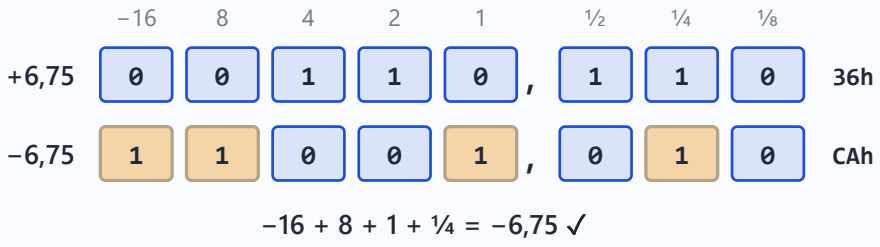
Sabit noktalı sayı (*fixed-point number*), kesir noktasının yeri önceden belirlenmiş ikili sayıdır. **Q*m.f*** biçiminde sayı, işaret biti dahil m tam sayı biti ve f kesir bitinden oluşur. Deseni bir tam sayı olarak okuyup 2^f'ye bölmek, sayının değerini verir.

8 bitlik Q5.3 biçimini düşünelim: son üç bit kesirdir. Basamak ağırlıkları soldan sağa −16, 8, 4, 2, 1, ½, ¼, ⅛'dir. En soldaki bitin ağırlığı ikiye tümleyende olduğu gibi negatiftir. Gösterilebilen değerler −16 ile +15,875 arasındadır ve ardışık iki değer arasındaki fark 1/8 = 0,125'tir.

Örnek 1.2.13

−6,75'i Q5.3 biçiminde yazalım. Önce +6,75 yazılır: 6,75 = 4 + 2 + ½ + ¼, yani 00110,110b = 36h. Negatif için ikiye tümleyen alınır; virgül yok sayılır. 0011 0110 ters çevrilince 1100 1001 olur; 1 eklenince 1100 1010 = **CAh** bulunur. Virgül yerine konunca 11001,010b okunur.

Sağlama ağırlıklarla yapılır: −16 + 8 + 1 + ¼ = −6,75. Daha kısa bir yol da vardır: sayı $2^3 = 8$ ile çarpılır ve tam sayı gibi işlenir. −6,75 × 8 = −54 ve 256 − 54 = 202 = CAh olur.



Şekil 1.2.26: Q5.3 biçiminde +6,75 = 36h ve −6,75 = CAh. İşaret bitinin ağırlığı −16'dır: −16 + 8 + 1 + ¼ = −6,75.

Sabit noktalı sayıların asıl üstünlüğü, toplamanın sıradan tam sayı toplaması olmasıdır. İşlemci için CAh sıradan bir bayttır; virgülün yeri yalnız programcının kabulüdür. −6,75 + 2,5 işleminde 2,5 × 8 = 20 = 14h'dir. CAh + 14h = DEh olur. DEh = 11011,110b, ağırlıklarla −16 + 8 + 2 + 1 + ½ + ¼ = −4,25'tir; beklenen sonuç da budur. Toplama ve çıkarma için ek bir donanım gerekmez. Çarpmada ise sonucun kesir bit sayısı ikiye katlanır ve sonuç f bit sağa kaydırılarak yeniden ölçeklenir.

Sunumdaki bit gezgini, 8 bitlik bir desenin işaretsiz, işaretli, BCD ve Gray kodu yorumlarını birlikte gösterir. Bitlerden biri değiştirildiğinde dört yorumun nasıl değiştiğine bakmak, desen ile yorum arasındaki ayrımı pekiştirir. Aynı amaçla İkiye Tümleyen Laboratuvarı da kullanılabilir.

1.2.5 İkili ve onaltılık aritmetik

Bir mikroişlemcinin aritmetik-mantık birimi (*arithmetic logic unit, ALU*) toplama, çıkarma ve bit düzeyinde işlemleri ikili sayılar üzerinde yapar. Programcı ise sonuçları çoğu zaman onaltılık okur. Bu kesim iki tabanda toplama ve çıkarmayı elle yapmayı, çıkarmanın neden toplamaya indirgendiğini ve CF ile OF bayraklarının bir işlemden sonra nasıl kurulduğunu anlatır.

1.2.5.1 İkili toplama ve tam toplayıcı

İkili toplama, onluk toplamayla aynı biçimde yapılır: sayılar alt alta yazılır, en düşük basamaktan başlanır ve sütunlar sırayla toplanır. Fark, bir sütun toplamının 2'ye ulaşınca eldeye dönüşmesidir. Bir sütunda iki rakam için dört durum vardır: 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1 ve 1 + 1 = 10b. Son durumda sütuna 0 yazılır ve bir sonraki sütuna 1 **elde** (*carry*) aktarılır. Önceki sütundan gelen elde de eklenince toplam 3 (11b) olabilir; bu durumda sütuna 1 yazılır ve yine 1 elde çıkar.



Şekil 1.2.27: İki bitin toplanmasındaki dört durum. Yalnız 1 + 1'de elde oluşur: sütuna 0 yazılır, 1 elde çıkar.

a	b	c_g	s	$c_ç$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tablo 1.2.12: Tam toplayıcının doğruluk tablosu. a ve b toplanan bitler, c_g gelen elde, s toplam biti, $c_ç$ çıkan eldedir.

Tablo 1.2.12 tek bir sütunun bütün durumlarını verir. Bu tablo, sayısal tasarımdan bilinen **tam toplayıcı** (*full adder*) devresinin doğruluk tablosudur. Toplam biti üç girişin XOR'udur: $s = a \oplus b \oplus c_g$. Çıkan elde, girişlerden en az ikisi 1 olduğunda 1'dir: $c_ç = ab + c_g(a \oplus b)$. Sekiz tam toplayıcı, her birinin çıkan eldesi bir sonrakinin gelen eldesine bağlanarak zincirlenirse 8 bitlik bir toplayıcı elde edilir. Bu yapıya **dalgalı elde toplayıcısı** (*ripple-carry adder*) denir. En son tam toplayıcının çıkan eldesi, 8086'da CF bayrağına yazılan bittir.

Dalgalı elde toplayıcısının bir zayıflığı vardır: en üst bitin sonucu, eldenin en alttan en üste kadar bütün kapılardan geçmesini bekler. 16 bitlik bir toplamada gecikme, tek bir tam toplayıcının gecikmesinin yaklaşık 16 katıdır. Hızlı işlemciler bu yüzden eldeyi önceden hesaplayan (*carry-lookahead*) devreler kullanır. Mantık aynıdır; yalnız eldeler paralel hesaplanır.

Örnek 1.2.14

110001101b (397) ile 100001011b (267) sayılarını toplayalım. Sütunlar sağdan sola numaralanır.

1. Basamak 0: 1 + 1 = 2 → 0 yaz, 1 elde.
2. Basamak 1: 0 + 1 + 1 (elde) = 2 → 0 yaz, 1 elde.
3. Basamak 2: 1 + 0 + 1 (elde) = 2 → 0 yaz, 1 elde.
4. Basamak 3: 1 + 1 + 1 (elde) = 3 → 1 yaz, 1 elde.
5. Basamak 4: 0 + 0 + 1 (elde) = 1 → 1 yaz, elde yok.
6. Basamak 5: 0 + 0 = 0 → 0 yaz.
7. Basamak 6: 0 + 0 = 0 → 0 yaz.
8. Basamak 7: 1 + 0 = 1 → 1 yaz.
9. Basamak 9: 1 + 1 = 2 → 0 yaz, 1 elde.
10. Basamak 9: yalnız elde 1 → 1 yaz.

Sonuç **1010011000b**'dir. Sağlama: 512 + 128 + 16 + 8 = 664 = 397 + 267. Toplanan sayılar 9 bitti, sonuç 10 bit oldu. 9 bitlik bir toplayıcıda en üstteki 1 sığmaz ve elde olarak dışarı çıkar.

elde	1				1	1	1	1	
397		1	1	0	0	0	1	1	0	1
+ 267		1	0	0	0	0	1	0	1	1
= 664	1	0	1	0	0	1	1	0	0	0

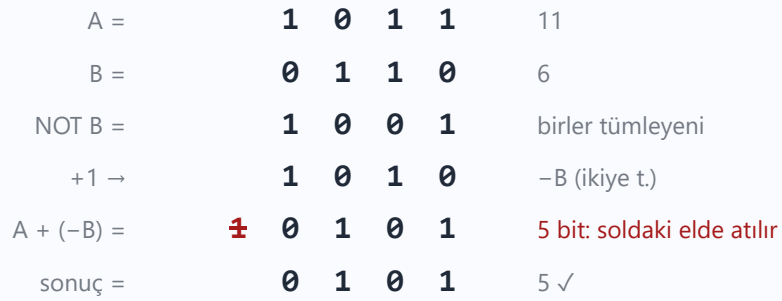
Şekil 1.2.28: 110001101b + 100001011b = 1010011000b (397 + 267 = 664). Üst satır her sütuna gelen eldeyi gösterir.

1.2.5.2 İkili çıkarma: tümleyenle toplama

Çıkarma da ödünç alarak sütun sütun yapılabilir. İşlemciler ise genellikle ayrı bir çıkarıcı devre kullanmaz. Çıkarma, toplamaya çevrilir: *A − B = A + (−B)*. −B, B'nin ikiye tümleyenidir; ikiye tümleyen de bitleri ters çevirip 1 eklemekle bulunur. Bit ters çevirme her bit için bir XOR kapısıyla, 1 ekleme ise en alt tam toplayıcının gelen eldesini 1 yaparak sağlanır. Böylece aynı toplayıcı hem toplar hem çıkarır.

Örnek 1.2.15

4 bitte 1011 − 0110 (11 − 6) işlemini yapalım. A = 1011, B = 0110'dır. B'nin bitleri ters çevrilir: 1001. Buna 1 eklenir: 1010; bu, 4 bitte −6'dır. A ile toplanır: 1011 + 1010 = 10101. Sonuç 5 bittir. 4 bite sığmayan soldaki 1 atılır ve **0101** = 5 kalır. Sağlama: 11 − 6 = 5.



Şekil 1.2.29: 1011 − 0110 işleminin ikiye tümleyenle yapılması: NOT B = 1001, −B = 1010, A + (−B) = 1 0101; soldaki elde atılınca 0101 = 5.

Bu örnekte atılan elde 1'dir ve sonuç doğrudur. A < B olsaydı ne olurdu? 0110 − 1011 (6 − 11) için 1011'in ikiye tümleyeni 0101'dir. 0110 + 0101 = 1011 olur ve elde çıkmaz. 1011 işaretli yorumda −5'tir; doğru sonuç da −5'tir. Elde çıkmaması, işaretsiz yorumda çıkarmanın ödünç gerektirdiğini, yani A'nın B'den küçük olduğunu gösterir.

8086 bu ilişkiyi CF bayrağında tersine çevirerek saklar. `SUB` ve `CMP` komutlarından sonra CF, işaretsiz yorumda **ödünç** (*borrow*) alınıp alınmadığını gösterir. Tümleyenle toplamada elde 1 çıktıysa ödünç yoktur ve CF = 0 olur. Elde çıkmadıysa ödünç vardır ve CF = 1 olur. Bu kural sayesinde `CMP AX, BX` sonrası CF = 1, işaretsiz yorumda AX < BX anlamına gelir. `JB` ("küçükse atla") tam olarak bu bayrağa bakar.

1.2.5.3 Onaltılık toplama

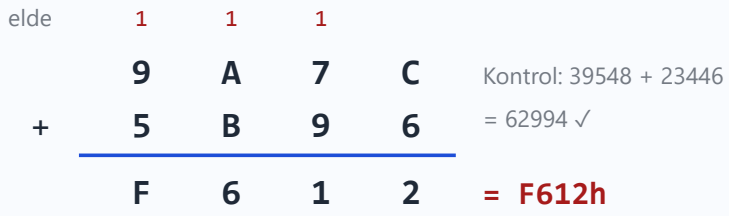
Onaltılık sayılar da sütun sütun toplanır. Rakamların değerleri onluk düşünülür (A = 10, …, F = 15). Bir sütunun toplamı 16'dan küçükse doğrudan yazılır. 16 ya da daha büyükse toplamdan 16 çıkarılır, kalan sütuna yazılır ve sola 1 elde verilir. Kural onluk toplamadakiyle aynıdır; yalnız eşik 10 yerine 16'dır.

Örnek 1.2.16

9A7Ch + 5B96h toplamını bulalım.

1. C + 6 = 12 + 6 = 18 ≥ 16 → 18 − 16 = 2 yaz, 1 elde.
2. 7 + 9 + 1 = 17 ≥ 16 → 17 − 16 = 1 yaz, 1 elde.
3. A + B + 1 = 10 + 11 + 1 = 22 ≥ 16 → 22 − 16 = 6 yaz, 1 elde.
4. 9 + 5 + 1 = 15 < 16 → F yaz, elde yok.

Sonuç **F612h**'dir. Sağlama onlukta: 9A7Ch = 39.548, 5B96h = 23.446 ve 39.548 + 23.446 = 62.994 = F612h. 8086'da AX = 9A7Ch, BX = 5B96h iken `ADD AX, BX` aynı sonucu verir. En soldaki sütundan elde çıkmadığı için CF = 0 olur.



Şekil 1.2.30: 9A7Ch + 5B96h = F612h. Üç sütundan elde çıkar, en soldaki sütundan çıkmaz.

Bu toplamın bir ayrıntısı daha vardır. İşaretli yorumda 9A7Ch negatiftir (MSB = 1), 5B96h pozitiftir. Farklı işaretli iki sayının toplamı hiçbir zaman işaretli taşma üretmez; bu yüzden OF = 0'dır. F612h işaretli yorumda −2542'dir: −25.988 + 23.446 = −2542.

1.2.5.4 Onaltılık çıkarma

Onaltılık çıkarmada üstteki rakam alttakinden küçükse soldaki sütundan 1 **ödünç** alınır. Ödünç alınan 1, sağdaki sütunda 16 değerindedir. Ödünç veren sütunun rakamı 1 azalır.

Örnek 1.2.17

9A7Ch − 5B96h farkını bulalım.

1. C − 6 = 12 − 6 = 6; ödünç yok.
2. 7 − 9 olmaz: soldan ödünç alınır. 7 + 16 − 9 = 14 = E; soldaki A, 9'a iner.
3. 9 − B olmaz: yine ödünç. 9 + 16 − 11 = 14 = E; soldaki 9, 8'e iner.
4. 8 − 5 = 3; ödünç yok.

Sonuç **3EE6h**'dir. Sağlama: 39.548 − 23.446 = 16.102 = 3EE6h. En soldaki sütunda ödünç gerekmediği için `SUB AX, BX` sonrası CF = 0 olur.



Şekil 1.2.31: 9A7Ch − 5B96h = 3EE6h. "−1" işaretleri ödünç veren sütunları gösterir.

1.2.5.5 15'e ve 16'ya tümleyen

İkili tabandaki birler ve ikiye tümleyenin onaltılık karşılıkları 15'e ve 16'ya tümleyendir. Bunlar onaltılık çıkarmayı, tıpkı ikilide olduğu gibi, toplamaya çevirir.

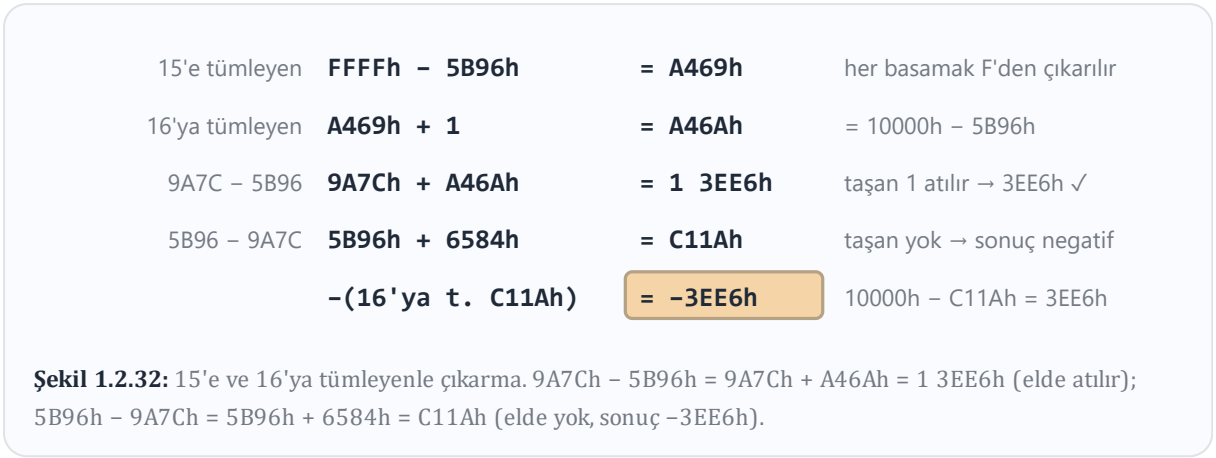
Tanım 1.2.14

k basamaklı onaltılık bir sayının **15'e tümleyeni**, her basamağın F'den çıkarılmasıyla bulunur ve $16^k - 1 - x$'e eşittir. **16'ya tümleyeni**, 15'e tümleyene 1 eklenerek bulunur ve $16^k - x$'e eşittir. 4 basamaklı onaltılık sayı 16 bit olduğu için 16'ya tümleyen, 16 bitlik ikiye tümleyenle aynı desendir.

Örnek 1.2.18

5B96h'nin 15'e tümleyeni: F − 5 = A, F − B = 4, F − 9 = 6, F − 6 = 9 → A469h. 16'ya tümleyeni A469h + 1 = A46Ah'dir; bu 10000h − 5B96h'ye eşittir. Şimdi 9A7Ch − 5B96h, 9A7Ch + A46Ah olarak hesaplanır: toplam 13EE6h'dir. Dört basamağı aşan 1 atılır ve **3EE6h** kalır. Doğrudan çıkarmayla aynı sonuç bulunmuştur.

Ters yönde 5B96h − 9A7Ch hesaplayalım. 9A7Ch'nin 16'ya tümleyeni 10000h − 9A7Ch = 6584h'dir. 5B96h + 6584h = C11Ah olur ve dört basamağı aşan elde **çıkmaz**. Elde çıkmaması sonucun negatif olduğunu gösterir. Büyüklüğü bulmak için C11Ah'nin 16'ya tümleyeni alınır: 10000h − C11Ah = 3EE6h. Sonuç −3EE6h, yani −16.102'dir.



Şekil 1.2.32: 15'e ve 16'ya tümleyenle çıkarma. 9A7Ch − 5B96h = 9A7Ch + A46Ah = 1 3EE6h (elde atılır); 5B96h − 9A7Ch = 5B96h + 6584h = C11Ah (elde yok, sonuç −3EE6h).

8086'da AX = 5B96h ve BX = 9A7Ch iken `SUB AX, BX` doğrudan AX = C11Ah verir. Ödünç alındığı için CF = 1 olur. C11Ah işaretli yorumda −16.102'dir; programcı sonucu işaretli okuyorsa ek bir işlem gerekmez. İşaretsiz okuyorsa CF = 1, sonucun anlamsız olduğunu, çünkü küçük sayıdan büyüğün çıkarıldığını söyler.

1.2.5.6 Elde ve işaretli taşma

Sabit genişlikteki bir yazmaçta her işlemin sonucu yazmaca sığmayabilir. Sığmama iki farklı biçimde olur ve 8086 bunları iki ayrı bayrakla bildirir. Bu iki bayrağı karıştırmak, assembly programlamadaki en yaygın hatalardan biridir.

Tanım 1.2.15

Elde bayrağı (CF), işaretsiz yorumda sonucun hedefe sığmadığını gösterir: toplamada en üst bitten elde çıkmıştır, çıkarmada ödünç alınmıştır. **Taşma bayrağı** (OF), işaretli yorumda sonucun hedefin aralığına sığmadığını gösterir (*overflow*). Donanım OF'yi şöyle kurar: işaret bitine *giren* elde ile işaret bitinden *çıkan* elde farklıysa OF = 1'dir.

İşaretli taşmanın belirtisi, sonucun işaretinin beklenenin tersi çıkmasıdır. İki pozitif sayının toplamı negatif çıkarsa ya da iki negatif sayının toplamı pozitif çıkarsa taşma olmuştur. Farklı işaretli iki sayının toplamı ise hiçbir zaman taşmaz: sonuç her zaman iki sayının arasında kalır ve aralığın içindedir. Çıkarmada durum terstir: *A − B = A + (−B)* olduğu için farklı işaretli sayıların farkı taşabilir, aynı işaretli sayıların farkı taşamaz. Tablo 1.2.13 altı durumu 8 bitlik örneklerle verir.

İşlem	İşaretler	Örnek	İşaretli yorum	OF	CF
A + B	+ ile +	`50h + 40h = 90h`	80 + 64 → −112 (yanlış)	1	0
A + B	− ile −	`9Ch + CEh = 6Ah`	−100 + (−50) → +106 (yanlış)	1	1
A + B	farklı	`64h + CEh = 32h`	100 + (−50) = 50	0	1
A − B	+ eksi −	`64h - CEh = 96h`	100 − (−50) → −106 (yanlış)	1	1
A − B	− eksi +	`9Ch - 32h = 6Ah`	−100 − 50 → +106 (yanlış)	1	0

İşlem	İşaretler	Örnek	İşaretli yorum	OF	CF
A − B	aynı	`9Ch − CEh = CEh`	−100 − (−50) = −50	0	1

Tablo 1.2.13: 8 bitte işaretli taşma kuralı. CF sütunu aynı işlemin işaretsiz yorumunu gösterir.

Tablonun CF sütunu iki bayrağın birbirinden bağımsız olduğunu gösterir. Birinci satırda CF = 0 ama OF = 1'dir: 80 + 64 = 144 işaretsiz yorumda 255'e sığar, işaretli yorumda 127'yi aşar. Üçüncü satırda CF = 1 ama OF = 0'dır: işaretsiz yorumda 100 + 206 = 306 bayta sığmaz, işaretli yorumda 100 − 50 = 50 doğrudur. Hangi bayrağın anlamlı olduğu, programcının sayıları nasıl yorumladığına bağlıdır. İşaretsiz sayılarla çalışan bir program CF'ye, işaretli sayılarla çalışan bir program OF'ye bakar.

Örnek 1.2.19

50h + 40h işleminde donanım kuralını uygulayalım. 0101 0000 + 0100 0000 toplanırken bit 6'da 1 + 1 = 10b olur: bit 6'dan bit 7'ye (işaret bitine) 1 elde girer. Bit 7'de 0 + 0 + 1 = 1 olur; işaret bitinden elde çıkmaz. Giren elde 1, çıkan elde 0'dır; ikisi farklı olduğu için OF = 1'dir. Çıkan elde 0 olduğu için CF = 0'dır. Sonuç 1001 0000 = 90h'dir.

9Ch + CEh işleminde ise 1001 1100 + 1100 1110 toplanır. Bit 6'da 0 + 1 + (bit 5'ten gelen 0) = 1 olur; işaret bitine elde girmez. Bit 7'de 1 + 1 = 10b olur; elde çıkar. Giren 0, çıkan 1 olduğu için OF = 1, çıkan elde 1 olduğu için CF = 1'dir. Sonuç 0110 1010 = 6Ah'dir.

Kuralın mantığı şöyle açıklanır. İşaret bitinden çıkan elde, işaretsiz taşmayı gösterir. İşaret bitine giren elde ise alt yedi bitin toplamının 127'yi aştığını gösterir. İki pozitif sayıda çıkan elde yoktur; giren elde varsa sonuç işaret bitini bozmuştur. İki negatif sayıda çıkan elde her zaman vardır; giren elde yoksa sonuç işaret bitini 0'a düşürmüştür. Giren ile çıkan eşitse işaret biti doğru kalmıştır.

Genişlikleri farklı işaretli sayılar toplanacaksa kısa olan önce işaretle genişletilmelidir. FBh (−5) ile 0100h (256) toplanacak olsun. FBh işaretle genişletilince FFFBh olur: FFFBh + 0100h = 1 00FBh ve 16 bitlik sonuç 00FBh = 251'dir. Doğru sonuç da −5 + 256 = 251'dir. FBh sıfırla genişletilseydi 00FBh + 0100h = 01FBh = 507 çıkardı; bu yanlıştır. 8086'da `CBW` bu işi tek komutla yapar.

Sık yapılan hata

CF = 1'i her durumda “sonuç yanlış” diye okumak. İşaretli sayılarla çalışırken CF anlamsızdır: −5 + 5 işleminde CF = 1'dir ama sonuç (0) doğrudur. Tersine, işaretsiz sayılarla çalışırken OF anlamsızdır: 80 + 64 işleminde OF = 1'dir ama işaretsiz sonuç (144) doğrudur. Bayrak, yalnız programcının seçtiği yoruma göre okunur.

Bayrak Laboratuvarı simülatörü, seçilen iki işlenen için toplama ve çıkarma sonrası CF, OF, SF, ZF ve diğer bayrakları gösterir. Tablodaki satırları orada denemek, iki bayrak arasındaki farkı somutlaştırır.

1.2.5.7 Assembler'ın sabit ifadeleri

emu8086 assembler'ı sabit ifadelerde bir dizi işleç kabul eder. Bu işleçler programın yazılmasını kolaylaştırır: bir maske ya da bir adres farkı elle hesaplanmak yerine ifade olarak yazılabilir. Tablo 1.2.14 desteklenen işleçleri örnekleriyle listeler.

İşleç	Anlam	Örnek → değer	Çalışma zamanındaki karşılığı
`~`	bütün bitleri tersler	`~0Fh` → F0h (8 bit)	`NOT`
`*`	çarpma	`3*4` → 12	`MUL`
`/`	tam sayı bölme	`17/5` → 3	`DIV` (bölüm)
`%`	kalan	`17%5` → 2	`DIV` (kalan)
`+`	toplama	`40h+2` → 42h	`ADD`
`-`	çıkarma, tekli eksi	`-5` → FBh	`SUB` , `NEG`
`<<`	sola kaydırma	`1<<4` → 16	`SHL`
`>>`	sağa kaydırma	`80h>>3` → 10h	`SHR`
`&`	bit düzeyinde VE	`0F0h&3Ch` → 30h	`AND`
`^`	bit düzeyinde XOR	`0Fh^05h` → 0Ah	`XOR`
`|`	bit düzeyinde VEYA	`0F0h|0Fh` → FFh	`OR`

Tablo 1.2.14: emu8086'nın sabit ifadelerde desteklediği işleçler.

Bu ifadeler **derleme zamanında** hesaplanır. `MOV AX, 3*4+1` yazıldığında assembler 13'ü hesaplar ve makine koduna `MOV AX, 13` komutunu yazar. İşlemci çalışırken çarpma yapmaz. Bu yüzden ifadeler yalnız sabitlerle çalışır: `MOV AX, BX*2` geçersizdir, çünkü BX'in değeri derleme zamanında bilinmez. Çalışma zamanında aynı işlemler tablonun son sütunundaki komutlarla yapılır. BX'i 2 ile çarpmak için örneğin `SHL BX, 1` kullanılır.

1.2.6 Bitin donanımdaki karşılığı

Buraya kadar bitler soyut 0 ve 1'ler olarak ele alındı. Bir mikroişlemcide ise her bit fiziksel bir büyüklüktür: bir hattaki gerilim ya da bir devrenin kararlı durumu. Bu kesim, gerilim düzeylerinin bitlere nasıl karşılık geldiğini ve bitlerin flip-flop'larda nasıl saklandığını anlatır. Konular sayısal tasarımdan bilinir; burada 8086'nın yazmaçları ve saat sinyaliyle ilişkileri öne çıkarılır.

1.2.6.1 İkili sinyaller ve gerilim düzeyleri

Bir programın komutları ve verileri bellekte rakam öbekleri olarak durur. Bu rakamlar devrelerde gerilim ve akım olarak var olur. Sayısal bilgisayarlar iki tür sinyal kullanır: **yüksek** düzey 1'i, **düşük** düzey 0'ı temsil eder. 8086 gibi 5 V ile beslenen bir devrede yüksek düzey 5 V'a, düşük düzey 0 V'a yakındır.

Gerilimin tam 5 V ya da tam 0 V olması gerekmez. Her mantık ailesi bir giriş için eşik aralıkları tanımlar. TTL (*transistor-transistor logic*) ailesinde 2,0 V ve üzeri bir giriş 1, 0,8 V ve altı bir giriş 0 kabul edilir. Arada kalan bölge tanımsızdır ve kararlı çalışan bir devrede bir sinyal orada ancak geçiş sırasında bulunur. Çıkış tarafında ise daha sıkı sınırlar vardır: bir TTL çıkışı 1 için en az 2,4 V, 0 için en çok 0,4 V verir. Çıkış ile giriş eşikleri arasındaki 0,4 V'luk pay **gürültü payıdır** (*noise margin*). Sinyale bu kadar gürültü eklense bile alıcı bit değerini doğru okur. Sayısal devrelerin gürültüye dayanıklı olmasının nedeni budur. Analog bir devrede sinyale eklenen her gürültü değeri bozar; sayısal devrede eşiği aşmayan gürültü hiçbir iz bırakmaz.

8086'nın adres ve veri yolları bu tür hatlardan oluşur. 20 adres hattı aynı anda 20 biti, 16 veri hattı aynı anda 16 biti taşır. Bir `MOV` komutu bellekten bir sözcük okuduğunda bellek yongası 16 veri hattının her birini yüksek ya da düşük düzeye çeker. İşlemci bu düzeyleri saatin belirli bir anında örnekler ve yazmacına alır.

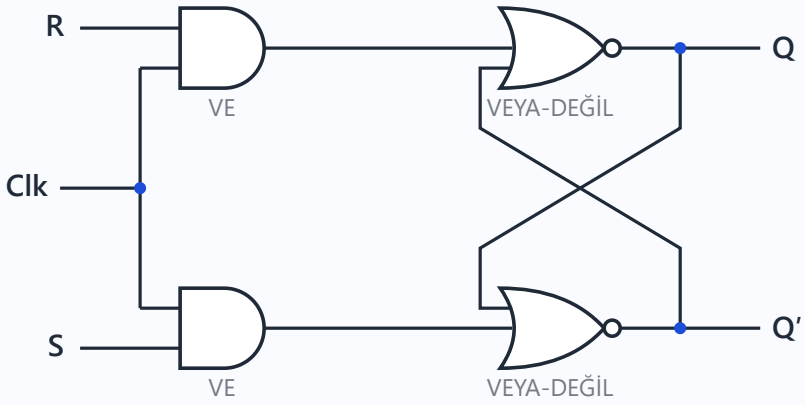
1.2.6.2 Flip-flop

Bir biti bir sonraki işleme kadar saklamak için belleği olan bir devre gerekir. Kapılardan oluşan bileşimsel bir devre girişleri değişince çıkışını da değiştirir; geçmişi hatırlamaz. Geri beslemeli devreler ise hatırlar.

Tanım 1.2.16

Flip-flop, bir biti saklayan ve durumunu saat sinyaline bağlı olarak değiştiren temel bellek devresidir. İki çıkışı vardır: saklanan bit Q ve onun tersi Q'. Girişindeki değer, saat sinyalinin belirli bir anında devreye yüklenir ve bir sonraki o ana kadar saklanır.

Şekil 1.2.33 saat girişli bir SR flip-flop'u gösterir. Devrenin kalbi, çapraz bağlanmış iki VEYA-DEĞİL (NOR) kapısıdır. Her kapının çıkışı ötekinin girişine geri beslenir. Bu geri besleme, iki kararlı durum yaratır: Q = 1, Q′ = 0 ya da Q = 0, Q′ = 1. Girişler 0 iken devre bulunduğu durumda kalır; biti “hatırlar”. S ve R girişleri saat sinyaliyle VE kapılarından geçer. Böylece girişler yalnız saat 1 iken etkili olur.



Şekil 1.2.33: Saat girişli SR flip-flop. R ve S girişleri Clk ile VE kapılarından geçer; çapraz bağlı iki VEYA-DEĞİL kapısı Q ve Q′ çıkışlarını üretir.

1.2.6.3 Flip-flop türleri

Uygulamada dört flip-flop türü kullanılır. Hepsi bir bit saklar; girişlerinin anlamı farklıdır. Tablo 1.2.15 dört türün davranışını özetler.

Tür	Girişler	Davranış	Özellik
SR (set-reset)	S, R	S = 1 → Q = 1; R = 1 → Q = 0; S = R = 0 → değişmez	S = R = 1 yasaktır
D (veri)	D	Q ← D	yazmaç ve bellek hücresi
JK	J, K	J = 1 → Q = 1; K = 1 → Q = 0; J = K = 0 → değişmez; J = K = 1 → Q tersine döner	yasak durumu yoktur
T (tersleme)	T	T = 1 → Q tersine döner; T = 0 → değişmez	sayaç, frekans bölücü

Tablo 1.2.15: Flip-flop türleri ve saat darbesindeki davranışları.

SR flip-flop'ta S = R = 1 girişi yasaktır. Bu durumda iki NOR kapısının çıkışı da 0 olur ve Q ile Q′ birbirinin tersi olmaktan çıkar. Girişler aynı anda 0'a dönerse devrenin hangi duruma düşeceği kapı gecikmelerine bağlıdır ve öngörülemez. JK flip-flop bu durumu yararlı bir işe çevirir: J = K = 1 iken her saat darbesinde çıkış tersine döner. JK'yı esnek yapan bu tersleme özelliğidir. J ve K girişleri birleştirilip tek bir T girişi yapılınca T flip-flop elde edilir. T = 1 tutulan bir T flip-flop'un çıkışı her iki saat darbesinde bir tam periyot üretir, yani saat frekansını ikiye böler. Zincirlenen T flip-flop'lar ikili bir sayaç oluşturur.

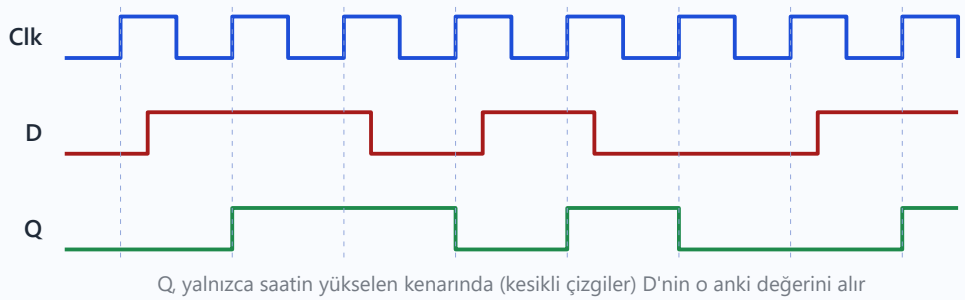
Örnek 1.2.20

Bir JK flip-flop'ta başlangıçta Q = 0'dır ve J = K = 1 tutulurken üç yükselen saat kenarı gelir. Her kenarda çıkış tersine döner: 0 → 1 → 0 → 1. Üç kenardan sonra Q = 1'dir. Kenar sayısı çift ol-

saydı Q yine 0 olurdu.

1.2.6.4 D flip-flop ve yazmaç

Mikroişlemcilerde en çok kullanılan tür D flip-flop'tur. Tek bir veri girişi vardır ve saatin yükselen kenarında D'deki değer Q'ya kopyalanır. Şekil 1.2.34 bir zamanlama diyagramı üzerinde bu davranışı gösterir.



Şekil 1.2.34: D flip-flop zamanlaması. Q yalnız saatin yükselen kenarında (kesikli çizgiler) D'nin o anki değerini alır; kenarlar arasında D'deki değişiklikler Q'yu etkilemez.

Diyagramda D girişi saat kenarları arasında birkaç kez değişir, ama Q bu değişikliklerden etkilenmez. Yalnız yükselen kenar anında D örneklenir ve değer bir sonraki kenara kadar Q'da saklanır. Bu, 1 bitlik bir bellektir. Yan yana konmuş 16 D flip-flop ortak bir saat ve bir yükleme sinyaliyle sürülürse 16 bitlik bir yazmaç elde edilir. 8086'nın AX, BX gibi yazmaçları ilke olarak böyle kurulmuştur. Bir `MOV AX, BX` komutu, BX'in çıkışlarının iç veri yolu üzerinden AX'in D girişlerine taşınması ve bir saat kenarında AX'e yüklenmesidir. Bayrak yazmacındaki her bayrak da bir flip-flop'tur.

Saat sinyali bütün bu yüklemelerin ne zaman olacağını belirler. 8086'nın bütün iç işlemleri saat turlarına bölünür. 5 MHz'lik bir saatte bir **saat turu** (*clock cycle*) 1 / 5.000.000 s = 200 ns sürer. Bir komutun kaç tur sürdüğü, kaç iç adımdan oluştuğuna bağlıdır. Yazmaçtan yazmaca bir `MOV` 2 tur, 16 bitlik bir `DIV` 144–162 tur sürer. Bir bellek okuması ise en az 4 turluk bir yol döngüsü (T1–T4) gerektirir.

1.2.7 Kayan noktalı sayılar: IEEE 754

Sabit noktalı gösterim, virgülün yeri sabit olduğu için ya çok büyük ya çok küçük sayıları tutabilir; ikisini birlikte tutamaz. Q5.3 biçiminde en küçük adım 0,125'tir ve en büyük değer 16'nın altındadır. Bilimsel hesaplarda ise hem 6×10^{23} gibi büyük hem $1{,}6 \times 10^{-19}$ gibi küçük sayılar gerekir. Bilimsel gösterimdeki fikir ikiliye taşınınca kayan nokta biçimi elde edilir: sayı bir kesir ile 2'nin bir kuvvetinin çarpımı olarak yazılır ve virgül, üsse göre “kayar”.

1.2.7.1 Tek duyarlıklı biçim

Tanım 1.2.17

IEEE 754 tek duyarlıklı (*single precision*) **kayan noktalı sayı** 32 bittir ve üç alandan oluşur: 1 bitlik **işaret** s (bit 31), 8 bitlik **üs** e (bit 30–23) ve 23 bitlik **kesir** (*fraction, mantissa*) f (bit 22–0). Üs 127 kaydırmalı saklanır. Normalize bir sayının değeri

$$\text{değer} = (-1)^{s} \times 1{,}f \times 2^{e-127}$$

ifadesidir. Buradaki 1, saklanmayan **gizli bittir**.



Şekil 1.2.35: IEEE 754 tek duyarlıklı biçimin alanları: bit 31 işaret, bit 30–23 üs, bit 22–0 kesir. Gösterilen desen 3E200000h'dir.

Biçimin üç ayrıntısı açıklama ister. Birincisi, işaret ayrı bir bittir: kayan noktalı sayılar ikiye tümleyen değil, işaret-büyüklük mantığıyla saklanır. Bir sayının negatifini almak için yalnız bit 31 ters çevrilir. İkincisi, üs **kaydırmalı** (*biased*) saklanır: gerçek üse 127 eklenir. Böylece −126 … +127 arasındaki üsler 1 … 254 arasındaki işaretsiz sayılarla yazılır. Kaydırmanın yararı, iki pozitif kayan noktalı sayının desenlerinin işaretsiz tam sayı gibi karşılaştırılabilmesidir. Üçüncüsü, normalize bir ikili sayı her zaman 1 ile başlar: $1{,}xxx \times 2^{k}$. Baştaki 1 hep aynı olduğu için saklanmaz ve kesir alanına bir bit daha sığar. Bu yüzden 23 bitlik alan 24 bitlik duyarlık verir.

8086'nın kayan nokta komutu yoktur. Kayan nokta işlemleri ya 8087 matematik yardımcı işlemcisiyle ya da tam sayı komutlarıyla yazılmış yazılım yordamlarıyla yapılır. 8087, 8086'nın yanına takılan ve kendi yazmaç yığınıyla bu biçimdeki sayılar üzerinde çalışan ayrı bir yongadır. 64 bitlik çift duyarlıklı (*double*) biçim aynı fikri 1 + 11 + 52 bitle uygular; üs kaydırması 1023'tür.

1.2.7.2 Çözümleme

Örnek 1.2.21

`0 01111100 01000000000000000000000` deseninin, yani 3E200000h'nin değerini bulalım.

1. İşaret biti 0'dır: sayı pozitiftir, çarpan (+1).
2. Üs alanı 0111 1100b = 124'tür. Gerçek üs 124 − 127 = −3'tür: çarpan 2^{-3}.
3. Kesir alanı 0100…0'dır: $0 \times 2^{-1} + 1 \times 2^{-2} = 0{,}25$. Gizli bitle birlikte 1,25 olur.

4. Değer $(+1) \times 1{,}25 \times 2^{-3} = 1{,}25 / 8 =$ **0,156 25**'tir.

Desenin onaltılık yazımı alanlar bir araya getirilip dörtlü öbeklenerek bulunur: 0011 1110 0010 0000 … = 3E200000h.

1.2.7.3 Kodlama

Örnek 1.2.22

−6,5'i 32 bitlik IEEE 754 biçimine çevirelim.

1. Sayı negatiftir: işaret biti **1**.
2. Büyüklük ikiliye çevrilir: 6 = 110b, 0,5 = 0,1b; 6,5 = 110,1b.
3. Normalize edilir: virgül iki basamak sola kayar, $110{,}1b = 1{,}101b \times 2^2$.
4. Üs: 2 + 127 = 129 = **1000 0001b**.
5. Kesir: gizli 1 atılır, kalan 101 sağa 0'larla 23 bite tamamlanır: **101**0 0000 0000 0000 0000 000.
6. Alanlar birleştirilir: 1 10000001 10100000000000000000000 = 1100 0000 1101 0000 0000 … = **C0D00000h**.



Şekil 1.2.36: −6,5'in IEEE 754 deseni: işaret 1, üs 1000 0001 (129), kesir 101 ve sıfırlar; C0D00000h.

8086 belleğinde bu desen düşük adresten başlayarak 00 00 D0 C0 sırasıyla durur; en düşük bayt en düşük adrese yazılır (little-endian düzen).

1.2.7.4 Özel desenler

Üs alanının iki uç değeri (0 ve 255) özel anlamlara ayrılmıştır. Tablo 1.2.16 bunları listeler.

Desen	Anlam
e = 0, f = 0	±0 (işaret bitine göre +0 ya da −0)
e = 255, f = 0	±∞ (ör. 1/0)
e = 255, f ≠ 0	NaN, "sayı değil" (ör. 0/0)
e = 0, f ≠ 0	normalize edilmemiş, çok küçük sayı: $0{,}f \times 2^{-126}$

Tablo 1.2.16: IEEE 754'ün özel desenleri.

Sıfır, gizli bit kuralıyla yazılamaz; 1,f her zaman en az 1'dir. Bu yüzden sıfır için ayrı bir desen tanımlanmıştır. İşaret bitinin ayrı olması +0 ile −0'ı doğurur; standart ikisinin karşılaştırmada eşit sayılmasını ister. Sonsuz ve NaN değerleri, bir hesabın hata verip durmak yerine sürmesini sağlar. Normalize edilmemiş sayılar ise sıfıra yakın bölgedeki boşluğu doldurur. En küçük normalize sayı 2^{-126}'dır; bunun altındaki değerler gizli bit olmadan $0{,}f \times 2^{-126}$ olarak saklanır.

Kayan noktalı sayıların duyarlığı sınırlıdır. Tek duyarlıkta 24 bitlik anlamlı kısım, yaklaşık 7 onluk basamağa karşılık gelir. 0,1 gibi ikilide sonlu olmayan bir değer bu 24 bite yuvarlanarak saklanır: tek duyarlıkta 3DCCCCCDh deseni 0,100 000 001 49… değerini gösterir. Bu küçük hata, toplamalar art arda yapıldığında birikir.

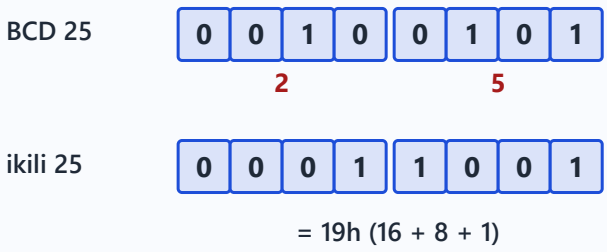
1.2.8 İkili kodlu onluk (BCD)

İkili sistem hesap için verimlidir, ama insanlar onluk okur. Bir sayıyı ekrana basmak için onu her seferinde 10'a bölüp basamaklarına ayırmak gerekir; bu da yavaş bir `DIV` dizisi ister. Sayıların sık sık gösterildiği ve az hesaplandığı uygulamalarda başka bir kodlama kullanılır: her onluk basamak ayrı ayrı ikili yazılır.

Tanım 1.2.18

İkili kodlu onluk (*binary-coded decimal, BCD*) kodlamada her onluk basamak dört bitle gösterilir: 0 = 0000, …, 9 = 1001. 1010 … 1111 desenleri geçersizdir. **Paketli** BCD bir bayta iki basamak koyar (25 → 25h); **paketsiz** BCD her bayta bir basamak koyar (25 → 02h 05h).

BCD ile ikili arasındaki fark bir örnekle görülür. 25 BCD olarak 0010 0101'dir: soldaki nibble 2'yi, sağdaki 5'i gösterir. Aynı sayının ikili karşılığı ise 0001 1001 = 19h'dir. Paketli BCD'de bir baytın onaltılık yazımı, onluk değerin rakamlarının aynısıdır: BCD 25 = 25h. Bu özellik, BCD sayıların bellek dökümünde okunmasını çok kolaylaştırır. Bedeli, alanın verimsiz kullanılmasıdır: bir bayt ikili olarak 0 … 255'i, paketli BCD olarak yalnız 0 … 99'u tutar.



Şekil 1.2.37: 25'in BCD ve ikili gösterimi. BCD'de her nibble bir onluk basamaktır (0010 0101); ikilide sayı bütün olarak yazılır (0001 1001 = 19h).

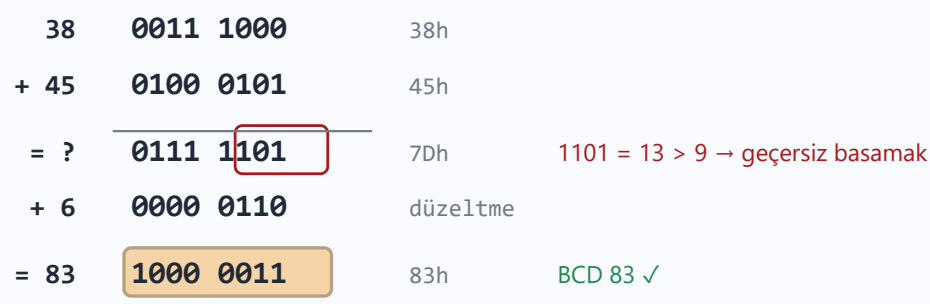
BCD, onluk basamakların doğrudan gösterildiği yerlerde kullanılır. Bilgisayarlardaki gerçek zamanlı saat (RTC) yongası saati, dakikayı ve tarihi BCD olarak tutar; 14:35, 14h ve 35h baytlarıyla saklanır. 7 bölütlü göstergeler bir BCD basamağı alıp ilgili bölütleri yakan kod çözücülerle sürülür. Hesap makineleri ve finansal yazılımlar, 0,1 gibi onluk kesirlerin tam saklanması için BCD aritmetiği kullanır.

1.2.8.1 BCD toplama ve düzeltme

İki paketli BCD sayı ikili toplayıcıyla toplanırsa sonuç her zaman geçerli BCD olmaz. Bir nibble'ın toplamı 9'u aşınca ikili toplayıcı onu 10 … 15 arasında bırakır; oysa onluk sistemde 10'a ulaşan basamak eldeye dönüşmelidir. Bu fark bir **düzeltme** ile giderilir: geçersiz basamağa 6 eklenir. 6 sayısı, bir nibble'ın 16 deseni ile onluk basamağın 10 değeri arasındaki farktır. 6 eklemek, kullanılmayan altı deseni (1010 … 1111) atlar ve eldeyi bir üst nibble'a geçirir.

Örnek 1.2.23

BCD 38 ile BCD 45'i toplayalım. 38 = 0011 1000, 45 = 0100 0101'dir. İkili toplam 0111 1101 = 7Dh'dir. Düşük nibble 1101 = 13'tür ve BCD'de geçersizdir. Bu nibble'a 6 eklenir: 7Dh + 06h = 83h = 1000 0011. Sonuç BCD **83**'tür ve 38 + 45 = 83 doğrudur.



Şekil 1.2.38: BCD 38 + 45. İkili toplam 7Dh'nin düşük basamağı (13) geçersizdir; 6 eklenince 83h, yani BCD 83 elde edilir.

Düzeltme gereken ikinci bir durum vardır: düşük nibble geçerli görünür ama aslında taşmıştır. 29 + 18 işleminde ikili toplam 29h + 18h = 41h'dir. Düşük nibble 1'dir ve geçerli görünür; oysa 9 + 8 = 17'dir. Düşük nibble'dan bir üst nibble'a elde çıkmış ve 16 değerini götürmüştür. 8086 bu eldeyi **yardımcı elde bayrağında** (AF) tutar. AF = 1 ise düşük nibble'a yine 6 eklenir: 41h + 6 = 47h, yani BCD 47 = 29 + 18. Yüksek basamak 9'u aşarsa ya da ondan elde çıkarsa ona da 6 eklenir, yani bayta 60h eklenir; bu durumda CF = 1 olur ve sonuç 99'u aşmıştır.

8086 bu kuralların hepsini tek bir komutla uygular: `DAA` (*decimal adjust after addition*). `ADD AL, BL` ile iki paketli BCD sayı toplandıktan hemen sonra `DAA` çalıştırılırsa AL geçerli BCD'ye düzeltilir. DAA, AL'nin düşük nibble'ına ve AF'ye, sonra yüksek nibble'ına ve CF'ye bakarak 06h, 60h ya da 66h ekler. Çıkarma için eşi `DAS`'tır. DAA yalnız AL üzerinde çalışır; çok basamaklı BCD sayılar bayt bayt, eldeyle (`ADC`) toplanıp her bayttan sonra DAA uygulanarak işlenir.

Sık yapılan hata

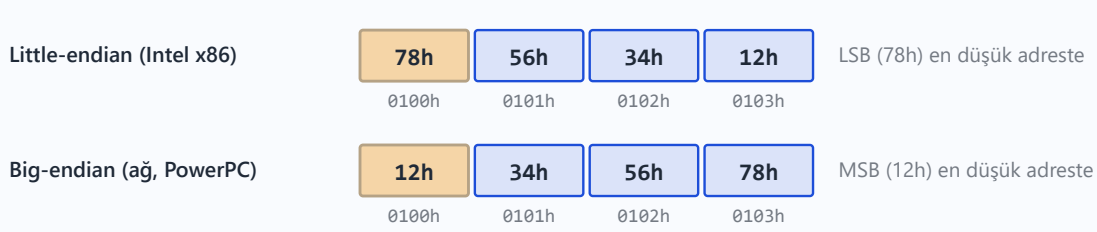
Yalnız “nibble 9'dan büyük mü?” sorusuna bakıp AF'yi unutmak. 29h + 18h = 41h'de düşük nibble 1'dir ve düzeltme gerekmiyor gibi görünür; ama sonuç 41 değil 47 olmalıdır. Bir başka hata, BCD sayıyı onaltılık sayı gibi okumaktır: 47h onaltılık olarak 71'dir, ama BCD olarak 47'dir. 2Fh ise onluk 47'nin ikili karşılığıdır ve BCD değildir.

1.2.9 Bayt sıralaması: little-endian ve big-endian

Bellek bayt bayt adreslenir. 16 ya da 32 bitlik bir değer belleğe yazılırken baytlarının hangi sırayla dizileceği bir tasarım kararıdır. İki karşıt gelenek vardır.

Tanım 1.2.19

Little-endian düzende çok baytlı bir değerin en az anlamlı baytı en düşük adrese yazılır. **Big-endian** düzende en anlamlı bayt en düşük adrese yazılır. 8086 ve bütün x86 ailesi little-endian'dır.



Şekil 1.2.39: 12345678h değerinin 0100h adresinden başlayarak iki düzende yerleşimi. Little-endian: 78 56 34 12; big-endian: 12 34 56 78.

Şekil 1.2.39 32 bitlik 12345678h'yi iki düzende gösterir. Little-endian'da 0100h adresinde 78h, 0103h adresinde 12h durur. Bellek dökümü soldan sağa okununca değer “ters” görünür. Big-endian'da ise baytlar yazıldığı sırayla durur ve döküm doğrudan okunur. Big-endian kullanan sistemler arasında Motorola 68000, çoğu PowerPC yapılandırması ve ağ protokolleri vardır. İnternet protokolleri çok baytlı alanları big-endian iletir; bu düzene “ağ bayt sırası” (*network byte order*) denir. Bir x86 makinesi ağdan gelen 16 bitlik bir port numarasını kullanmadan önce baytlarını yer değiştirmelidir.

Little-endian düzenin bir kolaylığı vardır: bir değerin adresi, değerin genişliğinden bağımsız olarak düşük baytı gösterir. 0100h adresindeki bir sözcüğün düşük baytını okumak için yine 0100h adresine bakılır. 8086'da `MOV AX, [0100h]` komutu AL'yi 0100h'den, AH'yi 0101h'den yükler. Aynı adresten `MOV AL, [0100h]` yalnız düşük baytı alır ve aynı değerin düşük baytını bulur.

Örnek 1.2.24

`DD 0A1B2C3D4h` ile DS:0300h adresine 32 bitlik A1B2C3D4h yazılmış olsun. Little-endian düzende baytlar 0300h: D4h, 0301h: C3h, 0302h: B2h, 0303h: A1h olur. Şimdi şu üç komutu düşünelim. `MOV AX, [0302h]` AL'yi 0302h'den (B2h), AH'yi 0303h'den (A1h) alır: AX = A1B2h. `MOV BL, [0301h]` tek bayt okur: BL = C3h. `MOV CX, [0301h]` CL'yi 0301h'den (C3h), CH'yi 0302h'den (B2h) alır: CX = B2C3h. Son okuma tek adresten başlayan bir sözcük okumasıdır; 8086 bunu iki ayrı yol döngüsüyle yapar ve fazladan dört saat turu harcar.

Sık yapılan hata

Little-endian'ı bitlerin ters dizilmesi sanmak. Sıralama bayt düzeyindedir; her baytın kendi içindeki bitler değişmez. 1234h belleğe 34 12 diye yazılır, 43 21 ya da 2C 48 diye değil. Bir başka hata, `DW 1234h` ile tanımlanmış bir sözcüğü `MOV AL, [adres]` ile okuyunca 12h beklemektir; düşük adreste 34h vardır.

1.2.10 Gray kodu

Sıradan ikili sayımda ardışık iki sayı arasında birden çok bit değişebilir. 3'ten 4'e geçerken 011 → 100 olur ve üç bit birden değişir. Bir döner kodlayıcının ya da bir sayacın çıkışında bu bitler fiziksel olarak tam aynı anda değişmez. Geçiş anında okunan değer, örneğin 111 (7) ya da 000 (0) gibi, ne 3 ne 4 olan bir ara değer olabilir. Gray kodu bu sorunu kökten çözer.

Tanım 1.2.20

Gray kodu (*Gray code*), ardışık her iki değeri arasında yalnız bir bitin değiştiği kodlamadır. İkili bir *B* sayısının Gray kodu *G = B XOR (B >> 1)* formülüyle bulunur.

Onluk	İkili	Gray	Değişen bit
0	`000`	`000`	—
1	`001`	`001`	1
2	`010`	`011`	2
3	`011`	`010`	1
4	`100`	`110`	3
5	`101`	`111`	1
6	`110`	`101`	2
7	`111`	`100`	1

Tablo 1.2.17: 3 bitlik ikili ve Gray kodları. “Değişen bit”, Gray ko-

dunda bir önceki satıra göre değişen bitin sağdan sırasıdır.

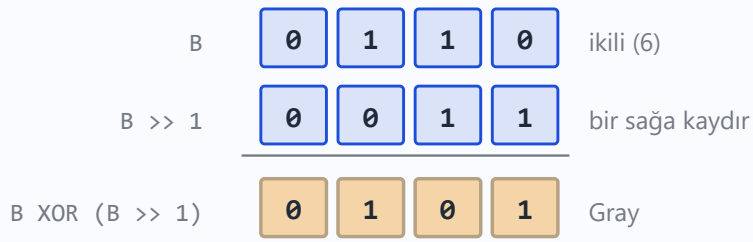
Tablo 1.2.17 3 bitlik diziyi verir: 000, 001, 011, 010, 110, 111, 101, 100. 3'ten 4'e geçiş Gray kodunda 010 → 110'dur; yalnız en üst bit değişir ve hatalı bir ara değer okunamaz. Dizi kendi içinde de kapanır: 7'den (100) 0'a (000) geçişte de tek bit değişir. Bu yüzden Gray kodu, bir milin açısını ölçen döner kodlayıcılarda (*rotary encoder*) çok kullanılır. Analog-sayısal dönüştürücülerin bazı türlerinde ve Karnaugh haritalarının satır-sütun sıralamasında da aynı kod karşımıza çıkar. Tek bit değişimi, geçişlerin elektriksel gürültüye ve zamanlama farklarına karşı dayanıklı olmasını sağlar.

1.2.10.1 İkiliden Gray koduna ve geri

Formülün anlamı şudur: Gray kodunun en yüksek biti ikiliyle aynıdır; her sonraki bit, ikilideki kendisi ile solundaki bitin XOR'udur. *B >> 1*, her bitin solundaki biti kendi konumuna getirir; XOR da iki biti karşılaştırır. Gray kodunda bir bit 1 ise ikilide o bit ile solundaki bit farklıdır.

Örnek 1.2.25

B = 0110 (6) için Gray kodunu bulalım. B bir bit sağa kaydırılır: B >> 1 = 0011. Bit bit XOR alınır: 0110 ⊕ 0011 = **0101**. 6'nın Gray kodu 0101'dir. Kuralla sağlama: en yüksek bit 0; sonra 0 ⊕ 1 = 1, 1 ⊕ 1 = 0, 1 ⊕ 0 = 1 → 0101.



Şekil 1.2.40: 6 = 0110'nın Gray kodu: B >> 1 = 0011 ve 0110 ⊕ 0011 = 0101.

Gray kodundan ikiliye dönüş ters yönde ilerler. İkilinin en yüksek biti Gray'in en yüksek bitine eşittir. Sonraki her bit, ikilide bir önce bulunan bit ile Gray'deki aynı konumdaki bitin XOR'udur: $B_i = B_{i+1} \oplus G_i$. 0101 için: $B_3 = 0$; $B_2 = 0 \oplus 1 = 1$; $B_1 = 1 \oplus 0 = 1$; $B_0 = 1 \oplus 1 = 0$. Sonuç 0110 = 6'dır. Dönüş, ileri yön gibi tek adımda değil, bitten bite ilerleyen bir zincirle yapılır.

1.2.10.2 Kod izleme: SHR ve XOR ile Gray kodu

Formül 8086'da üç komutla yazılır: değeri kopyala, kopyayı bir sağa kaydır, iki değeri XOR'la. Aşağıdaki program ardışık iki sayının, 11 ve 12'nin Gray kodunu hesaplar.

```
ORG 100h
MOV AL, 1011b  ; 11
MOV BL, AL
SHR BL, 1      ; B >> 1
XOR BL, AL     ; G
MOV AL, 1100b  ; 12
MOV BH, AL
SHR BH, 1
XOR BH, AL     ; G
RET
```

`MOV AL, 1011b` AL'ye 11 yükler. `MOV BL, AL` değeri BL'ye kopyalar; kaydırma kopya üzerinde yapılacak, B değeri AL'de korunacaktır. `SHR BL, 1` BL'yi bir sağa kaydırır: 0000 1011 → 0000 0101. En sağdaki 1 CF'ye düşer, en sola 0 girer. `XOR BL, AL` iki değeri bit bit karşılaştırır: 0101 ⊕ 1011 = 1110. BL = 0Eh, 11'in Gray kodudur. XOR komutu CF ve OF'yi her zaman 0 yapar; ZF ve SF'yi sonuca göre kurar. Programın ikinci yarısı aynı işi 12 için BH üzerinde yapar: 1100 → SHR → 0110 → XOR → 1010 = 0Ah.

Komut	AL	BL	BH	Açıklama
`MOV AL, 1011b`	0Bh	00h	00h	B = 11
`MOV BL, AL`	0Bh	0Bh	00h	kopya
`SHR BL, 1`	0Bh	05h	00h	0101b; düşen bit CF = 1
`XOR BL, AL`	0Bh	0Eh	00h	1011 ⊕ 0101 = 1110: Gray(11)
`MOV AL, 1100b`	0Ch	0Eh	00h	B = 12
`MOV BH, AL`	0Ch	0Eh	0Ch	kopya
`SHR BH, 1`	0Ch	0Eh	06h	0110b; düşen bit CF = 0
`XOR BH, AL`	0Ch	0Eh	0Ah	1100 ⊕ 0110 = 1010: Gray(12)

Tablo 1.2.18: Gray kodu programının izlemesi.



Şekil 1.2.41: Programın sonunda: Gray(11) = 1110 (BL), Gray(12) = 1010 (BH). İki kod yalnız bit 2'de farklıdır.

Sonuç, Gray kodunun temel özelliğini doğrular. 11'den 12'ye geçişte ikilide üç bit değişir: 1011 → 1100. Gray kodunda ise yalnız bir bit değişir: 1110 → 1010. Programda BL ve BH aynı BX yazmacının iki yarısıdır; program bitince BX = 0A0Eh olur. BH ve BL yazmaçları ayrı ayrı adreslenebildiği için iki 8 bitlik sonuç tek bir 16 bitlik yazmaçta yan yana durur.

1.2.11 Özet

- **Konumsal sistemde** bir sayının değeri, her rakamın taban üzeri konum ağırlığıyla çarpımlarının toplamıdır: $N = \Sigma\, d_i \times b^i$. Romen rakamlarında sıfır ve basamak olmadığı için aritmetik zordur.
- Bilgisayarlar iki durumlu elemanlarla kolay kurulduğu için **ikili** sistemi kullanır. n bit 2^n desen verir: bayt 0 … 255, sözcük 0 … 65.535.
- **Onaltılık** bir rakam tam bir nibble'dır; ikili ile onaltılık arasında sağdan dörtlü öbeklerle geçilir. Assembly'de `b`, `h`, `o` sonekleri tabanı belirtir; harfle başlayan onaltılık sayının önüne 0 konur. Taban yalnız yazım biçimidir: `65`, `41h` ve `'A'` aynı `B0 41` kodunu üretir.
- Onluktan başka tabana tam sayı kısmı **bölme-kalan** yöntemiyle çevrilir ve kalanlar aşağıdan yukarı okunur. Kesirli kısım **2 ile çarpma** yöntemiyle çevrilir ve bitler yukarıdan aşağı okunur. Paydası 2'nin kuvveti olmayan kesirler (0,1) ikilide sonlu değildir.
- `DIV BX` DX:AX'i böler, bölümü AX'e, kalanı DX'e yazar; DX önceden sıfırlanmalıdır. `SHR` 2'ye böler ve kalanı CF'ye düşürür; `RCR` CF'yi bir yazmaca taşır.
- Bit deseni anlam taşımaz; yorumu program verir. **İkiye tümleyende** $-x = NOT\ x + 1 = 2^n - x$'tir; `NEG` bunu tek komutta yapar. MSB işaret bitidir; tek bir sıfır vardır; 8 bitte aralık −128 … +127'dir. **Birler tümleyeninde** (`NOT`) ve işaret-büyüklükte iki sıfır bulunur.
- **İşaret genişletme** üst bitlere işaret bitini kopyalar (`CBW`, `CWD`): −5 = FBh = FFFBh. **Sabit noktalı** sayılarda virgülün yeri programcının kabulüdür; toplama sıradan tam sayı toplamasıdır.
- İkili toplama tam toplayıcılarla yapılır; çıkarma, ikiye tümleyenle toplamaya çevrilir. Onaltılıkta sütun toplamı 16'ya ulaşınca elde oluşur; 15'e ve 16'ya tümleyen, birler ve ikiye tümleyenin karşılığıdır.
- **CF** işaretsiz yorumda sonucun sığmadığını (elde ya da ödünç), **OF** işaretli yorumda sığmadığını gösterir. OF yalnız aynı işaretli sayılar toplanırken ya da farklı işaretli sayılar çıkarılırken 1 olabilir; donanım, işaret bitine giren ve çıkan eldeyi karşılaştırır.
- Bitler devrede gerilim düzeyleridir; eşik aralıkları gürültü payı sağlar. **Flip-flop** bir bit saklar; SR'de S = R = 1 yasaktır, JK'de J = K = 1 tersler. Yazmaçlar, saatin yükselen kenarında yüklenen D flip-flop'lardan kurulur.
- **IEEE 754** tek duyarlıklı sayı 1 işaret, 8 üs (127 kaydırmalı) ve 23 kesir bitinden oluşur; değer $(-1)^s \times 1{,}f \times 2^{e-127}$'dir. 8086'nın kayan nokta komutu yoktur; bu iş 8087'ye ya da yazılıma kalır.
- **BCD** her onluk basamağı dört bitle tutar; toplamadan sonra geçersiz basamağa ya da AF = 1 ise 6 eklenir (`DAA`). 8086 **little-endian**'dır: düşük bayt düşük adrestedir. **Gray kodunda** ardışık değerler tek bitte farklıdır: $G = B \oplus (B \gg 1)$.

1.2.12 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

konumsal sayı sistemi *positional number system* · taban *base (radix)* · basamak *digit* · onluk *decimal* · ikili *binary* · onaltılık *hexadecimal* · sekizli *octal* · sonek *suffix* · bit *bit* · nibble *nibble* · bayt *byte* · sözcük *word* · çift sözcük *double word* · en anlamlı / en az anlamlı bit *MSB / LSB* · işaretli / işaretsiz sayı *signed / unsigned* · işaret biti *sign bit* · işaret-büyüklük *sign-magnitude* · birler tümleyeni *one's complement* · ikiye tümleyen *two's complement* · işaret genişletme *sign extension* · sabit noktalı sayı *fixed-point number* · elde *carry* · ödünç *borrow* · taşma *overflow* · elde bayrağı (CF) *carry flag* · taşma bayrağı (OF) *overflow flag* · işaret bayrağı (SF) *sign flag* · sıfır bayrağı (ZF) *zero flag* · yardımcı elde bayrağı (AF) *auxiliary carry flag* · kaydırma *shift* · döndürme *rotate* · flip-flop *flip-flop* · saat turu *clock cycle* · kayan noktalı sayı (IEEE 754) *floating-point number* · işaret / üs / kesir *sign / exponent / fraction* · ikili kodlu onluk (BCD) *binary-coded decimal* · little-endian / big-endian *byte order* · Gray kodu *Gray code*

1.2.13 Kendini Dene

1. 300 sayısını onaltılık ve ikili olarak yazın. Bu değer AL'ye sığar mı?

Cevap

300 ÷ 16 = 18, kalan 12 (C); 18 ÷ 16 = 1, kalan 2; 1 ÷ 16 = 0, kalan 1. Sonuç 12Ch'dir; rakamlar 4 bite açılınca 1 0010 1100b bulunur. Değer 255'ten büyük olduğu için 8 bitlik AL'ye sığmaz; dokuz bit gerekir. AX'e sığar.

2. 0,3125 sayısını ikiliye çevirin. Sonuç sonlu mudur? Neden?

Cevap

0,3125 × 2 = 0,625 → 0; 0,625 × 2 = 1,25 → 1; 0,25 × 2 = 0,5 → 0; 0,5 × 2 = 1,0 → 1. Bitler yukarıdan aşağı okununca 0,0101b çıkar: 1/4 + 1/16 = 0,3125. Sonuç sonludur, çünkü 0,3125 = 5/16 ve payda 2'nin kuvvetidir.

3. 1101 0110b deseninin işaretsiz, ikiye tümleyen ve birler tümleyeni yorumları nedir?

Cevap

İşaretsiz: 128 + 64 + 16 + 4 + 2 = 214. İkiye tümleyen: 214 − 256 = −42 (ya da ağırlıklarla −128 + 64 + 16 + 4 + 2 = −42). Birler tümleyeni: bitler ters çevrilince 0010 1001 = 41 çıkar; yorum −41'dir.

4. AL = 80h iken `CBW` çalışıyor. AX ne olur? Aynı değer sıfırla genişletilseydi ne bulunurdu?

Cevap

AL'nin MSB'si 1'dir; CBW AH'yi FFh yapar ve AX = FF80h olur. Bu, 16 bitte −128'dir (65.536 − 128 = 65.408 = FF80h). Sıfırla genişletme 0080h = +128 verirdi; işaretli değer korunmazdı.

5. AL = 7Fh iken `ADD AL, 1` çalışıyor. AL, CF, OF, SF ve ZF ne olur? Sonucu işaretsiz ve işaretli yorumlayın.

Cevap

0111 1111 + 1 = 1000 0000, yani AL = 80h. En üst bitten elde çıkmaz: CF = 0. İşaret bitine elde girer ama çıkmaz: OF = 1. SF = 1, ZF = 0. İşaretsiz yorumda 127 + 1 = 128 doğrudur; işaretli yorumda +127 + 1 sonucu −128 olarak görünür ve taşmadır. Sayı çemberinde 0111'den 1000'e geçişin 8 bitlik karşılığıdır.

6. Q5.3 biçimindeki F4h hangi değeri gösterir?

Cevap

F4h = 11110,100b. Ağırlıklarla −16 + 8 + 4 + 2 + ½ = −1,5. Kısa yol: F4h = 244, işaretli olarak 244 − 256 = −12; −12 / 8 = −1,5.

7. 0,75 sayısını 32 bitlik IEEE 754 biçiminde yazın.

Cevap

0,75 = 0,11b = 1,1b × 2^{-1}. İşaret 0; üs −1 + 127 = 126 = 0111 1110b; kesir 1000…0. Desen 0 01111110 10000000000000000000000 = 0011 1111 0100 0000 … = 3F400000h.

8. AL = 66h ve BL = 57h paketli BCD sayılardır. `ADD AL, BL` ve ardından `DAA` çalışınca AL ve CF ne

olur?

Cevap

İkili toplam 66h + 57h = BDh'dir. Düşük nibble D (13) geçersizdir: 6 eklenir, C3h olur. Yüksek nibble C (12) de geçersizdir: 60h eklenir, 123h olur. AL = 23h, CF = 1. Sonuç BCD 123'tür: 66 + 57 = 123; yüzler basamağı CF'dedir.

9. DS:0200h'den başlayarak `DW 1234h, 0ABCDh` tanımlanmış. `MOV AX, [0201h]` sonrası AX nedir?

Cevap

Little-endian düzende baytlar 0200h: 34h, 0201h: 12h, 0202h: CDh, 0203h: ABh'dir. AL, 0201h'den (12h), AH, 0202h'den (CDh) yüklenir: AX = CD12h. Okuma tek adresten başladığı için iki yol döngüsü sürer.

10. Gray kodu 1001 olan 4 bitlik sayının ikili karşılığı nedir? Sonucu formülle sağlayın.

Cevap

$B_3 = G_3 = 1$; $B_2 = 1 \oplus 0 = 1$; $B_1 = 1 \oplus 0 = 1$; $B_0 = 1 \oplus 1 = 0$. İkili karşılık 1110 = 14'tür. Sağlama: $1110 \oplus 0111 = 1001$.

1.2.14 İleri okuma

- M. M. Mano ve M. D. Ciletti, *Digital Design* (5. baskı), Bölüm 1, 4 ve 5: sayı sistemleri, tümleyenler, BCD ve Gray kodu ile toplayıcı devreler ve flip-flop'lar, sayısal tasarım bakış açısından ayrıntılı ele alınır.
- B. B. Brey, *The Intel Microprocessors* (8. baskı), Bölüm 1: sayı sistemleri, işaretli sayılar, BCD ve bayt sıralaması doğrudan 8086 ailesinin veri biçimleriyle ilişkilendirilir.
- D. Goldberg, “What Every Computer Scientist Should Know About Floating-Point Arithmetic”, *ACM Computing Surveys*, 23(1), 1991: IEEE 754'ün yuvarlama, özel değerler ve hata birikimi konularını açıklayan klasik makaledir.
- D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Kesim 4.1: konumsal sayı sistemlerinin tarihini ve taban dönüşümü yöntemlerini kanıtlarıyla verir.

1.2.15 Kaynaklar

1. M. Morris Mano, Michael D. Ciletti, *Digital Design*, 5. baskı, Pearson, 2013.
2. Barry B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson, 2009, Bölüm 1.
3. IEEE Std 754-2019, *IEEE Standard for Floating-Point Arithmetic*.

4. emu8086 belgeleri, *Numbering systems tutorial*.
5. Mehmet Bodur, *Mikroişlemcilere Giriş: Assembler ile Yazılım ve Arayüz*, EMO, 2014.
6. Şadi Çağatay Öztürk, Oğuz Ergin, *Intel 8086 ile Mikroişlemci Programlamaya Giriş*, 2009.
7. David Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic", *ACM Computing Surveys*, 23(1):5–48, 1991.
8. Donald E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3. baskı, Addison-Wesley, 1997.

KISIM I · BÖLÜM 2

8086 Mimarisi ve Yazmaçlar

Intel 8086, 1978'de tanıtılan ve x86 ailesini başlatan 16 bitlik bir mikroişlemcidir. Bölümde sistem yolları, yol arayüz birimi (BIU) ile yürütme birimi (EU), önden getirme kuyruğu ve kesimli bellek düzeninde 20 bitlik fiziksel adresin hesaplanması anlatılır. Ardından 8086'nın on dört yazmacı gruplarıyla incelenir: AX, BX, CX ve DX genel amaçlı yazmaçları, SP, BP, SI ve DI işaretçi ve indis yazmaçları, kesim yazmaçları ve IP. Durum ve kontrol bayraklarını taşıyan bayrak yazmacı ile bu bayrakları değiştiren komutlar da ele alınır.

Kod (8086): `github.com/sercankulcu/microprocessors-8086/Bolum02`

BU BÖLÜMÜN NOTLARI

2.1 8086 Mimarisi

2.2 8086 Yazmaçları

KISIM I · BÖLÜM 2.1

8086 Mimarisi

Assembly dilinde yazılan her satır, işlemcinin içindeki somut bir donanım parçasını çalıştırır: bir yazmaca değer yükler, ALU'ya toplama yaptırır ya da adres yoluna 20 bitlik bir adres koyar. Bu yüzden 8086'yı programlamadan önce içini tanımak gerekir. Bu not, Intel 8086'nın genel özellikleriyle başlar ve işlemcinin belleğe ve G/Ç aygıtlarına yollar üzerinden nasıl bağlandığını anlatır. Ardından iki iç birimi, yol arayüz birimini (BIU) ve yürütme birimini (EU), önden getirme kuyruğunu ve boru hattını inceler. ALU'nun işlemlerini ve komutların bellekteki bayt uzunluklarını kod izlemeleriyle gösterir. Son kesimler yazmaç kümesini ve 8086'nın en ayırt edici özelliği olan kesimli belleği ele alır: 16 bitlik iki değerden 20 bitlik fiziksel adresin nasıl üretildiğini, aynı baytın neden birçok adla anılabildiğini ve hangi erişimin hangi kesimi kullandığını.

ÖĞRENME ÇIKTILARI

Genel özellikleri açıklamak
8086'nın veri yolu ve adres yolu genişliğini, adresleyebildiği bellek miktarını, x86 ailesindeki yerini ve bir 8086 sisteminin hangi yongalardan oluştuğunu söylemek.

Yolları izlemek
Adres, veri ve kontrol yollarının görevini ayırmak; bir bellek ya da G/Ç okuma ve yazma işleminde yollarda hangi değerlerin bulunduğunu kod üzerinde göstermek.

İç yapıyı anlatmak
BIU ve EU birimlerinin görevlerini, önden getirme kuyruğunun işleyişini, boru hattının kazancını ve dallanmanın kuyruğa etkisini açıklamak; bir komutun getirilip yürütülmesini blok diyagram üzerinde izlemek.

Makine kodunu okumak
Bir komutun işlem kodu, ModR/M baytı, yer değiştirme ve anlık veri alanlarını ayırmak; komut uzunluklarından IP'nin her adımdaki değerini bulmak.

Yazmaçları gruplamak
14 yazmacı gruplarına ayırmak; kesim yazmaçlarının kullanım kurallarını ve IP'nin görevini açıklamak.

Fiziksel adres hesaplamak
`kesim × 10h + bağıl konum` ile 20 bitlik fiziksel adresi bulmak; bir adresi normalleştirmek; 1 MB sınırındaki sarmayı ve varsayılan kesim kurallarını kod izlemeleriyle göstermek.

2.1.1 8086'ya genel bakış

Intel 8086, 1978'de piyasaya çıkan 16 bitlik bir mikroişlemcidir. Bugün masaüstü ve dizüstü bilgisayarlarda çalışan x86 işlemci ailesinin ilk üyesidir. Kırk yılı aşkın sürede bu ailenin işlemcileri 32 ve 64 bite büyüdü, saat hızları binlerce kat arttı. Yine de her yeni üye, 8086 için yazılmış komutları çalıştırmayı sürdürdü. Bu yüzden 8086'nın yazmaçlarını, adres hesabını ve komut biçimini öğrenmek, bugünkü işlemcilerin en alt katmanını öğrenmek demektir.

Bu kesim önce 8086'nın temel sayılarını verir: kaç bitlik veriyle çalıştığını, ne kadar bellek adresleyebildiğini, hangi teknolojiyle üretildiğini. Ardından 8086'nın ailedeki yerini ve 20 bitlik adres yolunun neden seçildiğini anlatır. Son olarak bir mikroişlemcinin tek başına bir bilgisayar olmadığını, çevresinde hangi yongalara gerek duyduğunu gösterir.

2.1.1.1 Temel özellikler

Bir işlemciyi "16 bitlik" diye anmak, onun aynı anda üzerinde çalışabildiği verinin genişliğini anlatır. 8086'nın genel amaçlı yazmaçları 16 bittir. Aritmetik-mantık birimi (ALU, *arithmetic logic unit*) iki 16 bitlik sayıyı tek işlemde toplar. Dış veri yolu da 16 hatlıdır, yani işlemci bellekle tek seferde 2 bayt alışverişi yapar. Bu üç genişliğin birbirine eşit olması 8086'yı "tamamen 16 bitlik" bir işlemci yapar.

Adres yolu ise 20 hatlıdır. Her hat bir bit taşıdığı için işlemci 2^{20} farklı adres üretebilir. 2^{20} = 1.048.576 bayttır, yani 1 MB. Adresler onaltılık olarak beş basamakla yazılır: en düşük adres `00000h`, en yüksek adres `FFFFFh`'dir.

Özellik	Değer	Anlamı
Veri yolu	16 bit	bir yol döngüsünde en çok 2 bayt taşınır
Adres yolu	20 bit	2^{20} bayt = 1 MB bellek adreslenir
Yazmaçlar, ALU	16 bit	sözcük boyu 16 bittir
Paket	40 pin DIP	çift sıralı plastik ya da seramik kılıf
Saat	5–10 MHz	8086: 5 MHz, 8086-2: 8 MHz, 8086-1: 10 MHz
Transistör	yaklaşık 29.000	HMOS (yüksek yoğunluklu n kanallı MOS) teknolojisi
Besleme	+5 V	tek besleme gerilimi
Mimari	CISC	1–6 bayt uzunlukta komutlar, bellek üzerinde işlem yapan komutlar

Tablo 2.1.1: 8086'nın temel özellikleri.

Tablo 2.1.1 bu sayıları toplar. Saat hızı, işlemcinin içindeki bütün işlemleri eş zamanlayan kare dalganın frekansıdır. 5 MHz'lik bir saatin bir periyodu 1 / 5.000.000 s = 200 ns sürer. Bu süreye **saat turu** (*clock cycle*) denir. 8086'da en basit komut birkaç saat turunda biter; çarpma ve bölme gibi karmaşık komutlar yüzlerce tur sürebilir.

CISC (*complex instruction set computer*, karmaşık komut kümeli bilgisayar) tasarımında komutlar farklı uzunluklardadır ve bir komut doğrudan bellekteki bir işlenenle çalışabilir. Örneğin `ADD AX, [BX]` komutu bellekten bir sözcük okur ve AX'e ekler. Bu tasarım bellek pahalıyken programları kısa tutmayı amaçlıyordu: sık kullanılan işler tek bir komuta sığdırılırdı.

8086'nın bir de kardeşi vardır: 8088. 8088'in yazmaçları ve ALU'su 16 bittir, ama dış veri yolu 8 hatlıdır. Bir sözcüğü bellekten okumak için iki ayrı yol döngüsü yapar. IBM, 1981'deki ilk kişisel bilgisayarında 8088'i seçti; 8 bitlik yol, o yılların ucuz 8 bitlik çevre yongalarıyla kolayca birleşiyordu. Programcı açısından iki işlemci aynı komutları çalıştırır.

> **Sık yapılan hata**
>
> "16 bitlik işlemci 64 KB bellek adresler" düşüncesi yanlıştır. Veri yolunun genişliği bir seferde ne kadar veri taşındığını, adres yolunun genişliği ne kadar bellek seçilebildiğini belirler. İkisi birbirinden bağımsızdır. 8086'nın verisi 16 bit, adresi 20 bittir; bu yüzden 64 KB değil 1 MB adresler.

2.1.1.2 x86 ailesindeki yeri

8086'dan sonra gelen işlemciler, öncekiler için yazılmış programları değiştirmeden çalıştırabilecek biçimde tasarlandı. Bu ilkeye **geriye dönük uyumluluk** (*backward compatibility*) denir. Tablo 2.1.2 ailenin ilk beş üyesini karşılaştırır.

İşlemci	Yıl	İç / dış veri yolu	Adres yolu	Bellek	Belirgin özellik
8086	1978	16 / 16 bit	20 bit	1 MB	6 baytlık önden getirme kuyruğu
8088	1979	16 / 8 bit	20 bit	1 MB	4 baytlık kuyruk; IBM PC (1981)
80186	1982	16 / 16 bit	20 bit	1 MB	tümleşik çevre birimleri
80286	1982	16 / 16 bit	24 bit	16 MB	korumalı mod
80386	1985	32 / 32 bit	32 bit	4 GB	32 bitlik x86 (IA-32)

Tablo 2.1.2: x86 ailesinin ilk üyeleri.

Tablodaki bellek sütunu doğrudan adres yolundan gelir: 24 bit 2^{24} = 16.777.216 bayt (16 MB), 32 bit 2^{32} = 4.294.967.296 bayt (4 GB) eder. 80186, 8086'nın çekirdeğine saat üreteci, zamanlayıcı ve kesme denetleyicisi gibi çevre birimlerini ekleyerek gömülü sistemler için tek yongada daha çok iş toplar. 80286 korumalı modu getirir: bu modda programlar birbirinin belleğine erişemez ve 16 MB'ın tamamı kullanılabilir.

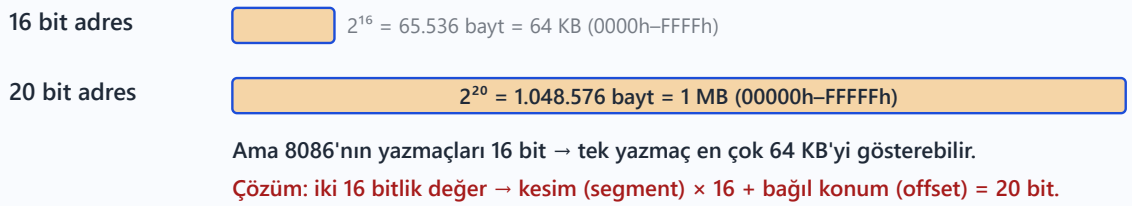
Tanım 2.1.1

Gerçek mod (*real mode*), x86 işlemcilerin 8086 gibi davrandığı çalışma modudur: adresler 16 bitlik kesim ve bağıl konum çiftinden 20 bitlik fiziksel adres olarak hesaplanır, bellek koruması yoktur ve doğrudan erişilebilen alan 1 MB'tır. 80286'dan bu yana her x86 işlemci açıldığında gerçek modda başlar.

Bu derste emu8086 ile yazılan bütün programlar 8086 gerçek modunda çalışır. Bugünkü bir bilgisayarda da açılışın ilk anlarında işlemci gerçek moddadır; işletim sistemi daha sonra onu korumalı ya da 64 bitlik moda geçirir.

2.1.1.3 Neden 20 bit adres?

Bellek, her biri bir bayt tutan hücrelerin sıralanmasıdır. Her hücrenin bir numarası, yani adresi vardır. İşlemci bir hücreye erişmek için onun adresini adres yoluna koyar. n hatlı bir adres yolu 2^n farklı bit deseni taşıyabilir, bu yüzden 2^n farklı bayt seçebilir.



Şekil 2.1.1: 16 bitlik adres 64 KB'ı, 20 bitlik adres 1 MB'ı seçer. 8086'nın yazmaçları 16 bit olduğu için 20 bitlik adres iki 16 bitlik değerden üretilir.

Örnek 2.1.1

16 ve 20 bitlik adres yollarının kapasitesini karşılaştıralım.

1. 16 bit: 2^{16} = 65.536 bayt. 1 KB = 2^{10} = 1024 bayt olduğundan 65.536 / 1024 = 64 KB. Adresler `0000h` – `FFFFh`.
2. 20 bit: 2^{20} = 1.048.576 bayt. 1 MB = 2^{20} bayt olduğundan tam 1 MB. Adresler `00000h` – `FFFFFh`.
3. Oran: $2^{20} / 2^{16} = 2^4 = 16$. Dört ek hat, adreslenebilen belleği 16 katına çıkarır.

Her onaltılık basamak dört bittir; 20 bit tam beş basamak eder. Bu yüzden fiziksel adresler hep beş basamakla yazılır.

8086'nın tasarımcıları burada bir çelişkiyle karşılaştı. 1978'de 64 KB bellek artık dar geliyordu; daha büyük bir adres alanı gerekiyordu. Ama işlemcinin yazmaçları 16 bitti ve tek bir yazmaç ancak 64 KB'lık bir aralığı gösterebilirdi. Yazmaçları 20 bite çıkarmak yongayı büyütür ve 8 bitlik işlemcilerden gelen yazılımla uyumu bozardı. Çözüm, adresi iki 16 bitlik değerden üretmek oldu: biri bir bellek bölgesinin, yani **kesimin** (*segment*) başlangıcını, öteki o bölge içindeki uzaklığı, yani **bağıl konumu** (*offset*) verir. İşlemci kesim değerini 16 ile çarpar ve bağıl konumu ekler. Bu hesap 20 bitlik bir sonuç verir. Şekil 2.1.1 bu fikri özetler.

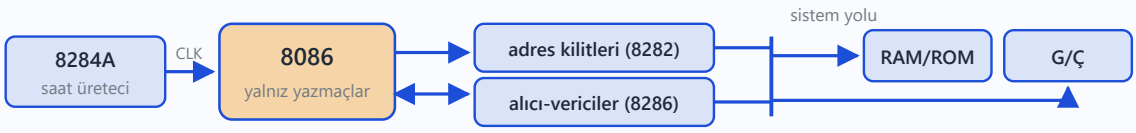
2.1.1.4 Mikroişlemci ve mikrodenetleyici

8086 bir **mikroişlemcidir**: merkezî işlem biriminin (CPU, *central processing unit*) bütün işlevlerini tek bir tümleşik devrede toplar. Ama içinde program ya da veri saklayacak bir bellek yoktur. Yonganın içinde yalnız yazmaçlar bulunur; ara ve son sonuçlar bu yazmaçlarda tutulur. Program kodu, değişkenler ve yığın dışarıdaki RAM ve ROM yongalarındadır. İşlemci bu belleğe ve G/Ç (giriş/çıkış) aygıtlarına **sistem yolu** üzerinden erişir.

Tanım 2.1.2

Mikroişlemci, CPU'nun komut getirme, çözme ve yürütme işlevlerini tek yongada toplayan, ama bellek ve G/Ç birimlerini dışarıda bırakan tümleşik devredir. **Mikrodenetleyici** (MCU, *microcontroller*) ise CPU ile birlikte program belleğini (ROM ya da Flash), RAM'i, zamanlayıcıları ve G/Ç portlarını aynı yongada taşır.

İki yaklaşımın kullanım alanı farklıdır. Bir mikrodenetleyici (8051, AVR, PIC) tek başına bir sistem gibi çalışabilir: besleme ve bir kristal bağlanınca programını kendi belleğinden okur, portlarından lambaları yakar, sensör okur. Çamaşır makinesi, uzaktan kumanda ve sensör düğümü gibi gömülü uygulamalarda düşük maliyeti ve düşük güç tüketimi öne çıkar. İşlem gücü ise genellikle sınırlıdır. Mikroişlemci ise esnektir: bellek miktarı ve çevre birimleri uygulamaya göre seçilir, ama sistem birçok yongadan oluşur.



Şekil 2.1.2: 8086 tabanlı bir sistemin yongaları: 8284A saat üreteci, 8086, adres kilitleri (8282), veri yolu alıcı-vericileri (8286), bellek ve G/Ç birimleri.

Şekil 2.1.2 en küçük 8086 sisteminin parçalarını gösterir. Her yonganın görevi şöyledir:

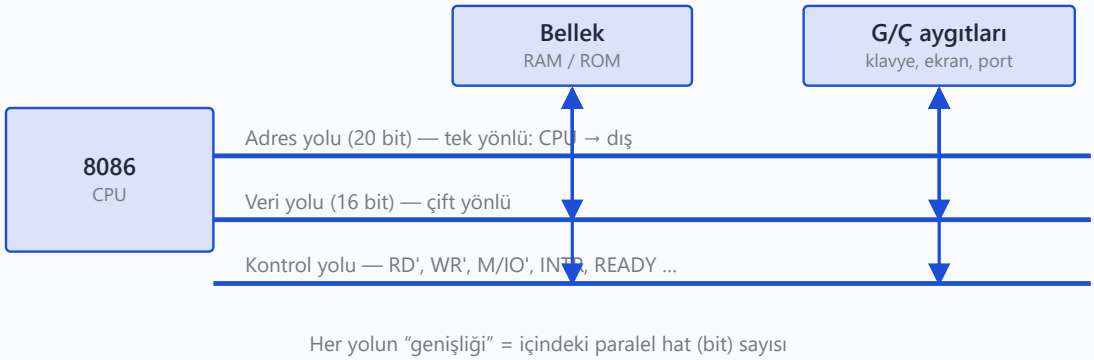
- **8284A saat üreteci** 8086'ya saat sinyalini (CLK) verir. Dışarıdaki kristalin frekansını üçe böler: 15 MHz'lik bir kristal 5 MHz'lik saat üretir. Ayrıca RESET ve READY sinyallerini saatle eş zamanlar.

- **Adres kilitleri** (*latch*, 8282 ya da 74LS373) adresi yakalayıp yol döngüsü boyunca tutar. 8086'nın 40 pini her sinyale ayrı bir pin vermeye yetmez. Bu yüzden adresin alt 16 biti ile veri aynı pinlerden (AD0–AD15) farklı zamanlarda çıkar. Döngünün başında pinlerde adres vardır; işlemci ALE (*address latch enable*) darbesi üretir ve kilitler adresi yakalar. Sonra aynı pinler veriye geçer, ama bellek kilitteki adresi görmeye devam eder. 20 bit için 8 bitlik üç kilit gerekir.
- **Veri yolu alıcı-vericileri** (*transceiver*, 8286) veri hatlarını güçlendirir ve yönünü belirler: okumada bellekten işlemciye, yazmada işlemciden belleğe. 16 bit için iki tane kullanılır.
- **Bellek** (RAM ve ROM) program ve veriyi tutar; **G/Ç birimleri** klavye, ekran ve portlar gibi dış dünyayla bağlantıyı kurar.

Bu ek yongalar mikroişlemcinin bedelidir. Buna karşılık sistemin belleği 1 MB'a kadar büyütülebilir ve her türlü çevre birimi eklenebilir. Kişisel bilgisayarlar bu yolu seçti.

2.1.2 Sistem yolları

İşlemci, bellek ve G/Ç aygıtları birbirine paralel tel demetleriyle bağlanır. Her demete **yol** (*bus*) denir. Bir yolun genişliği içindeki hat sayısıdır; her hat aynı anda bir bit taşır. Sistemde üç yol vardır ve her birinin görevi ayrıdır.



Şekil 2.1.3: İşlemci, bellek ve G/Ç aygıtları adres, veri ve kontrol yollarıyla bağlanır. Adres yolu tek yönlü, veri yolu çift yönlüdür.

2.1.2.1 Adres, veri ve kontrol yolu

Tanım 2.1.3

Sistem yolu (*system bus*), CPU, bellek ve G/Ç aygıtlarını birbirine bağlayan üç yolun bütünüdür. **Adres yolu** erişilecek bellek konumunu ya da G/Ç portunu seçer ve yalnız işlemciden dışarı doğru çalışır. **Veri yolu** komut ve veri baytlarını iki yönde taşır. **Kontrol yolu** işlemin türünü (okuma, yazma, bellek ya da G/Ç) ve zamanlamasını bildiren sinyallerden oluşur.

Yol	Ne taşır?	Yön	8086'da
Adres	okunacak ya da yazılacak bellek konumu, G/Ç port numarası	CPU → dış	20 bit → 1 MB bellek; G/Ç için 16 bit → 64 K port
Veri	işlemci ile bellek ya da G/Ç arasındaki komut ve veri baytları	çift yönlü	16 bit, tek seferde bir sözcük (2 bayt)
Kontrol	okuma, yazma, bellek/G/Ç seçimi, kesme istekleri, durum bilgisi	sinyale göre	RD', WR', M/IO', ALE, INTR, READY …

Tablo 2.1.3: 8086'nın üç yolu.

Adresi yalnız işlemci üretir; bellek ya da G/Ç aygıtı adres yoluna hiçbir şey koymaz. Bu yüzden adres yolu tek yönlüdür. Veri ise okumada dışarıdan işlemciye, yazmada işlemciden dışarıya akar; veri yolu iki yönlüdür. Kontrol yolundaki sinyallerin bir kısmını işlemci üretir (RD', WR', M/IO', ALE), bir kısmını dış aygıtlar üretir (INTR kesme isteği, READY hazır sinyali).

Sinyal adlarının sonundaki kesme işareti (') sinyalin **etkin düşük** olduğunu gösterir: sinyal, mantık 0 düzeyindeyken görevini yapar. RD' = 0 "şimdi oku", WR' = 0 "şimdi yaz" anlamına gelir. M/IO' sinyali erişimin hedefini seçer: 1 iken bellek, 0 iken G/Ç. Bellek yongaları yalnız M/IO' = 1 iken, G/Ç aygıtları yalnız M/IO' = 0 iken yanıt verir. Böylece aynı adres ve veri hatları iki ayrı adres uzayına hizmet eder.

G/Ç adres uzayı bellekten küçüktür. G/Ç erişiminde adresin yalnız alt 16 biti (A15–A0) kullanılır, A19–A16 hatları 0 olur. 2^{16} = 65.536 port numarası vardır. `IN` ve `OUT` komutları port numarasını ya komutun içinde 8 bitlik bir sabit olarak (0–FFh) ya da DX yazmacında (0–FFFFh) alır.

2.1.2.2 Yol döngüsü

İşlemcinin bellekten ya da G/Ç'den bir kez okuması ya da bir kez yazması bir **yol döngüsü** (*bus cycle*) sürer. 8086'da bir yol döngüsü en az dört saat turundan oluşur. Bu turlara T durumları denir ve T1, T2, T3, T4 diye adlandırılır.

> **Tanım 2.1.4**
>
> **Yol döngüsü**, bellek ya da G/Ç ile tek bir okuma veya yazma işleminin baştan sona sürdüğü zaman dilimidir. 8086'da dört T durumundan (T1–T4) oluşur; yavaş bir aygıt READY sinyalini 0 tutarak T3 ile T4 arasına bekleme durumları (Tw) ekletebilir.

Bir bellek okuma döngüsü şu sırayla ilerler:

1. **T1:** İşlemci 20 bitlik adresi adres hatlarına koyar, M/IO' ile hedefi bildirir ve ALE darbesi verir. Kilitler adresi yakalar.
2. **T2:** AD0–AD15 pinleri adresten çekilir ve veri için serbest kalır. İşlemci RD' sinyalini 0 yapar: "oku".
3. **T3:** Bellek, adresteki baytı ya da sözcüğü veri yoluna koyar. İşlemci READY'yi yoklar; aygıt hazırsa veriyi alır.

4. **T4:** RD' yeniden 1 olur, bellek veri yolunu bırakır. Döngü biter.

Yazma döngüsünde sıra aynıdır; farkı, veriyi işlemcinin T2'den başlayarak veri yoluna koyması ve RD' yerine WR' sinyalini 0 yapmasıdır. Bellek, WR' etkinken veri yolundaki değeri hücresine yazar.

Örnek 2.1.2

5 MHz'lik bir 8086 bekleme durumu olmadan bellekten 100 sözcük okuyor. En az ne kadar sürer?

1. Bir saat turu: 1 / 5 MHz = 200 ns.
2. Bir yol döngüsü: 4 × 200 ns = 800 ns.
3. Sözcükler çift adreslerde duruyorsa her sözcük tek döngüde gelir: 100 × 800 ns = 80.000 ns = 80 µs.

Bu, yolun en yüksek aktarım hızının 2 bayt / 800 ns = 2,5 MB/s olduğunu gösterir. Sözcükler tek adreslerde dursaydı her biri iki döngü alır, süre 160 µs olurdu. 8086'nın belleği çift ve tek adresli iki bayt bankasına bölünmüştür. Tek adresten başlayan bir sözcüğün baytları farklı satırlarda kalır ve tek döngüde birlikte okunamaz.

Bu süreler yazmaçlarla karşılaştırılınca bir şey açıkça görülür: bellek yavaştır. Bir yazmaçtan yazmaca `MOV AX, BX` 8086'da 2 saat turu sürer. Aynı değeri bellekten okuyan `MOV AX, [BX]` ise adres hesabı ve yol döngüsüyle birlikte 13 tur sürer. Bu yüzden sık kullanılan değerler yazmaçlarda tutulur ve 8086'nın iç yapısı bellek beklemesini gizlemek için düzenlenmiştir.

2.1.2.3 Kod izleme: yollar üzerinde bellek ve G/Ç

Aşağıdaki program dört erişim yapar: bellekten okur, G/Ç portuna yazar, G/Ç portundan okur ve belleğe yazar. Her erişimde yollarda hangi değerlerin bulunduğunu izlemek, adres, veri ve kontrol yollarının görevini somutlaştırır. Programın başında bellekte 10200h adresinde 5Ah, 51h numaralı G/Ç portunda 3Ch değerinin bulunduğu varsayılır.

```
org 100h
mov ax, 1000h
mov ds, ax
mov bx, 0200h
mov al, [bx]     ; bellek oku
out 50h, al      ; G/Ç yaz
in al, 51h       ; G/Ç oku
mov [bx+1], al   ; bellek yaz
ret
```

`org 100h` bir komut değil, assembler'a verilen bir yönergedir. Programın .COM biçiminde, kesimin 100h bağıl konumundan başlayacağını bildirir. İlk iki komut DS yazmacına 1000h yükler. Değer önce AX'e, sonra AX'ten DS'ye taşınır; çünkü 8086'da bir kesim yazmacına doğrudan sabit yüklenemez. DS = 1000h olunca veri kesimi fiziksel 10000h adresinden başlar. `mov bx, 0200h` BX'e bağıl konumu koyar. Bu üç komut yalnız yazmaçları değiştirir; veri için yol kullanılmaz. Yollar yalnız komut baytlarının getirilmesi için çalışır.

`mov al, [bx]` ilk veri erişimidir. Köşeli parantez "BX'in gösterdiği bellek konumu" anlamına gelir. İşlemci fiziksel adresi DS × 10h + BX = 10000h + 0200h = 10200h olarak hesaplar ve bir bellek okuma döngüsü başlatır: adres yolunda 10200h, M/IO' = 1, RD' = 0. Bellek 5Ah'yi veri yoluna koyar, işlemci onu AL'ye yazar.

`out 50h, al` AL'deki değeri 50h numaralı porta gönderir. Bu bir G/Ç yazma döngüsüdür. Adres yolunda yalnız port numarası bulunur: 0050h. M/IO' = 0 olduğu için bellek yongaları bu döngüyü yok sayar; WR' = 0 ve veri yolunda 5Ah vardır. Port 50h'ye bağlı aygıt değeri alır.

`in al, 51h` bir G/Ç okuma döngüsüdür: adres 0051h, M/IO' = 0, RD' = 0. Aygıt 3Ch'yi veri yoluna koyar ve AL = 3Ch olur. Son veri komutu `mov [bx+1], al` belleğe yazar. Etkin adres BX + 1 = 0201h, fiziksel adres 10000h + 0201h = 10201h'dir. Döngüde M/IO' = 1, WR' = 0 ve veri yolunda 3Ch bulunur. `ret` programı bitirip denetimi işletim sistemine geri verir.

Komut	AL	Adres yolu	Veri yolu	M/IO'	RD'	WR'
`mov ax,1000h` / `mov ds,ax`	00h	—	—	—	—	—
`mov bx,0200h`	00h	—	—	—	—	—
`mov al,[bx]`	5Ah	10200h	5Ah (bellekten)	1	0	1
`out 50h,al`	5Ah	0050h	5Ah (dışarı)	0	1	0
`in al,51h`	3Ch	0051h	3Ch (aygıttan)	0	0	1
`mov [bx+1],al`	3Ch	10201h	3Ch (dışarı)	1	1	0

Tablo 2.1.4: Yollar programının izlemesi. "—" o komutta veri için yol döngüsü olmadığını gösterir.



Şekil 2.1.4: Solda bellek okuma (`mov al,[bx]`): adres 10200h, M/IO' = 1, RD' = 0. Sağda G/Ç yazma (`out 50h,al`): adres 0050h, M/IO' = 0, WR' = 0.



Şekil 2.1.5: Solda G/Ç okuma (`in al,51h`): aygıt 3Ch'yi veri yoluna koyar. Sağda bellek yazma (`mov [bx+1],al`): 3Ch, 10201h adresine yazılır.

Tablo 2.1.4, Şekil 2.1.4 ve Şekil 2.1.5 dört erişimin ortak noktasını gösterir: hepsi aynı adres ve veri hatlarını kullanır. Bellek ile G/Ç'yi M/IO', okuma ile yazmayı RD' ve WR' ayırır. Bellek erişiminde adres yolunda 20 bitlik fiziksel adres, G/Ç erişiminde 16 bitlik port numarası bulunur. Program emu8086'da çalıştırıldığında da AL'nin son değeri 3Ch, 10201h'deki bayt 3Ch ve port 50h'ye yazılan değer 5Ah çıkar.

Sık yapılan hata

`out 50h, al` komutunun 50h adresli bellek hücresine yazdığını düşünmek yanlıştır. Port numarası ile bellek adresi farklı adres uzaylarındadır. M/IO' = 0 olduğu için bellek yongaları bu döngüye hiç yanıt vermez; 00050h'deki bellek baytı değişmez. 8088'de bu sinyalin adı ve kutbu terstir (IO/M'), ama ayrım ilkesi aynıdır.

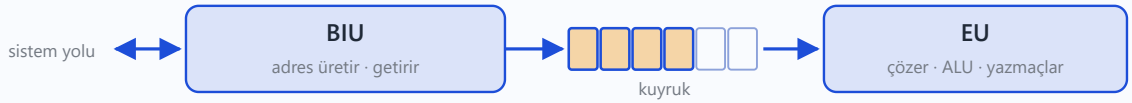
2.1.3 İç yapı: BIU ve EU

Basit bir işlemci her komut için aynı üç adımı sırayla yapar: komutu bellekten getirir, ne yapacağını çözer ve yürütür. Getirme sırasında işlemcinin hesap birimleri boş bekler; yürütme sırasında ise yol boş kalır. 8086 bu boşluğu kapatmak için içini iki bağımsız birime böler. Biri yolla ilgili bütün işleri yapar, öteki komutları yürütür. İki birim aynı anda çalışabildiği için bir komut yürütülürken sonraki komutun baytları getirilebilir.

2.1.3.1 İki temel birim

Tanım 2.1.5

Yol arayüz birimi (BIU, *bus interface unit*), 8086'nın dış dünyayla bütün iletişimini yürüten birimdir: komutları bellekten getirir, 20 bitlik fiziksel adresi hesaplar, işlenenleri okur ve sonuçları yazar. **Yürütme birimi** (EU, *execution unit*) ise komutları çözen ve yürüten birimdir: ALU'yu, genel amaçlı yazmaçları ve bayrakları içerir, dış yollara doğrudan bağlı değildir.

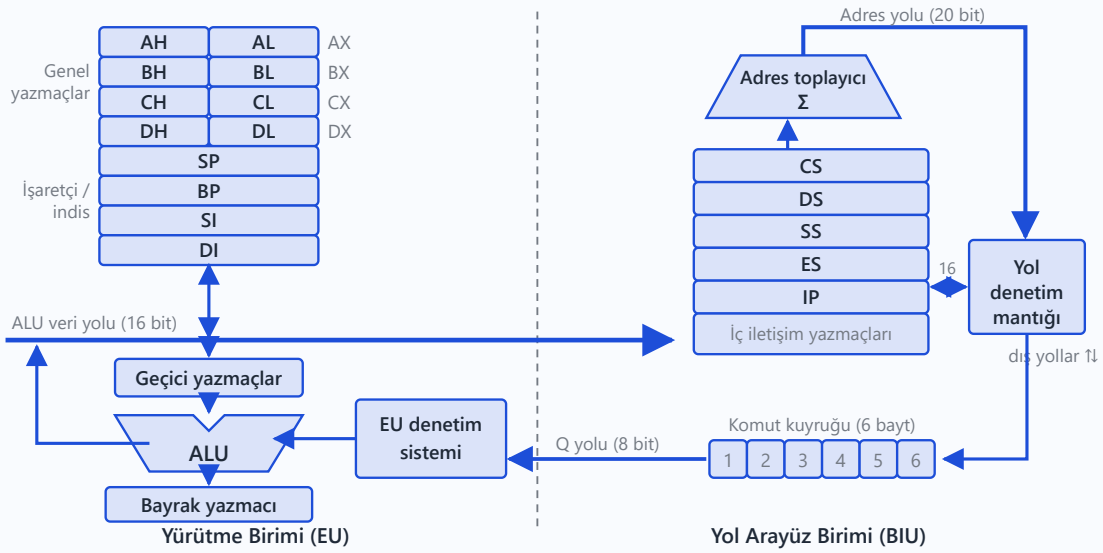


Şekil 2.1.6: BIU sistem yoluna bağlıdır, adres üretir ve komut baytlarını kuyruğa yazar. EU baytları kuyruktan alır, çözer ve ALU ile yazmaçlar üzerinde yürütür.

İki birim arasındaki köprü, önden getirme kuyruğudur. BIU bellekten getirdiği komut baytlarını kuyruğun sonuna ekler; EU sıradaki komutun baytlarını kuyruğun başından alır. EU'nun bellekte bir işlenene ihtiyacı olursa, örneğin `ADD AX, [BX]` yürütülürken, adresin bağıl konumunu hesaplar ve okuma işini BIU'dan ister. BIU yol döngüsünü yapar ve gelen veriyi EU'ya iletir. EU hiçbir zaman adres yoluna kendisi bir şey koymaz.

Bu iş bölümü sayesinde iki birim **paralel** çalışır. EU bir toplama yaparken BIU sonraki komutun baytlarını getirir. Getirme ile yürütmenin zaman içinde örtüşmesine **boru hattı** denir. 8086'nın boru hattı yalnız iki aşamalıdır, ama 1978 için önemli bir hız kazancı sağlıyordu.

2.1.3.2 Blok diyagram



Şekil 2.1.7: 8086'nın iç blok diyagramı. Solda EU (genel yazmaçlar, işaretçi ve indis yazmaçları, ALU, bayrak yazmacı, kontrol sistemi), sağda BIU (adres toplayıcı, kesim yazmaçları, IP, 6 baytlık komut kuyruğu, yol kontrol mantığı).

Şekil 2.1.7 işlemcinin içini iki yarıya ayırır. Sol yarıdaki EU'da en üstte dört genel amaçlı yazmaç durur: AX, BX, CX, DX. Her biri iki 8 bitlik yarıya bölünebilir (AH-AL, BH-BL, CH-CL, DH-DL). Altlarında işaretçi ve indis yazmaçları vardır: SP, BP, SI, DI. Bütün bu yazmaçlar 16 bitlik **ALU veri yoluna** bağlıdır. ALU, işlenenlerini geçici yazmaçlardan alır; programcı bu geçici yazmaçları göremez. ALU'nun çı-

kışı veri yoluna geri döner ve sonucu bir yazmaca götürür. Aynı anda işlemin sonucuna göre bayrak yazmacı güncellenir. EU'nun kontrol sistemi kuyruktan gelen işlem kodunu çözer ve bütün bu parçaları sırayla çalıştıran sinyalleri üretir.

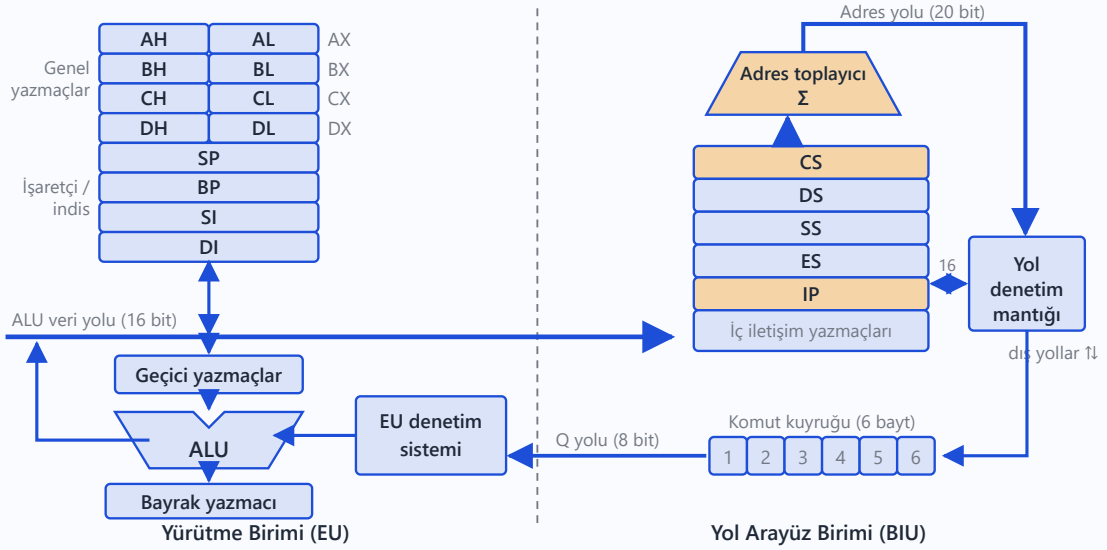
Sağ yarıdaki BIU'nun en belirgin parçası **adres toplayıcıdır** (Σ). Bir kesim yazmacındaki değeri 16 ile çarpar ve bir bağıl konumla toplar; sonuç 20 bitlik fiziksel adrestir ve adres yoluna çıkar. Toplayıcının altında dört kesim yazmacı (CS, DS, SS, ES) ve komut işaretçisi IP bulunur. İç iletişim yazmaçları EU ile BIU arasında taşınan veriyi geçici olarak tutar. Yol kontrol mantığı dış yollara bağlanır ve yol döngülerini yürütür. Bellekten gelen komut baytları 6 baytlık **komut kuyruğuna** yazılır; EU bu baytları 8 bitlik **Q yolu** üzerinden alır.

Yazmaçların hangi birimde bulunduğu, görevleri hakkında da bilgi verir. Kesim yazmaçları ve IP BIU'dadır, çünkü ikisi de adres üretmek içindir. Veri üzerinde işlem yapan yazmaçlar ve bayraklar EU'dadır, çünkü yürütmenin parçasıdırlar.

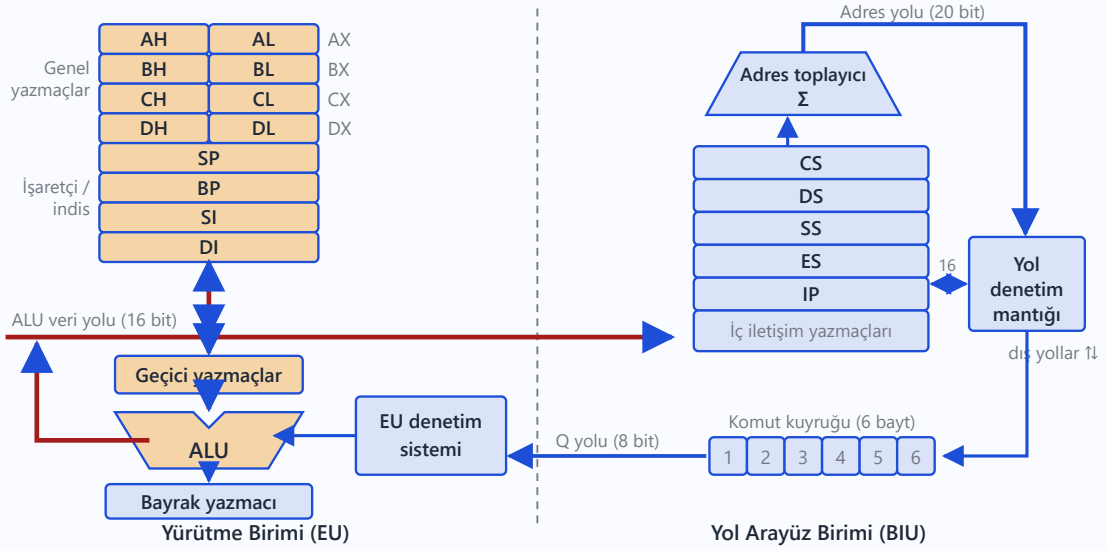
2.1.3.3 Bir komutun yolculuğu

İki birimin birlikte nasıl çalıştığını görmek için tek bir komutu, `ADD AX, BX`'i, bellekten alınmasından sonucun yazılmasına kadar izleyelim.

1. BIU, getirilecek komutun fiziksel adresini adres toplayıcıda hesaplar: **CS × 10h + IP**.
2. 20 bitlik adres dış adres yoluna konur; yol kontrol mantığı bir bellek okuma döngüsü başlatır.
3. Gelen komut baytları 6 baytlık kuyruğa yazılır. Getirilen bayt sayısı kadar IP ilerler.
4. Sıra bu komuta gelince EU, Q yolu üzerinden işlem kodunu alır. Kontrol sistemi komutu çözer: işlem toplama, işlenenler AX ve BX'tir.
5. Kontrol sinyalleri AX ve BX'i ALU veri yolundan geçici yazmaçlara, oradan ALU'ya taşır. ALU toplar.
6. Sonuç AX'e yazılır; bayraklar (CF, ZF, SF, OF, PF, AF) sonuca göre güncellenir. Bu sırada BIU sonraki baytları getirmeyi sürdürür.



Şekil 2.1.8: Birinci adım: BIU'daki adres toplayıcı CS ve IP'den komutun fiziksel adresini hesaplar. Vurgulanan parçalar o adımda çalışan birimlerdir.



Şekil 2.1.9: Beşinci adım: AX ve BX, ALU veri yolu ve geçici yazmaçlar üzerinden ALU'ya gelir; toplama yapılır.

Adımların ilk üçü BIU'da, son üçü EU'da geçer (Şekil 2.1.8, Şekil 2.1.9). Bu sınır, boru hattının temelidir. EU dördüncü adımda kuyruktan bir komut aldığı anda kuyrukta yer açılır. BIU, EU'nun beşinci ve altıncı adımlarını beklemeden yeni bir getirme döngüsüne başlar. `ADD AX, BX` bellek işleneni kullanmadığı için EU yürütme boyunca yola hiç ihtiyaç duymaz ve yol bütünüyle önden getirmeye ayrılır.

2.1.3.4 Yol arayüz birimi (BIU)

BIU'nun görevlerini topluca sıralamak, onun neden ayrı bir birim olduğunu açıklar:

- Bellek ve G/Ç aygıtlarıyla sistem yolu üzerinden arayüz kurar; bütün yol döngülerini o yürütür.

- Kesim yazmacı ve bağıl konumdan 20 bitlik fiziksel adresi adres toplayıcıda üretir.
- Komutları bellekten getirir ve kuyruğa yazar.
- EU'nun istediği işlenenleri bellekten ya da G/Ç'den okur, sonuçları yazar.
- 6 baytlık önden getirme kuyruğunu yönetir; boru hattını bu kuyruk mümkün kılar.

BIU'nun içinde kesim yazmaçları (CS, DS, SS, ES), komut işaretçisi (IP), adres toplayıcı, 6 baytlık komut kuyruğu ve yol kontrol mantığı bulunur. EU bir bellek işleneni istediğinde BIU, o sırada süren getirme döngüsünü bitirir ve ardından EU'nun isteğini yerine getirir. EU'nun isteği önden getirmeden önce gelir; çünkü EU o veri gelmeden ilerleyemez, kuyruk ise biraz bekleyebilir.

2.1.4 Önden getirme kuyruğu ve boru hattı

BIU ile EU'nun paralel çalışabilmesi, ikisinin arasında baytları biriktiren bir tampon olmasına bağlıdır. Bu tampon olmasaydı BIU getirdiği baytı EU alana kadar bekletmek zorunda kalırdı. Kuyruk iki birimi birbirinden ayırır: BIU kuyruğu doldurur, EU boşaltır ve her biri kendi hızında çalışır.

2.1.4.1 Kuyruğun işleyişi

Tanım 2.1.6

Önden getirme (*prefetch*), komut baytlarını yürütülmelerinden önce bellekten alıp beklet-mektir. 8086'nın **önden getirme kuyruğu** (*prefetch queue*), BIU'nun getirdiği komut baytları-nı tutan 6 baytlık bir FIFO (*first in, first out*, ilk giren ilk çıkar) tampondur: baytlar getirildikle-ri sırayla tüketilir.



Şekil 2.1.10: BIU bellekten getirdiği baytları kuyruğun sonuna yazar, EU baştan alır. Şekilde kuyrukta MOV AX,1234h (B8 34 12) ve ADD AX,BX (01 D8) komutlarının baytları sırada bekliyor.

Kuyruğun davranışını dört kural belirler:

1. **Getirme kuralı.** Kuyrukta en az 2 bayt boş yer açıldığında ve EU yolu istemiyorsa BIU sıradaki sözcüğü (2 bayt) getirir. 16 bitlik veri yolu tek döngüde iki bayt taşıdığı için getirme birimi sözcük-tür.
2. **Tüketme kuralı.** EU sıradaki komutun baytlarını kuyruğun başından alır. Kuyruk boşsa EU, ilk bayt gelene kadar bekler.

3. **Dolu kuyruk.** Kuyruk doluysa ve EU bir bellek işleneni istemiyorsa BIU boş durur. Bu durumda yol döngüsü yapılmaz.
4. **Dallanma kuralı.** Bir dallanma (JMP, CALL, RET, koşullu atlama, kesme) yürütülünce kuyruk **temizlenir**. Önden getirilmiş baytlar sıradaki adreslerden gelmiştir; program ise artık başka bir adresten devam eder. BIU yeni adresten getirmeye başlar.

Kuyruğun 6 bayt olması rastgele değildir. 8086 komutları öneksiz olarak 1 ile 6 bayt arasındadır. 6 baytlık bir kuyruk en uzun komutu da tümüyle tutabilir. 8088'in dış veri yolu 8 bit olduğu için bir döngüde yalnız bir bayt gelir; kuyruğu 4 bayttır ve 1 bayt boşalınca getirmeye başlar.

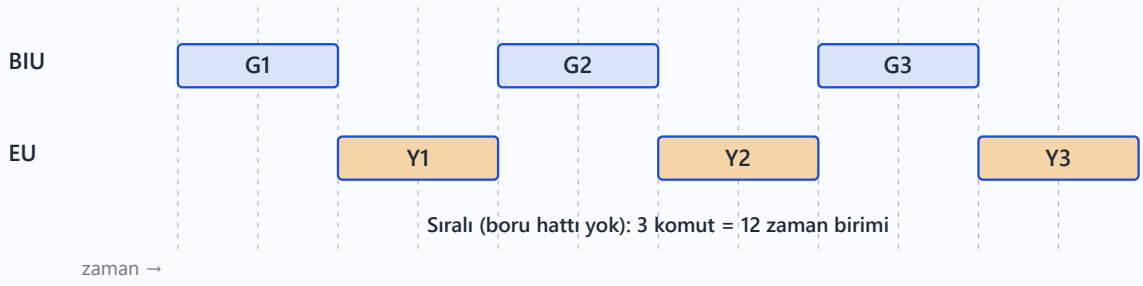
Kuyruğun kazancı, EU'nun çoğu zaman beklemeden komut bulmasıdır. Bellek erişim süresi, EU yürütürken arka planda geçer ve programcıya görünmez. Kaybı ise dallanmada ortaya çıkar: kuyruk boşaltılır, yeniden dolana kadar EU bekler. Döngülerle ve koşullu atlamalarla dolu programlarda bu kayıp önemli bir payı tutar.

2.1.4.2 Boru hattı

Tanım 2.1.7

Boru hattı (*pipeline*), ardışık komutların farklı aşamalarının aynı anda işlendiği düzendir. 8086'da iki aşama vardır: BIU'daki getirme (G) ve EU'daki çözme ile yürütme (Y). EU bir komutu yürütürken BIU sonraki komutu getirir.

Boru hattının kazancını görmek için basit bir model kullanalım: her getirme 2 zaman birimi, her çözme ile yürütme de 2 zaman birimi sürsün. Gerçekte komutların süreleri farklıdır, ama model fikri açıkça gösterir.



Şekil 2.1.11: Boru hattı olmasaydı: önce getir, sonra yürüt. BIU ve EU sırayla boş bekler; 3 komut 12 zaman birimi sürer.



Şekil 2.1.12: 8086'nın boru hattı: EU Y1'i yürütürken BIU G2'yi getirir. 3 komut 8 birimde biter; sonraki her komut yalnız 2 birim ekler.

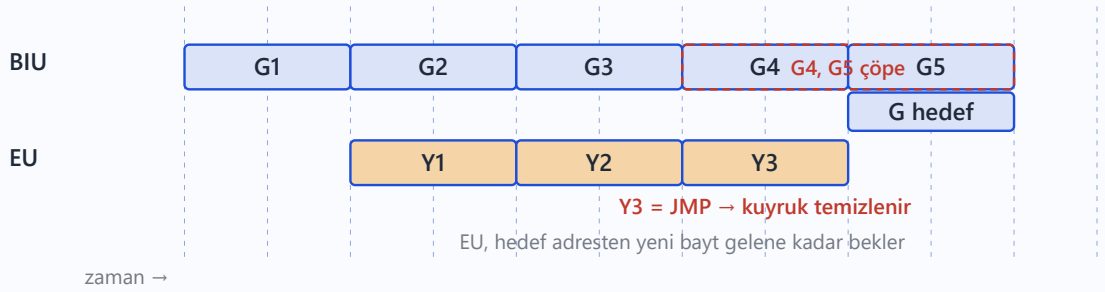
Sıralı düzende (Şekil 2.1.11) her komut 2 + 2 = 4 birim sürer ve *n* komut *4n* birim tutar. Boru hattında (Şekil 2.1.12) ilk getirme 2 birim sürer; ondan sonra her 2 birimde bir komut biter. *n* komut *2 + 2n* birim tutar. Komut sayısı arttıkça 4n / (2n + 2) oranı 2'ye yaklaşır: verim yaklaşık iki katına çıkar.

Örnek 2.1.3

Aynı modelle 5 komutun süresini üç durumda hesaplayalım.

1. Boru hattı yok: 5 × (2 + 2) = 20 birim.
2. Boru hattı var: 2 + 5 × 2 = 12 birim. G1 0–2, Y1 2–4 (G2 aynı anda), Y2 4–6, Y3 6–8, Y4 8–10, Y5 10–12.
3. Üçüncü komut bir JMP: Y3 sırasında (6–8) önden getirilen dördüncü komut yanlış yoldadır ve atılır. Hedef ancak Y3 bitince getirilebilir: G4 8–10, Y4 10–12, Y5 12–14. Toplam 14 birim.

Dallanma, boru hattının kazancından 2 birim geri alır. 100 komutluk bir programda boru hattı 400 birimi 202 birime indirir; her atlama bu sayıya yeniden 2 birim ekler.



Şekil 2.1.13: Dallanma: Y3 bir JMP ise önden getirilen G4 ve G5 geçersizdir. Kuyruk boşaltılır, EU hedef adresten yeni bayt gelene kadar bekler.

Şekil 2.1.13 bu kaybı zaman çizelgesinde gösterir. Modern işlemciler aynı sorunu dallanma tahminiyle hafifletir: atlamanın sonucunu önceden kestirir ve hedeften getirmeye erken başlar. 8086'da böyle bir düzenek yoktur; her alınan dallanma kuyruğu sıfırlar.

2.1.4.3 Kuyruk adım adım

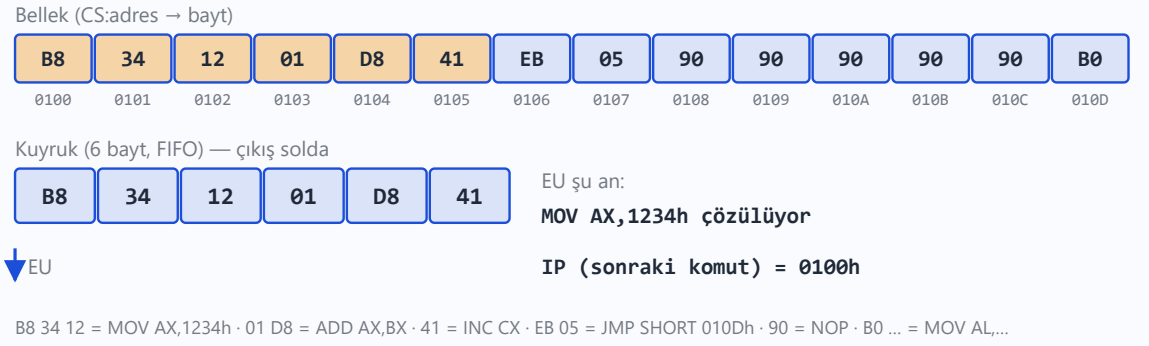
Kuyruğun kurallarını bir program üzerinde izlemek, bayt düzeyinde neler olduğunu gösterir. Program CS:0100h'den başlar ve bellekte şu baytları tutar:

```
0100: B8 34 12     mov ax, 1234h
0103: 01 D8        add ax, bx
0105: 41           inc cx
0106: EB 05        jmp short 010Dh
0108: 90 90 90 90 90   ; beş NOP
010D: B0 ...       mov al, ...
```

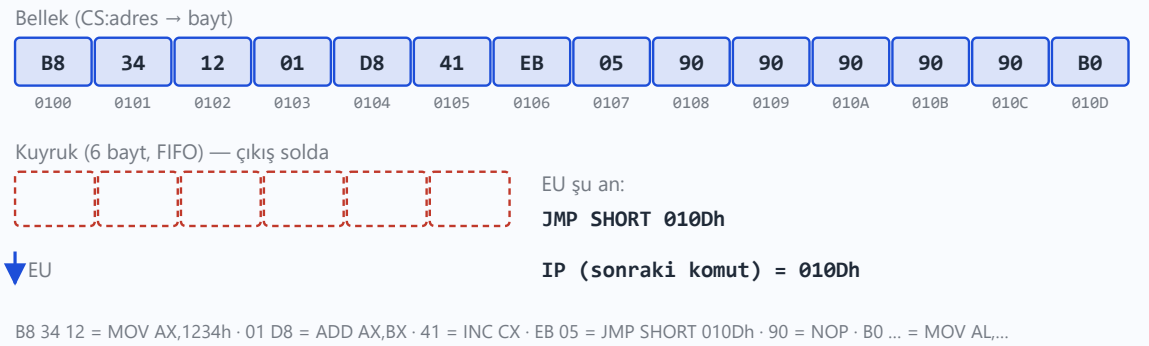
Her satırın başındaki sayı komutun kod kesimindeki bağıl konumudur. `B8` "AX'e 16 bitlik sabit yükle" işlem kodudur ve sabit, düşük baytı önce gelecek biçimde (34 12) ardından gelir. `01 D8`, `ADD AX, BX`'in makine kodudur. `41` tek baytlık `INC CX`'tir. `EB 05` kısa atlamadır: atlamadan sonraki komutun adresine 5 eklenir. `90` hiçbir şey yapmayan `NOP` komutudur.

Adım	Kuyruk	EU	Sıradaki komutun IP'si	Olay
1	(boş)	bekliyor	0100h	başlangıç ya da dallanma sonrası
2	B8 34 12 01 D8 41	MOV AX,1234h çözülüyor	0100h	BIU üç sözcük getirdi (B8 34, 12 01, D8 41)
3	01 D8 41	MOV AX,1234h	0103h	EU 3 bayt çekti; 3 boş yer var
4	01 D8 41 EB 05	MOV AX,1234h	0103h	BIU EB 05'i getirdi; EU beklemedi
5	41 EB 05 90 90	ADD AX,BX	0105h	EU 01 D8'i aldı; BIU 90 90'ı getirdi
6	EB 05 90 90 90 90	INC CX	0106h	EU 41'i aldı; BIU iki NOP daha getirdi, kuyruk dolu
7	(temizlendi)	JMP SHORT 010Dh	010Dh	IP = 0108h + 5 = 010Dh; dört 90 atıldı
8	B0	bekliyor	010Dh	BIU 010Dh'den getirir; tek adres olduğu için önce 1 bayt

Tablo 2.1.5: Kuyruğun adım adım durumu. Kuyruk sütununda en soldaki bayt kuyruğun başıdır.



Şekil 2.1.14: İkinci adım: BIU üç sözcük getirerek kuyruğu doldurdu; EU ilk bayt gelir gelmez MOV AX,1234h'yi çözmeye başladı.



Şekil 2.1.15: Yedinci adım: EU JMP'yi yürütür, IP 010Dh olur. Kuyruktaki NOP baytları yanlış yoldadır ve kuyruk temizlenir.

Tablo 2.1.5 birkaç ayrıntıyı ortaya koyar. İkinci adımda EU, kuyruk tümüyle dolmadan çalışmaya başlar; ilk bayt gelince çözme başlayabilir. Üçüncü adımda EU, 3 baytlık komutu kuyruktan çekince 3 boş yer açılır. Kural “en az 2 boş yer” olduğu için BIU hemen bir sözcük getirir. Bundan sonra BIU, EU'nun hep önünde kalır: dördüncü ve altıncı adımlarda EU hiç beklemez.

Yedinci adımda atlama yürütülür (Şekil 2.1.15). Kısa atlamanın uzaklığı, atlamadan sonraki komutun adresine göre hesaplanır. JMP 0106h'de başlayıp iki bayt sürdüğü için sonraki adres 0108h'dir; 0108h + 5 = 010Dh. Kuyrukta bekleyen dört 90 baytı 0108h–010Bh adreslerinden gelmiştir ve artık yürütülmeyecektir. Kuyruk temizlenir. Sekizinci adımda BIU 010Dh'den getirmeye başlar. 010Dh tek bir adrestir; 8086 bir sözcüğü tek döngüde ancak çift adresten okuyabildiği için önce yalnız bir bayt (B0) getirir, sonraki getirmeler yeniden çift adreslerden sözcük sözcük ilerler. EU bu sürede bekler; bu bekleme dallanmanın bedelidir.

`ADD AX, BX`'in burada `01 D8` olarak kodlandığına dikkat etmek gerekir. 8086'da iki yazmaç arasındaki bir işlem iki biçimde kodlanabilir. `01` işlem kodu “yazmacı r/m alanına ekle”, `03` işlem kodu “r/m alanını yazmaca ekle” anlamındadır. Hedef de kaynak da yazmaç olunca ikisi aynı işi yapar: `01 D8` ile `03 C3` eşdeğerdir. Çoğu assembler, emu8086 da dahil, ikinci biçimi üretir.

2.1.4.4 Önden getirme ve çözme işlevleri

Önden getirme ve çözme, işlemci tasarımında iki ayrı işlev olarak ele alınır. **Önden getirme işlevi** komutları bellekten getirip bir tamponda saklar. Bir program sayacı, getirilecek sonraki baytın adresini tutar. 8086'da bu iş IP ve BIU'nun içindeki bir sayaçla yapılır: IP programcının gördüğü "sıradaki komut" adresidir, BIU'nun sayacı ise kuyruğa girecek sıradaki baytı gösterir. Kuyruk dolu olduğunda bu sayaç IP'nin birkaç bayt önündedir. EU komutları bellek yerine tampondan aldığı için bellek erişim süresi gizlenir ve birim zamanda daha çok komut yürütülür.

Çözme işlevi makine kodunu kontrol sinyallerine ve mikro işlemlere çevirir. Bir işlem kodunun hangi işlemi, hangi işlenenleri ve hangi adresleme modunu istediğini belirler. Yazmaçtan yazmaca, bellekten yazmaca aktarımları ve koşullu atlama, CALL, RET gibi karmaşık komutları adım adım yönlendirir. Çözme de önden getirmeyle paralel çalışır.

8086'da bu işlevler ayrı birimler değildir. Önden getirme BIU'da, çözme EU'nun kontrol sisteminde yapılır. Kontrol sistemi **mikrokod** tabanlıdır: her makine komutu, yonganın içindeki bir ROM'da saklanan kısa bir mikro işlem dizisine karşılık gelir. Karmaşık bir komut, örneğin bölme, uzun bir mikro işlem dizisi çalıştırır; bu yüzden yüzlerce saat turu sürer. Ayrı önden getirme ve çözme birimleri 80286 ile ortaya çıktı; 80286'da adres hesabı ve komut çözme kendi birimlerine taşındı ve boru hattı dört aşamaya çıktı.

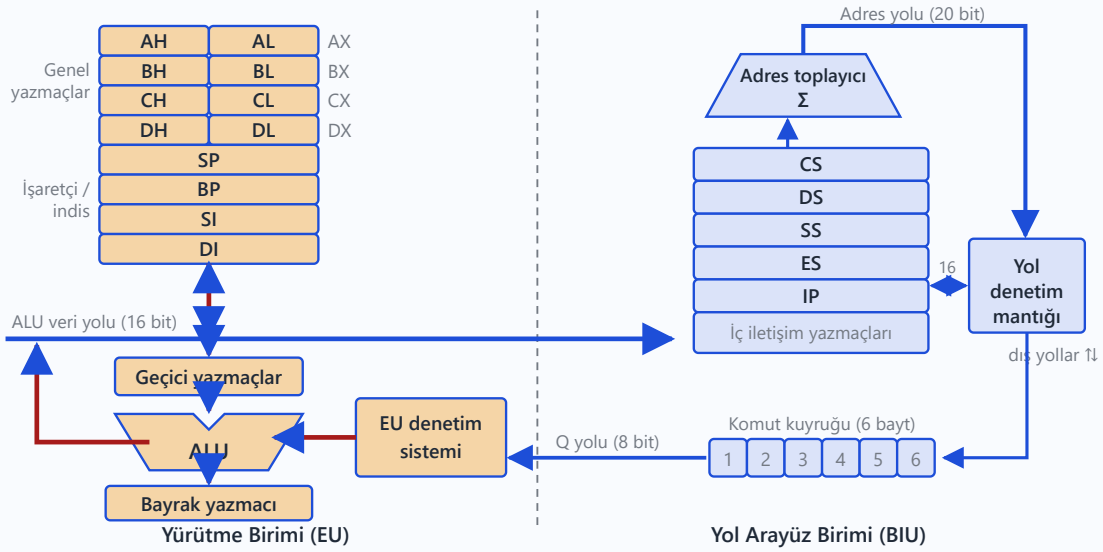
2.1.5 Yürütme birimi

EU, komutların gerçekten "yapıldığı" yerdir. Kuyruktan gelen baytları çözer, işlenenleri yazmaçlardan alır, ALU'ya işlem yaptırır ve sonucu geri yazar. Bu kesim EU'nun parçalarını, ALU'nun yaptığı işlemleri ve EU'yu yöneten kontrol birimini anlatır.

2.1.5.1 EU'nun bileşenleri

EU'nun başlıca bileşenleri şunlardır:

- **Genel amaçlı yazmaçlar** (AX, BX, CX, DX): veri saklar, aritmetik ve mantık işlemlerine katılır.
- **Aritmetik-mantık birimi** (ALU): 8 ve 16 bitlik hesapları yapar.
- **İşaretçi ve indis yazmaçları** (SP, BP, SI, DI): yığın ve bellek adreslemesinde bağıl konum tutar.
- **Komut yazmacı ve komut çözücü**: kuyruktan alınan işlem kodunu tutar ve çözer.
- **Bayrak yazmacı** (FLAGS): son işlemin sonucunu özetleyen durum bitlerini ve işlemcinin davranışını belirleyen kontrol bitlerini tutar.
- **Kontrol sistemi**: bütün bu parçaları çalıştıran sinyalleri sırasıyla üretir.



Şekil 2.1.16: EU'nun bileşenleri blok diyagramda vurgulanmıştır: genel yazmaçlar, işaretçi ve indis yazmaçları, geçici yazmaçlar, ALU, bayrak yazmacı ve kontrol sistemi.

2.1.5.2 Aritmetik-mantık birimi

Tanım 2.1.8

Aritmetik-mantık birimi (ALU), iki işlenen üzerinde aritmetik ya da mantıksal bir işlem yapıp sonucu üreten birimdir. 8086'nın ALU'su 8 ve 16 bitlik işlenenlerle çalışır ve her işlemden sonra sonuca göre durum bayraklarını günceller.

ALU'nun yaptığı işler iki gruba ayrılır. Aritmetik işlemler toplama, çıkarma, çarpma, bölme, bir artırma, bir azaltma ve karşılaştırmadır. Mantık işlemleri AND, OR, XOR ve NOT ile bitleri sola ya da sağa kaydırma ve döndürmedir. Karşılaştırma (`CMP`) aslında bir çıkarmadır; sonucu bir yere yazılmaz, yalnız bayraklar kurulur.

Komut	İşlem	Boyut
`ADD AX, BX`	AX ← AX + BX	16 bit
`SUB AL, 5`	AL ← AL − 5	8 bit
`MUL BL`	AX ← AL × BL	8 → 16 bit
`DIV CX`	AX ← DX:AX ÷ CX, DX ← kalan	32 → 16 bit
`AND AL, 0Fh`	AL'nin üst 4 biti sıfırlanır	8 bit
`XOR AX, AX`	AX ← 0 (ZF = 1)	16 bit

Tablo 2.1.6: ALU işlemlerinden örnekler.

Çarpma ve bölmenin boyut sütunu dikkat ister. İki 8 bitlik sayının çarpımı 16 bite kadar büyüyebilir: 255 × 255 = 65.025. Bu yüzden `MUL BL` sonucu AL'ye değil, AX'in tamamına yazar. Bölmede tersi geçerlidir: 16 bitlik bir bölme 32 bitlik bir bölünenle başlar. Bölünenin üst yarısı DX'te, alt yarısı AX'tedir ve bu çift DX:AX diye yazılır. Bölüm AX'e, kalan DX'e gider.

Bayraklar ALU'nun her işlemden sonra bıraktığı izdir. Altı durum bayrağı vardır:

- **CF** (elde bayrağı): işaretsiz işlemde en üst bitten elde ya da ödünç çıktıysa 1.
- **ZF** (sıfır bayrağı): sonuç sıfırsa 1.
- **SF** (işaret bayrağı): sonucun en anlamlı biti; işaretli yorumda negatifse 1.
- **OF** (taşma bayrağı): işaretli işlemin sonucu hedefe sığmadıysa 1.
- **PF** (eşlik bayrağı): sonucun düşük baytında çift sayıda 1 varsa 1.
- **AF** (yardımcı elde bayrağı): bit 3'ten bit 4'e elde ya da ödünç olduysa 1.

Bunlara ek olarak programcının kurduğu üç kontrol bayrağı (TF, IF, DF) vardır; bunlar ALU sonucundan etkilenmez. Her komut bütün bayrakları değiştirmez: `MOV` hiçbirine dokunmaz, `INC` ve `DEC` CF'yi korur, `MUL` ve `DIV` sonrası bazı bayraklar **tanımsızdır**. Tanımsız bir bayrak işlemcide 0 ya da 1 olabilir, ama değerine güvenilemez; program ona bakarak karar vermemelidir.

2.1.5.3 Kod izleme: ALU ne yapar?

Aşağıdaki program Tablo 2.1.6'daki işlemleri sırayla çalıştırır. Her adımda hangi yazmacın değiştiği ve bayrakların ne olduğu izlenir.

```
org 100h
mov ax, 0010h
mov bx, 0030h
add ax, bx      ; AX ← AX + BX
sub al, 5       ; AL ← AL − 5
mov bl, 4
mul bl          ; AX ← AL × BL
mov dx, 0
mov cx, 3
div cx          ; AX ← DX:AX ÷ CX
and al, 0Fh     ; üst 4 biti sil
xor ax, ax      ; AX ← 0
ret
```

İlk iki `MOV` işlenenleri yazmaçlara yükler: AX = 0010h, BX = 0030h. `MOV` ALU'yu kullanmaz; değer yalnız bir yerden ötekine kopyalanır ve bayraklar değişmez. `add ax, bx` 16 bitlik toplamadır: 0010h + 0030h = 0040h. Onluk olarak 16 + 48 = 64'tür. En üst bitten elde çıkmaz (CF = 0), sonuç sıfır değildir (ZF = 0), en üst bit 0'dır (SF = 0).

`sub al, 5` 8 bitlik bir çıkarmadır ve yalnız AL'yi etkiler. AL = 40h'dir; 40h − 05h = 3Bh (64 − 5 = 59). AH değişmez, AX = 003Bh olur. Sonuç işaretsiz olarak sığdığı için ödünç gerekmez ve CF = 0 kalır. Düşük nibble'da 0 − 5 işlemi bit 4'ten ödünç aldığı için AF = 1 olur.

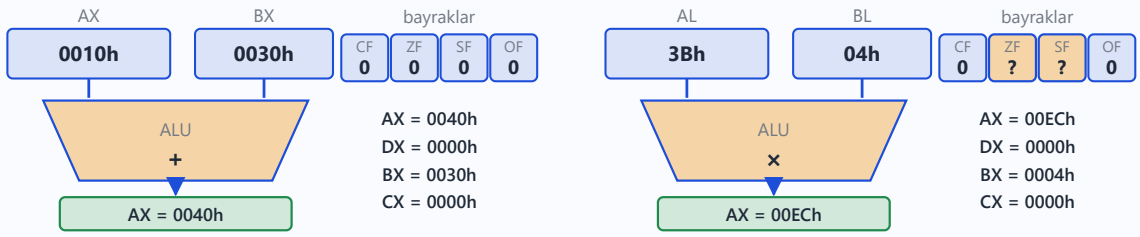
`mov bl, 4` yalnız BX'in düşük yarısını değiştirir: BH zaten 00h olduğu için BX = 0004h olur. `mul bl` AL ile BL'yi çarpar: 3Bh × 04h = ECh, yani 59 × 4 = 236. Sonuç 16 bit olarak AX'e yazılır: AX = 00ECh. Sonucun üst yarısı (AH) sıfır olduğu için CF = OF = 0 olur; bu iki bayrak "sonuç AL'ye sığdı" bilgisini verir. SF, ZF, AF ve PF ise `MUL` sonrası tanımsızdır.

Bölmeye hazırlık iki satır sürer. `mov dx, 0` bölünenin üst yarısını sıfırlar; böylece DX:AX = 0000 00ECh = 236 olur. `mov cx, 3` böleni yükler. `div cx` 236'yı 3'e böler: bölüm 78 = 004Eh AX'e, kalan 2 = 0002h DX'e yazılır. Sağlama: 78 × 3 + 2 = 236. `DIV` sonrası altı durum bayrağının hepsi tanımsızdır.

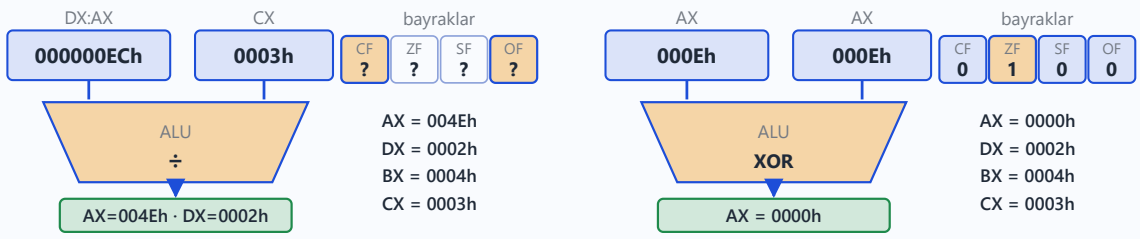
`and al, 0Fh` bir maskeleme işlemidir. 4Eh = 0100 1110, 0Fh = 0000 1111'dir; AND bitleri tek tek çarpar: 0000 1110 = 0Eh. Maskenin 0 olduğu üst dört bit silinir, 1 olduğu alt dört bit korunur. AND her zaman CF = OF = 0 yapar. Son komut `xor ax, ax` her biti kendisiyle XOR'lar. Aynı iki bitin XOR'u 0 olduğu için AX = 0000h ve ZF = 1 olur. Bu, `mov ax, 0`'dan kısa bir sıfırlama yoludur: `33 C0` iki bayt, `B8 00 00` üç bayttır. Farkı da vardır: XOR bayrakları değiştirir, MOV değiştirmez.

Satır	AX	BX	CX	DX	CF	ZF	SF	OF
2–3 `mov ax / mov bx`	0010h	0030h	0000h	0000h	0	0	0	0
4 `add ax, bx`	0040h	0030h	0000h	0000h	0	0	0	0
5 `sub al, 5`	003Bh	0030h	0000h	0000h	0	0	0	0
6–7 `mov bl / mul bl`	00ECh	0004h	0000h	0000h	0	?	?	0
8–9 `mov dx / mov cx`	00ECh	0004h	0003h	0000h	0	?	?	0
10 `div cx`	004Eh	0004h	0003h	0002h	?	?	?	?
11 `and al, 0Fh`	000Eh	0004h	0003h	0002h	0	0	0	0
12 `xor ax, ax`	0000h	0004h	0003h	0002h	0	1	0	0

Tablo 2.1.7: ALU programının izlemesi. "?" tanımsız bayrak.



Şekil 2.1.17: Solda `add ax, bx` : 16 bitlik toplama, AX = 0040h. Sağda `mul bl` : 8 bitlik işlenenlerin 16 bitlik çarpımı, AX = 00ECh; ZF ve SF tanımsız.



Şekil 2.1.18: Solda `div cx` : DX:AX = 236, bölüm AX = 004Eh, kalan DX = 0002h; bütün durum bayrakları tanımsız. Sağda `xor ax, ax` : AX = 0000h, ZF = 1.

Sık yapılan hata

`div cx` 'ten önce DX'i sıfırlamayı unutmak sık görülür. 16 bitlik `DIV` her zaman DX:AX çiftini böler. DX'te örneğin 0005h kalmışsa bölünen 5 × 65.536 + 236 = 327.916 olur. Bu sayının 3'e bölümü 109.305'tir ve 16 bite sığmaz. İşlemci bu durumda bir bölme hatası kesmesi (INT 0) üretir ve program durur.

Bu komutların süreleri de çok farklıdır. 8086'da `ADD AX, BX` 3 saat turu sürer. `MUL BL` 70–77 tur, `DIV CX` 144–162 tur alır; çünkü mikrokod çarpma ve bölmeyi bit bit kaydırıp toplayarak yapar. 5 MHz'de tek bir 16 bitlik bölme 30 µs'ye yaklaşır. Bölen 2'nin bir kuvvetiyse bölme yerine kaydırma komutları çok daha hızlıdır.

2.1.5.4 Komut yazmacı, çözücü ve kontrol birimi

EU'nun içinde, programcının hiç görmediği ama her komutta çalışan üç parça vardır.

- **Komut yazmacı**: EU kuyruktan işletilecek komutu alır; işlem kodu çözülürken burada geçici olarak tutulur. Programcı bu yazmaca erişemez.
- **Komut çözücü**: işlem kodunu çözer. Hangi işlemin yapılacağını, işlenenlerin nerede olduğunu ve hangi adresleme modunun kullanıldığını belirler ve bu bilgiyi kontrol devresine gönderir.
- **Kontrol birimi**: komut ve verinin işlemci içindeki akışını yönetir. Getir–çöz–yürüt döngüsünü sıralar ve diğer bileşenlerin ne zaman ne yapacağını belirler.

Kontrol biriminin görevleri bir döngü olarak özetlenebilir: komutu BIU aracılığıyla getirmek, çözmek, yürütmek, işlemcinin durumunu (yazmaçlar, bayraklar) güncellemek ve her komutun sonunda **kesme isteklerini** denetlemek. İstek iki hattan gelebilir: INTR'den maskelenebilir (yalnız IF = 1 iken kabul edilir), NMI'dan maskelenemeyen istek. Bir istek varsa işlemci sıradaki komuta geçmeden kesme hizmet programına dallanır. Bazı hata durumları da kesme üretir; sıfıra bölme ya da sığmayan bölüm INT 0'ı tetikler.

8086'nın kontrol birimi mikrokod tabanlıdır. Güç yönetimi gibi özellikler 8086'da yoktur; bunlar çok sonraki işlemcilerle, örneğin 386SL ile gelmiştir. 8086 ya çalışır ya da `HLT` komutuyla bir kesme gelene kadar durur.

2.1.6 Komut uzunlukları ve makine kodu

Bellekte duran bir program, assembly satırları değil, bayt dizisidir. Her komut, işlemcinin çözdüğü bir veya daha çok bayta çevrilir. 8086'da bu bayt sayısı komuttan komuta değişir. Uzunluğu bilmek, kuyruğun ve IP'nin davranışını anlamanın anahtarıdır: IP her komutta tam o komutun uzunluğu kadar ilerler.

2.1.6.1 Komut biçimi

Tanım 2.1.9

Bir 8086 makine komutu şu alanlardan oluşur: **işlem kodu** (*opcode*, 1 bayt; hangi işlemin yapılacağını belirtir), varsa **ModR/M baytı** (işlenenleri ve adresleme modunu belirtir), varsa 8 ya da 16 bitlik **yer değiştirme** (*displacement*; bellek adresine eklenen sabit) ve varsa 8 ya da 16 bitlik **anlık değer** (*immediate*; komutun içindeki sabit işlenen). Öneksiz bir komut 1 ile 6 bayt arasındadır.

Komut	Makine kodu	Bayt	Yapı
`INC CX`	`41`	1	yalnız işlem kodu
`MOV AX, BX`	`8B C3`	2	işlem kodu + ModR/M
`ADD AL, 7Ch`	`04 7C`	2	AL'ye özel kısa kod + 8 bitlik anlık
`MOV BX, 4050h`	`BB 50 40`	3	işlem kodu + 16 bitlik anlık
`MOV WORD PTR [BX+1234h], 5678h`	`C7 87 34 12 78 56`	6	işlem kodu + ModR/M + 16 bitlik yer değiştirme + 16 bitlik anlık

Tablo 2.1.8: Farklı uzunlukta komutlar.

Tablo 2.1.8'deki en kısa komut `INC CX`'tir. 8086'da 16 bitlik bir yazmacı bir artıran komutların işlem kodu 40h'den başlar ve yazmaç numarası işlem koduna eklenir: AX için 40, CX için 41, DX için 42, BX için 43. Yazmaç işlem kodunun içinde olduğu için ayrı bir bayt gerekmez. Benzer biçimde `MOV BX, 4050h`'de BB işlem kodu "BX'e 16 bitlik sabit yükle" anlamına gelir (B8 + 3, BX'in numarası). Sabit ardından gelir.

Anlık değerler belleğe **little-endian** düzende yazılır: önce düşük bayt, sonra yüksek bayt. 4050h, bellekte `50 40` sırasıyla durur. Aynı kural yer değiştirmeler için de geçerlidir: 1234h yer değiştirmesi `34 12`, 5678h anlık değeri `78 56` olarak yazılır.

`ADD AL, 7Ch` için 8086'nın AL ve AX'e ayrılmış kısa kodlarından biri kullanılır. Genel biçim `80 C0 7C` diye üç bayt sürerdi; AL'ye özel `04` işlem kodu ModR/M baytına gerek bırakmaz ve komut iki bayta iner. Akümülatörün (AX/AL) bu ayrıcalığı, sık yapılan işlemleri kısaltmak içindir.

İşlenenlerden biri genel bir yazmaç ya da bellek olduğunda ModR/M baytı devreye girer. Bu bayt üç alana bölünür:

> ModR/M = mod (2 bit) · reg (3 bit) · r/m (3 bit)

- **mod** r/m alanının nasıl yorumlanacağını söyler: 11 ise r/m bir yazmaçtır; 00, 01, 10 ise bellek işlenenidir ve sırasıyla yer değiştirme yok, 8 bitlik ya da 16 bitlik yer değiştirme vardır.
- **reg** işlenenlerden birinin yazmacını verir: 000 AX, 001 CX, 010 DX, 011 BX, 100 SP, 101 BP, 110 SI, 111 DI (8 bitlik komutlarda AL, CL, DL, BL, AH, CH, DH, BH).
- **r/m** mod = 11 ise ikinci yazmacı, değilse adres biçimini verir: 000 [BX+SI], 001 [BX+DI], 010 [BP+SI], 011 [BP+DI], 100 [SI], 101 [DI], 110 [BP] (mod = 00 iken doğrudan adres), 111 [BX].

Örnek 2.1.4

İki ModR/M baytını çözelim.

1. `8B C3` : 8B işlem kodu "r/m16'yı reg16'ya taşı" (MOV r16, r/m16) anlamındadır. C3h = 1100 0011 = 11 000 011. mod = 11 (yazmaç), reg = 000 (AX), r/m = 011 (BX). Komut `MOV AX, BX` 'tir.
2. `C7 87 …` : C7 işlem kodu "r/m16'ya 16 bitlik sabit yaz" anlamındadır; reg alanı burada yazmaç değil, işlemin alt türüdür ve 000 olmalıdır. 87h = 1000 0111 = 10 000 111. mod = 10 (16 bitlik yer değiştirme), r/m = 111 ([BX]). Ardından yer değiştirme `34 12` = 1234h ve anlık değer `78 56` = 5678h gelir. Komut `MOV WORD PTR [BX+1234h], 5678h` 'tir.

Aynı mantıkla `89 D8` çözülürse: 89 "reg16'yı r/m16'ya taşı", D8h = 11 011 000, reg = BX, r/m = AX. Bu da `MOV AX, BX` 'tir. Yazmaçtan yazmaca bir MOV'un iki eşdeğer kodlaması vardır.

2.1.6.2 Kod izleme: baytlar ve IP

Aşağıdaki program Tablo 2.1.8'deki komutları sırayla çalıştırır. İzlemenin amacı, her komutun bellekte kaç bayt tuttuğunu ve IP'nin buna göre nasıl ilerlediğini görmektir.

```
org 100h
inc cx
mov bx, 4050h
mov ax, bx
add al, 7Ch
mov word ptr [bx+1234h], 5678h
ret
```

Program 0100h'den başlar ve kod kesiminin 0100h–010Eh aralığını, toplam 15 baytı kaplar: `41 BB 50 40 8B C3 04 7C C7 87 34 12 78 56 C3`. `inc cx` yalnız işlem kodundan oluşur; CX = 1 olur ve IP 0101h'ye ilerler. `mov bx, 4050h` üç bayttır (BB 50 40); IP = 0104h. `mov ax, bx` iki bayttır (8B C3); AX = 4050h, IP = 0106h.

`add al, 7Ch` AL'ye özel kısa koddur (04 7C). AL = 50h + 7Ch = CCh olur; onluk olarak 80 + 124 = 204. AH değişmediği için AX = 40CCh'dir. IP = 0108h. Altı baytlık `mov word ptr [bx+1234h], 5678h` etkin adresi BX + 1234h = 4050h + 1234h = 5284h olarak hesaplar ve oraya 5678h yazar. Bellekte DS:5284h'de 78, DS:5285h'te 56 durur. IP = 010Eh. Son komut `ret` tek bayttır (C3).

Komut	Bayt	Uzunluk	IP	CX	BX	AX	Bellek
`inc cx`	41	1	0101h	0001h	0000h	0000h	—
`mov bx, 4050h`	BB 50 40	3	0104h	0001h	4050h	0000h	—
`mov ax, bx`	8B C3	2	0106h	0001h	4050h	4050h	—
`add al, 7Ch`	04 7C	2	0108h	0001h	4050h	40CCh	—
`mov word ptr [bx+1234h], 5678h`	C7 87 34 12 78 56	6	010Eh	0001h	4050h	40CCh	DS:5284h = 78, DS:5285h = 56
`ret`	C3	1	0000h	0001h	4050h	40CCh	—

Tablo 2.1.9: Komut uzunlukları programının izlemesi. IP, komut yürütüldükten sonraki değerdir.



Şekil 2.1.19: Solda `mov bx, 4050h` : 3 bayt, IP 0101h'den 0104h'ye. Sağda altı baytlık `mov word ptr [bx+1234h], 5678h` : IP 0108h'den 010Eh'ye; 5678h, DS:5284h'ye little-endian yazılır.

Tablo 2.1.9'un IP sütunu, her komutun uzunluğunun bir önceki IP'ye eklenmesiyle elde edilir: 0100h + 1 = 0101h, + 3 = 0104h, + 2 = 0106h, + 2 = 0108h, + 6 = 010Eh. Son satırda ise IP 0000h olur. `RET` bir dallanmadır: IP'ye ardışık bir değer değil, yığının tepesindeki sözcüğü yükler. .COM programı başlarken DOS yığına 0000h koyar; kesimin 0000h bağıl konumunda programın önündeki 256 baytlık PSP başlığı vardır ve onun ilk komutu programı sonlandırır. Böylece `RET` denetimi işletim sistemine geri verir.

Altı komut 15 bayt tuttu: uzunluklar 1 ile 6 bayt arasında değişti. Kuyruğun 6 bayt olması bu yüzden yeterlidir. emu8086'nın ürettiği makine kodu listesi ve izleme, tablodaki baytları ve değerleri birebir verir.

Sık yapılan hata

Makine kodunu okurken `BB 50 40` 'ı "BX'e 5040h yükle" diye yorumlamak yanlıştır. 8086 little-endian'dır: ilk gelen bayt düşük bayttır. Doğru okuma 4050h'dir. Aynı hata bellek dökümlerinde de yapılır: DS:5284h'den başlayan `78 56` baytları sözcük olarak 5678h'dir.

2.1.7 Yazmaç kümesi

Bir assembly programcısının elindeki en önemli araç yazmaçlardır. Her komut ya bir yazmaçtan okur, ya bir yazmaca yazar, ya da bir yazmacın gösterdiği bellek konumuna erişir. Bu kesim 8086'nın yazmaçlarını gruplarına ayırır ve sözcük boyunun anlamını açıklar. Adres üretiminde rol oynayan iki grup, kesim yazmaçları ve IP, ayrıntılı ele alınır.

2.1.7.1 14 yazmaç

Tanım 2.1.10

Yazmaç (*register*), işlemcinin içinde bulunan, çok hızlı, küçük bir bellek hücresidir. Bir işlemcinin programcıya sunduğu yazmaçların tümüne **yazmaç kümesi** (*register set*) denir. 8086'nın yazmaç kümesinde 16 bitlik 14 yazmaç vardır.



Şekil 2.1.20: 8086'nın 14 yazmacı: dört genel amaçlı (veri) yazmaç, dört işaretçi ve indis yazmacı, dört kesim yazmacı ve iki özel yazmaç (IP, FLAGS).

Şekil 2.1.20 yazmaçları dört gruba ayırır: 4 + 4 + 4 + 2 = 14.

- **Genel amaçlı yazmaçlar** (AX, BX, CX, DX) veri saklar ve aritmetik ile mantık işlemlerine katılır. Her biri 8 bitlik iki yarıya bölünebilir: AX'in yüksek yarısı AH, düşük yarısı AL'dir. AX'e akümülatör denir; çarpma, bölme ve G/Ç komutları onu kullanır. BX taban yazmacıdır ve bellek adreslemede bağıl konum tutabilir. CX sayaçtır; döngü ve yineleme komutları onu azaltır. DX veri yazmacıdır; çarpma ve bölmede AX'in üst yarısı olur, G/Ç'de port numarasını tutar.
- **İşaretçi ve indis yazmaçları** (SP, BP, SI, DI) bellekteki konumları gösterir. SP yığın işaretçisidir; yığının tepesinin bağıl konumunu tutar. BP taban işaretçisidir; yığındaki parametre ve yerel değişkenlere erişir. SI kaynak indisi, DI hedef indisidir; dizi ve dizgi işlemlerinde kullanılır. Bu yazmaçlar ikiye bölünemez.
- **Kesim yazmaçları** (CS, DS, SS, ES) bellekteki dört etkin kesimin başlangıcını gösterir.
- **Özel yazmaçlar**: IP sıradaki komutu, FLAGS son işlemin sonucunu ve işlemcinin kontrol bitlerini gösterir.

Genel amaçlı yazmaçların ikiye bölünebilmesi önemli bir ayrıntıdır. AL'ye yazılan bir değer AX'in düşük baytını değiştirir, AH'yi değiştirmez. Önceki kod izlemelerinde bu davranış görüldü: `sub al, 5` yalnız AL'yi, `mov bl, 4` yalnız BL'yi değiştirdi. Yazmaç adları da bu yapıyı anlatır: H yüksek (*high*), L düşük (*low*) baytı gösterir.

2.1.7.2 Sözcük boyu

8086, 16 bitlik bir **tam sayı** işlemcisidir. Her komut tek bir veri öğesi üzerinde çalışır; aynı anda birden çok veriyi işleyen vektör komutları yoktur. Kayan noktalı sayılarla hesap da donanımda yoktur; bu iş için ayrı bir yardımcı işlemci, 8087, sisteme eklenebilir. 8087 yoksa kayan nokta işlemleri yazılımla, tam sayı komutlarından kurulan uzun alt programlarla yapılır.

Yonga içindeki yazmaçların genişliği, işlemcinin aynı anda üzerinde çalışabileceği veri miktarını, yani **sözcük boyunu** belirler. 8086'da sözcük 16 bittir. Bu genişlik değer aralıklarını da belirler:

Genişlik	Örnek	İşaretsiz aralık	İşaretli aralık
8 bit	AL, BH	0 … 255 (FFh)	−128 … +127
16 bit	AX, SI	0 … 65.535 (FFFFh)	−32.768 … +32.767
32 bit	DX:AX	0 … 4.294.967.295	−2.147.483.648 … +2.147.483.647

Tablo 2.1.10: Yazmaç genişliğine göre değer aralıkları.

32 bitlik bir değer tek bir yazmaca sığmaz; DX:AX gibi iki yazmaçlık bir çiftte tutulur. Çarpma ve bölme bu çifti kendiliğinden kullanır. Toplama ve çıkarmada ise iki adımla yapılır: önce düşük sözcükler `ADD` ile, sonra yüksek sözcükler elde ile birlikte `ADC` ile toplanır.

Yazmaçlar işlemcinin içindeki en hızlı depolama alanıdır. Bir yazmaca erişmek için yol döngüsü gerekmez; değer iç veri yolunda hemen hazırdır. Belleğe erişim ise en az bir yol döngüsü, yani en az dört saat turu ister. Bu fark, iyi bir assembly programında sık kullanılan değerlerin yazmaçlarda tutulmasının nedenidir.

2.1.7.3 Kesim yazmaçları

Tanım 2.1.11

Kesim (*segment*), 8086 belleğinde 16 baytlık bir sınırda başlayan ve en çok 64 KB uzunluğunda olan bellek bloğudur. **Kesim yazmacı** (*segment register*), bir kesimin başlangıç adresinin 16'ya bölümünü tutar; kesimin fiziksel başlangıcı, yani **kesim tabanı**, yazmaçtaki değerin 10h ile çarpımıdır.

Kesimler bellektedir; kesim yazmaçları işlemcinin içindedir. Bu ayrım önemlidir: kesim yazmacı bir bellek bölgesini "gösterir", o bölgeyi içermez. Yazmaç değiştirilince gösterilen bölge kayar, ama bellekteki baytlar yerinde kalır. Dört kesim yazmacının her birinin belirli bir görevi vardır:

Yazmaç	Adı	Gösterdiği kesim	Birlikte kullanıldığı bağıl konum
CS	kod kesimi (*code segment*)	programın komutları	IP (CS:IP sıradaki komut)
DS	veri kesimi (*data segment*)	değişkenler ve veriler	BX, SI, DI, doğrudan adres
SS	yığın kesimi (*stack segment*)	yığın	SP, BP (SS:SP, SS:BP)
ES	ek kesim (*extra segment*)	ek veri; dizgi hedefi	DI (ES:DI)

Tablo 2.1.11: Kesim yazmaçları ve görevleri.

Kesim yazmaçlarının kullanımında üç kural vardır. Bu kurallar, kesim yazmaçlarının ALU'ya değil BIU'ya bağlı olmasından ve komut kodlamasında onlara ayrılan yerin dar olmasından gelir.

1. **Anlık değer yüklenemez.** `MOV DS, 2000h` geçersizdir; 8086'nın komut kümesinde kesim yazmacına sabit yükleyen bir işlem kodu yoktur. Değer önce genel bir yazmaçtan geçer: `MOV AX, 2000h`, ardından `MOV DS, AX`. Değer bellekten de alınabilir: `MOV DS, [BX]` geçerlidir.
2. **CS, MOV ile değiştirilmez.** CS değişince bir sonraki komut başka bir yerden alınır; bu bir dallanmadır. CS yalnız uzak JMP, uzak CALL, RETF, INT ve IRET ile değişir; bu komutlar IP'yi de aynı anda ayarlar. Assembler `MOV CS, AX` satırını kabul etmez.
3. **Kesimden kesime doğrudan MOV yoktur.** `MOV ES, DS` geçersizdir. Değer ya genel bir yazmaçtan (`MOV AX, DS` / `MOV ES, AX`) ya da yığından (`PUSH DS` / `POP ES`) geçer.

Sık yapılan hata

Veri kesimini ayarlamak için `MOV DS, 1000h` yazmak derlenmez. Assembler bu satırda hata verir. Doğru yazım iki satırdır: `MOV AX, 1000h` ve `MOV DS, AX`. Bu kalıp, bütün kesim ayarlamalarında görülür ve kod izlemelerinin hepsinde kesim yazmaçlarına değerin AX üzerinden gitmesinin nedenidir.

2.1.7.4 Komut işaretçisi (IP)

Tanım 2.1.12

Komut işaretçisi (IP, *instruction pointer*), kod kesiminde yürütülecek sonraki komutun bağıl konumunu tutan 16 bitlik yazmaçtır. Sonraki komutun fiziksel adresi CS × 10h + IP'dir.

IP her komut getirildikçe ilerler; bir komut yürütüldüğünde IP o komutun uzunluğu kadar artmış olur. Komut uzunlukları izlemesinde bu davranış görüldü: 1 baytlık `INC CX` IP'yi 1, 6 baytlık `MOV` 6 artırdı. Dallanma komutları ise (JMP, koşullu atlamalar, CALL, RET, LOOP, INT) IP'ye ardışık olmayan yeni bir değer yükler. Programın akışı bu değerle değişir.

Programcı IP'yi doğrudan okuyamaz ve yazamaz; `MOV IP, …` diye bir komut yoktur. IP'yi değiştirmenin tek yolu dallanmadır. IP'nin değerini öğrenmek için ise küçük bir hile kullanılır: `CALL` komutu dönüş adresini, yani sonraki komutun IP'sini yığına iter; hemen ardından `POP` ile bu değer bir yazmaca alınabilir.

Örnek 2.1.5

CS = 4321h ve IP = 1000h ise sonraki komut hangi fiziksel adrestedir?

1. CS × 10h: onaltılık bir sayıyı 10h ile çarpmak, sağına bir 0 eklemektir. 4321h × 10h = 43210h.
2. IP'yi sağa hizalayarak ekle: 43210h + 01000h.
3. Basamaklar sağdan toplanır: 0 + 0 = 0, 1 + 0 = 1, 2 + 0 = 2, 3 + 1 = 4, 4 + 0 = 4. Sonuç 44210h; elde oluşmadı.

Sonraki komut 44210h fiziksel adresindedir. Bu adres 20 bittir ve doğrudan adres yoluna çıkar.

2.1.8 Kesimli bellek

8086'yı öteki birçok işlemciden ayıran özellik, belleği kesimler üzerinden adreslemesidir. Bir programcı hiçbir zaman 20 bitlik bir adresi doğrudan yazmaz; her bellek erişimi bir kesim yazmacı ile bir bağıl konumun birleşiminden doğar. Bu kesim, bu birleşimin nasıl yapıldığını, aynı baytın neden birçok adla anılabildiğini, 1 MB sınırında ne olduğunu ve hangi erişimin hangi kesimi kullandığını anlatır.

2.1.8.1 Mantıksal ve fiziksel adres

8086 belleği **kesimli** (*segmented*) olarak adresler. Her bellek adresi iki 16 bitlik parçayla verilir: kesim ve bağıl konum. Yazılışta iki parça iki nokta üst üste ile ayrılır: `2000h:0010h`. Kesim yazmacı kesimin başlangıcını gösterir; bağıl konum istenen baytın kesimin başından itibaren uzaklığıdır ve 0000h ile FFFFh arasında değişir. İki parça BIU'daki adres toplayıcıda birleştirilerek 20 bitlik fiziksel adres elde edilir.

Tanım 2.1.13

Mantıksal adres, programın kullandığı `kesim:bağıl konum` çiftidir. **Fiziksel adres** (FA), adres yoluna çıkan gerçek 20 bitlik adrestir ve şöyle hesaplanır:

> FA = kesim × 10h + bağıl konum

Toplam 20 biti aşarsa 8086 yalnız alt 20 biti kullanır.



Şekil 2.1.21: Adres toplayıcı: kesim değeri 2000h, 4 bit sola kaydırılarak 20000h kesim tabanına çevrilir; bağıl konum 0010h eklenir; 20010h fiziksel adresi A19–A0 hatlarına çıkar.

10h ile çarpma, ikili sayıda 4 bit sola kaydırmaya eşittir; donanımda bir çarpıcıya gerek yoktur. Kesim değerinin bitleri toplayıcının girişine dört konum kaydırılmış olarak bağlanır ve sağdan dört 0 eklenir. Böylece 16 bitlik kesim değeri 20 bitlik bir tabana dönüşür. Bağıl konum ise kaydırılmadan, sağa hizalı olarak eklenir. Şekil 2.1.21 bu düzeni gösterir.

Kaydırmanın sonucunda kesim tabanının son onaltılık basamağı her zaman 0'dır. Kesim tabanları bu yüzden 16'nın katlarıdır. Bellekte 16 baytlık bu sınırlara **paragraf** denir. Bir kesim yazmacının alabileceği 65.536 değerin her biri bir paragraf sınırını gösterir: 1.048.576 / 16 = 65.536.

2.1.8.2 Fiziksel adres hesabı adım adım

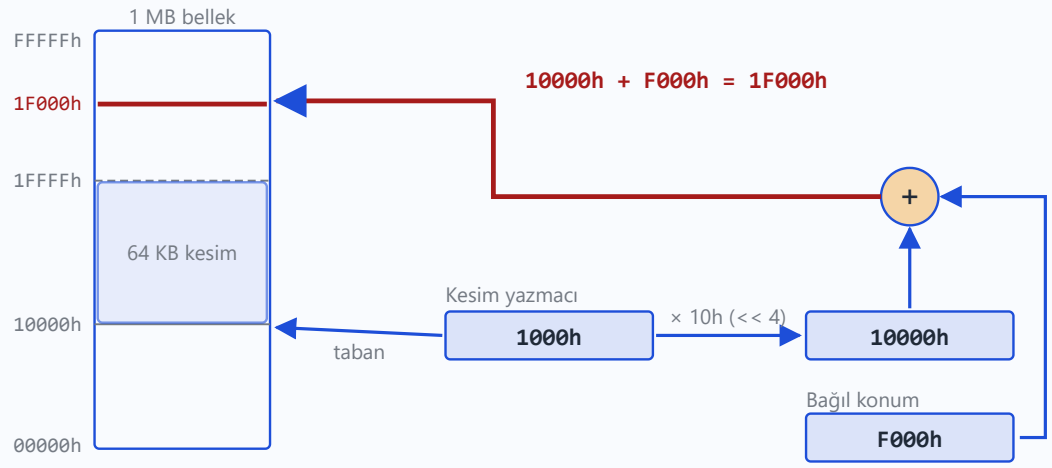
Hesap her zaman üç adımda yapılır: kesim değerinin sağına bir 0 eklenir, bağıl konum sağa hizalanarak altına yazılır ve iki sayı onaltılık olarak toplanır.

Örnek 2.1.6

Kesim yazmacı 1000h, bağıl konum F000h olsun.

1. Kesim tabanı: 1000h × 10h = 10000h.
2. Bağıl konum F000h, yani 0F000h.
3. Toplam: 10000h + 0F000h = 1F000h.

Bağıl konum en çok FFFFh olabildiği için bu kesim 10000h–1FFFFh aralığını kapsar. Aralığın uzunluğu 10000h bayt, yani 64 KB'tır.



Şekil 2.1.22: Kesim yazmacındaki 1000h, 4 bit sola kaydırılarak 10000h tabanına dönüşür; bağıl konum F000h ile toplanınca 1F000h fiziksel adresi bulunur. 64 KB'lık kesim 10000h–1FFFFh aralığını kaplar.

Örnek 2.1.7

CS = F000h ve IP = FFF0h ise komutun fiziksel adresi nedir?

1. F000h × 10h = F0000h.
2. IP = FFF0h sağa hizalanır: 0FFF0h.
3. F0000h + 0FFF0h = FFFF0h.

FFFF0h, 1 MB'ın son 16 baytının başıdır. 8086 sıfırlandığında CS = FFFFh ve IP = 0000h yükler: FFFF0h + 0000h = FFFF0h. Farklı iki çift aynı baytı gösterir; işlemci ilk komutunu her durumda buradan alır. Bilgisayarlarda bu adres BIOS ROM'undadır ve orada BIOS'un başlangıç koduna uzak bir atlama bulunur.



Şekil 2.1.23: Basamak basamak toplama. Solda 43210h + 1000h = 44210h, sağda F0000h + FFF0h = FFFF0h. Kesimin sağına bir 0 eklenir; bağıl konum sağa hizalanır.

Sık yapılan hata

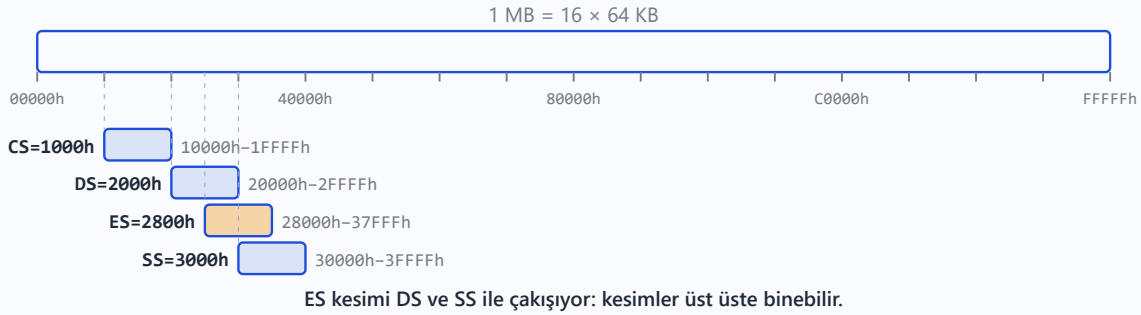
İki hata sık görülür. Birincisi kesim değerini 10h ile çarpmayı unutup 4321h + 1000h = 5321h bulmaktır; bu 16 bitlik bir sayıdır ve fiziksel adres olamaz. İkincisi kaydırmayı yanlış parçaya uygulamaktır: bağıl konumu kaydırıp 4321h + 10000h = 14321h bulmak. Kural tektir: **kesim** kaydırılır, bağıl konum olduğu gibi eklenir.

2.1.8.3 Fiziksel adres hesaplayıcı

Sunumdaki etkileşimli hesaplayıcı ve Kesim:Bağıl Konum Hesaplayıcı simülatörü, girilen herhangi bir kesim ve bağıl konum için aynı üç satırı üretir: kesim, kesim × 10h ve toplam. Ayrıca 1 MB'lık bellek haritası üzerinde kesimin 64 KB'lık aralığını ve fiziksel adresin konumunu çizer, adresin normalleştirilmiş yazımını verir. Örneğin 4321h:1000h için kesim aralığı 43210h–5320Fh, fiziksel adres 44210h ve normalleştirilmiş yazım 4421h:0000h'dir. Toplam 1 MB'ı aşarsa hesaplayıcı 21. biti atar ve sonucun başa sardığını gösterir.

2.1.8.4 Bellek kesimleme

20 bitlik adres yolu 1 MB'ı adresler; bir kesim ise en fazla 64 KB'tır, çünkü bağıl konum 16 bittir. Herhangi bir anda dört kesim etkindir: CS, DS, SS ve ES'nin gösterdikleri. Kesim yazmaçları değiştirilmeden en çok 4 × 64 KB = 256 KB'a doğrudan erişilebilir. Belleğin başka bir bölgesine erişmek için bir kesim yazmacına yeni değer yüklenir.



Şekil 2.1.24: 1 MB bellekte dört etkin kesim: CS = 1000h (10000h–1FFFFh), DS = 2000h (20000h–2FFFFh), ES = 2800h (28000h–37FFFh), SS = 3000h (30000h–3FFFFh). ES kesimi DS ve SS ile çakışır.

Kesimler belleği sabit 16 parçaya bölmez. 1 MB / 64 KB = 16 yalnız bir hesaptır; 8086'da kesim başlangıçları sabit değildir. Bir kesim her paragraf sınırında başlayabilir; 65.536 olası başlangıç vardır. Bu yüzden kesimler ayrık olabileceği gibi çakışabilir de. Şekil 2.1.24'te ES = 2800h kesimi 28000h'de başlar ve 37FFFh'de biter. Bu aralığın ilk yarısı DS kesiminin ikinci yarısıyla (28000h–2FFFFh), ikinci yarısı SS kesiminin ilk yarısıyla (30000h–37FFFh) aynı baytlardır. Aynı bayta hem DS hem ES üzerinden, farklı bağıl konumlarla erişilebilir: fiziksel 2A000h, DS:A000h ile de ES:2000h ile de okunur.

Örnek 2.1.8

ES = 2800h kesiminin aralığını bulalım ve 30010h adresine ES ile erişilip erişilemeyeceğine bakalım.

1. Kesim tabanı: 2800h × 10h = 28000h.
2. Son bayt: 28000h + FFFFh = 37FFFh.
3. 30010h bu aralıktadır. Bağıl konum: 30010h − 28000h = 8010h. FFFFh'den küçük olduğu için geçerlidir: `ES:8010h`.

Aynı bayt SS = 3000h ile `SS:0010h` olarak da yazılır. DS = 2000h ile ise erişilemez: 30010h − 20000h = 10010h, 16 bite sığmaz.

2.1.8.5 Neden kesimleme?

Kesimleme, 16 bitlik bir işlemcide 1 MB adreslemenin ucuz bir yoluydu. Ama getirdiği kolaylıkların yanında yükleri de vardır.

Artıları	Eksileri
16 bitlik yazmaçlarla 20 bitlik (1 MB) bellek adreslenir.	Tek bir dizi ya da yapı 64 KB'ı aşamaz; aşarsa kesim değiştirmek gerekir.
Kod, veri ve yığın birbirinden ayrılır; program düzeni netleşir.	Aynı fiziksel adresin birçok kesim:bağıl konum yazımı vardır; işaretçi karşılaştırması zorlaşır.
Yeniden konumlanabilirlik: yalnız kesim yazmaçları değiştirilerek program bellekte başka yere yüklenebilir.	Kesimler arasında koruma yoktur; hatalı bir yazma başka bir kesimi bozabilir.
Komutlar kısa kalır: çoğu adres 16 bitlik bağıl konumla yazılır.	Adres hesabı programcıya ek bir zihinsel yük getirir.

Tablo 2.1.12: Kesimlemenin artıları ve eksileri.

Yeniden konumlanabilirlik özellikle değerlidir. Bir .COM programı içindeki bütün adresleri kesim başından itibaren bağıl konumla kullanır. İşletim sistemi programı hangi paragrafa yüklerse yüklesin, CS, DS, ES ve SS'yi o paragrafa ayarlaması yeter; programın hiçbir baytını değiştirmeye gerek kalmaz. emu8086 .COM programlarını 0700h kesimine yükler: CS = DS = ES = SS = 0700h ve program 0700h:0100h'den başlar.

Eksiler ise zamanla ağırlaştı. 64 KB'tan büyük veri yapıları için programcılar kesimleri elle değiştirmek zorunda kaldı; derleyiciler “yakın” ve “uzak” işaretçi türleri tanımladı. Koruma eksikliği çok görevli işletim sistemlerini zorlaştırdı. 80286'nın korumalı modu ve 80386'nın 32 bitlik düz adres alanı bu sorunları çözmek için geldi.

2.1.8.6 Aynı adres, farklı yazımlar

Kesimler çakışabildiği için bir fiziksel adres tek bir mantıksal adresle gösterilmez. Kesim bir paragraf artırılırsa taban 16 bayt ilerler; bağıl konum 16 azaltılırsa toplam değişmez. Tablo 2.1.13 bunu sayılarla gösterir.

Mantıksal adres	Hesap	Fiziksel adres
`1234h:0005h`	12340h + 0005h	**12345h**
`1000h:2345h`	10000h + 2345h	**12345h**
`1200h:0345h`	12000h + 0345h	**12345h**
`0234h:FFF5h`	02340h + FFF5h	**12335h** (12345h değil)
`FFFFh:0010h`	FFFF0h + 0010h = 100000h	**00000h** (8086'da 20 bite sarar)

Tablo 2.1.13: Farklı mantıksal adresler ve fiziksel karşılıkları.

İlk üç satır aynı baytı gösterir. Dördüncü satır yanıltıcıdır: kesim 0234h, 1234h'den 1000h küçüktür, yani taban 10000h bayt geridedir. Aynı bayta ulaşmak için bağıl konumun 10005h olması gerekirdi, ama bu 16 bite sığmaz. FFF5h ile 16 bayt geride kalınır. Son satırda toplam 1 MB'ı aşar ve 20 bite sarar.

Tanım 2.1.14

Normalleştirilmiş adres, bağıl konumu 0h–Fh arasında olan mantıksal adrestir. Beş basamaklı fiziksel adresin ilk dört basamağı kesimi, son basamağı bağıl konumu verir: 12345h → `1234h:0005h`.

Bir fiziksel adresin en çok 4096 farklı yazımı vardır. Bağıl konum 16'şar azaltılırken kesim birer artırılır; bağıl konum 0000h–FFFFh aralığında kaldığı sürece her adım geçerli bir yazım verir. Bağıl konumun son basamağı sabit kaldığı için seçilebilecek değer sayısı 65.536 / 16 = 4096'dır. 12345h için yazımlar `0235h:FFF5h` 'den `1234h:0005h` 'ye kadar uzanır: 1234h − 0235h + 1 = 1000h = 4096. Belleğin en başına yakın adreslerde yazım sayısı daha azdır; çünkü kesim 0'ın altına inemez.

Örnek 2.1.9

3A2C5h fiziksel adresini normalleştirelim ve DS = 3000h ise gereken bağıl konumu bulalım.

1. İlk dört basamak kesim, son basamak bağıl konum: `3A2Ch:0005h`. Sağlama: 3A2C0h + 5h = 3A2C5h.

2. DS = 3000h ise kesim tabanı 30000h'dir. Bağıl konum = 3A2C5h − 30000h = A2C5h.
3. A2C5h ≤ FFFFh olduğu için geçerlidir: `3000h:A2C5h`.

Aynı soru CS = 2F00h ve IP = 01A4h için tersinden sorulabilir: FA = 2F000h + 01A4h = 2F1A4h; normalleştirilmiş yazım `2F1Ah:0004h` 'tür. emu8086'da 2F00h:01A4h adresine yazılan bir bayt, ES = 2F1Ah iken `ES:[0004h]` ile aynen okunur.

2.1.8.7 Kod izleme: 1 MB sınırında adres başa sarar

Kesim × 10h + bağıl konum toplamı FFFFFh'yi aşabilir: en büyük durum FFFF0h + FFFFh = 10FFEFh'dir ve 21 bit tutar. 8086'nın yalnız 20 adres hattı (A19–A0) vardır. 21. bit adres yoluna çıkmaz, yani atılır. Sonuç olarak adres belleğin başına sarar. Aşağıdaki program bu davranışı gösterir.

```
org 100h
mov ax, 0
mov ds, ax
mov byte ptr [0500h], 77h
mov ax, 0FFFFh
mov ds, ax
mov bl, [0510h]    ; FFFF:0510
ret
```

İlk iki komut DS = 0000h yapar; kesim belleğin en başındadır. `mov byte ptr [0500h], 77h` 0000h:0500h adresine, yani fiziksel 00500h'ye 77h yazar. `byte ptr` belirteci yazılacak verinin bir bayt olduğunu bildirir; sabit 77h tek başına bunu belirlemez. Ardından DS = FFFFh yapılır. Bu kesimin tabanı FFFF0h'dir, yani 1 MB'ın son 16 baytı. `mov bl, [0510h]` fiziksel adresi FFFF0h + 0510h = 100500h olarak hesaplar. Bu sayı 21 bittir. 8086'nın 20 adres hattı yalnız 00500h'yi taşır ve işlemci, ilk adımda 77h yazılan baytın aynısını okur: BL = 77h.

Satır	DS	BL	Hesap	00500h'deki bayt
2–3 `mov ax,0 / mov ds,ax`	0000h	00h	—	00h
4 `mov byte ptr [0500h], 77h`	0000h	00h	00000h + 0500h = 00500h	77h
5–6 `mov ax,0FFFFh / mov ds,ax`	FFFFh	00h	taban FFFF0h	77h
7 `mov bl, [0510h]`	FFFFh	77h	FFFF0h + 0510h = 100500h → 00500h	77h

Tablo 2.1.14: Sarma programının izlemesi.



Şekil 2.1.25: `mov bl, [0510h]` : FFFF0h + 0510h = 100500h. 21. bit atılır, adres 00500h'ye sarar ve BL = 77h olur.

Bu kural, FFFFh kesiminde FFFFh:0010h'den FFFFh:FFFFh'ye kadarki bütün adreslerin belleğin başındaki 00000h–0FFEFh bölgesine düştüğü anlamına gelir. Sarma, 8086'da bir hata değil, 20 bitlik adres yolunun doğal sonucudur. Bazı eski programlar bu davranışa güvenerek yazıldı.

Durum 80286 ile değişti. 80286'nın 24 adres hattı vardır ve gerçek modda bile 21. hat (A20) açık olabilir. Bu durumda FFFFh:0510h gerçekten 100500h'ye, 1 MB'ın hemen üstüne ulaşır. Eski programlarla uyumu korumak için IBM PC/AT bilgisayarlarında A20 hattını kapatıp açan bir devre eklendi. A20 açıkken FFFFh kesimiyle 1 MB'ın üstündeki 65.520 bayta (FFFF0h + FFFFh − FFFFFh = FFF0h) gerçek modda erişilebilir.

Sık yapılan hata

FFFFh:0510h için 100500h'yi fiziksel adres olarak yazmak 8086'da yanlıştır; bu sayı 20 bite sığmaz. Toplam 1 MB'ı aşarsa en üst bit atılır ve 100000h çıkarılır: 100500h → 00500h. Aynı kontrol her fiziksel adres hesabının sonunda yapılmalıdır.

2.1.8.8 Varsayılan kesimler ve kesim öneki

Bir komut yalnız bağıl konum yazdığında, örneğin `mov al, [bx]`, hangi kesim yazmacının kullanılacağına işlemci kendisi karar verir. Bu karar erişimin türüne ve adres hesabında kullanılan yazmaca bağlıdır.

Tanım 2.1.15

Varsayılan kesim (*default segment*), bir bellek erişiminde kesim belirtilmemişse işlemcinin kullandığı kesimdir. **Kesim öneki** (*segment override prefix*), komutun başına eklenen ve varsayılan kesim yerine başka bir kesim yazmacını seçtiren bir baytlık önektir: ES için 26h, CS için 2Eh, SS için 36h, DS için 3Eh.

Erişim türü	Kesim	Bağıl konum kaynağı	Önekle değiştirilebilir mi?
Komut getirme	CS	IP	hayır

Erişim türü	Kesim	Bağıl konum kaynağı	Önekle değiştirilebilir mi?
Yığın işlemleri (PUSH, POP, CALL, RET)	SS	SP	hayır
BP ile adresleme `[BP+…]`	SS	etkin adres	evet: `DS:[BP]`
Genel veri `[BX]`, `[SI]`, `[DI]`, `[1234h]`	DS	etkin adres	evet: `ES:[BX]`
Dizgi kaynağı (MOVSB, LODSB)	DS	SI	evet
Dizgi hedefi (MOVSB, STOSB)	ES	DI	hayır

Tablo 2.1.15: Erişim türlerine göre varsayılan kesimler.

Tablo 2.1.15'teki kuralların mantığı yazmaçların görevinden gelir. Komutlar her zaman kod kesimindedir. Yığın her zaman yığın kesimindedir. BP yığındaki parametrelere ve yerel değişkenlere erişmek için tasarlandığından, BP içeren her adres varsayılan olarak SS'yi kullanır: `[BP]`, `[BP+SI]`, `[BP+DI+4]`. Geri kalan bütün veri erişimleri DS'yi kullanır. Dizgi komutları kaynağı DS:SI'den okur, hedefe ES:DI üzerinden yazar; böylece iki farklı kesim arasında blok kopyalanabilir.

Varsayılanı değiştirmek için kesim öneki yazılır: `MOV AL, ES:[BX]` fiziksel adresi ES × 10h + BX olarak hesaplar. Önek komuta bir bayt ekler (burada 26h) ve yürütmeyi iki saat turu uzatır. Komut getirme, yığın işlemleri ve dizgi hedefi önekle değiştirilemez; işlemci bu erişimlerde önek yokmuş gibi davranır.

Örnek 2.1.10

SS = 2000h, DS = 3000h, BP = 0100h ve SI = 0020h iken `mov byte ptr [bp+si+4], 0AAh` hangi adrese yazar? Aynı komut `ds:` önekiyle yazılırsa ne değişir?

1. Etkin adres: BP + SI + 4 = 0100h + 0020h + 0004h = 0124h.
2. Adreste BP vardır; varsayılan kesim SS'dir. FA = 20000h + 0124h = 20124h.
3. `ds:` önekiyle FA = 30000h + 0124h = 30124h olur. Makine kodu `C6 42 04 AA` 'dan `3E C6 42 04 AA` 'ya uzar.

emu8086 iki komutun 20124h ve 30124h adreslerine yazdığını ve önekli komutun 5 bayt tuttuğunu gösterir.

Sık yapılan hata

`[BX]` ile `[BP]` 'nin aynı kesimi kullandığını düşünmek yaygın bir hatadır. BX ve BP aynı değeri tutsa bile `[BX]` DS'ye, `[BP]` SS'ye göre hesaplanır ve iki komut farklı baytlara erişir. Bir .COM programında DS = SS olduğu için fark görünmez; kesimler ayrı ayarlanınca hata ortaya çıkar.

2.1.8.9 Kod izleme: aynı bağıl konum, üç kesim

Aşağıdaki program aynı bağıl konumu, 0010h'yi, üç farklı kesimle kullanır ve üç farklı bayta erişir. Program bir .COM programı olarak başladığı için başlangıçta CS = DS = ES = SS = 0700h'dir.

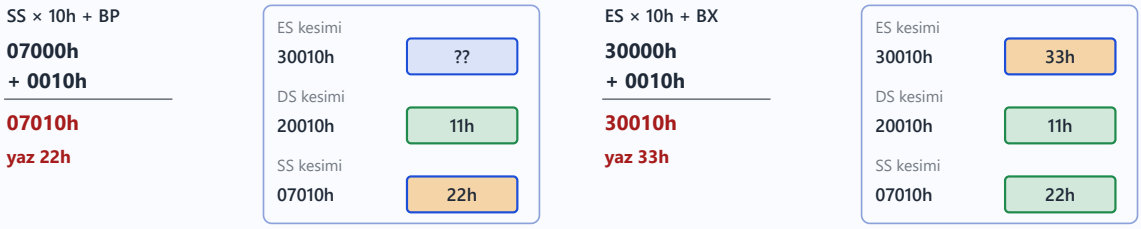
```
org 100h
mov ax, 2000h
mov ds, ax
mov ax, 3000h
mov es, ax
mov bx, 0010h
mov bp, 0010h
mov byte ptr [bx], 11h    ; DS
mov byte ptr [bp], 22h    ; SS
mov byte ptr es:[bx], 33h ; ES
mov al, [bx]              ; DS
mov ah, [bp]              ; SS
ret
```

İlk dört komut DS = 2000h ve ES = 3000h yapar; değerler AX üzerinden geçer. SS dokunulmadan 0700h kalır. Böylece üç farklı kesim tabanı oluşur: 07000h (SS), 20000h (DS), 30000h (ES). BX ve BP aynı bağıl konumu, 0010h'yi tutar. Hangi bayta erişileceğini bağıl konum değil, kullanılan kesim belirleyecektir.

`mov byte ptr [bx], 11h` varsayılan olarak DS'yi kullanır: 20000h + 0010h = 20010h adresine 11h yazar. `mov byte ptr [bp], 22h` BP içerdiği için SS'yi kullanır: 07000h + 0010h = 07010h adresine 22h yazar. `mov byte ptr es:[bx], 33h`'teki `es:` öneki DS yerine ES'yi seçtirir: 30000h + 0010h = 30010h adresine 33h yazar; makine kodunun başında 26h öneki vardır. Son iki komut okur: `mov al, [bx]` yine DS:0010h'yi okur ve AL = 11h olur, ES:0010h'deki 33h değil. `mov ah, [bp]` SS:0010h'yi okur ve AH = 22h olur. Sonuç AX = 2211h'dir.

Satır	Kesim	Fiziksel adres	İşlem	AX
2–5	—	—	DS = 2000h, ES = 3000h, SS = 0700h	3000h
6–7	—	—	BX = BP = 0010h	3000h
8 `[bx] ← 11h`	DS	20010h	yaz 11h	3000h
9 `[bp] ← 22h`	SS	07010h	yaz 22h	3000h
10 `es:[bx] ← 33h`	ES	30010h	yaz 33h	3000h
11 `al ← [bx]`	DS	20010h	oku 11h → AL	3011h
12 `ah ← [bp]`	SS	07010h	oku 22h → AH	2211h

Tablo 2.1.16: Üç kesim programının izlemesi.



Şekil 2.1.26: Solda `[bp]` : SS × 10h + BP = 07010h'ye 22h yazılır. Sağda `es:[bx]` : önek ES'yi seçer, 30010h'ye 33h yazılır. Aynı bağıl konum üç ayrı bayta karşılık gelir.

11. satırdan sonra AX = 3011h olur: AL yeni değeri aldı, AH ise önceki `mov ax, 3000h`'den kalan 30h'yi korudu. 12. satır AH'yi 22h yapar. Bu ayrıntı, 8 bitlik bir yazmaca yazmanın 16 bitlik yazmacın öteki yarısına dokunmadığını bir kez daha gösterir.

2.1.8.10 Kod izleme: kesim:bağıl konum ile erişim

Son izleme, aynı baytın iki farklı mantıksal adresle okunabildiğini ve bir kesimin son baytının nerede olduğunu gösterir.

```
mov ax, 2000h
mov ds, ax          ; DS = 2000h
mov bx, 0010h
mov byte ptr [bx], 41h
mov ax, 2001h
mov ds, ax          ; +16 bayt
mov bx, 0000h
mov al, [bx]        ; aynı bayt
mov si, 0FFFFh
mov ah, [si]        ; son bayt
```

DS = 2000h iken veri kesimi 20000h–2FFFFh aralığıdır. `mov byte ptr [bx], 41h` 20000h + 0010h = 20010h adresine 41h yazar; 41h, 'A' karakterinin ASCII kodudur. Ardından DS = 2001h yapılır. Kesim değeri 1 arttı, ama taban 16 bayt, yani bir paragraf ilerledi: yeni kesim 20010h–3000Fh aralığıdır. BX = 0000h ile `mov al, [bx]` 20010h + 0000h = 20010h'yi okur ve AL = 41h olur. `2000h:0010h` ile `2001h:0000h` aynı bayttır. Son iki komut SI = FFFFh ile kesimin en büyük bağıl konumunu kullanır: 20010h + FFFFh = 3000Fh. Bu, DS = 2001h kesiminin son baytıdır ve AH oradaki değeri alır.

Satır	AX	DS	BX	SI	FA	[20010h]
1 `mov ax, 2000h`	2000h	?	0000h	0000h	—	?
2 `mov ds, ax`	2000h	2000h	0000h	0000h	—	?
3 `mov bx, 0010h`	2000h	2000h	0010h	0000h	—	?
4 `mov byte ptr [bx], 41h`	2000h	2000h	0010h	0000h	20010h	41h
5 `mov ax, 2001h`	2001h	2000h	0010h	0000h	—	41h

Satır	AX	DS	BX	SI	FA	[20010h]
6 `mov ds, ax`	2001h	2001h	0010h	0000h	—	41h
7 `mov bx, 0000h`	2001h	2001h	0000h	0000h	—	41h
8 `mov al, [bx]`	2041h	2001h	0000h	0000h	20010h	41h
9 `mov si, 0FFFFh`	2041h	2001h	0000h	FFFFh	—	41h
10 `mov ah, [si]`	??41h	2001h	0000h	FFFFh	3000Fh	41h

Tablo 2.1.17: Kesim:bağıl konum programının izlemesi. AH'nin son değeri 3000Fh'deki bayta bağlıdır.

Doğrulayıcıda 3000Fh'ye önceden 99h konduğunda program AX = 9941h ile biter; AH gerçekten 3000Fh'deki baytı alır. Sekizinci satırda AX'in 2041h olması da aynı ilkeyi gösterir: AL 41h oldu, AH beşinci satırdan kalan 20h'yi korudu.

2.1.9 Özet

- **8086**, 1978'de çıkan 16 bitlik bir mikroişlemcidir ve x86 ailesinin ilk üyesidir. Yazmaçları, ALU'su ve veri yolu 16 bit, adres yolu 20 bittir: 2^{20} bayt = 1 MB bellek adresler. CISC tasarımıdır; komutları 1–6 bayt uzunluğundadır. Sonraki bütün x86 işlemciler açılışta **gerçek modda** 8086 gibi çalışır.
- Mikroişlemcinin içinde bellek yoktur; çalışması için 8284A saat üreteci, adres kilitleri (8282), veri yolu alıcı-vericileri (8286), bellek ve G/Ç birimleri gerekir. **Mikrodenetleyici** bunları tek yongada toplar.
- **Sistem yolu** üç yoldan oluşur: tek yönlü 20 bitlik adres yolu, çift yönlü 16 bitlik veri yolu ve kontrol yolu (RD', WR', M/IO', ALE, INTR, READY). M/IO' bellek ile G/Ç'yi ayırır; G/Ç portları 16 bitlik ayrı bir adres uzayındadır.
- Bir **yol döngüsü** en az dört saat turu (T1–T4) sürer; 5 MHz'de 800 ns. Bellek, yazmaçlardan çok yavaştır.
- 8086 iki birimden oluşur. **BIU** adres üretir, komutları getirir ve bütün yol döngülerini yürütür; kesim yazmaçları, IP, adres toplayıcı ve kuyruk ondadır. **EU** komutları çözer ve yürütür; genel yazmaçlar, ALU, bayraklar ve kontrol sistemi ondadır.
- 6 baytlık **önden getirme kuyruğu** bir FIFO'dur. En az 2 bayt boşalınca BIU bir sözcük getirir. Getirme ile yürütmenin örtüşmesi iki aşamalı bir **boru hattıdır** ve verimi yaklaşık iki katına çıkarır. Dallanma kuyruğu temizler ve bu kazancın bir kısmını geri alır.
- **ALU** 8 ve 16 bitlik aritmetik ve mantık işlemleri yapar ve durum bayraklarını (CF, ZF, SF, OF, PF, AF) günceller. `MUL` sonucu genişletir (AL × BL → AX), `DIV` DX:AX'i böler; `MOV` bayrakları değiştirmez.
- Bir makine komutu işlem kodu, ModR/M baytı (mod · reg · r/m), yer değiştirme ve anlık değerden oluşur. Çok baytlı sabitler **little-endian** yazılır. IP her komutta o komutun uzunluğu kadar ilerler; dallanmalar IP'ye yeni değer yükler.

- 8086'nın **14 yazmacı** vardır: AX, BX, CX, DX · SP, BP, SI, DI · CS, DS, SS, ES · IP, FLAGS. Kesim yazmacına anlık değer yüklenemez, CS MOV ile değişmez, kesimden kesime doğrudan MOV yoktur.
- **Fiziksel adres** = kesim × 10h + bağıl konum. Kesim en çok 64 KB'tır, bir paragraf (16 bayt) sınırında başlar ve kesimler çakışabilir. Aynı adresin en çok 4096 yazımı vardır; **normalleştirilmiş** yazımda bağıl konum 0h–Fh arasındadır.
- Toplam 1 MB'ı aşarsa 8086'da 21. bit atılır ve adres başa **sarar** (FFFFh:0510h → 00500h). A20 hattı açık bir 80286'da aynı adres 1 MB'ın üstüne ulaşır.
- **Varsayılan kesimler**: komut getirme CS:IP, yığın SS:SP, BP'li adresler SS, öteki veri erişimleri DS, dizgi hedefi ES:DI. **Kesim öneki** (26h, 2Eh, 36h, 3Eh) veri erişimlerinde varsayılanı değiştirir.

2.1.10 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

mikroişlemci *microprocessor* · mikrodenetleyici (MCU) *microcontroller* · CISC *complex instruction set computer* · gerçek mod *real mode* · saat turu *clock cycle* · saat üreteci *clock generator* · kilit *latch* · tampon *buffer (bus transceiver)* · yol *bus* · sistem yolu *system bus* · adres yolu *address bus* · veri yolu *data bus* · kontrol yolu *control bus* · yol döngüsü *bus cycle* · yol arayüz birimi (BIU) *bus interface unit* · yürütme birimi (EU) *execution unit* · önden getirme *prefetch* · önden getirme kuyruğu *prefetch queue* · boru hattı *pipeline* · aritmetik-mantık birimi (ALU) *arithmetic logic unit* · bayrak *flag* · durum bayrağı *status flag* · komut yazmacı *instruction register* · komut çözücü *instruction decoder* · kontrol birimi *control unit* · işlem kodu *opcode* · ModR/M baytı *ModR/M byte* · yer değiştirme *displacement* · anlık değer *immediate value* · etkin adres (EA) *effective address* · little-endian *little-endian* · yazmaç *register* · yazmaç kümesi *register set* · genel amaçlı yazmaç *general-purpose register* · sözcük *word* · kesim *segment* · kesim yazmacı *segment register* · kod / veri / yığın / ek kesim *code / data / stack / extra segment* · bağıl konum *offset* · kesim:bağıl konum (mantıksal adres) *segment:offset* · fiziksel adres *physical address* · paragraf *paragraph* · kesimli bellek *segmented memory* · kesimleme *segmentation* · komut işaretçisi (IP) *instruction pointer* · varsayılan kesim *default segment* · kesim öneki *segment override prefix*

2.1.11 Kendini Dene

1. 80286'nın adres yolu 24 bittir. Kaç bayt adresler ve en yüksek adresi onaltılık olarak nasıl yazılır? 8086'nın adres alanının kaç katıdır?

   Cevap

   2^{24} = 16.777.216 bayt = 16 MB. En yüksek adres altı onaltılık basamakla FFFFFFh'dir. 2^{24} / 2^{20} = 2^4 = 16 kat.

2. 8 MHz'lik bir 8086-2, bekleme durumu olmadan bir yol döngüsünü kaç nanosaniyede tamamlar? Bu hızda tek adreslerden başlayan 50 sözcüğü okumak en az ne kadar sürer?

   Cevap

   Saat turu 1 / 8 MHz = 125 ns; yol döngüsü 4 × 125 = 500 ns. Tek adresten başlayan bir sözcük iki döngü ister: 50 × 2 × 500 ns = 50.000 ns = 50 µs. Sözcükler çift adreste olsaydı süre 25 µs olurdu.

3. Getirme 2, çözme ile yürütme 2 zaman birimi süren modelde 8 komut çalışıyor ve dördüncü komut bir JMP. Boru hattı olmadan ve boru hattıyla toplam süreler nedir?

   Cevap

   Boru hattı olmadan 8 × 4 = 32 birim. Boru hattıyla dallanma olmasaydı 2 + 8 × 2 = 18 birim olurdu. JMP Y4 sırasında (8–10) getirilen beşinci komutu geçersiz kılar; hedef 10–12'de getirilir ve Y5 12–14, Y6 14–16, Y7 16–18, Y8 18–20 olur. Toplam 20 birim.

4. `ADD CX, [BX+SI+10h]` komutunun makine kodu `03 48 10`'dur. ModR/M baytını alanlarına ayırın ve her alanın anlamını yazın.

   Cevap

   48h = 0100 1000 = 01 001 000. mod = 01: bellek işleneni, 8 bitlik yer değiştirme var (10h). reg = 001: CX. r/m = 000: [BX+SI]. 03 işlem kodu "r/m16'yı reg16'ya ekle" olduğu için komut CX ← CX + [BX+SI+10h]'dir; adreste BP olmadığından kesim DS'dir. Komut 3 bayttır.

5. SS = 9000h ve SP = FFFEh iken `PUSH AX` çalışıyor (AX = 1234h). SP'nin yeni değeri nedir ve 34h ile 12h baytları hangi fiziksel adreslere yazılır?

   Cevap

   PUSH önce SP'yi 2 azaltır: SP = FFFCh. Sözcük SS:FFFCh'ye little-endian yazılır. Taban 90000h olduğundan 34h → 9FFFCh, 12h → 9FFFDh.

6. `0ABCDh:1234h` mantıksal adresinin fiziksel adresini bulun ve normalleştirin.

   Cevap

   ABCD0h + 01234h = ACF04h. Toplam FFFFFh'yi aşmaz. Normalleştirilmiş yazım: ilk dört basamak kesim, son basamak bağıl konum → `ACF0h:0004h`.

7. `F800h:9000h` adresi 8086'da hangi fiziksel adrese karşılık gelir? A20 hattı açık bir 80286'da ne değişir?

   Cevap

   F8000h + 09000h = 101000h; 21 bit. 8086'da 21. bit atılır ve adres 01000h'ye sarar. A20 açık bir 80286'da adres gerçekten 101000h'dir, yani 1 MB'ın 4 KB üstü.

8. SS = 2000h, DS = 3000h, BP = 0100h, SI = 0020h. `MOV AL, [BP+SI+4]` hangi fiziksel adresi okur? Aynı veriyi DS kesiminden okumak için komut nasıl yazılır ve kaç bayt uzar?

   Cevap

   Etkin adres 0124h; BP içerdiği için varsayılan kesim SS: 20000h + 0124h = 20124h. DS'den okumak için kesim öneki yazılır: `MOV AL, DS:[BP+SI+4]` → 30124h. Önek (3Eh) komuta bir bayt ekler.

9. DS ile ES'nin değerleri, genel amaçlı yazmaçlar bozulmadan takas edilmek isteniyor. `MOV ES, DS` neden kullanılamaz? Yığınla çalışan bir çözüm yazın.

   Cevap

   8086'da kesimden kesime doğrudan MOV yoktur. Yığınla: `PUSH DS`, `PUSH ES`, `POP DS`, `POP ES`. Yığın LIFO olduğu için önce ES'nin değeri DS'ye, sonra DS'nin eski değeri ES'ye çıkar. DS = 1111h, ES = 2222h ile başlanırsa sonuç DS = 2222h, ES = 1111h olur.

2.1.12 İleri okuma

- Intel, *The 8086 Family User's Manual* (1979), Bölüm 2: BIU ve EU, kuyruk, kesimleme ve komut kodlaması tasarımcının kendi anlatımıyla, zamanlama çizelgeleriyle birlikte verilir.
- B. B. Brey, *The Intel Microprocessors* (8. baskı), Bölüm 2 ve 3: gerçek mod bellek adreslemesi, varsayılan kesimler ve adresleme modları bol örnekle ele alınır.
- M. A. Mazidi, J. G. Mazidi ve D. Causey, *The x86 PC: Assembly Language, Design, and Interfacing* (5. baskı), Bölüm 1: 8086'nın iç yapısı, boru hattı ve kesim:bağıl konum hesabı DEBUG örnekleriyle anlatılır.

2.1.13 Kaynaklar

1. Intel, *8086 16-Bit HMOS Microprocessor*, veri sayfası (231455).
2. Intel, *The 8086 Family User's Manual*, 1979.
3. Barry B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson, 2009.
4. Muhammad Ali Mazidi, Janice Gillispie Mazidi, Danny Causey, *The x86 PC: Assembly Language, Design, and Interfacing*, 5. baskı, Pearson, 2010.
5. emu8086 belgeleri ve komut kümesi başvurusu.

KISIM I · BÖLÜM 2.2

8086 Yazmaçları

Bir assembly programının yaptığı işlerin çoğu, işlemcinin içindeki birkaç küçük bellek hücresinde olup biter. Bu hücrelere yazmaç denir. 8086'nın on dört yazmacı vardır ve her birinin adı bir görevi anlatır: AX akümülatördür, CX sayaçtır, SP yığının tepesini gösterir, IP sıradaki komutun yerini tutar. Not önce yazmacın ne olduğunu ve bellekten neden çok daha hızlı olduğunu açıklar. Ardından 8086'nın yazmaç kümesini gruplar hâlinde inceler: dört veri yazmacının 8 ve 16 bitlik kullanımı, komutların bu yazmaçları örtük olarak nasıl kullandığı, işaretçi ve indis yazmaçlarıyla yığına ve dizgilere erişim, kesim yazmaçları ve IP. Notun ikinci yarısı bayrak yazmacına ayrılmıştır. Altı durum bayrağının hangi sonuçta kurulduğu, ALU'nun bu bitleri donanımda nasıl ürettiği ve üç kontrol bayrağının işlemcinin çalışmasını nasıl değiştirdiği örneklerle gösterilir. Her kod parçası, yazmaçların, bayrakların ve belleğin her adımdaki değerleriyle satır satır izlenir.

ÖĞRENME ÇIKTILARI

Yazmaç kümesini tanımak
8086'nın on dört yazmacını gruplarına göre saymak; hangilerinin 8 bitlik iki yarıya bölünebildiğini ve bir yarıya yazmanın öbür yarıyı neden değiştirmediğini açıklamak.

Örtük görevleri bilmek
MUL, DIV, LOOP, REP, IN, OUT, PUSH, POP ve dizgi komutlarının hangi yazmaçları komutta yazılmadan kullandığını söylemek.

Yazmaç çiftlerini kullanmak
CS:IP, SS:SP, SS:BP, DS:BX, DS:SI ve ES:DI çiftlerinin hangi erişimde kullanıldığını bilmek ve fiziksel adresi hesaplamak.

Durum bayraklarını hesaplamak
Bir toplama ya da çıkarmadan sonra CF, PF, AF, ZF, SF ve OF bayraklarını elle bulmak; CF ile OF arasındaki farkı işaretsiz ve işaretli yorumla açıklamak.

Kontrol bayraklarını yönetmek
DF, IF ve TF'nin işlemcinin davranışını nasıl değiştirdiğini açıklamak; CLD/STD, CLI/STI, LAHF/SAHF ve PUSHF/POPF ile bayrakları okumak ve yazmak.

Kod izlemek
Kısa bir 8086 programını yazmaçların, bayrakların, belleğin ve yığının her adımdaki durumunu tablo hâlinde yazarak izlemek.

2.2.1 Yazmaç kavramı

2.2.1.1 Yazmaç nedir?

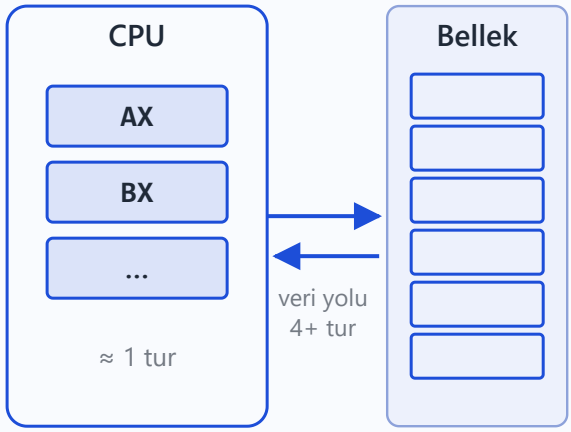
Sayısal tasarım dersinde D tipi flip-flop'lar yan yana dizilip ortak bir saat sinyaline bağlandığında bir **yazmaç** elde edilir. Saatin yükselen kenarında girişteki bitler flip-flop'lara yüklenir ve bir sonraki yüklemeye kadar orada kalır. Mikroişlemcinin içindeki yazmaçlar da böyle kurulmuştur. Fark, sayılarında ve görevlerindedir: işlemci aynı anda birkaç düzine yazmacı kullanır ve her birine komutlar aracılığıyla erişir.

Tanım 2.2.1

Yazmaç (*register*), işlemcinin içinde bulunan, çok hızlı ve küçük bir bellek hücresidir. 8086'nın yazmaçları 16 bit genişliğindedir. Programcının komutlarla doğrudan okuyup yazabildiği yazmaçların tümüne **yazmaç kümesi** (*register set*) denir; 8086'nın yazmaç kümesinde on dört yazmaç vardır.

İşlemcinin içinde programcının göremediği yazmaçlar da bulunur. Çözülmekte olan komutu tutan komut yazmacı, ALU'nun girişlerini besleyen geçici yazmaçlar ve önden getirme kuyruğu bunlardandır. Bu notta “yazmaç” denince programcının adıyla kullanabildiği on dört yazmaç anlaşılır.

Bir program çalışırken geçici veriler, bellek adresleri, döngü sayaçları ve işlem sonuçları yazmaçlarda tutulur. Şekil 2.2.1 işlemci ile bellek arasındaki ilişkiyi gösterir. Yazmaçlar işlemcinin içindedir ve ALU'ya kısa iç yollarla bağlıdır. Bellek ise işlemcinin dışında, adres ve veri yollarının öbür ucundadır.



Şekil 2.2.1: Yazmaçlar işlemcinin içinde, bellek dışındadır. Bir yazmaç erişimi işlemcinin iç yolunda biter; bellek erişimi dış veri yolu üzerinden en az dört saat turu sürer.

2.2.1.2 Yazmaç ve bellek: hız farkı

8086 dış bellekle **yol döngüleri** (*bus cycle*) hâlinde konuşur. Bir yol döngüsü en az dört saat turundan (T1, T2, T3, T4) oluşur. T1'de adres yola konur, T2'de okuma ya da yazma sinyali verilir, T3'te veri yolda hazır olur, T4'te çevrim kapanır. Bellek yavaşsa araya bekleme turları da girer. 5 MHz'lik bir 8086'da bir saat turu 200 ns sürer; dolayısıyla tek bir bellek okuması en az 800 ns'dir.

Yazmaç erişimi ise dış yola hiç çıkmaz. Yazmacın içeriği aynı saat turu içinde ALU'ya ulaşır. Tablo 2.2.1, Intel'in 8086 kılavuzundaki saat turu sayılarından birkaç örnek verir. EA, etkin adresin hesaplanması için harcanan ek turdur. Değeri adres biçimine bağlıdır: `[BX]` gibi tek yazmaçlı bir adres için 5, `[0500h]` gibi doğrudan adres için 6, `[BX+2]` gibi yazmaç artı sabit için 9 turdur.

Komut	İşlenenler	Saat turu	5 MHz'de süre
`MOV AX, BX`	yazmaç ← yazmaç	2	0,4 µs
`MOV AX, [BX]`	yazmaç ← bellek	8 + 5 = 13	2,6 µs
`ADD AX, BX`	yazmaç ← yazmaç + yazmaç	3	0,6 µs
`ADD AX, [BX]`	yazmaç ← yazmaç + bellek	9 + 5 = 14	2,8 µs
`ADD [BX], AX`	bellek ← bellek + yazmaç	16 + 5 = 21	4,2 µs

Tablo 2.2.1: Aynı işin yazmaçla ve bellekle yapılmasının saat turu karşılığı (8086 kılavuzu).

Tablo açık bir sonuç verir: aynı toplama, işlenenlerden biri bellekte olduğunda yaklaşık beş kat, sonuç belleğe yazıldığında yedi kat uzun sürer. `ADD [BX], AX` komutu belleği önce okur, sonra yazar; iki yol döngüsü harcar. Bu yüzden iyi yazılmış bir assembly programı sık kullanılan değerleri yazmaçlarda tutar ve belleğe yalnız gerektiğinde gider. Yüksek düzeyli dillerin derleyicileri de aynı ilkeyle çalışır: bir döngünün sayacını ve ara sonuçlarını mümkün olduğunca yazmaçlara yerleştirir.

Yazmaçların sayısı ise azdır. Her yazmaç işlemci yongasında yer kaplar. Komutun içinde de yazmacı seçen bir alan bulunur: 8086 komutlarında bu alan 3 bittir ve $2^3 = 8$ yazmaçtan birini seçer. Yazmaç sayısını artırmak hem yongayı büyütür hem komutları uzatır. 1978'in teknolojisinde 8086'nın tasarımcıları sekiz genel amaçlı yazmaçla yetinmiştir.

2.2.1.3 Bellekten belleğe işlem yoktur

8086'nın iki işlenenli komutlarında (MOV, ADD, SUB, AND, CMP ...) işlenenlerden en az biri yazmaç ya da sabit olmalıdır. Bir bellek hücresini doğrudan başka bir bellek hücresine kopyalayan ya da iki bellek hücresini toplayan bir MOV veya ADD yoktur. Bu kural komut biçiminden gelir: bir komutun içinde yalnız bir bellek adresini tarif edecek yer vardır.

```
x  DW 1
y  DW 2
   MOV x, y        ; HATA: bellekten belleğe MOV yok

   MOV AX, y       ; doğrusu: önce yazmaca
   MOV x, AX       ; sonra belleğe
```

Assembler ilk satırı kabul etmez ve bir hata iletisi verir. Doğru yol, değeri bir yazmaç üzerinden taşımaktır. Yazmaç burada bir aktarma istasyonu gibi çalışır. Bu kuralın tek istisnası dizgi komutlarıdır. `MOVSB` bir baytı DS:SI'dan ES:DI'ya doğrudan taşır. İki adresi de komutta yazmadan, örtük olarak SI ve DI yazmaçlarından alır.

> **Sık yapılan hata**
>
> Değişkenleri yüksek düzeyli dildeki gibi düşünüp `MOV toplam, sayi` ya da `ADD sonuc, deger` yazmak. İki işlenen de bellekte olduğu için assembler bu satırları kabul etmez. Değerlerden biri önce bir yazmaca alınır.

2.2.2 8086 yazmaç kümesi

2.2.2.1 On dört yazmaç ve grupları

8086'nın on dört yazmacı görevlerine göre dört grupta toplanır (Şekil 2.2.2):

- **Veri yazmaçları**: AX, BX, CX, DX. Her biri 16 bittir ve 8 bitlik iki yarıya bölünebilir: AH/AL, BH/BL, CH/CL, DH/DL.
- **İşaretçi ve indis yazmaçları**: SP (yığın işaretçisi), BP (taban işaretçisi), SI (kaynak indisi), DI (hedef indisi). Bunlar bellek içinde bir konumu gösterir; 8 bitlik parçaları yoktur.

- **Kesim yazmaçları**: CS (kod), DS (veri), SS (yığın), ES (ek). Her biri bellekteki bir kesimin başlangıcını belirler.
- **Özel yazmaçlar**: IP (komut işaretçisi) ve FLAGS (bayrak yazmacı).



Şekil 2.2.2: 8086 yazmaç kümesi. Dört veri yazmacı 8 bitlik yarılarıyla, dört işaretçi ve indis yazmacı, dört kesim yazmacı, IP ve FLAGS. Hepsi 16 bittir; yalnız AX, BX, CX ve DX ikiye bölünebilir.

Sayım şöyledir: 4 veri + 4 işaretçi/indis + 4 kesim + IP + FLAGS = 14. Bu derste ilk sekiz yazmaç, yani AX, BX, CX, DX, SP, BP, SI ve DI, **genel amaçlı yazmaçlar** olarak sayılır. Bu sekizinin ortak özelliği, MOV, ADD, AND gibi komutlarda işlenen olarak serbestçe kullanılabilmeleridir. `ADD SI, 4` ya da `MOV BP, AX` geçerli komutlardır. Kesim yazmaçları, IP ve FLAGS ise aritmetik komutlarda işlenen olamaz.

Yazmaç adlarındaki X harfi "genişletilmiş" anlamında kullanılmıştır. 8086'dan önceki 8080 işlemcisinin A, B, C, D gibi 8 bitlik yazmaçları vardı. 8086 bunları 16 bite genişletti ve AX, BX, CX, DX adlarını verdi; 8 bitlik yarılar H (yüksek) ve L (düşük) harfleriyle anıldı. Bu adlandırma 8080 programlarının 8086'ya kolay aktarılması için seçilmişti. Bugünkü x86 işlemcilerinde de sürer: 32 bitlik işlemcilerde EAX, 64 bitlik işlemcilerde RAX yazmacının en alt 16 biti hâlâ AX'tir.

2.2.2.2 Yazmaçların BIU ve EU'daki yeri

8086 iki birimden oluşur: belleğe ve G/Ç'ye erişen **yol arayüz birimi** (BIU) ile komutları çözüp yürüten **yürütme birimi** (EU). On dört yazmaç bu iki birime paylaştırılmıştır ve bu paylaşım, yazmaçların görevlerini anlamayı kolaylaştırır.

Birim	Yazmaçlar	Adres üretimindeki rolü
EU	AX, BX, CX, DX	veri; BX ayrıca bağıl konum tutabilir
EU	SP, BP, SI, DI	16 bitlik bağıl konum
EU	FLAGS	yok; ALU sonucunu ve kontrol bitlerini tutar
BIU	CS, DS, SS, ES	kesim başlangıcı (paragraf numarası)
BIU	IP	kod kesimi içindeki bağıl konum

Tablo 2.2.2: Yazmaçların bulunduğu birim ve adres üretimindeki rolü.

EU yalnız 16 bitlik sayılarla çalışır. Bir bellek işleneni gerektiğinde etkin adresi, yani kesim içindeki bağıl konumu, BX, BP, SI, DI ve komuttaki sabitten hesaplar ve BIU'ya verir. 20 bitlik fiziksel adresi kesim yazmacıyla birleştiren BIU'dur. Kesim yazmaçlarının ve IP'nin BIU'da durmasının nedeni budur: BIU, EU'dan bağımsız olarak sıradaki komut baytlarını CS:IP'den önceden getirir ve kuyruğa koyar. Bu düzen, "hangi yazmaç adresin hangi parçasını verir?" sorusunun cevabını da belirler. Kesim yazmacı adresin üst kısmını, işaretçi ve indis yazmaçları kesim içindeki konumu verir.

2.2.2.3 Genel amaçlı yazmaçlar ve örtük görevler

Genel amaçlı yazmaçlar adlarından anlaşılacağı gibi çoğu komutta birbirinin yerine kullanılabilir. `ADD AX, BX` yerine `ADD SI, DX` yazmak mümkündür. Bununla birlikte bazı komutlar belirli bir yazmacı, komutta adı geçmeden kullanır.

> **Tanım 2.2.2**
>
> **Örtük işlenen** (*implicit operand*), komutun yazılışında görünmeyen ama komutun tanımı gereği okunan ya da yazılan işlenendir. `MUL BX` komutunda yalnız BX yazılır; çarpılan AX ve sonucun yüksek yarısını alan DX örtük işlenenlerdir.

Tablo 2.2.3 sekiz genel amaçlı yazmacın adlarını ve örtük görevlerini bir arada verir. Yazmaçların İngilizce adları görevlerinin kısaltmasıdır ve tabloyu akılda tutmayı kolaylaştırır.

Yazmaç	Ad	8 bitlik yarılar	Örtük kullanım (örnek)
`AX`	akümülatör (*accumulator*)	AH, AL	MUL, DIV, IN, OUT, LODS/STOS
`BX`	taban (*base*)	BH, BL	bellek adresleme `[BX]`, XLAT
`CX`	sayaç (*counter*)	CH, CL	LOOP, REP, kaydırma sayısı CL
`DX`	veri (*data*)	DH, DL	MUL/DIV yüksek yarısı, port adresi
`SP`	yığın işaretçisi (*stack pointer*)	—	PUSH, POP, CALL, RET
`BP`	taban işaretçisi (*base pointer*)	—	yığın çerçevesi `[BP+n]`
`SI`	kaynak indisi (*source index*)	—	dizgi kaynağı DS:SI
`DI`	hedef indisi (*destination index*)	—	dizgi hedefi ES:DI

Tablo 2.2.3: Genel amaçlı yazmaçlar ve örtük kullanımları.

Örtük görevler 8086'nın komut kodlamasını kısaltmak için seçilmiştir. LOOP komutu sayacın hangi yazmaçta olduğunu belirtmek zorunda kalsaydı, her LOOP'a bir yazmaç alanı eklenmesi gerekirdi. Sayaç her zaman CX olduğu için LOOP yalnız iki bayttır: işlem kodu ve atlama uzaklığı. Bedeli esnekliktir. Bir programda CX başka bir iş için kullanılıyorsa LOOP'tan önce saklanmalıdır. Örtük görevleri bilmeyen bir programcı, bir MUL komutunun DX'teki değeri sessizce sildiğini fark etmeyebilir.

2.2.2.4 8 ve 16 bitlik kullanım: AX = AH : AL

AX, BX, CX ve DX hem 16 bitlik tek bir yazmaç hem de 8 bitlik iki ayrı yazmaç olarak kullanılabilir. AX'in yüksek sıradaki 8 biti (bit 15–8) **AH**, düşük sıradaki 8 biti (bit 7–0) **AL** adını taşır. Bunlar ayrı donanım değildir; aynı 16 flip-flop'un iki yarısıdır. Bu yüzden şu eşitlik her an geçerlidir:

> AX = AH × 256 + AL (onaltılık: AX = AH : AL)

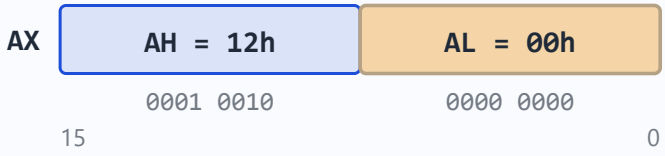
Bir komut AL'ye yazdığında yalnız alt 8 flip-flop yüklenir; üst yarı yerinde kalır. Bu davranışı görmenin en iyi yolu kısa bir programı izlemektir. Kod, sunumdaki programın aynısıdır:

```
MOV AX, 1234h   ; AH=12h, AL=34h
MOV AL, 0FFh    ; yalnız alt yarı
ADD AL, 1       ; 8 bitlik toplama
MOV AX, 12FFh
ADD AX, 1       ; 16 bitlik toplama
```

Satır	Komut	AX	AH	AL	CF	ZF
1	`MOV AX, 1234h`	1234h	12h	34h	0	0
2	`MOV AL, 0FFh`	12FFh	12h	**FFh**	0	0
3	`ADD AL, 1`	1200h	12h	**00h**	**1**	**1**
4	`MOV AX, 12FFh`	12FFh	12h	FFh	1	1
5	`ADD AX, 1`	1300h	**13h**	00h	**0**	**0**

Tablo 2.2.4: AX = AH : AL programının izlemesi (her satır yürütüldükten sonraki durum).

Birinci satır AX'e 1234h yazar. Onaltılık gösterimde her iki basamak bir bayttır; bu yüzden AH = 12h, AL = 34h olur. İkinci satır yalnız AL'yi değiştirir: AL = FFh, AH = 12h kalır, AX = 12FFh olur. Üçüncü satır en öğretici olanıdır. `ADD AL, 1` 8 bitlik bir toplamadır: FFh + 01h = 100h. Sonuç 9 bit tutar ama hedef 8 bittir. AL'ye alt 8 bit, yani 00h yazılır. Taşan 1 AH'ye eklenmez; **elde bayrağına** (CF) gider. Sonuç sıfır olduğu için sıfır bayrağı da kurulur (ZF = 1). AX artık 1200h'dir (Şekil 2.2.3).



Şekil 2.2.3: `ADD AL, 1` sonrasında AX. AL = FFh + 1 = 00h olur; elde AH'ye geçmez, CF'ye gider. AH = 12h yerinde kalır.

Dördüncü satır AX'i yeniden 12FFh yapar. MOV bayrakları değiştirmediği için CF = 1 ve ZF = 1 korunur. Beşinci satır aynı "+1" işlemini bu kez 16 bit üzerinde yapar: 12FFh + 0001h = 1300h. Alt bayttan çıkan elde bu sefer AH'ye geçer ve 12h, 13h olur. 16 bitlik sonuç 16 bite sığdığı için CF = 0 olur. Sonuç sıfır olmadığından ZF de 0'dır.

Bu iki toplamanın farkı, ALU'nun hangi genişlikte çalıştığından gelir. İşlemci işlem genişliğini işlenenlerin adından anlar: AL 8 bitlik, AX 16 bitlik bir yazmaçtır. Komut kodunda bu bilgi tek bir bitle, W bitiyle taşınır: W = 0 bayt işlemi, W = 1 sözcük işlemi demektir. `ADD AL, 1` komutu `04 01`, `ADD AX, 1` komutu `05 01 00` olarak kodlanır; işlem kodları yalnız en düşük bitte ayrılır.

Örnek 2.2.1

BX = 12FFh iken `INC BL` çalışırsa BX kaç olur?

INC 8 bitlik BL üzerinde çalışır: FFh + 1 = 100h, BL'ye 00h yazılır. Elde BH'ye geçmez; BH = 12h kalır ve BX = 1200h olur. ZF = 1'dir. INC komutu elde bayrağını hiç değiştirmez, CF önceki değerini korur. CF önceden 1 yapılmışsa INC'ten sonra da 1 kalır.

8 bitlik yarılar bellekteki düşük bayt önce düzeniyle de ilişkilidir. `MOV [0300h], AX` komutu AX = 1234h iken çalışırsa 0300h adresine AL'deki 34h, 0301h adresine AH'deki 12h yazılır. Düşük bayt düşük adrese gider. Bellekten okurken de aynı kural geçerlidir: `MOV AX, [0300h]` AL'yi 0300h'den, AH'yi 0301h'den doldurur.

Sık yapılan hata

AL'ye yazıldığında AX'in tamamının değiştiğini sanmak. `MOV AL, 5` AH'ye dokunmaz. AX daha önce 1234h idiyse komuttan sonra 1205h olur, 0005h olmaz. AX'in bütünüyle 5 olması isteniyorsa `MOV AX, 5` yazılır.

Yalnız dört veri yazmacının yarıları vardır. `MOV SL, 5` ya da `MOV BPL, 0` gibi bir yazım 8086'da yoktur. SI'nın alt baytına erişmek isteyen bir program SI'yı önce AX'e kopyalar ve AL üzerinde çalışır.

2.2.3 Veri yazmaçları

2.2.3.1 AX: akümülatör

AX, 8086'nın en çok kullanılan yazmacıdır. Adını eski işlemcilerdeki **akümülatörden** (*accumulator*) alır. Akümülatörlü bir işlemcide her aritmetik işlem bir işleneni akümülatörden alır ve sonucu yine oraya yazar; sonuçlar orada "birikir". 8086 bu zorunluluğu kaldırmıştır: `ADD BX, CX` akümülatöre dokunmadan çalışır. Yine de birçok komut AX'i ya da AL'yi örtük olarak kullanır:

- `MUL`, `IMUL`, `DIV`, `IDIV`: çarpılan ve bölünen AX'te (ya da AL'de, DX:AX'te) durur, sonuç AX'e yazılır.
- `IN` ve `OUT`: porttan okunan veri AL'ye ya da AX'e gelir; porta yazılacak veri AL'de ya da AX'te bulunur.
- `CBW`: AL'deki işaretli baytı AX'e genişletir. `DAA` ve `DAS`: AL'deki BCD sonucunu düzeltir.
- `LODSB` baytı AL'ye yükler, `STOSB` AL'deki baytı belleğe yazar.

AX'in ayrıcalığı komut uzunluğunda da görülür. 8086'nın birçok komutunun akümülatöre özel kısa bir biçimi vardır. Tablo 2.2.5 assembler'ın ürettiği makine kodlarını yan yana koyar.

Komut	Makine kodu	Uzunluk
`ADD AX, 1234h`	`05 34 12`	3 bayt
`ADD BX, 1234h`	`81 C3 34 12`	4 bayt
`MOV AX, [0300h]`	`A1 00 03`	3 bayt
`MOV BX, [0300h]`	`8B 1E 00 03`	4 bayt

Tablo 2.2.5: Akümülatörlü ve akümülatörsüz komutların makine kodu.

Genel biçimde işlem kodundan sonra işlenenleri tarif eden bir bayt gelir (ModR/M baytı: `C3`, `1E`). Akümülatör biçiminde hedefin AX olduğu işlem kodunun içinde saklıdır; bu bayta gerek kalmaz. Bir bayt az, hem bellekte yer hem de önden getirme kuyruğunda bir yol döngüsü kazancıdır. Bu yüzden derleyiciler ve deneyimli programcılar sık yapılan hesapları AX üzerinde kurar.

Sunumdaki kısa örnek AX'i kendisiyle toplar:

```
ADD AX, AX      ; AX = AX + AX
MOV AX, 0005h
ADD AX, AX      ; AX = 000Ah
MOV AH, 4Ch     ; DOS: programı bitir
INT 21h
```

Komut	AX	Açıklama
`ADD AX, AX`	0000h	0 + 0 = 0; ZF = 1, PF = 1
`MOV AX, 0005h`	0005h	AX'e 5 yazılır
`ADD AX, AX`	000Ah	5 + 5 = 10 = 000Ah; ZF = 0
`MOV AH, 4Ch`	4C0Ah	yalnız AH değişir; AL = 0Ah kalır
`INT 21h`	4C0Ah	DOS 4Ch hizmeti: program biter, çıkış kodu AL = 10

Tablo 2.2.6: AX'i kendisiyle toplayan programın izlemesi (.COM programı, başlangıçta AX = 0).

Bir değeri kendisiyle toplamak 2 ile çarpmaktır. İkili sayıda bu, bütün bitlerin bir sola kaymasıdır: 0101b + 0101b = 1010b. Son iki satır da ilginçtir. DOS'un 4Ch hizmeti AH'de hizmet numarasını, AL'de programın çıkış kodunu bekler. `MOV AH, 4Ch` AL'ye dokunmadığı için program 10 çıkış koduyla biter. Çıkış kodunun 0 olması isteniyorsa `MOV AX, 4C00h` yazılır; bu tek komut hem AH'yi hem AL'yi doldurur.

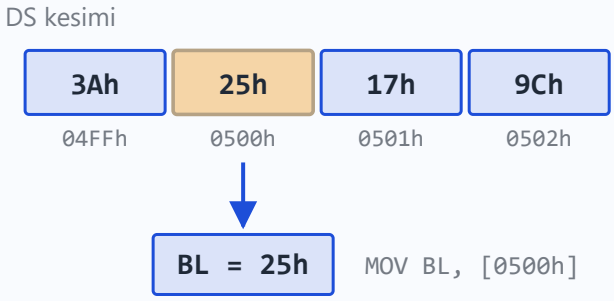
2.2.3.2 BX: taban yazmacı

BX de AX gibi 16 bittir ve BH ile BL olarak ikiye ayrılır. Onu öbür veri yazmaçlarından ayıran özellik, **bellek adreslemede** kullanılabilmesidir. Köşeli parantez içindeki bir yazmaç, yazmacın değerinin bir bağıl konum olarak kullanılacağını söyler. `[BX]`, "BX'in gösterdiği bellek hücresi" demektir.

> **Tanım 2.2.3**
>
> **Taban yazmacı** (*base register*), dolaylı adreslemede adresin taban kısmını tutan yazmaçtır. 8086'da taban yazmaçları BX ve BP'dir. Köşeli parantez içinde kullanılabilen yazmaçlar yalnız BX, BP, SI ve DI'dır; `[AX]`, `[CX]`, `[DX]` yazılamaz.

Bu kısıt da komut kodlamasından gelir. 8086'nın adresleme alanı (R/M alanı) 3 bitlik bir kodla sekiz adres biçiminden birini seçer: `[BX+SI]`, `[BX+DI]`, `[BP+SI]`, `[BP+DI]`, `[SI]`, `[DI]`, `[BP]` (ya da doğrudan adres) ve `[BX]`. Listede AX, CX ve DX yoktur. `MOV AX, [CX]` satırı bu yüzden assembler'dan hata alır. CX'teki bir adresi kullanmak gerekiyorsa önce BX'e kopyalanır: `MOV BX, CX` ve `MOV AX, [BX]`.

Köşeli parantezin anlamı yeni başlayanların en sık karıştırdığı noktadır. Şekil 2.2.4 farkı gösterir.



Şekil 2.2.4: `MOV BL, [0500h]` komutu DS:0500h adresindeki 25h baytını BL'ye kopyalar. BL'ye 500h sayısı yazılmaz; köşeli parantez "bu adresteki değer" anlamına gelir.

`MOV BL, [0500h]` bellekten okur: DS:0500h adresinde 25h varsa BL = 25h olur. `MOV BX, 0500h` ise belleğe hiç gitmez; BX'e 0500h sayısını yazar. Birinci komut 4 bayttır ve bir yol döngüsü harcar. İkincisi 3 bayttır ve sayı komutun içindedir. Bu ayrım, sabitle bellek içeriği arasındaki temel ayrımdır. Bir sayıyı yazmaca koymak için köşeli parantez kullanılmaz.

Sunumdaki izleme programı BX'i bir işaretçi olarak kullanır. Bellekte 0500h–0503h adreslerinde 25h, 17h, 00h ve 9Ch baytları bulunsun.

```
MOV BX, 0500h     ; BX = adres
MOV AL, [BX]      ; AL = [0500h]
MOV DL, [0501h]   ; doğrudan adres
ADD AL, DL        ; AL = 25h + 17h
MOV [BX+2], AL    ; [0502h] = AL
```

Satır	Komut	BX	AL	DL	0500h–0503h
1	`MOV BX, 0500h`	**0500h**	00h	00h	25 17 00 9C
2	`MOV AL, [BX]`	0500h	**25h**	00h	25 17 00 9C
3	`MOV DL, [0501h]`	0500h	25h	**17h**	25 17 00 9C
4	`ADD AL, DL`	0500h	**3Ch**	17h	25 17 00 9C
5	`MOV [BX+2], AL`	0500h	3Ch	17h	25 17 **3C** 9C

Tablo 2.2.7: BX ile dolaylı erişimin izlemesi. Bellek sütunu 0500h–0503h baytlarıdır.

Birinci satır BX'e bir sayı yazar; bu sayı bundan sonra bir adres olarak kullanılacaktır. İkinci satırda `[BX]`, BX'in gösterdiği bayttır: AL = 25h. Üçüncü satır aynı türden bir okumayı adresi komutun içine yazarak yapar ve DL = 17h olur. Dördüncü satır iki yazmacı toplar: 25h + 17h = 3Ch. Beşinci satırda `[BX+2]` etkin adresi 0500h + 2 = 0502h'dir; AL bu adrese yazılır. BX değişmez, değişen bellektir (Şekil 2.2.5).



Şekil 2.2.5: Programın sonunda bellek. BX hâlâ 0500h'yi gösterir; `[BX+2]` ile 0502h adresine 3Ch yazılmıştır.

Programdaki adresler bağıl konumdur. İşlemci bunları DS kesimi içinde arar, çünkü BX ile yapılan erişimlerin varsayılan kesimi DS'dir. emu8086 bir .COM programını DS = 0700h ile yükler. Bu durumda 0500h bağıl konumunun fiziksel adresi 0700h × 10h + 0500h = 07000h + 0500h = 07500h'dir. Beşinci satırda yazılan bayt emu8086'nın bellek penceresinde 07502h fiziksel adresinde görünür.

BX'in adres tutabilmesi onu dizi ve tablo işlemlerinde vazgeçilmez yapar. BX bir dizinin başını gösterirken `[BX+SI]` biçimi dizinin SI'ncı elemanına erişir; SI her turda artırılarak dizi gezilir. XLAT komutu da BX'i örtük kullanır: AL'yi BX'in gösterdiği tablonun indisi sayar ve `AL ← [BX+AL]` işini tek komutta yapar.

2.2.3.3 CX: sayaç yazmacı

CX, CH ve CL olarak ikiye ayrılan 16 bitlik bir yazmaçtır. Döngü, tekrar ve sayma işlerinde kullanılır; adı da buradan gelir (*count register*). Üç komut CX'i örtük olarak kullanır:

- `LOOP etiket` : önce CX ← CX − 1 yapar, sonra CX ≠ 0 ise etikete atlar. Bayrakları etkilemez.
- `REP` öneki: arkasındaki dizgi komutunu CX kez tekrarlar; her turda CX bir azalır.
- Kaydırma ve döndürme komutları: sayı 1'den büyükse CL'de verilir: `SHL AX, CL`.

> **Tanım 2.2.4**
>
> `LOOP etiket` komutu şu iki adımı tek komutta yapar: (1) CX ← CX − 1; (2) CX ≠ 0 ise IP ← etiketin adresi, değilse bir sonraki komuta geçilir. Azaltma, sınamadan **önce** yapılır.

Sunumdaki program 5 + 4 + 3 + 2 + 1 toplamını hesaplar. CX hem sayaç hem de toplanan değerdir.

```
MOV CX, 5          ; sayaç
MOV AX, 0          ; toplam
TEKRAR: ADD AX, CX
LOOP TEKRAR        ; CX--, ≠0 ise atla
MOV DX, AX         ; sonuç
```

Adım	Komut	CX	AX	DX	Sonraki
1	`MOV CX, 5`	0005h	0000h	0000h	2. satır
2	`MOV AX, 0`	0005h	0000h	0000h	TEKRAR
3	`ADD AX, CX`	0005h	**0005h**	0000h	LOOP
4	`LOOP TEKRAR`	**0004h**	0005h	0000h	atla
5	`ADD AX, CX`	0004h	**0009h**	0000h	LOOP
6	`LOOP TEKRAR`	**0003h**	0009h	0000h	atla
7	`ADD AX, CX`	0003h	**000Ch**	0000h	LOOP
8	`LOOP TEKRAR`	**0002h**	000Ch	0000h	atla
9	`ADD AX, CX`	0002h	**000Eh**	0000h	LOOP
10	`LOOP TEKRAR`	**0001h**	000Eh	0000h	atla
11	`ADD AX, CX`	0001h	**000Fh**	0000h	LOOP
12	`LOOP TEKRAR`	**0000h**	000Fh	0000h	atlamaz
13	`MOV DX, AX`	0000h	000Fh	**000Fh**	—

Tablo 2.2.8: CX ve LOOP ile 5 + 4 + 3 + 2 + 1 toplamının izlemesi.

Döngü gövdesi CX = 5, 4, 3, 2, 1 değerleriyle beş kez çalışır. Beşinci LOOP CX'i 1'den 0'a indirir ve atlamaz; program döngüden çıkar. Gövde CX = 0 değeriyle **hiç** çalışmaz. Toplam 15 = 000Fh'dir. Gauss'un formülüyle sağlama: 5 × 6 / 2 = 15.

Döngünün süresi de kolayca hesaplanır. LOOP atladığında 17, atlamadığında 5 saat turu sürer; `ADD AX, CX` 3 turdur. Beş tur için 5 × 3 + 4 × 17 + 5 = 88 tur harcanır. Bu değer, gövde uzadıkça döngü denetiminin toplam süreye oranının nasıl küçüldüğünü de gösterir.

CX = 0 tuzağı

LOOP sınamadan önce azalttığı için CX = 0 ile döngüye girmek beklenmedik bir sonuç verir. İlk LOOP, 0000h − 1 = FFFFh hesaplar. FFFFh sıfır değildir; işlemci atlar. Döngü bundan sonra FFFFh, FFFEh, …, 0001h değerleriyle döner ve toplam tur sayısı 65.536 olur.

```
    MOV CX, 0
    MOV BX, 0
    MOV DX, 0
D:  ADD BX, 1      ; tur sayacı (alt sözcük)
    ADC DX, 0      ; üst sözcük
    LOOP D
```

Bu program tur sayısını 32 bitlik DX:BX çiftinde sayar. Program bittiğinde DX:BX = 0001:0000h = 65.536 olur. Tur sayısının 16 bite sığmadığına dikkat edilmelidir; sayaç yalnız BX olsaydı 65.536 tur sonunda yeniden 0'a dönerdi ve döngünün hiç çalışmadığı sanılırdı.

Bu tuzaktan korunmanın yolu `JCXZ` komutudur: “CX sıfırsa atla”. Döngüden hemen önce konur ve sayaç sıfırsa döngünün tamamını atlar.

```
    MOV CX, 0
    MOV AX, 0
    JCXZ BOS       ; CX = 0 ise döngüye hiç girme
D:  ADD AX, CX
    LOOP D
BOS: MOV DX, AX
```

Bu programda JCXZ'den doğrudan BOS etiketine geçilir; döngü gövdesi bir kez bile çalışmaz ve AX = DX = 0 kalır. JCXZ de LOOP gibi bayraklara bakmaz, doğrudan CX'i sınar.

CL ile kaydırma

8086'da kaydırma ve döndürme komutları sayıyı ya 1 olarak ya da CL'de alır. `SHL AX, 4` yazımı 8086'da yoktur; 80186 ile gelmiştir. 8086 için aynı iş şöyle yazılır:

```
MOV AX, 0003h
MOV CL, 4
SHL AX, CL       ; AX = 0030h
```

Dört bitlik sola kaydırma $2^4 = 16$ ile çarpmaktır: 3 × 16 = 48 = 0030h. Onaltılık gösterimde bu, sayının sonuna bir 0 eklemek demektir. Aynı işlem kesim yazmacını fiziksel adrese çevirirken de karşımıza çıkar.

2.2.3.4 DX: veri yazmacı

DX, DH ve DL olarak ikiye ayrılır. İki önemli örtük görevi vardır: geniş çarpma ve bölmede AX'in yüksek yarısını tutmak ve G/Ç port numarasını taşımak.

Çarpma. İki 16 bitlik sayının çarpımı 32 bite kadar çıkabilir: FFFFh × FFFFh = FFFE0001h. Sonuç tek bir 16 bitlik yazmaca sığmaz. 8086 bu yüzden 16 bitlik çarpmanın sonucunu iki yazmaca böler: yüksek 16 bit DX'e, düşük 16 bit AX'e yazılır. Bu çift **DX:AX** diye gösterilir (Şekil 2.2.6). 8 bitlik çarpmada ise sonuç 16 bittir ve tamamı AX'e sığar: `MUL BL` → AX = AL × BL. Bu durumda DX değişmez.



Şekil 2.2.6: `MUL BX` : AX ile BX çarpılır, 32 bitlik sonuç DX:AX çiftine yazılır. DX yüksek, AX düşük yarıdır.

Bölme. 16 bitlik bölmede bölünen 32 bitlik DX:AX'tir. Bölüm AX'e, kalan DX'e yazılır. Örneğin DX = 0012h, AX = 3405h, BX = 0100h iken `DIV BX`, 00123405h ÷ 0100h hesaplar. Bölüm 1234h, kalan 0005h'tir: AX = 1234h, DX = 0005h olur. Bölüm 16 bite sığmazsa işlemci tip 0 kesmesi (bölme hatası) üretir. Bu yüzden 16 bitlik bir sayıyı bölmeden önce DX sıfırlanır; DX'te artık bir değer kalmışsa bölünen beklenenden çok büyük olur.

Port numarası. IN ve OUT komutları port numarasını iki yolla alır. 0–255 arasındaki portlar komutun içine sabit olarak yazılabilir: `IN AL, 60h`. 255'ten büyük portlar için numara DX'e konur: `OUT DX, AL`. PC'de birinci seri port (COM1) 03F8h'den, yazıcı portu 0378h'den başlar; bunlara yalnız DX ile erişilir. DX'in 16 bit olması 65.536 portluk G/Ç adres alanını tümüyle kapsar.

DOS çağrıları. INT 21h hizmetlerinin birçoğu bir bellek adresini DX'te bekler. Ekrana dizgi yazan 09h hizmeti dizginin adresini DS:DX'ten alır: `LEA DX, mesaj`.

Sunumdaki program bu görevlerden ikisini birlikte gösterir:

```
MOV AX, 1234h
MOV BX, 0100h
MUL BX          ; DX:AX = AX*BX
MOV AX, 00FFh
MOV BL, 2
MUL BL          ; AX = AL*BL
MOV DX, 03F8h   ; port adresi
OUT DX, AL      ; porta yaz
```

Satır	Komut	AX	BX	DX	CF	OF	SF ZF AF PF
1	`MOV AX, 1234h`	1234h	0000h	0000h	0	0	0 0 0 0
2	`MOV BX, 0100h`	1234h	0100h	0000h	0	0	0 0 0 0
3	`MUL BX`	**3400h**	0100h	**0012h**	**1**	**1**	? ? ? ?
4	`MOV AX, 00FFh`	00FFh	0100h	0012h	1	1	? ? ? ?
5	`MOV BL, 2`	00FFh	**0102h**	0012h	1	1	? ? ? ?
6	`MUL BL`	**01FEh**	0102h	0012h	1	1	? ? ? ?
7	`MOV DX, 03F8h`	01FEh	0102h	**03F8h**	1	1	? ? ? ?
8	`OUT DX, AL`	01FEh	0102h	03F8h	1	1	? ? ? ?

Tablo 2.2.9: MUL ve OUT programının izlemesi. “?” MUL sonrasında tanımsız olan bayrakları gösterir.

Üçüncü satırda 1234h × 0100h = 00123400h hesaplanır. 0100h ile çarpmak onaltılık sayıyı iki basamak sola kaydırmaktır. 32 bitlik sonuç ikiye bölünür: DX = 0012h, AX = 3400h. MUL, sonucun yüksek yarısı sıfır değilse CF = OF = 1 yapar. Bu iki bayrak “sonuç alt yarıya sığmadı, DX'e de bakılmalı” anlamındadır. MUL'den sonra SF, ZF, AF ve PF **tanımsızdır**: Intel bu bayrakların değerini belirtmemiştir ve program onlara güvenmemelidir. Tablodaki “?” işareti bunu gösterir.

Beşinci satır yalnız BL'yi değiştirir. BX önceden 0100h olduğundan BH = 01h kalır ve BX = 0102h olur. Altıncı satır 8 bitlik çarpmadır: AL × BL = FFh × 02h = 255 × 2 = 510 = 01FEh. Sonuç AX'in tamamına yazılır; DX'teki 0012h olduğu gibi durur. Yüksek yarı AH = 01h sıfır olmadığı için yine CF = OF = 1'dir.

Yedinci satırda DX bu kez bir veri değil, bir port numarası tutar: 03F8h, birinci seri portun veri yazmacıdır. Sekizinci satır AL'deki FEh'yi bu porta gönderir. OUT, AX'in yalnız alt baytını kullanır; AH'deki 01h porta gitmez. Porta tek bir bayt, FEh yazılır.

Sık yapılan hata

MUL'ün sonucunun yalnız AX'te olduğunu sanıp DX'i başka bir iş için kullanmaya devam etmek. 16 bitlik bir MUL, DX'teki değeri her zaman siler; sonuç küçük olsa bile DX = 0 yazılır. DX'te saklanması gereken bir değer varsa MUL'den önce yığına itilir.

Örnek 2.2.2

AL = 10h ve BL = 03h iken `MUL BL` sonrasında AX, CF ve OF nedir?

10h × 03h = 16 × 3 = 48 = 0030h. AX = 0030h olur. Yüksek yarı AH = 00h olduğu için sonuç AL'ye sığmıştır ve CF = OF = 0'dır. Program bu durumda yalnız AL'ye bakarak devam edebilir. CF = 1 olsaydı sonucun AX'in tamamında aranması gerekirdi.

2.2.4 İşaretçi ve indis yazmaçları

SP, BP, SI ve DI yazmaçları 16 bittir ve 8 bitlik yarıları yoktur. Dördü de çoğunlukla bir veri değil, bir **bağıl konum** tutar: bellekteki bir hücrenin, ait olduğu kesimin başından itibaren uzaklığı. SP ve BP yığın kesiminde (SS), SI ve DI ise veri kesimlerinde (DS, ES) çalışır. Aritmetik komutlarda da kullanılabilirler: `ADD SI, 2` bir diziyi gezerken indisi ilerletmenin olağan yoludur.

2.2.4.1 SP: yığın işaretçisi

Tanım 2.2.5

Yığın (*stack*), son giren ilk çıkar (LIFO) düzeninde çalışan bir bellek bölgesidir. **Yığın işaretçisi** (SP, *stack pointer*), yığının en üstündeki öğenin bağıl konumunu tutar. Yığının fiziksel adresi her zaman `SS:SP` çiftiyle verilir.

8086'nın yığını üç özelliğiyle tanınır. Birincisi, öğeler her zaman **16 bittir**: `PUSH AX` geçerlidir, `PUSH AL` değildir. Assembler `PUSH AL` satırını hatalı sayar. İkincisi, yığın belleğin **aşağısına doğru** büyür; her PUSH daha düşük bir adrese yazar. Üçüncüsü, işlem sırası sabittir:

- `PUSH kaynak` : önce SP ← SP − 2, sonra [SS:SP] ← kaynak. Düşük bayt SP'ye, yüksek bayt SP + 1'e yazılır.
- `POP hedef` : önce hedef ← [SS:SP], sonra SP ← SP + 2.

SP her zaman yığındaki son geçerli öğeyi gösterir. Yığına yeni bir öğe konacağı zaman önce yer açılır, sonra yazılır. POP, okuduğu hücreyi silmez; değer bellekte durur ama SP onun üstüne çıktığı için artık yığının parçası sayılmaz. Bir sonraki PUSH o hücrenin üzerine yazar.

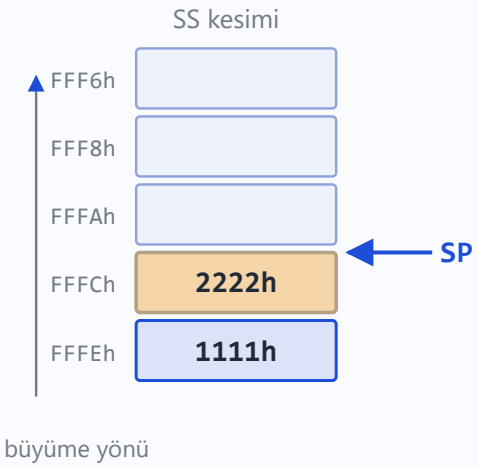
Aşağıdaki programı izlemek bu kuralları somutlaştırır. Yığının boş olduğu ve SP'nin 0000h'den başladığı varsayılsın.

```
MOV  AX, 1111h
PUSH AX
MOV  BX, 2222h
PUSH BX
POP  CX
POP  DX
```

Komut	SP	Yığın (adres: değer)	CX	DX
`MOV AX, 1111h`	0000h	boş	0000h	0000h
`PUSH AX`	**FFFEh**	FFFEh: 1111h	0000h	0000h
`MOV BX, 2222h`	FFFEh	FFFEh: 1111h	0000h	0000h
`PUSH BX`	**FFFCh**	FFFCh: 2222h, FFFEh: 1111h	0000h	0000h
`POP CX`	**FFFEh**	FFFEh: 1111h	**2222h**	0000h
`POP DX`	**0000h**	boş	2222h	**1111h**

Tablo 2.2.10: PUSH ve POP izlemesi (başlangıçta SP = 0000h). Yığın sütunu SP'den yukarı doğru öğeleri verir.

İlk PUSH, SP'yi 0000h − 2 hesabıyla FFFEh yapar. 16 bitlik aritmetikte 0'dan aşağı inmek en üste sarmak demektir; bu yüzden SP = 0000h, 64 KB'lık yığın kesiminin tamamının kullanılabildiği bir başlangıçtır. 1111h değeri FFFEh–FFFFh baytlarına yazılır. İkinci PUSH, SP'yi FFFCh'ye indirir ve 2222h'yi oraya koyar. Şekil 2.2.7 bu andaki yığını gösterir. SP en üstteki öğeyi, yani en son itilen 2222h'yi gösterir.



Şekil 2.2.7: İki PUSH'tan sonra yığın kesimi: FFFEh'de 1111h, FFFCh'de 2222h. SP = FFFCh en üstü gösterir. Yığın düşük adreslere doğru büyür.

POP'lar ters sırada çalışır. `POP CX` en üstteki 2222h'yi alır, SP FFFEh'ye çıkar. `POP DX` 1111h'yi alır, SP yeniden 0000h olur. Sonuçta CX = 2222h ve DX = 1111h'dir: değerler itildikleri sıranın tersiyle geri gelmiştir. LIFO düzeninin anlamı budur. Bir programda AX ve BX'i saklayıp geri almak için `PUSH AX`, `PUSH BX` ve ardından `POP BX`, `POP AX` yazılır; POP sırası PUSH sırasının tersidir.

"Yığın boşken SP = FFFEh" bilgisi genel bir kural değildir. DOS bir .COM programını yüklerken SP'yi FFFEh yapar ve FFFEh adresine 0000h sözcüğünü koyar. Böylece programın sonundaki `RET` bu sıfırı IP'ye alır ve programın başındaki DOS çıkış koduna döner. emu8086 da .COM programlarını bu düzenle başlatır. Aynı program bir .COM dosyası olarak çalıştırılsa SP önce FFFCh'ye, sonra FFFAh'ya inerdi; değerler ve sıralar değişmezdi.

SP'nin örtük kullanıcıları yalnız PUSH ve POP değildir. `CALL` dönüş adresini yığına iter, `RET` onu yığından alır. Bir kesme FLAGS, CS ve IP'yi iter; `IRET` üçünü geri alır. `PUSHF` ve `POPF` bayrak yazmacını taşır. Bütün bu komutlar SP'yi kendileri günceller. Programcı SP'ye doğrudan yazarsa (`MOV SP, …`) yığının konumunu değiştirmiş olur; bu yalnız program başında ya da yığın kesimini değiştirirken yapılır.

> **Sık yapılan hata**
>
> PUSH ve POP sayılarını eşitlememek. Bir prosedür içinde fazladan bir PUSH bırakılırsa RET dönüş adresi yerine o değeri IP'ye alır ve program rastgele bir adrese atlar. Bir koşullu dalda POP'un atlanması da aynı sonucu verir. Her çıkış yolunda yığın, girişteki hâline dönmelidir.

2.2.4.2 BP: taban işaretçisi

BP (*base pointer*), yığın üzerindeki **sabit bir konumu** göstermek için kullanılır. BP ile yapılan bellek erişimleri varsayılan olarak SS kesimini kullanır: `[BP+4]`, SS:BP+4 adresidir. BX ile yazılan `[BX+4]` ise DS kesimine gider. İki taban yazmacının farkı yalnız bu varsayılan kesimdir.

SP yığının tepesini gösterir ama prosedür içinde sürekli değişir: her PUSH ve POP onu kaydırır. Parametrelere SP'ye göre erişmek bu yüzden zahmetlidir; bir PUSH'tan sonra aynı parametrenin uzaklığı 2 bayt artar. BP bu sorunu çözer. Prosedürün başında SP'nin o anki değeri BP'ye kopyalanır. Bundan sonra SP ne kadar değişirse değişsin BP sabit kalır ve her parametre BP'den hep aynı uzaklıktadır.

```
        MOV  BX, 1234h
        PUSH BX          ; parametre
        CALL TOPLA
        MOV  DX, AX      ; sonuç
        RET

TOPLA PROC
  PUSH BP
  MOV  BP, SP     ; çerçeve tabanı
  MOV  AX, [BP+4] ; 1. parametre
  POP  BP
  RET  2
TOPLA ENDP
```

Prosedür gövdesi sunumdakinin aynısıdır. Çağıran kısım, parametreyi yığına iten ve sonucu DX'e alan üç satırla eklenmiştir. Program .COM olarak, SP = FFFEh ile başlar.

Komut	SP	BP	AX	Yığın (adres: değer)
`MOV BX, 1234h`	FFFEh	0000h	0000h	FFFE: 0000
`PUSH BX`	**FFFCh**	0000h	0000h	FFFC: 1234, FFFE: 0000
`CALL TOPLA`	**FFFAh**	0000h	0000h	FFFA: 0107, FFFC: 1234, ...
`PUSH BP`	**FFF8h**	0000h	0000h	FFF8: 0000, FFFA: 0107, FFFC: 1234, ...
`MOV BP, SP`	FFF8h	**FFF8h**	0000h	aynı
`MOV AX, [BP+4]`	FFF8h	FFF8h	**1234h**	aynı
`POP BP`	**FFFAh**	**0000h**	1234h	FFFA: 0107, FFFC: 1234, ...
`RET 2`	**FFFEh**	0000h	1234h	FFFE: 0000
`MOV DX, AX`	FFFEh	0000h	1234h	FFFE: 0000

Tablo 2.2.11: Yığın çerçevesinin izlemesi. Yığın sütunu SP'den yukarı doğrudur.

CALL, dönüş adresi olan 0107h'yi yığına iter. Bu, CALL'dan sonraki komutun, yani `MOV DX, AX`'in adresidir: program 0100h'den başlar, `MOV BX, 1234h` 3 bayt, `PUSH BX` 1 bayt, `CALL` 3 bayttır ve 0100h + 7 = 0107h eder. Prosedür girişindeki `PUSH BP`, çağıranın BP değerini saklar. `MOV BP, SP` ile BP = FFF8h olur ve çerçevenin tabanı sabitlenir. Artık çerçeve şöyle okunur:

- `[BP]` = FFF8h: saklanan eski BP;
- `[BP+2]` = FFFAh: dönüş adresi 0107h;
- `[BP+4]` = FFFCh: birinci parametre 1234h.

`MOV AX, [BP+4]` parametreyi AX'e alır. `POP BP` çağıranın BP'sini geri yükler. `RET 2` iki iş yapar: dönüş adresini IP'ye alır ve SP'ye ayrıca 2 ekleyerek yığındaki parametreyi atar. SP girişteki FFFEh değerine döner; yığın, çağrıdan önceki hâlindedir. Prosedürde fazladan PUSH yapılsaydı bile `[BP+4]` değişmezdi; BP'nin değeri budur.

Örnek 2.2.3

TOPLA iki parametre alacak biçimde çağrılsın: önce `PUSH a`, sonra `PUSH b`. Prosedür içinde a ve b hangi adreslerdedir, dönüşte RET kaç bayt atmalıdır?

Son itilen b, dönüş adresinin hemen üstündedir: `[BP+4]` = b, ondan önce itilen a `[BP+6]`'dadır. İki parametre 4 bayt tutar; dönüş `RET 4` ile yapılır. Parametre sayısı her arttığında ilk parametrenin uzaklığı 2 bayt artar, ama son parametre her zaman `[BP+4]`'tedir.

2.2.4.3 SI ve DI: indis yazmaçları

SI (*source index*, kaynak indisi) ve DI (*destination index*, hedef indisi), adlarını dizgi komutlarındaki görevlerinden alır. Dizgi komutları bellekteki ardışık bayt ya da sözcükler üzerinde çalışır: kopyalama (MOVS), karşılaştırma (CMPS), tarama (SCAS), yükleme (LODS) ve saklama (STOS). Bu komutların işlenenleri komutta yazılmaz; örtüktür:

- Kaynak her zaman **DS:SI**'dır: MOVSB, CMPSB ve LODSB okumayı buradan yapar.
- Hedef her zaman **ES:DI**'dır: MOVSB, STOSB ve SCASB yazmayı ya da karşılaştırmayı burada yapar.
- Her adımdan sonra SI ve DI, DF bayrağına göre 1 (bayt) ya da 2 (sözcük) artar veya azalır.

Dizgi komutları dışında SI ve DI sıradan indis yazmaçlarıdır. `MOV AL, [SI]` ya da `MOV [DI], AL` yazıldığında ikisi de varsayılan olarak **DS** kesimini kullanır. ES:DI kuralı yalnız dizgi komutlarının hedefi içindir. Bu ayrım, DS ile ES farklı kesimleri gösterdiğinde önem kazanır.

Tanım 2.2.6

İndis yazmacı (*index register*), bir dizi ya da dizgi içindeki konumu tutan yazmaçtır. 8086'nın indis yazmaçları SI ve DI'dır. Taban yazmacıyla birlikte kullanıldıklarında (`[BX+SI]`, `[BP+DI+5]`) etkin adres, iki yazmacın ve sabitin toplamıdır.

Sunumdaki program "ABC" dizgisini 0200h'den 0300h'ye kopyalar. Bir .COM programında DS = ES olduğu için iki alan aynı kesimdedir.

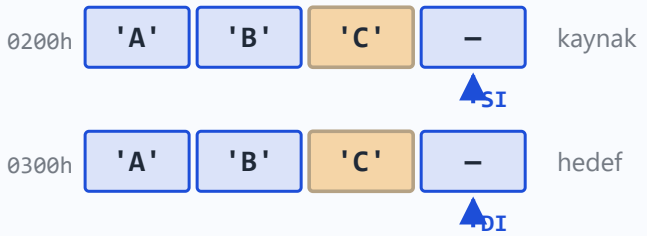
```
CLD             ; DF=0: artan adres
MOV SI, 0200h   ; kaynak
MOV DI, 0300h   ; hedef
MOV CX, 3       ; bayt sayısı
REP MOVSB       ; ES:[DI] ← DS:[SI]
```

Adım	Komut	SI	DI	CX	DF	0300h–0302h
1	`CLD`	0000h	0000h	0000h	0	— — —
2	`MOV SI, 0200h`	**0200h**	0000h	0000h	0	— — —

Adım	Komut	SI	DI	CX	DF	0300h–0302h
3	`MOV DI, 0300h`	0200h	**0300h**	0000h	0	— — —
4	`MOV CX, 3`	0200h	0300h	**0003h**	0	— — —
5	`REP MOVSB` (1. tur)	**0201h**	**0301h**	**0002h**	0	A — —
6	`REP MOVSB` (2. tur)	**0202h**	**0302h**	**0001h**	0	A B —
7	`REP MOVSB` (3. tur)	**0203h**	**0303h**	**0000h**	0	A B C

Tablo 2.2.12: REP MOVSB izlemesi. REP'in her turu ayrı bir satırdır.

CLD, DF'yi sıfırlar: SI ve DI her adımda artacaktır. SI kaynağın, DI hedefin başını gösterir; CX kaç bayt kopyalanacağını söyler. REP MOVSB her turda dört iş yapar: DS:SI'daki baytı ES:DI'ya kopyalar, SI'yı 1 artırır, DI'yı 1 artırır, CX'i 1 azaltır. CX sıfır olunca tekrar biter. Üç tur sonunda “ABC” hedefe kopyalanmıştır ve SI, DI kopyalanan alanın hemen arkasını gösterir (Şekil 2.2.8).



Şekil 2.2.8: Üçüncü turdan sonra kaynak ve hedef alanları. 'A', 'B', 'C' baytları 0300h'den başlayarak yazılmıştır; SI = 0203h, DI = 0303h, CX = 0.

REP öneki CX'i, MOVSB SI ve DI'yı örtük olarak kullanır. Böylece tek bir iki baytlık komut (`F3 A4`) bütün bir döngünün işini görür. Aynı kopyayı `MOV AL, [SI]` , `MOV [DI], AL` , `INC SI` , `INC DI` ve `LOOP` ile yazmak beş komut ve her turda daha fazla saat turu gerektirir. 8086 tasarımında yazmaçlara örtük görevler verilmesinin en açık kazancı burada görülür.

2.2.5 Kesim yazmaçları ve IP

8086'nın adres yolu 20 bittir ve 1 MB'lık belleği adresler. Yazmaçları ise 16 bittir ve tek başlarına yalnız 64 KB'ı gösterebilir. İki uç, belleğin **kesimlere** bölünmesiyle birleştirilir: her bellek erişimi bir kesim yazmacı ile 16 bitlik bir bağıl konumdan oluşur. Bu kesim, kesimlemeyi yazmaçlar açısından ele alır: kesim yazmaçlarına nasıl yazılır, hangi yazmaç hangi kesimle çiftleşir ve IP nasıl ilerler.

2.2.5.1 Kesim yazmaçlarına yazmak

Dört kesim yazmacı vardır: **CS** (kod kesimi), **DS** (veri kesimi), **SS** (yığın kesimi) ve **ES** (ek kesim). Her biri 16 bitlik bir paragraf numarası tutar. Kesimin başladığı fiziksel adres bu sayının 16 katıdır; kesim, oradan başlayarak en çok 64 KB uzanır.

Tanım 2.2.7

Fiziksel adres (20 bit) = kesim yazmacı × 10h + bağıl konum. 10h (16) ile çarpmak, onaltılık sayının sonuna bir 0 eklemektir. DS = 2000h ve BX = 0500h ise `[BX]` erişiminin fiziksel adresi 20000h + 0500h = 20500h'dir.

Kesim yazmaçlarına yazmanın kuralları, öbür yazmaçlardan daha sıkıdır:

- **Sabit yüklenemez.** `MOV DS, 2000h` yazılamaz; assembler bu satırı reddeder. Değer önce bir genel amaçlı yazmaca konur: `MOV AX, 2000h` ve `MOV DS, AX`.
- **CS'ye MOV ile yazılamaz.** CS'nin değişmesi programın başka bir yere atlaması demektir; bu iş uzak JMP, uzak CALL, RETF, INT ve IRET ile yapılır. `MOV CS, AX` satırı da reddedilir.
- **Kesimden kesime MOV yoktur.** `MOV ES, DS` yazılamaz. Ya bir yazmaç üzerinden (`MOV AX, DS`, `MOV ES, AX`) ya da yığın üzerinden (`PUSH DS`, `POP ES`) aktarılır.
- **Kesim yazmaçlarıyla aritmetik yapılamaz.** `ADD DS, 10h` ya da `INC ES` yoktur.

Bu kısıtların nedeni, kesim yazmaçlarının EU'da değil BIU'da bulunmasıdır. ALU'nun çıkışı doğrudan kesim yazmaçlarına bağlı değildir; kesim yazmaçlarına yalnız MOV ve POP aracılığıyla, bir yazmaç ya da bellek sözcüğünden veri gelir. Aşağıdaki kısa program bir veri kesimini kurar ve kullanır. 20500h fiziksel adresinde 5Ah baytı bulunsun.

```
MOV AX, 2000h
MOV DS, AX       ; DS = 2000h: kesim 20000h'de başlar
MOV BX, 0500h
MOV AL, [BX]     ; AL ← [20500h]
MOV [BX+1], AL   ; [20501h] ← AL
```

Komut	AX	DS	BX	Fiziksel adres	20500h–20501h
`MOV AX, 2000h`	**2000h**	0700h	0000h	—	5A 00
`MOV DS, AX`	2000h	**2000h**	0000h	—	5A 00
`MOV BX, 0500h`	2000h	2000h	**0500h**	—	5A 00
`MOV AL, [BX]`	**205Ah**	2000h	0500h	20500h (okuma)	5A 00
`MOV [BX+1], AL`	205Ah	2000h	0500h	20501h (yazma)	5A **5A**

Tablo 2.2.13: DS yazmacının kurulması ve kullanılması.

DS başlangıçta .COM yükleyicisinin verdiği 0700h'dir. İkinci satırdan sonra BX ile yapılan her erişim 20000h tabanına göre olur. AL'ye 5Ah okunduğunda AH'deki 20h yerinde kaldığı için AX = 205Ah olur; 8 bitlik yarıların bağımsızlığı burada da görülür.

2.2.5.2 Yazmaç çiftleri

Bir bağıl konum tek başına bir adres değildir; hangi kesime ait olduğu da bilinmelidir. 8086 her erişim türü için bir kesim yazmacını varsayılan olarak seçer. Programcı bunu yazmadığı için bu çiftler de örtük görevlerin bir parçasıdır (Tablo 2.2.14).

Çift	Kullanım	Örnek
`CS:IP`	sıradaki komut	her komut getirme
`SS:SP`	yığının üstü	PUSH, POP, CALL, RET, INT
`SS:BP`	yığın çerçevesi	`MOV AX, [BP+4]`
`DS:BX`, `DS:SI`, `DS:DI`	veri	`MOV AL, [BX+SI]`
`DS:SI → ES:DI`	dizgi komutları	`MOVSB`, `CMPSB`

Tablo 2.2.14: Yazmaç çiftleri ve kullanıldıkları erişimler.

Kural kısaca şöyle özetlenir: adres BP içeriyorsa kesim SS'dir, içermiyorsa DS'dir. `[BP+SI]` SS'yi, `[BX+SI]` DS'yi kullanır. Dizgi komutlarının hedefi her zaman ES'dir. Varsayılan kesim, komutun önüne bir **kesim öneki** eklenerek değiştirilebilir: `MOV AL, ES:[BX]` veriyi ES kesiminden okur. Komut getirme (CS:IP) ve yığın işlemleri (SS:SP) için önek kullanılamaz. Dizgi hedefi ES:DI de değiştirilemez; yalnız kaynak DS:SI'nın kesimi önekle başka bir kesime yöneltilebilir.

Örnek 2.2.4

DS = 1000h, SS = 3000h, SI = 0020h ve BP = 0020h iken `MOV AL, [SI]` ve `MOV AL, [BP]` hangi fiziksel adresleri okur?

`[SI]` BP içermez; kesim DS'dir: 1000h × 10h + 0020h = 10000h + 0020h = 10020h. `[BP]` ise SS kesimini kullanır: 30000h + 0020h = 30020h. İki komutun bağıl konumu aynıdır ama fiziksel adresleri 128 KB uzaktadır.

Sık yapılan hata

Bir diziye BP ile erişmeye çalışmak. `MOV BP, OFFSET dizi` ve `MOV AL, [BP]` yazılırsa erişim SS kesimine gider. .COM programlarında DS = SS olduğu için hata gizli kalır; DS ile SS'nin farklı olduğu bir .EXE programında yanlış bellek okunur. Veri dizileri için BX, SI ya da DI kullanılır.

2.2.5.3 IP: komut işaretçisi

Tanım 2.2.8

Komut işaretçisi (IP, *instruction pointer*), kod kesimi içinde yürütülecek sıradaki komutun bağıl konumunu tutar. Sıradaki komutun fiziksel adresi CS × 10h + IP'dir. Başka işlemcilerde bu yazmaç program sayacı (PC) adını taşır.

Bir komut yürütülürken IP, komutun uzunluğu kadar ilerletilir ve bir sonraki komutu gösterir. 8086 komutları 1 ile 6 bayt arasındadır; bu yüzden IP düzenli adımlarla artmaz. Tablo 2.2.4'teki AX = AH : AL programının adresleri buna iyi bir örnektir (Tablo 2.2.15).

Komutun adresi	Komut	Makine kodu	Uzunluk	Sonraki IP
0100h	`MOV AX, 1234h`	`B8 34 12`	3	0103h
0103h	`MOV AL, 0FFh`	`B0 FF`	2	0105h
0105h	`ADD AL, 1`	`04 01`	2	0107h
0107h	`MOV AX, 12FFh`	`B8 FF 12`	3	010Ah
010Ah	`ADD AX, 1`	`05 01 00`	3	010Dh

Tablo 2.2.15: IP'nin komut uzunluklarına göre ilerleyişi (.COM programı, ORG 100h).

16 bitlik bir sabit taşıyan komutlar 3 bayt, 8 bitlik sabit taşıyanlar 2 bayttır. Makine kodundaki sabitin düşük baytı önce gelir: 1234h, `34 12` olarak yazılmıştır.

IP'yi sıranın dışına çıkaran komutlar şunlardır: JMP ve koşullu atlamalar, LOOP, CALL, RET, INT ve IRET. IP'ye `MOV` ile yazılamaz ve IP'den `MOV` ile okunamaz; IP, programcının adıyla eriştiği bir işlenen değildir. IP'nin değerini öğrenmenin dolaylı bir yolu vardır: `CALL` komutu dönüş adresini yığına iter ve bir `POP` ile bu değer bir yazmaca alınabilir.

BIU'nun önden getirme düzeni nedeniyle, işlemcinin içinde bir an için kuyruktaki baytlar ile IP'nin gösterdiği adres farklı olabilir. Programcı açısından bu fark görünmez; IP her zaman yürütülecek sıradaki komutun adresini verir. Bir atlama olduğunda kuyruk boşaltılır ve getirme yeni CS:IP'den yeniden başlar.

2.2.6 Bayrak yazmacı ve durum bayrakları

2.2.6.1 Bayrak yazmacının yapısı

Bir toplama yapıldığında işlemci yalnız sonucu üretmez; sonuç hakkında birkaç bilgiyi de kaydeder. Sonuç sıfır mı? Negatif mi? 8 bite sığdı mı? Bu bilgiler tek bitlik göstergelerdir ve **bayrak** (*flag*) adını taşır. 8086 bayraklarını 16 bitlik **bayrak yazmacında** (FLAGS) toplar. Bu yazmaca durum yazmacı

(*status register*) da denir.



Şekil 2.2.9: 16 bitlik bayrak yazmacı. Bit 0 CF, 2 PF, 4 AF, 6 ZF, 7 SF, 8 TF, 9 IF, 10 DF, 11 OF; öbür bitler kullanılmaz.

Yazmacın 16 bitinden yalnız 9'u kullanılır (Şekil 2.2.9). Bunlar iki gruba ayrılır:

Tanım 2.2.9

Durum bayrakları (*status flags*) CF, PF, AF, ZF, SF ve OF'dir. Aritmetik ve mantık komutlarından sonra işlemci bunları sonuca göre kendisi kurar (1) ya da temizler (0). **Kontrol bayrakları** (*control flags*) DF, IF ve TF'dir. Sonuçtan etkilenmezler; programcı onları ayarlayarak işlemcinin çalışma biçimini değiştirir.

Durum bayraklarının asıl kullanıcısı koşullu atlama komutlarıdır. `JZ` ZF'ye, `JC` CF'ye, `JS` SF'ye, `JO` OF'ye bakar; `JG` ve `JL` gibi işaretli karşılaştırmalar SF ile OF'yi birlikte değerlendirir. Yüksek düzeyli bir dildeki her `if` ve her döngü koşulu, assembly'de bir karşılaştırma ve ardından bayraklara bakan bir atlama hâline gelir. Bu yüzden bayrakların ne zaman ve nasıl kurulduğunu bilmek, 8086 programlamanın temelidir.

Bayrakların bit sırası rastgele değildir. Alt bayttaki beş bayrak (SF, ZF, AF, PF, CF), 8080 işlemcisinin bayrak baytıyla aynı yerlerdedir. Bu düzen, 8080 programlarının çevrilmesini kolaylaştırmak için korunmuştur ve LAHF/SAHF komutlarının varlık nedenidir. OF ve kontrol bayrakları 8086 ile eklenmiş ve üst bayta konmuştur.

2.2.6.2 Bayraklar donanımda nasıl üretilir?

Durum bayrakları ALU'nun çıkışından birkaç basit kapıyla üretilir. Sayısal tasarım bilgisiyle bu devreyi kurmak zor değildir. Bir n bitlik toplayıcıda r_i sonucun i'nci biti, c_i de i'nci bite giren elde olsun; c_n en yüksek bitten çıkan eldedir. 8 bitlik işlemde n = 8'dir.

Bayrak	Mantık ifadesi	Devre
CF	c_8 (çıkarmada ödünç)	son tam toplayıcının elde çıkışı
AF	c_4	bit 3'ten bit 4'e geçen elde
ZF	$\neg(r_7 + r_6 + \ldots + r_0)$	8 girişli NOR kapısı
SF	r_7	en yüksek bitin kopyası (kapı yok)

Bayrak	Mantık ifadesi	Devre
PF	$\neg(r_7 \oplus r_6 \oplus \ldots \oplus r_0)$	XOR ağacı ve çıkışta değil kapısı
OF	$c_7 \oplus c_8$	en yüksek bite giren ve çıkan eldenin XOR'u

Tablo 2.2.16: Durum bayraklarının toplayıcı çıkışından üretilmesi (8 bitlik işlem).

16 bitlik işlemde CF için c_{16}, SF için r_{15}, OF için $c_{15} \oplus c_{16}$ kullanılır; ZF 16 bitin hepsine bakar. AF ve PF ise 16 bitlik işlemde de yalnız düşük bayttan üretilir. Tabloda dikkat çeken nokta, bayrakların hepsinin aynı anda ve sonuçla birlikte üretilmesidir. İşlemci "işaretli mi, işaretsiz mi?" diye sormaz; her toplamadan sonra altı bayrağın hepsini birden kurar. Hangisinin anlamlı olduğuna program karar verir.

Çıkarma, ALU'da ikiye tümleyenle toplama olarak yapılır: $A - B = A + \neg B + 1$. Bu toplamada en yüksek bitten elde çıkması, ödünç alınmadığı anlamına gelir. 8086, çıkarmada CF'yi bu eldenin tersi olarak kurar; böylece CF = 1 her zaman "ödünç alındı" demektir. AF için de aynı ters çevirme yapılır.

2.2.6.3 SF: işaret bayrağı

İşaret bayrağı (SF, *sign flag*), sonucun en anlamlı bitinin (MSB) kopyasıdır. 8 bitlik işlemde bit 7, 16 bitlik işlemde bit 15'tir. İkiye tümleyen gösteriminde MSB işaret bitidir: 1 ise sayı negatif, 0 ise pozitif ya da sıfırdır. SF bu yüzden sonucun, işaretli olarak yorumlandığında negatif olup olmadığını söyler.

Sonuç (8 bit)	İkili	İşaretli değer	MSB	SF
`35h`	0011 0101	+53	0	0
`7Fh`	0111 1111	+127	0	0
`80h`	1000 0000	−128	1	1
`C2h`	1100 0010	−62	1	1
`FFh`	1111 1111	−1	1	1

Tablo 2.2.17: 8 bitlik sonuçlarda SF.

Kural kısadır: 00h–7Fh arasındaki sonuçlarda SF = 0, 80h–FFh arasındakilerde SF = 1'dir. 16 bitte sınır 7FFFh ile 8000h arasındadır. İşlemci sayının işaretli mi işaretsiz mi olduğunu bilmez. C2h baytı işaretsiz olarak 194, işaretli olarak −62'dir; iki yorumda da SF = 1 olur. SF'yi "negatif" diye okumak programcının seçimidir ve yalnız işaretli sayılarla çalışırken anlamlıdır.

> **Sık yapılan hata**
>
> SF'nin gerçek matematiksel sonucun işaretini verdiğini sanmak. 70h + 10h = 80h işleminde iki pozitif sayı toplanmıştır ve gerçek sonuç +128'dir. Yine de SF = 1 olur, çünkü 8 bitlik sonucun MSB'si 1'dir. Sonuç işaretli aralığa sığmamıştır; bunu OF gösterir. SF yalnız OF = 0 iken doğru işareti verir.

2.2.6.4 ZF ve PF: sıfır ve eşlik bayrakları

Sıfır bayrağı (ZF, *zero flag*), sonucun bütün bitleri sıfırsa 1 olur; aksi hâlde 0'dır. Bayrağın değeri ile sonucun değeri ters görünür: ZF = 1 "sonuç sıfır" demektir. Bu karışıklık bayrağı bir soru gibi okuyarak giderilir: "sonuç sıfır mı?" Evet ise 1.

ZF en çok eşitlik sınamasında kullanılır. `CMP AX, BX`, AX − BX'i hesaplar ama sonucu hiçbir yere yazmaz; yalnız bayrakları kurar. AX = BX ise fark sıfırdır ve ZF = 1 olur. Ardından gelen `JE` (eşitse atla, `JZ` ile aynı komut) ZF'ye bakar. AX = BX = 1234h iken CMP sonrasında ZF = 1, PF = 1 ve diğer durum bayrakları 0'dır.

Bir yazmacı sıfırlamanın üç yolu ZF açısından ilginç bir karşılaştırma verir. `SUB AL, AL` ve `XOR AL, AL` sonucu sıfır yaptıkları için ZF = 1, CF = 0 yapar. `MOV AL, 0` ise bayraklara hiç dokunmaz; önceki işlemden kalan bayraklar durur. Bir koşullu atlamadan hemen önce yazmaç sıfırlanacaksa bu fark önem taşır.

Eşlik bayrağı (PF, *parity flag*), sonucun **düşük 8 bitindeki** 1'lerin sayısı çift ise 1, tek ise 0 olur. 16 bitlik işlemlerde de yalnız düşük bayta bakılır. Şekil 2.2.10 iki örnek verir.



Şekil 2.2.10: Eşlik bayrağı. 03h'nin düşük baytında iki tane 1 vardır, PF = 1; 07h'de üç tane 1 vardır, PF = 0.

Sıfır sayısında hiç 1 yoktur; sıfır da çift sayı olduğundan 00h için PF = 1'dir. PF'nin tarihsel kullanımı seri iletişimde hata denetimiydi. Gönderilen her karaktere bir eşlik biti eklenir ve alıcı aldığı baytın eşliğini sınar; tek bitlik bir bozulma eşliği değiştirir ve fark edilir. Günümüz programlarında PF nadiren kullanılır, ama 8086'nın her aritmetik ve mantık işlemi onu kurmaya devam eder. `JP` ve `JNP` komutları bu bayrağa bakar.

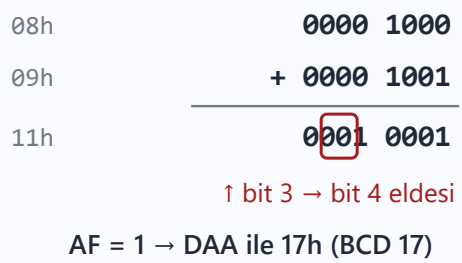
Örnek 2.2.5

AX = 7FF8h iken `ADD AX, 0009h` sonrasında ZF ve PF nedir?

7FF8h + 0009h = 8001h. Sonuç sıfır değildir: ZF = 0. PF için yalnız düşük bayt 01h = 0000 0001b'ye bakılır. İçinde bir tane 1 vardır, tek sayıdır: PF = 0. Üst bayt 80h'deki 1 hesaba katılmaz.

2.2.6.5 AF: yardımcı elde bayrağı

Yardımcı elde bayrağı (AF, *auxiliary carry flag*), işlem sırasında bit 3'ten bit 4'e elde (toplamada) ya da ödünç (çıkarmada) olursa 1, olmazsa 0 olur. Başka bir deyişle, düşük nibble'dan (4 bitlik gruptan) üst nibble'a bir taşma olup olmadığını gösterir.



Şekil 2.2.11: 08h + 09h = 11h. Düşük nibble'lar toplanınca 8 + 9 = 17 olur ve bit 3'ten bit 4'e elde çıkar: AF = 1. DAA bu bilgiyle sonucu BCD 17'ye düzeltir.

Bu bayrak **ikili kodlu onluk** (BCD) aritmetiğe özgüdür. BCD'de her onluk basamak ayrı bir nibble'da tutulur: 17 sayısı 0001 0111 = 17h olarak yazılır. BCD sayılar ikili toplayıcıyla toplandığında sonuç her zaman geçerli bir BCD olmaz. Şekil 2.2.11'deki örnekte 08h + 09h = 11h çıkar; doğru BCD cevap ise 17'dir. Düşük nibble 16'yı geçtiği için 10'luk sistemin 6'sı "kaybolmuştur". AF = 1, bu kaybın olduğunu söyler. `DAA` (toplamadan sonra onluk düzeltme) komutu AF'ye bakar ve düşük nibble'a 6 ekler: 11h + 06h = 17h. DAA'dan sonra AL = 17h olur.

AF'nin bir özelliği, programcının ona doğrudan ulaşamamasıdır. AF'yi sınayan bir koşullu atlama komutu yoktur. AF'yi kuran ya da temizleyen özel bir komut da yoktur. Yalnız DAA, DAS, AAA ve AAS onu kullanır. AF yine de okunabilir ve yazılabilir: `LAHF` ve `PUSHF` onu öbür bayraklarla birlikte bir yazmaca ya da yığına kopyalar; `SAHF` ve `POPF` geri yazar.

Örnek 2.2.6

`05h − 07h` ve `9Ch + 64h` işlemlerinde AF kaçtır?

05h − 07h'de düşük nibble'lar 5 − 7'dir; 5, 7'den küçük olduğu için bit 4'ten ödünç alınır: AF = 1. 9Ch + 64h'de düşük nibble'lar C + 4 = 12 + 4 = 16'dır; 15'i aştığı için bit 4'e elde geçer: AF = 1.

2.2.6.6 CF: elde bayrağı

Elde bayrağı (CF, *carry flag*), n bitlik bir toplamada sonuç n bite sığmazsa, yani en yüksek bitten dışarı bir elde çıkarsa 1 olur. Çıkarmada küçük bir sayıdan büyük bir sayı çıkarılır ve **ödünç** alınırsa da CF = 1 olur. CF, **işaretsiz** sayılar için taşma göstergesidir.

İşlem (8 bit)	İşaretsiz yorum	Sonuç	CF
`FFh + 01h`	255 + 1 = 256 > 255	00h	1
`80h + 80h`	128 + 128 = 256 > 255	00h	1
`70h + 10h`	112 + 16 = 128 ≤ 255	80h	0
`05h − 07h`	5 − 7 < 0 (ödünç)	FEh	1

İşlem (8 bit)	İşaretsiz yorum	Sonuç	CF
`07h - 05h`	7 − 5 = 2	02h	0

Tablo 2.2.18: 8 bitlik işlemlerde CF. İşaretsiz yorum sonucun neden sığdığını ya da sığmadığını açıklar.

05h − 07h örneğinde sonuç FEh'dir. İşaretsiz yorumda bu 254'tür ve gerçek sonuç −2 değildir; CF = 1 bunu bildirir. Aynı FEh işaretli yorumda −2'dir ve doğrudur. Bu yüzden işaretli bir çıkarmada CF = 1 olması bir hata değildir; hata olup olmadığını OF söyler.

CF'nin öbür bayraklardan farklı birkaç özelliği vardır:

- **Programcı değiştirebilir.** `STC` CF'yi 1, `CLC` 0 yapar, `CMC` tersine çevirir. Durum bayrakları içinde doğrudan kurulup silinebilen tek bayrak CF'dir.
- **INC ve DEC CF'ye dokunmaz.** AL = FFh iken `ADD AL, 1` CF = 1 yapar; `INC AL` ise aynı sonucu üretir ama CF'yi önceki değerinde bırakır. Bu tasarım, bir döngü sayacının INC/DEC ile değiştirilmesinin çok baytlı bir toplamanın eldesini bozmaması içindir.
- **Kaydırma ve döndürme** komutlarında dışarı atılan son bit CF'ye gider. `SHL AL, 1`, AL = 81h iken AL = 02h ve CF = 1 verir.
- **Çok sözcüklü aritmetik.** `ADC` (elde ile topla) CF'yi toplamaya ekler, `SBB` (ödünç ile çıkar) CF'yi çıkarır. Böylece 16 bitlik bir işlemci 32 ya da 64 bitlik sayıları parça parça toplayabilir.

32 bitlik toplama ADC'nin en bilinen kullanımıdır. DX:AX = 0001FFFFh'ye 1 eklenecek olsun:

```
MOV AX, 0FFFFh   ; alt sözcük
MOV DX, 0001h    ; üst sözcük
ADD AX, 1        ; AX = 0000h, CF = 1
ADC DX, 0        ; DX = 0001h + 0 + CF = 0002h
```

Alt sözcükte FFFFh + 1 = 10000h çıkar; AX = 0000h olur ve taşan 1 CF'de bekler. ADC üst sözcüğe hem 0'ı hem CF'yi ekler: DX = 0002h. Sonuç DX:AX = 00020000h'dir. Onluk bir toplamada elde nasıl sol basamağa taşınıyorsa, burada da CF sözcükten sözcüğe taşınmıştır.

2.2.6.7 OF: taşma bayrağı

Taşma bayrağı (OF, *overflow flag*), sonuç **işaretli** olarak yorumlandığında kullanılan bit sayısına sığmıyorsa 1 olur. 8 bitte işaretli aralık −128 … +127, 16 bitte −32.768 … +32.767'dir. CF işaretsiz taşmayı, OF işaretli taşmayı gösterir.

Toplama için OF'yi bulmanın iki eşdeğer yolu vardır:

- **İşaret kuralı.** İki pozitif sayı toplanıp negatif bir sonuç, ya da iki negatif sayı toplanıp pozitif bir sonuç çıkarsa OF = 1'dir. Farklı işaretli iki sayının toplamı hiçbir zaman taşmaz, çünkü sonuç iki sayının arasında kalır.

- **Elde kuralı.** MSB'ye giren elde ile MSB'den çıkan elde farklıysa OF = 1'dir (Tablo 2.2.16'daki $c_7 \oplus c_8$).

Çıkarmada işaret kuralı şöyle uyarlanır: pozitiften negatif çıkarılıp negatif sonuç, ya da negatiften pozitif çıkarılıp pozitif sonuç bulunursa OF = 1'dir. Örneğin 80h − 01h = 7Fh işleminde −128 − 1 = −129 hesaplanmak istenmiş, sonuç +127 çıkmıştır: OF = 1. Aynı işaretli iki sayının farkı ise hiçbir zaman taşmaz.

Tablo 2.2.19 CF ile OF'nin dört olası birleşimini birer örnekle verir. Her satır hem işaretsiz hem işaretli olarak okunmalıdır.

İşlem	İşaretsiz yorum	İşaretli yorum	Sonuç	CF	OF
`05h + 03h`	5 + 3 = 8	5 + 3 = 8	08h	0	0
`7Fh + 01h`	127 + 1 = 128	127 + 1 ≠ −128	80h	0	**1**
`FFh + 01h`	255 + 1 ≠ 0	−1 + 1 = 0	00h	**1**	0
`80h + 80h`	128 + 128 ≠ 0	−128 + (−128) ≠ 0	00h	**1**	**1**

Tablo 2.2.19: CF ve OF'nin dört birleşimi (8 bitlik toplama).

Satırlar elde kuralıyla da sağlanabilir. 7Fh + 01h'de alt yedi bitin hepsi 1'dir; toplamada bit 7'ye elde girer (c_7 = 1) ama bit 7'den elde çıkmaz (c_8 = 0). İkisi farklıdır: OF = 1. FFh + 01h'de hem c_7 hem c_8 1'dir: OF = 0, CF = 1. 80h + 80h'de bit 7'ye elde girmez (c_7 = 0) ama iki 1'in toplamı bit 7'den elde çıkarır (c_8 = 1): OF = 1, CF = 1.

CF ile OF birbirinden bağımsızdır; tabloda dört birleşimin dördü de görülür. İşlemci ikisini de her zaman hesaplar. Program işaretsiz sayılarla çalışıyorsa CF'ye (ve `JA`, `JB` gibi atlamalara), işaretli sayılarla çalışıyorsa OF'ye (ve `JG`, `JL`, `JO` gibi atlamalara) bakar.

Örnek 2.2.7

`MOV AL, 0C8h` ve `ADD AL, 64h` sonrasında AL ve altı durum bayrağı nedir?

C8h + 64h = 12Ch. AL'ye alt bayt 2Ch yazılır, taşan 1 CF'ye gider: CF = 1. İşaretli yorumda C8h = −56, 64h = +100'dür; −56 + 100 = 44 = 2Ch sonucu doğrudur, farklı işaretli sayıların toplamı taşmaz: OF = 0. 2Ch = 0010 1100b; MSB 0 olduğu için SF = 0, sonuç sıfır olmadığı için ZF = 0'dır. Düşük baytta üç tane 1 vardır: PF = 0. Düşük nibble'lar 8 + 4 = 12'dir, 15'i aşmaz: AF = 0.

Sık yapılan hata

"Taşma" kelimesini her iki bayrak için kullanmak ve CF = 1'i işaretli bir hesapta hata saymak. −1 + 1 işleminde (FFh + 01h) CF = 1 olur ama işaretli sonuç 0 tamamen doğrudur. İşaretli bir hesabın doğru olup olmadığını yalnız OF söyler.

INC ve DEC de OF'yi kurar. AL = 7Fh iken `INC AL`, AL = 80h yapar; +127 + 1 işaretli aralığı aşmıştır ve OF = 1, SF = 1 olur. NEG komutunda tek taşma durumu en küçük negatif sayıdır: `NEG AL`, AL = 80h iken yine 80h üretir, çünkü +128 8 bitte gösterilemez; bu durumda OF = 1'dir.

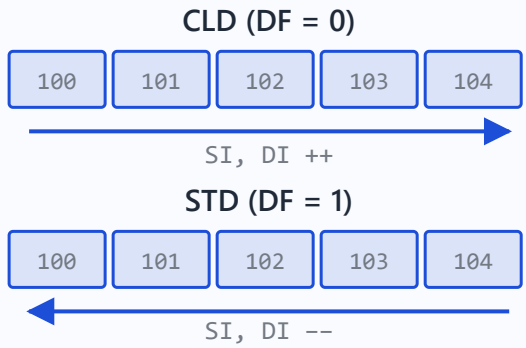
2.2.7 Kontrol bayrakları

DF, IF ve TF bir işlemin sonucunu anlatmaz; işlemcinin bundan sonra nasıl çalışacağını belirler. Aritmetik komutlar bu bayraklara dokunmaz. Programcı onları özel komutlarla ya da bayrak yazmacını yığın üzerinden değiştirerek ayarlar.

2.2.7.1 DF: yön bayrağı

Yön bayrağı (DF, *direction flag*), dizgi komutlarının belleği hangi yönde gezeceğini belirler (Şekil 2.2.12):

- **DF = 0**: düşük adresten yükseğe doğru; her adımda SI ve DI **artar**. `CLD` komutu DF'yi 0 yapar.
- **DF = 1**: yüksek adresten düşüğe doğru; her adımda SI ve DI **azalır**. `STD` komutu DF'yi 1 yapar.



Şekil 2.2.12: DF = 0 iken dizgi komutları adresleri artırarak, DF = 1 iken azaltarak ilerler.

Adım büyüklüğü komutun genişliğine bağlıdır: bayt komutlarında (MOVSB, LODSB ...) 1, sözcük komutlarında (MOVSW, LODSW ...) 2'dir. DF yalnız dizgi komutlarını etkiler. `MOV AL, [SI]` gibi sıradan bir komut SI'yı hiç değiştirmez.

Bir diziyi baştan sona işlemek için DF = 0 yeterlidir ve çoğu program bunu kullanır. Bu nedenle DF'nin değerine güvenmek yerine her dizgi işleminden önce `CLD` ya da `STD` yazmak iyi bir alışkanlıktır. Başka bir yordam DF'yi 1 bırakmışsa, CLD yazmayan bir kopyalama döngüsü belleği ters yönde gezer.

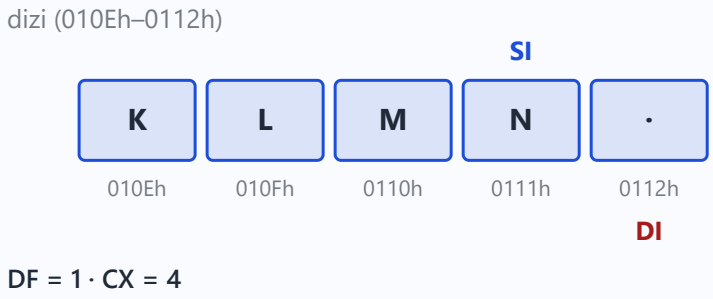
Geriye doğru gezinin gerçek gerekçesi **örtüşen alanlardır**. Kaynak ve hedef bellekte üst üste biniyorsa kopyalama yönü sonucu belirler. Hedef kaynağın ilerisindeyse (daha yüksek adresteyse) kopyalama sondan başa yapılmalıdır. Aksi hâlde kaynağın henüz okunmamış baytları, hedefe yazılan baytlarla ezilir.

2.2.7.2 Kod izleme: DF = 1 ile örtüşen kopyalama

Sunumdaki program bellekteki “KLMN” dizgisini bir bayt sağa kaydırır. Kaynak dizi..dizi+3, hedef dizi+1..dizi+4'tür; iki alan üç baytta örtüşür.

```
org 100h
mov si, offset dizi+3 ; kaynak
mov di, offset dizi+4 ; hedef
mov cx, 4
std                   ; DF = 1
rep movsb
cld
ret
dizi db 'KLMN', 0
```

Program 0100h'den başlar. Komutlar 3 + 3 + 3 + 1 + 2 + 1 + 1 = 14 bayt tutar; bu yüzden `dizi` 010Eh adresine yerleşir. 'K' 010Eh'de, 'L' 010Fh'de, 'M' 0110h'da, 'N' 0111h'de, sonlandırıcı 0 da 0112h'dadır. SI kaynağın son baytını (dizi+3 = 0111h), DI hedefin son baytını (dizi+4 = 0112h) gösterecek biçimde kurulur.

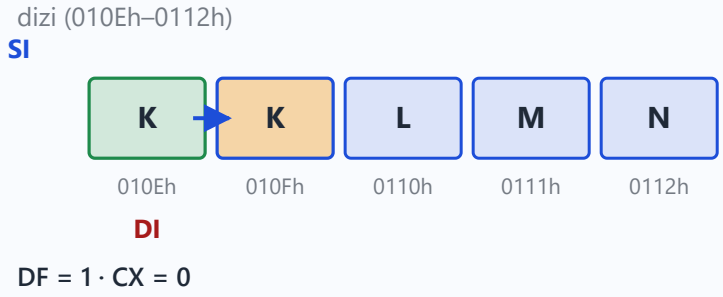


Şekil 2.2.13: `std` sonrasında, kopyalamadan önce: SI = 0111h ('N'), DI = 0112h, CX = 4, DF = 1.

Satır	Komut	SI	DI	CX	DF	010Eh … 0112h
2	`mov si, offset dizi+3`	**0111h**	0000h	0	0	K L M N ·
3	`mov di, offset dizi+4`	0111h	**0112h**	0	0	K L M N ·
4	`mov cx, 4`	0111h	0112h	**4**	0	K L M N ·
5	`std`	0111h	0112h	4	**1**	K L M N ·
6	`rep movsb` (1. tur)	**0110h**	**0111h**	**3**	1	K L M N **N**
6	`rep movsb` (2. tur)	**010Fh**	**0110h**	**2**	1	K L M **M** N
6	`rep movsb` (3. tur)	**010Eh**	**010Fh**	**1**	1	K L **L** M N
6	`rep movsb` (4. tur)	**010Dh**	**010Eh**	**0**	1	K **K** L M N
7	`cld`	010Dh	010Eh	0	**0**	K K L M N

Tablo 2.2.20: Geriye doğru örtüşen kopyalamanın izlemesi. Bellek sütunu 010Eh–0112h baytlarıdır (· = 0).

Birinci turda 0111h'deki 'N', 0112h'ye kopyalanır. Sonlandırıcı sıfırın üzerine yazılır ama bu bayt kaynağın parçası değildir. İkinci turda 0110h'deki 'M', 0111h'ye yazılır. 0111h'deki 'N' bir önceki turda zaten okunmuş ve taşınmıştı; üzerine yazılmasının zararı yoktur. Her tur, bir sonraki turda okunacak baytı değil, önceki turda okunmuş olanı ezer. Dördüncü turdan sonra bellekte "KKLMN" vardır: dizgi bir sağa kaymış, baştaki 'K' yerinde kalmıştır (Şekil 2.2.14). Son satırdaki `cld`, DF'yi programın geri kalanının beklediği değere döndürür.



Şekil 2.2.14: Dört turdan sonra bellek "KKLMN"dir. SI ve DI kopyalanan alanın bir bayt öncesini gösterir; CX = 0.

Aynı kaydırma ileri yönde, CLD ile denensin: SI = dizi, DI = dizi+1, CX = 4. Birinci tur 'K'yi 010Fh'ye yazar ve 'L'yi ezer. İkinci tur 010Fh'den okur, ama orada artık 'K' vardır; 'K' 0110h'ya yazılır. Her tur bir önceki turun yazdığı 'K'yi okur. Sonuç "KKKKK"dir. Kural şöyle özetlenir: hedef kaynağın ilerisindeyse geriye doğru (STD), gerisindeyse ileriye doğru (CLD) kopyalanır.

Sık yapılan hata

STD ile geriye kopyalarken SI ve DI'yı alanların **başına** kurmak. DF = 1 iken dizgi komutu adresleri azaltır; SI ve DI alanların **son** baytını göstermelidir. Sözcük kopyalamada (MOVSW) son sözcüğün adresi, yani son bayttan bir önceki adres verilir.

2.2.7.3 IF: kesme bayrağı

8086'nın INTR adında bir giriş ucu vardır. Klavye, zamanlayıcı ya da seri port gibi bir çevre birimi işlemciden hizmet istediğinde bu ucu etkinleştirir. İşlemci o anki komutu bitirir, programı keser ve bir kesme hizmet yordamı (ISR) çalıştırır. **Kesme bayrağı** (IF, *interrupt flag*) bu isteklerin kabul edilip edilmeyeceğini belirler:

- **IF = 1**: INTR'den gelen maskelenebilir kesme istekleri kabul edilir. `STI` IF'yi 1 yapar.
- **IF = 0**: bu istekler göz ardı edilir, yani **maskelenir**. `CLI` IF'yi 0 yapar.

IF her şeyi engellemez. NMI ucundan gelen **maskelenemez kesme** IF'den etkilenmez; güç kesintisi ya da bellek eşlik hatası gibi ertelenemeyecek olaylar için ayrılmıştır. `INT n` komutuyla programın kendisinin başlattığı yazılım kesmeleri de IF'ye bakmaz. IF yalnız dışarıdan, INTR ucundan gelen is-

tekleri süzer.

Bir programın kesmeleri neden kapatmak isteyebileceğini yığın örneği iyi gösterir. Yığını taşımak için SS ve SP'nin ikisi de değiştirilir. SS değiştikten sonra, SP değişmeden önce bir kesme gelirse işlemci FLAGS, CS ve IP'yi yarı yarıya kurulmuş, yanlış bir yığına iter. Bu iki komut kesilmemesi gereken bir bütündür:

```
CLI             ; kesmeleri kapat
MOV AX, 9000h
MOV SS, AX       ; yeni yığın kesimi
MOV SP, 0FFFEh   ; yeni yığın tepesi
STI             ; kesmeleri yeniden aç
```

8086 bu tehlikeye karşı kendisi de bir önlem alır: bir kesim yazmacına yazan komuttan sonra, bir sonraki komut bitene kadar kesme kabul etmez. `MOV SS, AX`'in hemen arkasına `MOV SP, …` yazılırsa iki komut kesintisiz çalışır. Yine de CLI/STI çifti niyeti açıkça gösterir. Kesmeleri uzun süre kapalı tutmak ise sakıncalıdır: klavye tuşları kaybolabilir, saat geri kalabilir. CLI ile STI arasındaki kod olabildiğince kısa tutulur.

İşlemci bir kesmeyi kabul ettiğinde IF'yi kendisi 0 yapar; böylece ISR, aynı türden yeni bir kesmeyle hemen kesilmez. Yordam sonundaki `IRET`, yığından eski FLAGS'i geri yükler ve IF eski değerine, yani 1'e döner.

2.2.7.4 TF: tuzak bayrağı

Tuzak bayrağı (TF, *trap flag*), yonga içi hata ayıklama için konmuştur. TF = 1 iken işlemci **tek adım** (*single step*) kipinde çalışır: her komut bittikten sonra kendiliğinden bir tip 1 kesmesi (INT 1) üretir (Şekil 2.2.15).



Şekil 2.2.15: TF = 1 iken her komuttan sonra INT 1 oluşur: FLAGS, CS ve IP yığına atılır, TF temizlenir, tek adım yordamı çalışır, IRET ile sonraki komuta dönülür.

Akış şöyledir. Program bir komutu yürütür. TF = 1 olduğu için işlemci FLAGS, CS ve IP'yi yığına iter, TF ile IF'yi temizler ve tip 1 kesme vektörünün gösterdiği yordama atlar. Bu yordam bir hata ayıklayıcının parçasıdır: yazmaçları ekrana yazar, kullanıcının bir tuşa basmasını bekler. Yordam çalışırken TF = 0 olduğundan kendisi adım adım yürütülmez. Sonundaki IRET yığından FLAGS'i alır; TF yeniden 1 olur ve programın bir sonraki komutu yürütülür. Ardından döngü yinelenir. DOS'un DEBUG programındaki `T` (trace) komutu ve emu8086'nın “single step” düğmesi bu düzeni kullanır.

TF'yi doğrudan kuran ya da silen bir komut yoktur. CF için STC/CLC, DF için STD/CLD, IF için STI/CLI vardır; TF için yoktur. TF, bayrak yazmacının tamamı yığın üzerinden değiştirilerek kurulur. TF bayrak yazmacının bit 8'idir ve maskesi 0100h'dir; yöntemin adım adım izlemesi Tablo 2.2.26'dadır.

Bayrak	Bit	1 iken	0 iken	Kuran / silen komut
DF	10	dizgi adresleri azalır	dizgi adresleri artar	`STD` / `CLD`
IF	9	INTR kesmeleri kabul edilir	INTR kesmeleri maskelenir	`STI` / `CLI`
TF	8	her komuttan sonra INT 1	normal çalışma	yok; PUSHF/POPF ile

Tablo 2.2.21: Kontrol bayraklarının karşılaştırması.

2.2.8 Bayrakları değiştiren komutlar

2.2.8.1 Komutların bayraklara etkisi

Bayraklar iki yolla değişir: bazı komutlar onları doğrudan kurar ya da siler, aritmetik ve mantık komutları ise yan etki olarak günceller. Tablo 2.2.22 doğrudan komutları verir.

Komut	Etki	Komut	Etki
`CLC`	CF = 0	`CLD`	DF = 0
`STC`	CF = 1	`STD`	DF = 1
`CMC`	CF = ¬CF	`CLI` / `STI`	IF = 0 / 1
`LAHF`	AH ← SF ZF – AF – PF – CF	`SAHF`	AH → düşük bayrak baytı
`PUSHF`	FLAGS'i yığına koy	`POPF`	yığından FLAGS'e yükle

Tablo 2.2.22: Bayrakları doğrudan değiştiren komutlar.

Komut gruplarının durum bayraklarına etkisi ise Tablo 2.2.23'te toplanmıştır. Tablo, bir koşullu atlamadan önce hangi komutun bayrakları en son değiştirdiğini bulmaya yarar.

Komut grubu	Bayraklara etkisi
ADD, ADC, SUB, SBB, CMP, NEG	CF, PF, AF, ZF, SF, OF hepsi güncellenir
INC, DEC	CF hariç hepsi güncellenir
AND, OR, XOR, TEST	CF = OF = 0; PF, ZF, SF güncellenir; AF tanımsız
MUL, IMUL	CF = OF = sonucun yüksek yarısı anlamlı mı; SF, ZF, AF, PF tanımsız
DIV, IDIV	altı durum bayrağı da tanımsız
SHL, SHR, SAR	CF = dışarı atılan son bit; PF, ZF, SF güncellenir
ROL, ROR, RCL, RCR	yalnız CF ve OF değişir
MOV, PUSH, POP, XCHG, LEA, LOOP, JMP, Jcc	bayrakları **etkilemez**

Tablo 2.2.23: Komut gruplarının durum bayraklarına etkisi.

"Tanımsız" sözcüğü, Intel'in bayrağın o komuttan sonraki değerini belgelemediği anlamına gelir. Gerçek işlemci bir değer bırakır ama bu değer işlemci modeline göre değişebilir. Doğru bir program tanımsız bayrağa bakmaz. Bir başka ince nokta TEST ve CMP'dir: ikisi de AND ve SUB gibi hesap yapar ama sonucu yazmaz, yalnız bayrakları kurar. Örneğin `TEST AL, 80h` AL'nin bit 7'sini sınar. AL = B5h iken sonuç 80h olur, ZF = 0 ve SF = 1 çıkar; AL değişmez.

2.2.8.2 Kod izleme: LAHF, SAHF ve CMC

LAHF (*load AH from flags*), bayrak yazmacının düşük baytını AH'ye kopyalar. **SAHF** (*store AH into flags*) ters yönde çalışır: AH'yi bayrak yazmacının düşük baytına yazar. Düşük baytın düzeni şöyledir:

Bit	7	6	5	4	3	2	1	0
Bayrak	SF	ZF	–	AF	–	PF	–	CF
Değer	SF	ZF	0	AF	0	PF	1	CF

Tablo 2.2.24: Bayrak yazmacının düşük baytı (LAHF/SAHF düzeni).

Bit 1 8086'da her zaman 1 okunur; bit 3 ve bit 5 ise 0'dır. OF, DF, IF ve TF üst baytta olduğu için LAHF ve SAHF onlara erişmez. Sunumdaki program bu iki komutu ve CF'yi değiştiren üç komutu birlikte izler:

```
org 100h
mov al, 0FFh
add al, 1       ; CF ZF AF PF = 1
lahf            ; AH ← bayraklar
cmc             ; CF tersle
clc             ; CF = 0
stc             ; CF = 1
mov ah, 0C1h    ; 1100 0001b
sahf            ; bayraklar ← AH
ret
```

Satır	Komut	AH	SF	ZF	AF	PF	CF
2	`mov al, 0FFh`	00h	0	0	0	0	0
3	`add al, 1`	00h	0	**1**	**1**	**1**	**1**
4	`lahf`	**57h**	0	1	1	1	1
5	`cmc`	57h	0	1	1	1	**0**
6	`clc`	57h	0	1	1	1	0
7	`stc`	57h	0	1	1	1	**1**
8	`mov ah, 0C1h`	**C1h**	0	1	1	1	1
9	`sahf`	C1h	**1**	1	**0**	**0**	1

Tablo 2.2.25: LAHF, SAHF ve CMC izlemesi.

Üçüncü satırda FFh + 1 = 00h hesaplanır. Dört bayrak birden kurulur: sonuç 8 bite sığmadı (CF = 1), sonuç sıfır (ZF = 1), düşük nibble F + 1 taştı (AF = 1), 00h'de sıfır tane 1 var ve sıfır çifttir (PF = 1). SF = 0'dır, çünkü 00h'nin MSB'si 0'dır.

Dördüncü satırda LAHF bu durumu AH'ye kopyalar. Bitler soldan sağa yazılırsa SF = 0, ZF = 1, 0, AF = 1, 0, PF = 1, 1, CF = 1 elde edilir: 0101 0111b = 57h (Şekil 2.2.16). Bit 1'in sabit 1 olduğu unutulursa 55h bulunur; bu sık bir hesap hatasıdır.



Şekil 2.2.16: `lahf` sonrasında bayrak baytı ve AH. AH = 57h = 0101 0111b; bit 1 her zaman 1'dir.

Beşinci satırda CMC, CF'yi 1'den 0'a çevirir. Altıncı satırdaki CLC CF'yi 0 yapar; CF zaten 0 olduğu için görünür bir değişiklik olmaz. Yedinci satırda STC, CF'yi 1 yapar. Bu üç komut yalnız CF'ye dokunur; ZF, AF ve PF'nin değişmediği tabloda açıkça görülür. Bu komutlar, örneğin bir yordamın başarılı ya da başarısız döndüğünü CF ile bildirmesinde ve ADC/SBB zincirinin başında eldeyi hazırlamakta kullanılır.

Sekizinci satır AH'ye C1h = 1100 0001b yazar. Bu bit dizisi bayrak baytının düzeninde okunursa bit 7 (SF), bit 6 (ZF) ve bit 0 (CF) 1'dir; AF (bit 4) ve PF (bit 2) 0'dır. Dokuzuncu satırda SAHF bu baytı bayraklara yazar: SF = 1, ZF = 1, CF = 1, AF = 0, PF = 0 olur (Şekil 2.2.17). SF ile ZF'nin birlikte 1 olması normal bir işlemin üretemeyeceği bir durumdur: sonuç hem sıfır hem negatif olamaz. SAHF bayrakları sonuçtan bağımsız olarak istenen değere getirir. OF, DF, IF ve TF bu komuttan etkilenmez.



Şekil 2.2.17: `sahf` sonrasında: AH = C1h bayrak baytına yazılmıştır; SF = ZF = CF = 1, AF = PF = 0.

Örnek 2.2.8

`MOV AL, 5` , `CMP AL, 5` ve `LAHF` sonrasında AH kaçtır?

CMP 5 − 5 = 0 hesaplar: ZF = 1, sonuç 00h olduğundan PF = 1, ödünç olmadığından CF = 0, nibble ödüncü olmadığından AF = 0, MSB 0 olduğundan SF = 0. Bayrak baytı SF ZF 0 AF 0 PF 1 CF = 0 1 0 0 0 1 1 0 = 0100 0110b = 46h'dir.

2.2.8.3 PUSHF ve POPF ile bütün bayraklar

LAHF ve SAHF yalnız düşük bayta erişir. Bayrak yazmacının 16 bitinin tamamı **PUSHF** ile yığına itilir ve **POPF** ile yığından geri alınır. İki komut arasında yığındaki sözcük bir yazmaca alınıp istenen bit değiştirilebilir. Doğrudan komutu olmayan TF'yi kurmanın tek yolu budur. Aşağıdaki program aynı yöntemi DF üzerinde gösterir; DF bit 10'dur ve maskesi 0400h'dir.

```
PUSHF             ; FLAGS → yığın
POP  AX           ; AX = FLAGS
OR   AX, 0400h    ; bit 10 (DF) ← 1
PUSH AX
POPF              ; FLAGS ← AX
PUSHF             ; sağlama:
POP  BX           ; BX = yeni FLAGS
```

Komut	AX	BX	SP	DF	Yığının tepesi
`PUSHF`	0000h	0000h	**FFFCh**	0	F202h
`POP AX`	**F202h**	0000h	**FFFEh**	0	0000h
`OR AX, 0400h`	**F602h**	0000h	FFFEh	0	0000h
`PUSH AX`	F602h	0000h	**FFFCh**	0	F602h
`POPF`	F602h	0000h	**FFFEh**	**1**	0000h
`PUSHF`	F602h	0000h	**FFFCh**	1	F602h
`POP BX`	F602h	**F602h**	**FFFEh**	1	0000h

Tablo 2.2.26: PUSHF/POPF ile bir bayrağın kurulması (.COM başlangıcı: IF = 1, öbür bayraklar 0).

Başlangıçta IF = 1, öbür bayraklar 0'dır. PUSHF'in ittiği sözcük F202h'dir. Bunun açılımı öğreticidir: F000h kısmı 12–15 bitleridir ve 8086'da bu bitler her zaman 1 okunur; 0200h bit 9'dur, yani IF = 1; 0002h bit 1'dir ve daima 1'dir. `OR AX, 0400h` yalnız bit 10'u 1 yapar, öbür bitleri korur: F602h. POPF bu değeri bayrak yazmacına yükler ve DF = 1 olur. Son iki satır sağlamadır: bayraklar yeniden okunduğunda BX = F602h çıkar.

TF için aynı kod `OR AX, 0100h` ile yazılır. Bir biti silmek için OR yerine AND ve ters maske kullanılır: `AND AX, 0FEFFh` TF'yi, `AND AX, 0FBFFh` DF'yi siler. Yığın üzerinden çalışmanın bir yararı da öbür bayrakları korumasıdır. Değişiklik bittiğinde SP girişteki değerine döner; PUSH ve POP sayıları eşittir.

PUSHF/POPF'in bir başka kullanımı bayrakları bir yordam boyunca saklamaktır. Bir yordam başında PUSHF, sonunda POPF yaparsa, içeride hangi aritmetik işlem yapılırsa yapılsın çağıran programın bayrakları bozulmaz. Kesmeler de aynı ilkeyle çalışır: işlemci kesmeye girerken FLAGS'i yığına iter, IRET onu geri alır.

2.2.9 Sık yapılan hatalar

Assembler, yazmaçlar ve bayraklarla ilgili hataların çoğunu yakalayamaz; program çalışır ama yanlış sonuç verir. Aşağıdaki hatalar yeni başlayanların programlarında en sık görülenlerdir.

Sık yapılan hata

8 bitlik eldenin komşu yarıya geçtiğini sanmak. AX = 12FFh iken `ADD AL, 1` sonucu 1300h değil 1200h'dir. 8 bitlik işlemin eldesi CF'ye gider. Elde AH'ye aktarılmak isteniyorsa ardından `ADC AH, 0` yazılır ya da işlem baştan 16 bit üzerinde yapılır.

Sık yapılan hata

Köşeli parantezi unutmak ya da gereksiz yere koymak. `MOV AX, BX` BX'in değerini kopyalar; `MOV AX, [BX]` BX'in gösterdiği bellek sözcüğünü okur. `MOV SI, [dizi]` dizinin adresini değil ilk sözcüğünü SI'ya yükler; adres için `MOV SI, OFFSET dizi` ya da `LEA SI, dizi` yazılır.

Sık yapılan hata

Örtük işlenenleri hesaba katmamak. CX bir döngünün sayacıyken döngü içinde `SHL AX, CL` için CL'yi değiştirmek, ya da DX'te bir değer saklarken 16 bitlik MUL veya DIV kullanmak. Komutların hangi yazmaçları gizlice değiştirdiği Tablo 2.2.3'te toplanmıştır; çakışan yazmaç yığına itilip sonra geri alınır.

Sık yapılan hata

DIV'den önce DX'i sıfırlamamak. AX'teki 16 bitlik bir sayıyı BX'e bölmek için yalnız `DIV BX` yazılırsa bölünen DX:AX olur. DX'te önceki bir işlemden kalan değer varsa bölüm 16 bite sığmayabilir ve işlemci bölme hatası kesmesi üretir. İşaretsiz bölmeden önce `MOV DX, 0`, işaretli bölmeden önce `CWD` yazılır.

Sık yapılan hata

MOV'un bayrakları değiştirdiğini sanmak. `MOV AX, 0` ve ardından `JZ` yazmak, AX'in sıfır olduğunu sınamaz; JZ, MOV'dan önceki son aritmetik işlemin ZF'sine bakar. Bir yazmacın sıfır olup olmadığı `CMP AX, 0`, `TEST AX, AX` ya da `OR AX, AX` ile sınanır.

Sık yapılan hata

CF ile OF'yi karıştırmak. İşaretsiz sayılarla çalışan bir programda taşmayı OF ile, işaretli sayılarla çalışanda CF ile sınamak. 7Fh + 01h işleminde CF = 0'dır ama işaretli sonuç yanlıştır; FFh + 01h işleminde CF = 1'dir ama işaretli sonuç doğrudur.

Sık yapılan hata

LAHF bayt hesabında bit 1'i unutmak. LAHF'in ürettiği baytta bit 1 her zaman 1'dir. Bayraklar SF = 0, ZF = 1, AF = 1, PF = 1, CF = 1 iken AH = 57h olur, 55h değil.

Sık yapılan hata

Kesim yazmacına sabit yazmak ya da kesimi BP ile karıştırmak. `MOV DS, @data` yazılamaz; önce AX'e alınır. `[BP+SI]` erişimi DS'de değil SS'de yapılır.

2.2.10 Alıştırmalar ve çözümleri

Sunumdaki beş alıştırma burada ara adımlarıyla çözülmüştür. Sonuçlar emu8086'da çalıştırılarak sağlanabilir.

1. AX = 3A7Fh iken `MOV AL, 0` ve ardından `INC AH` çalışınca AX kaç olur?

`MOV AL, 0` yalnız düşük baytı değiştirir: AH = 3Ah kalır, AL = 00h olur ve AX = 3A00h'dir. `INC AH` yalnız yüksek baytı artırır: 3Ah + 1 = 3Bh. Sonuç AX = **3B00h**'dir. İki komut da öbür yarıya dokunmaz.

2. `MOV AL, 9Ch` / `ADD AL, 64h` sonrası AL, CF, ZF, SF, OF, PF değerlerini bulun.

Önce ikili toplama yapılır: 1001 1100 + 0110 0100 = 1 0000 0000. Sonuç 100h'dir; AL'ye alt bayt 00h yazılır ve taşan 1 CF'ye gider: CF = 1. Sonuç sıfır olduğu için ZF = 1, MSB 0 olduğu için SF = 0'dır. İşaretli yorumda 9Ch = −100, 64h = +100'dür; −100 + 100 = 0 doğrudur ve farklı işaretli iki sayının toplamı taşmaz: OF = 0. 00h'de sıfır tane 1 vardır, çifttir: PF = 1. Düşük nibble'lar C + 4 = 16'dır; bit 3'ten elde çıkar: AF = 1.

3. `MOV AL, 50h` / `ADD AL, 50h`: CF ve OF kaçtır? Neden farklılar?

50h + 50h = A0h. İşaretsiz yorumda 80 + 80 = 160'tır ve 255'ten küçüktür; sonuç 8 bite sığar: CF = 0. İşaretli yorumda iki pozitif sayı, +80 ile +80 toplanmıştır; +160 işaretli aralığın üst sınırı +127'yi aşar. A0h'nin MSB'si 1 olduğu için sonuç −96 gibi görünür. İki pozitifin toplamı negatif çıkmıştır: OF = 1 (SF de 1'dir). İki bayrak farklı soruların cevabıdır: CF işaretsiz, OF işaretli taşmayı bildirir.

4. SP = 0100h iken 3 PUSH ve 1 POP sonrası SP kaçtır?

Her PUSH SP'den 2 çıkarır, her POP 2 ekler: 0100h − 3 × 2 + 2 = 0100h − 6 + 2 = 0100h − 4 = **00FCh**. Ara değerler 00FEh, 00FCh, 00FAh ve POP'tan sonra 00FCh'dir. Yığında iki öğe kalmıştır.

5. `MOV CX, 0` ile başlayan bir LOOP döngüsü kaç kez döner?

LOOP önce azaltır, sonra sınar. İlk LOOP 0 − 1 = FFFFh hesaplar ve atlar. Döngü CX = FFFFh'den 1'e kadar inerken 65.535 kez daha döner; ilk turla birlikte toplam **65.536** kez çalışır. Tur sayısı 32 bitlik DX:BX çiftinde sayılırsa 0001:0000h = 65.536 çıkar. Bu durumu önlemek için döngüden önce `JCXZ` konur.

2.2.11 Özet

- **Yazmaç**, işlemcinin içindeki çok hızlı ve küçük bir bellek hücresidir. Yazmaç erişimi dış yola çıkmaz; bellek erişimi en az dört saat turluk bir yol döngüsü sürer. 8086'da bellekten belleğe MOV ya da ADD yoktur.
- 8086'nın **14 yazmacı** vardır: 4 veri yazmacı (AX, BX, CX, DX), 4 işaretçi ve indis yazmacı (SP, BP, SI, DI), 4 kesim yazmacı (CS, DS, SS, ES), IP ve FLAGS. Hepsi 16 bittir; yalnız AX, BX, CX, DX 8 bitlik iki yarıya bölünür. Bir yarıya yazmak öbür yarıyı değiştirmez ve 8 bitlik işlemin eldesi komşu yarıya değil CF'ye gider.
- Bazı komutlar yazmaçları **örtük** kullanır: MUL/DIV → AX ve DX; LOOP/REP → CX; kaydırma sayısı → CL; IN/OUT → AL/AX ve 255'ten büyük port için DX; PUSH/POP/CALL/RET → SP; dizgi komutları → DS:SI ve ES:DI.
- Köşeli parantez içinde yalnız **BX, BP, SI ve DI** kullanılabilir. BP içeren adresler SS'yi, öbürleri DS'yi kullanır.
- **SP** yığının tepesini gösterir. PUSH önce SP'yi 2 azaltır, sonra yazar; POP önce okur, sonra SP'yi 2 artırır. **BP** prosedür boyunca sabit kalarak parametrelere `[BP+4]` gibi sabit uzaklıklarla erişmeyi sağlar.
- Kesim yazmaçlarına sabit yüklenemez, CS'ye MOV ile yazılamaz. **Fiziksel adres** = kesim × 10h + bağıl konum. **IP** sıradaki komutun bağıl konumudur; komut uzunluğu kadar ilerler, MOV ile yazılamaz.
- Bayrak yazmacının 9 biti kullanılır. **Durum bayrakları**: CF (işaretsiz taşma, ödünç), PF (düşük baytta çift sayıda 1), AF (bit 3'ten bit 4'e elde), ZF (sonuç sıfır), SF (sonucun MSB'si), OF (işaretli taşma). **Kontrol bayrakları**: DF (dizgi yönü), IF (INTR kesmelerine izin), TF (tek adım).
- Donanımda CF son eldedir, ZF bir NOR kapısıdır, SF MSB'nin kopyasıdır, PF bir XOR ağacının tersidir, OF MSB'ye giren ve çıkan eldenin XOR'udur.

- CLC/STC/CMC CF'yi, CLD/STD DF'yi, CLI/STI IF'yi değiştirir. LAHF bayrakların düşük baytını AH'ye kopyalar, SAHF AH'yi bu bayta yazar; PUSHF/POPF bayrak yazmacının tamamını yığına taşır. TF yalnız PUSHF/POPF ile değiştirilebilir.
- MOV, PUSH, POP, XCHG, LEA, LOOP ve atlama komutları bayrakları etkilemez; INC ve DEC CF'ye dokunmaz; MUL ve DIV sonrasında bazı bayraklar tanımsızdır.

2.2.12 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

yazmaç *register* · yazmaç kümesi *register set* · genel amaçlı yazmaç *general-purpose register* · akümülatör *accumulator* · taban yazmacı *base register* · sayaç yazmacı *counter register* · yüksek / düşük bayt *high / low byte* · örtük işlenen *implicit operand* · işaretçi yazmacı *pointer register* · yığın işaretçisi (SP) *stack pointer* · taban işaretçisi (BP) *base pointer* · indis yazmacı *index register* · kaynak indisi (SI) / hedef indisi (DI) *source / destination index* · yığın *stack* · yığın çerçevesi *stack frame* · kesim yazmacı *segment register* · bağıl konum *offset* · fiziksel adres *physical address* · varsayılan kesim *default segment* · komut işaretçisi (IP) *instruction pointer* · bayrak *flag* · bayrak yazmacı (FLAGS) *flags register* · durum bayrağı *status flag* · kontrol bayrağı *control flag* · elde bayrağı (CF) *carry flag* · eşlik bayrağı (PF) *parity flag* · yardımcı elde bayrağı (AF) *auxiliary carry flag* · sıfır bayrağı (ZF) *zero flag* · işaret bayrağı (SF) *sign flag* · taşma bayrağı (OF) *overflow flag* · yön bayrağı (DF) *direction flag* · kesme bayrağı (IF) *interrupt flag* · tuzak bayrağı (TF) *trap flag* · tek adım *single-step* · maskelenebilir / maskelenemez kesme *maskable / non-maskable interrupt* · dizgi komutu *string instruction*

2.2.13 Kendini Dene

1. BX = 12FFh ve CF = 1 iken `INC BL` çalışıyor. BX, ZF ve CF ne olur?

Cevap

INC 8 bitlik BL üzerinde çalışır: FFh + 1 = 00h, BH değişmez ve BX = 1200h olur. Sonuç sıfır olduğundan ZF = 1'dir. INC CF'ye dokunmadığı için CF = 1 kalır; 1 olmasının nedeni bu INC değil, önceki durumdur.

2. AX = 0000h iken `MOV AL, 0F0h`, `ADD AL, 10h` ve `ADC AH, 0` çalışıyor. AX ve CF kaçtır?

Cevap

F0h + 10h = 100h: AL = 00h, CF = 1. ADC AH'ye 0 ile birlikte CF'yi ekler: AH = 00h + 0 + 1 = 01h. AX = 0100h olur. ADC'nin kendi toplamında elde çıkmadığı için son durumda CF = 0'dır. İki komut birlikte, 8 bitlik eldeyi üst bayta taşıyarak 16 bitlik bir toplama yapmıştır.

3. Bir .COM programının başında (SP = FFFEh) AX = ABCDh iken `PUSH AX` çalışıyor. SP kaç olur, hangi adrese hangi bayt yazılır?

Cevap

PUSH önce SP'yi 2 azaltır: SP = FFFCh. Sözcük düşük bayt önce yazılır: SS:FFFCh ← CDh, SS:FFFDh ← ABh. FFFEh'deki 0000h sözcüğü yerinde kalır.

4. `XOR AX, AX` ile `MOV AX, 0` arasında AX açısından fark var mıdır? Bayraklar açısından fark nedir?

Cevap

İkisinde de AX = 0000h olur. XOR bayrakları sonuca göre kurar: ZF = 1, PF = 1, SF = 0, CF = OF = 0 (AF tanımsız). MOV ise hiçbir bayrağı değiştirmez; önceki işlemden kalan bayraklar korunur. XOR ayrıca 2 bayt, `MOV AX, 0` 3 bayttır.

5. `MOV AX, 0FFFFh`, `MOV BX, 0002h`, `MUL BX` sonrasında DX, AX, CF ve OF nedir?

Cevap

FFFFh × 2 = 1FFFEh (65.535 × 2 = 131.070). 32 bitlik sonuç 0001FFFEh'dir: DX = 0001h, AX = FFFEh. Yüksek yarı sıfır olmadığı için CF = OF = 1'dir. SF, ZF, AF ve PF tanımsızdır.

6. 8 bitlik `80h - 01h` ve `7Fh - FFh` işlemlerinde CF ve OF nedir?

Cevap

80h − 01h = 7Fh. İşaretsiz 128 − 1 = 127 doğrudur, ödünç yok: CF = 0. İşaretli −128 − 1 = −129 sığmaz; sonuç +127 çıkmıştır: OF = 1. 7Fh − FFh = 80h. İşaretsiz 127 − 255 negatiftir, ödünç alınır: CF = 1. İşaretli +127 − (−1) = +128 sığmaz: OF = 1.

7. DF = 1 iken SI = 0205h ve 0204h, 0205h adreslerinde 11h ve 22h varsa iki `LODSB` 'den sonra AL ve

SI nedir?

Cevap

Birinci LODSB DS:0205h'deki 22h'yi AL'ye alır, SI = 0204h olur. İkinci LODSB 0204h'deki 11h'yi alır, SI = 0203h olur. Sonuç AL = 11h, SI = 0203h'dir.

8. `MOV AL, 7Fh` ve ardından `INC AL` : AL, SF, OF ve CF ne olur?

Cevap

AL = 80h olur. MSB 1 olduğu için SF = 1'dir. +127 + 1 = +128 işaretli aralığa sığmaz: OF = 1. INC CF'yi değiştirmez; CF önceki değerini (bu programda 0) korur.

9. Bir kesme hizmet yordamı başında kesmeler neden kendiliğinden kapalıdır? NMI de bu sırada engellenir mi?

Cevap

İşlemci bir kesmeyi kabul ederken FLAGS'i yığına iter ve ardından IF ile TF'yi temizler. IF = 0 olduğu için INTR'den gelen yeni istekler yordamı hemen kesemez; yordam isterse STI ile kesmeleri açar. IRET eski FLAGS'i geri yükler. NMI, IF'den etkilenmediği için bu sırada da kabul edilir.

2.2.14 İleri okuma

- Intel, *The 8086 Family User's Manual* (1979), Bölüm 2: yazmaç modeli, bayrakların tam tanımı ve her komutun saat turu sayısı bu kaynaktadır; Tablo 2.2.1'deki değerler buradan alınmıştır.
- B. B. Brey, *The Intel Microprocessors* (8. baskı), Bölüm 2 ve 3: programlama modeli, adresleme biçimleri ve varsayılan kesimler, 8086'dan güncel x86'ya genişlemeyle birlikte.
- M. A. Mazidi, J. G. Mazidi, *The x86 PC: Assembly Language, Design, and Interfacing* (5. baskı), Bölüm 1: yazmaçlar, bayrak yazmacı ve bayrakların elle hesaplandığı çok sayıda örnek.
- M. Morris Mano, *Digital Design*: toplayıcılar, eşlik üreteçleri ve yazmaçların kapı düzeyinde tasarımı; bayrak devrelerinin sayısal tasarım temeli için.

2.2.15 Kaynaklar

1. Intel, *The 8086 Family User's Manual*, 1979.
2. Barry B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson, 2009.
3. Muhammad Ali Mazidi, Janice G. Mazidi, *The x86 PC: Assembly Language, Design, and Interfacing*, 5. baskı, 2010.

4. emu8086 belgeleri: *8086 instructions* ve *Addressing modes* bölümleri.

5. Mehmet Bodur, *Mikroişlemcilere Giriş: Assembler ile Yazılım ve Arayüz*.

KISIM I · BÖLÜM 3

Assembly Dili

Yazılım, makine dilinden yüksek düzeyli dillere uzanan farklı soyutlama düzeylerinde yazılabilir. Bölümde makine dili ve komut biçimi, assembler ile bağlayıcının rolü, farklı platformlardaki assembly dilleri ile derlenen ve yorumlanan diller karşılaştırılır. Assembly kısmında komut satırının etiket, komut, işlenen ve açıklamadan oluşan yapısı, komut ile direktif ayrımı, işlenen türleri ve örtük işlenenler işlenir. Komut süreleri, IP'nin ilerleyişi, bayraklar ve emu8086 ile yazılan ilk programla bölüm tamamlanır.

Kod (8086): `github.com/sercankulcu/microprocessors-8086/Bolum03`

BU BÖLÜMÜN NOTLARI

3.1 Programlama Dilleri

3.2 Assembly

KISIM I · BÖLÜM 3.1

Programlama Dilleri

İşlemci yalnız bitleri tanır. Belleğinden okuduğu her bayt ya bir komutun parçasıdır ya da bir veridir; hangisi olduğunu programın akışı belirler. İnsan ise “a ile b'yi topla, sonucu x'e yaz” diye düşünür. Programlama dilleri bu iki dünya arasındaki uzaklığı kapatmak için doğdu. Not, bu uzaklığı aşağıdan yukarıya doğru katlar hâlinde ele alır. En altta işlemcinin doğrudan çalıştırdığı makine dili vardır; 8086'nın birkaç baytının bellekte nasıl getirilip yürütüldüğü adım adım izlenir. Küçük, uydurma bir komut biçimi üzerinde komutlar elle kodlanır ve çözülür. Ardından assembly dili, assembler'ın iki geçişi, bağlayıcı ve yükleyici gelir. Aynı “Merhaba” programı altı ayrı ortamda yazılarak assembly kodunun hem işlemciye hem işletim sistemine ne kadar bağlı olduğu gösterilir. Son kesimler yüksek düzeyli dilleri derlenen, yorumlanan ve hibrit diye ayırır. Bir sayının tek mi çift mi olduğunu yazan aynı program on dört dilde ve sonunda 8086 assembly ile okunur.

ÖĞRENME ÇIKTILARI

Dil düzeylerini ayırmak
Makine dili, assembly dili ve yüksek düzeyli diller arasındaki farkı soyutlama, taşınabilirlik ve donanım denetimi açısından açıklamak.

Makine kodunu okumak ve yazmak
Verilen bir komut biçimine göre ikili komutları kodlamak ve çözmek; 8086'nın `A1 00 02` gibi baytlarının bellekte nasıl getirilip yürütüldüğünü izlemek.

Araç zincirini anlatmak
Assembler'ın iki geçişini, sembol tablosunu, bağlayıcıyı ve yükleyiciyi açıklamak; liste çıktısını okumak.

Platform bağımlılığını göstermek
Aynı programın x86, 68000, ARM, PowerPC ve farklı işletim sistemlerinde neden farklı yazıldığını sistem çağrısı ve çağrı kuralı üzerinden açıklamak.

Derleme ve yorumlamayı karşılaştırmak
Derleyici, bağlayıcı, yorumlayıcı, ara kod ve JIT kavramlarını hız, taşınabilirlik ve hata yakalama açısından karşılaştırmak.

Diller arasında okumak
Aynı algoritmayı farklı dillerde ve 8086 assembly ile izlemek; `for` ve `if` yapılarının karşılaştırma ve koşullu atlamaya nasıl dönüştüğünü göstermek.

3.1.1 Yazılım ve dil düzeyleri

3.1.1.1 Yazılımın görevi ve katmanlar

Bir bilgisayarın donanımı tek başına hiçbir iş yapmaz. İşlemci, bellek ve giriş/çıkış birimleri elektrik verildiği anda çalışmaya hazırdır, ama neyi hangi sırayla yapacaklarını bilmezler. Bu bilgiyi **yazılım** (*software*) verir. Yazılım, donanımın düzgün, eşgüdümlü ve istenen biçimde çalışmasını sağlayan komutların ve verilerin bütünüdür. Elektronik bir devrede sinyallerin sırasını bir denetim birimi belirler; bilgisayarda bu sırayı bellekteki program belirler.

Yazılımın en temel parçası **işletim sistemidir** (*operating system*). İşletim sistemi donanımı yönetir: belleği programlara paylaştırır, diske ve ekrana erişimi düzenler, aynı anda çalışan programların birbirini bozmasını engeller. Uygulama programları donanıma doğrudan dokunmaz; bir dosya açmak ya da ekrana yazı yazmak istediklerinde işletim sisteminden hizmet ister. Bu isteğin adı **sistem çağrısıdır**; ayrıntıları "Merhaba" programlarında görülecek. İşletim sistemi de donanıma, aygıtın kendine özgü ayrıntılarını bilen **sürücüler** (*driver*) aracılığıyla ulaşır.

Şekil 3.1.1 bu düzeni dört katman olarak gösterir. Kullanıcı en üstte, donanım en alttadır. Her katman yalnız hemen altındakinin sunduğu hizmetleri kullanır. Tarayıcı, derleyici ya da oyun gibi uygulamalar işletim sistemini, işletim sistemi de işlemciyi, belleği ve giriş/çıkış birimlerini kullanır.

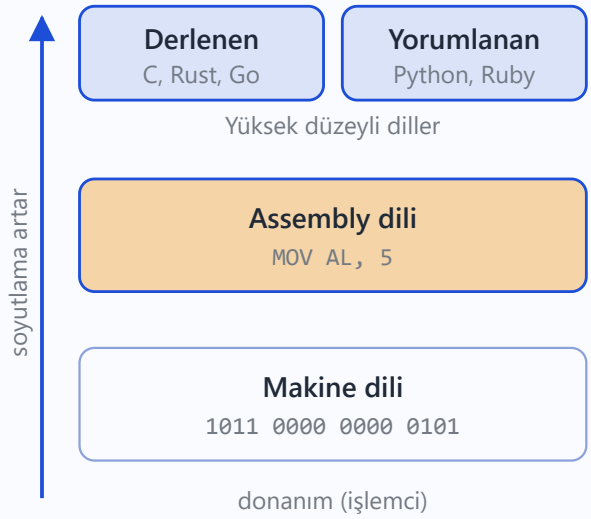


Şekil 3.1.1: Bilgisayar sisteminin katmanları. Uygulamalar donanıma işletim sistemi üzerinden ulaşır; işletim sistemi sistem çağrılarını karşılar ve sürücülerle donanımı yönetir.

Uygulamaları, işletim sistemini ve sürücüleri yazmak için **programlama dilleri** kullanılır. Bir programlama dili, bir işi adım adım tarif etmeye yarayan, kuralları kesin olarak tanımlanmış yapay bir dildir. Bugün yüzlerce dil vardır. Bazıları işletim sistemi yazmaya, bazıları sayısal hesaba, bazıları web sayfalarına yöneliktir. Dillerin tarihi, bilgisayar teknolojisinin tarihini de yansıtır. İlk bilgisayarlar kablolarla ve anahtarlarla programlanıyordu. Bellek büyüyüp işlemciler hızlandıkça programcının yükünü makineye devreden diller ortaya çıktı.

3.1.1.2 Üç dil düzeyi

Programlama dilleri, donanıma ne kadar yakın olduklarına göre üç düzeyde toplanır. Şekil 3.1.2 bu düzeyleri üst üste dizer; yukarı çıkıldıkça **soyutlama** artar, yani programcı donanımın ayrıntılarını daha az görür.



Şekil 3.1.2: Dil düzeyleri. En altta işlemcinin doğrudan çalıştırdığı makine dili, onun üstünde makine komutlarının sembolik yazımı olan assembly dili, en üstte derlenen ve yorumlanan yüksek düzeyli diller.

Tanım 3.1.1

Makine dili (*machine language*), işlemcinin hiçbir çeviri olmadan doğrudan çalıştırdığı ikili komutlardır. **Assembly dili** (*assembly language*), her makine komutunu `MOV`, `ADD` gibi akılda kalır bir adla, yani bir **anımsatıcıyla** (*mnemonic*) yazan dildir. Bir assembly satırı çoğunlukla tek bir makine komutuna karşılık gelir. **Yüksek düzeyli dil** (*high-level language*), işlemcinin komutlarından bağımsız olarak değişkenler, ifadeler, döngüler ve fonksiyonlarla yazılan dildir; bir satırı genellikle birden çok makine komutuna dönüşür.

Yüksek düzeyli diller de kendi içinde ikiye ayrılır. **Derlenen** dillerde (C, Rust, Go) kaynak kod, program çalışmadan önce bütünüyle makine diline çevrilir. Ortaya çıkan dosya doğrudan işlemcide çalışır. **Yorumlanan** dillerde (Python, Ruby) kaynak kod çalışma anında bir **yorumlayıcı** programca okunup

işlenir. Şekildeki örnekler bu ayrımı gösterir: `MOV AL, 5` satırı assembly dilinde, onun karşılığı olan `1011 0000 0000 0101` bitleri makine dilindedir.

Düzey yükseldikçe üç şey birlikte değişir. Birincisi **okunabilirlik**tir. `x = a + b` satırını okuyan herkes ne yapıldığını anlar; aynı işin makine kodunu çözmek için komut tablosu gerekir. İkincisi **taşınabilirliktir**. C ile yazılmış bir program x86 için de ARM için de derlenebilir. 8086 makine kodu ise yalnız 8086 ailesinde çalışır. Üçüncüsü **doğrudan denetimdir** ve bu ters yönde değişir. Yüksek düzeyli dilde değişkenin hangi yazmaçta tutulacağına, bir döngünün hangi atlama komutuyla kurulacağına derleyici karar verir. Assembly yazan programcı ise her yazmacı, her bayrağı ve her bellek erişimini kendisi seçer. Aygıt sürücüsü, kesme yordamı ya da önyükleyici gibi donanıma yakın işlerde bu denetim gereklidir.

Elektrik-elektronik mühendisliği açısından bu düzeyler tanıdık bir benzetmeyle de düşünülebilir. Makine dili, bir sayısal devrenin denetim girişlerine uygulanan bit örüntüleridir. Assembly dili, bu örüntülere ad veren bir sembol listesidir. Yüksek düzeyli dil ise bir donanım tanımlama dilinde davranışı tarif edip ayrıntıyı sentez aracına bırakmaya benzer. Her düzeyde iş aynıdır; değişen, ayrıntıyı kimin doldurduğudur.

3.1.1.3 Aynı işlem üç düzeyde

Düzeyler arasındaki farkı görmenin en iyi yolu aynı işi üç düzeyde yazmaktır. İş basittir: `a` ile `b` değişkenlerini toplayıp sonucu `x` değişkenine yazmak. Üç değişken de 16 bitlik tam sayıdır.

C dilinde bu iş tek satırdır:

```
x = a + b;
```

Bu satır değişkenlerin bellekte nerede durduğunu, toplamanın hangi yazmaçta yapılacağını söylemez. Bu kararlar derleyiciye bırakılmıştır. 8086 assembly dilinde aynı iş üç satır tutar:

```
MOV AX, [a]     ; a'yı AX yazmacına yükle
ADD AX, [b]     ; AX = AX + b
MOV [x], AX     ; sonucu x'e yaz
```

Burada her şey açıktır. Toplama AX yazmacında yapılır. Köşeli parantez, değişkenin adresindeki bellek hücresinin içeriği anlamına gelir. 8086'da iki bellek işleneni tek bir komutta birlikte kullanılamaz. Bu yüzden önce `a` yazmaca alınır, sonra `b` eklenir, en son sonuç belleğe geri yazılır. Her satır tek bir makine komutudur.

Makine dilinde ise yalnız baytlar kalır. Değişkenlerin adresleri `a` = 0200h, `b` = 0202h, `x` = 0204h olsun. “h” soneki sayının onaltılık yazıldığını gösterir. Assembler bu üç satırı şu baytlara çevirir:

```
A1 00 02        ; MOV AX, [0200h]
03 06 02 02     ; ADD AX, [0202h]
A3 04 02        ; MOV [0204h], AX
```

Her komutun ilk baytı **işlem kodudur** (*opcode*): işlemciye hangi işin yapılacağını söyler. `A1`, "bellekteki bir sözcüğü AX'e yükle" demektir ve adres hemen arkasından gelir. `A3` bunun tersidir: "AX'i belleğe yaz". Bu iki komut AX'e özgü kısa biçimlerdir. `03` ise genel bir toplama komutudur: "16 bitlik bir yazmaca bir yazmacı ya da bellek sözcüğünü ekle". Hangi yazmaç ve hangi bellek hücresi olduğu ikinci bayttan anlaşılır. Bu bayta ModR/M baytı denir. `06` = `00 000 110` b; ortadaki `000` AX'i, kenarlardaki `00` ve `110` ise "adres komutun içinde doğrudan verilmiştir" durumunu kodlar.

Adres baytlarının sırası ilk bakışta tersmiş gibi görünür: 0200h adresi `00 02` diye yazılmıştır. 8086, çok baytlı sayıları bellekte **düşük bayt önce** (*little-endian*) saklar. 0200h sayısının düşük baytı 00h, yüksek baytı 02h'dir; önce 00h, sonra 02h gelir. Aynı kural verilere de uygulanır: `a` = 0015h ise 0200h adresinde 15h, 0201h adresinde 00h bulunur.

> **Sık yapılan hata**
>
> Makine kodundaki adres baytlarını yazıldığı sırayla okumak. `A1 00 02` komutundaki adres 0002h değil, 0200h'dir. 8086'da komut içindeki 16 bitlik her sayı, adres de olsa sabit de olsa, düşük bayt önce yazılır. Bellek dökümünde bir sözcük okurken de ikinci bayt yüksek bayttır.

3.1.1.4 Kod izleme: `x = a + b` bellekte

Bu üç komutun bellekte nasıl çalıştığını izlemek için onları küçük bir COM programına koyalım. COM programı, emu8086 ve DOS'un kullandığı en basit çalıştırılabilir dosya türüdür. Kod, veri ve yığın aynı 64 KB'lık kesimde durur; program kesimin 0100h adresinden başlar.

```
org 100h
mov ax, [0200h]   ; A1 00 02
add ax, [0202h]   ; 03 06 02 02
mov [0204h], ax   ; A3 04 02
ret               ; C3
```

`org 100h` bir **direktiftir**: makine kodu üretmez, yalnız assembler'a ilk baytın 0100h adresine yerleşeceğini söyler. Kod 0100h–010Ah aralığındaki 11 bayttır: üç komut 3 + 4 + 3 = 10 bayt, `ret` 1 bayt. Başlangıçta `a` = 0015h (21) ve `b` = 0027h (39) olsun; `x`'in değeri bilinmiyor.

İşlemci her komutu iki evrede işler. **Getirme** (*fetch*) evresinde komut baytları IP'nin gösterdiği adresten okunur ve IP komut uzunluğu kadar ilerler. **Yürütme** (*execute*) evresinde işlem kodu çözülür ve iş yapılır. 8086'da bu iki evre iki ayrı birime dağıtılmıştır. **Yol arayüz birimi** (BIU, *bus interface unit*) bellekten komut baytlarını okuyup altı baytlık bir **komut kuyruğuna** doldurur. **Yürütme birimi** (EU,

execution unit) baytları bu kuyruktan alıp çözer ve yürütür. Yürütme birimi bir işlemi yaparken veri yolu boştaysa BIU sıradaki baytları önceden getirir. İzleme tablosunda iki evre ayrı satırlarda gösterilir, ama gerçek işlemcide bir komutun yürütülmesi ile sonrakinin getirilmesi zaman içinde örtüşebilir.

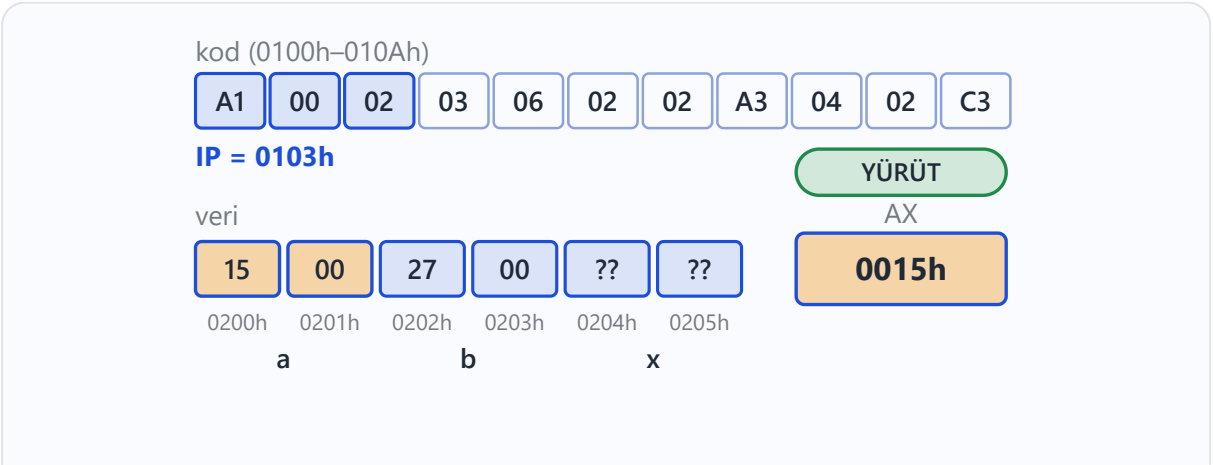
Adım	Evre	Komut baytları	IP	AX	0204h–0205h (x)
1	getir	`A1 00 02`	0100h → 0103h	????	?? ??
2	yürüt	0200h'den sözcük oku: `15 00`	0103h	0015h	?? ??
3	getir	`03 06 02 02`	0103h → 0107h	0015h	?? ??
4	yürüt	0202h'den `27 00` ; 15h + 27h	0107h	003Ch	?? ??
5	getir	`A3 04 02`	0107h → 010Ah	003Ch	?? ??
6	yürüt	AX'i 0204h'ye yaz	010Ah	003Ch	3C 00
7	getir, yürüt	`C3` (RET)	010Ah → dönüş	003Ch	3C 00

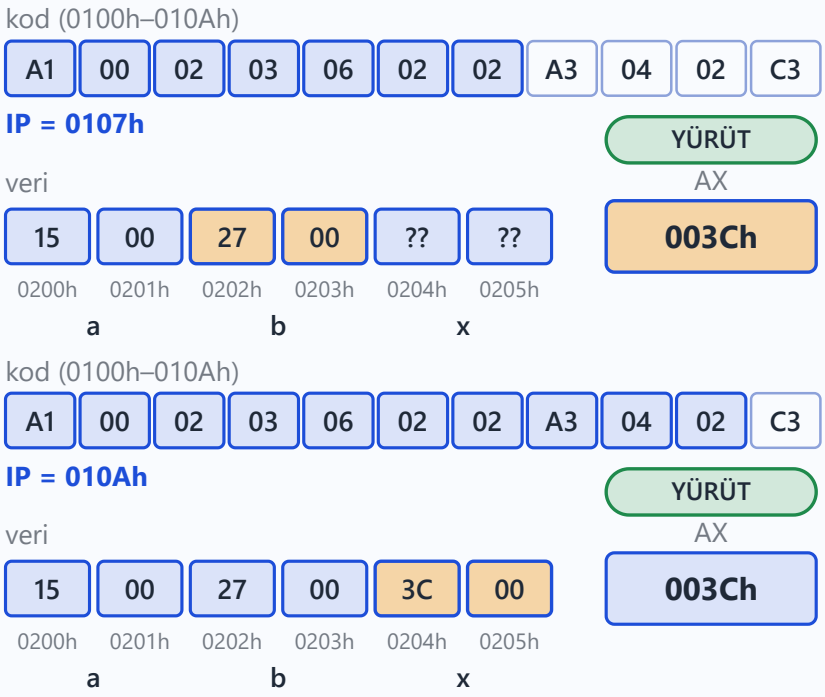
Tablo 3.1.1: `x = a + b` makine kodunun izlemesi (a = 0015h, b = 0027h).

Adım 1'de IP = 0100h'dir. İşlemci `A1` baytını okuyunca komutun üç bayt olduğunu anlar; arkadan gelen `00 02` baytları adresi oluşturur. Adres little-endian okunur: 0200h. IP 0103h'ye ilerler. Adım 2'de yürütme birimi veri kesiminin 0200h adresinden bir sözcük ister. BIU, 0200h'de duran 15h'yi ve 0201h'de duran 00h'yi okur; AX = 0015h olur. Bu okuma gerçek bir yol döngüsüdür. 8086'nın veri yolu 16 bit genişliğindedir ve 0200h çift bir adrestir. Bu yüzden sözcüğün iki baytı tek bir yol döngüsünde, yani en az dört saat turunda gelir. Sözcük tek adreste dursaydı iki ayrı çevrim gerekirdi.

Adım 3'te dört baytlık `03 06 02 02` getirilir. `03` toplama işlemini, `06` "hedef AX, kaynak doğrudan adres" bilgisini, `02 02` ise 0202h adresini taşır. Adım 4'te 0202h'den 0027h okunur ve aritmetik-mantık biriminde AX'e eklenir: 0015h + 0027h = 003Ch, ondalık olarak 21 + 39 = 60. Toplama bayrakları da günceller: sonuç sıfır olmadığından ZF = 0, en yüksek bit 0 olduğundan SF = 0 olur ve elde çıkmadığından CF = 0 kalır.

Adım 5'te `A3 04 02` getirilir ve adım 6'da AX belleğe yazılır. Yazma da düşük bayt önce yapılır: 0204h adresine 3Ch, 0205h adresine 00h gider. Artık `x` = 003Ch = 60'tır. Adım 7'deki `C3` baytı RET komutudur; COM programı denetimi kendisini çalıştıran sisteme geri verir. Şekil 3.1.3 bu izlemenin üç anını gösterir.





Şekil 3.1.3: `x = a + b` makine kodunun üç anı. Üstte birinci komut yürütülmüş ve AX = 0015h olmuştur; ortada toplama yapılmış ve AX = 003Ch olmuştur; altta AX, 0204h ve 0205h adreslerine 3C 00 olarak yazılmıştır. Vurgulu kutular o adımda kullanılan baytlardır.

Bu izleme, C'deki tek satırın donanımda neye dönüştüğünü ortaya koyar. Satır üç komuta ve on bayt makine koduna dönüşmüştür. Bunlara üç veri erişimi ve komut baytlarını getiren yol döngüleri eklenir. 8086 kullanıcı kılavuzundaki zamanlama tablosu bu komutların sürelerini de verir. Bellekten AX'e yükleme 10 saat turu sürer. Bellekteki bir sözcüğü yazmaca eklemek doğrudan adreslemede 9 + 6 = 15, AX'i belleğe yazmak yine 10 turdur. 5 MHz'lik bir 8086'da bir saat turu 200 ns olduğundan bu üç komut yaklaşık 35 × 200 ns = 7 µs tutar. Bu sürelere komut kuyruğu boşaldığında getirme için beklenen turlar eklenebilir.

Örnek 3.1.1

Aynı program `a` = 7FFFh, `b` = 0001h ile çalıştırılırsa `x` ve bayraklar ne olur?

Toplama 7FFFh + 0001h = 8000h verir. Belleğe 0204h ← 00h, 0205h ← 80h yazılır. Sonuç sıfır değildir (ZF = 0). En yüksek bit 1 olduğundan SF = 1 olur. İşaretsiz toplamda 16 bitten taşan elde yoktur (CF = 0). İşaretli sayılar açısından ise iki pozitif sayının toplamı negatif görünmüştür: +32767 + 1 sonucu −32768 çıkmıştır. Bu bir işaretli taşmadır ve OF = 1 olur. İşlemci taşmayı yalnız OF bayrağına yazar ve işine devam eder. C de işaretli taşmayı kendiliğinden bildirmez; dil standardı bu durumu tanımsız davranış sayar. Taşmayı denetlemek programcının işidir.

3.1.2 Makine dili

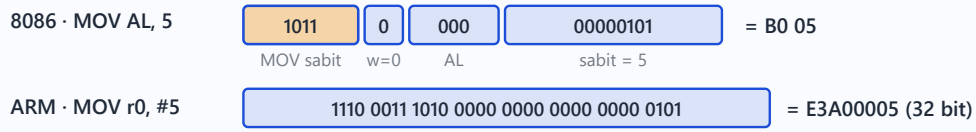
3.1.2.1 İşlemcinin tek anladığı dil

Makine dili, 0 ve 1'lerden oluşan en düşük düzeyli dildir. İşlemcinin denetim birimi bellekten okuduğu bit örüntüsünü çözer ve buna göre yazmaçları, aritmetik-mantık birimini ve veri yolunu sürer. Bu çözme işi kod çözücü devrelerle yapılır: belirli bir işlem kodu, belirli denetim hatlarını etkin kılar. Bu yüzden makine dili tamamen **işlemciye ve donanıma bağımlıdır**. Bir işlemci ailesi için yazılmış makine kodu başka bir ailenin kod çözücüsünde anlamsızdır ya da bambaşka bir iş yapar.

İlk genel amaçlı elektronik bilgisayarlardan ENIAC (1945) belleğe yüklenen bir programla çalışmıyordu. Program, birimler arasındaki kablolar takılarak ve anahtarlar ayarlanarak donanımın üzerine kuruluyordu. Yeni bir problem için kabloların yeniden düzenlenmesi günler sürebiliyordu. Programı da veri gibi belleğe koyma fikri bu sorunu çözdü. Bellekte duran komutlar artık sayılardı ve bu sayıların dili makine diliydi.

Günümüzde her işlemci ailesinin kendine özgü bir makine dili, başka bir deyişle bir **komut kümesi** (*instruction set*) vardır. Makine dili okumak ve yazmak için pratik değildir; bu yüzden programcılar onu doğrudan kullanmaz. Yine de bir mikroişlemci mühendisinin makine kodunu çözebilmesi gerekir. Bellek dökümünü okumak, bir mantık çözümleyicisinde veri yolundaki baytları anlamak ya da bir derleyicinin ne ürettiğini denetlemek bu beceriye dayanır.

Şekil 3.1.4, “AL yazmacına 5 yükle” işinin iki ayrı makine dilinde nasıl kodlandığını gösterir. 8086'da bu iş iki bayttır: `B0 05`. İlk baytın üst dört biti `1011`, “yazmaca sabit yükle” komutunu belirtir. Sonraki bit `w` = 0, işlemin 8 bitlik olduğunu söyler. Son üç bit `000` hedef yazmacı seçer; 8 bitlik yazmaçlarda 000 AL'dir. İkinci bayt sabitin kendisidir: 00000101 = 5. Aynı komut 16 bitlik AX için yazılsaydı `w` = 1 olur ve sabit iki bayt tutardı: `MOV AX, 5` = `B8 05 00`.



Şekil 3.1.4: Aynı iş iki makine dilinde. 8086'da `MOV AL, 5` iki baytlık `B0 05`'tir: işlem kodu 1011, `w` biti, yazmaç alanı ve sabit. ARM'da `MOV r0, #5` tek bir 32 bitlik sözcüktür: `E3A00005`.

ARM işlemcilerinde ise bütün komutlar 32 bit uzunluğundadır. `MOV r0, #5` komutu `E3A00005` sözcüğüne çevrilir. Bu sözcüğün en üstteki dört biti `1110` “her zaman çalıştır” koşuludur. Sonraki alanlar komutun bir veri işleme komutu olduğunu, işlemin MOV olduğunu, hedef yazmacın r0 olduğunu ve en alttaki sekiz bitte 5 sabitinin durduğunu kodlar. Aynı iş için 8086 iki bayt, ARM dört bayt kullanır. Alanların yeri ve anlamı da tamamen farklıdır. Bu fark, 8086 kodunun bir ARM işlemcide neden çalışmadığını tek örnekle açıklar.

Makine dilinin bu bağımlılığı assembly dilleriyle yumuşatılır. Assembly dili de işlemciye bağlıdır, ama programcıyı bit alanlarıyla uğraşmaktan kurtarır ve daha insan dostu bir programlama sağlar.

3.1.2.2 Örnek bir komut biçimi

Gerçek bir işlemcinin komut kümesi yüzlerce komut ve onlarca biçim içerir. Makine dilinin ana fikrini görmek için çok daha küçük, uydurma bir işlemci tasarlamak yeterlidir. Bu işlemcinin dört yazmacı (R0–R3) olsun ve her komutu 8 bit tutsun. Şekil 3.1.5 komutun alanlarını gösterir.



Şekil 3.1.5: Örnek işlemcinin 8 bitlik komut biçimi. Bit 7–4 işlem kodu, bit 3–2 hedef yazmaç, bit 1–0 kaynak yazmaçtır.

Tanım 3.1.2

Komut biçimi (*instruction format*), bir makine komutunun bitlerinin hangi alanlara ayrıldığını ve her alanın neyi kodladığını belirleyen düzendir. **İşlem kodu** (*opcode*) alanı yapılacak işlemi, **işlenen** (*operand*) alanları ise işlemin hangi yazmaç, bellek hücresi ya da sabit üzerinde yapılacağını belirtir.

Örnek işlemcide komutun anlamı her zaman aynı kalıba uyar: **hedef ← hedef işlem kaynak**. Sonuç hedef yazmaca yazılır; kaynak yazmaç değişmez. Yazmaçlar iki bitle kodlanır: R0 = `00`, R1 = `01`, R2 = `10`, R3 = `11`. İki bit dört yazmacı ayırt etmeye tam olarak yeter. İşlem kodları Tablo 3.1.2'de verilmiştir.

İşlem kodu	Anımsatıcı	Anlamı
`0000`	HALT	programı durdur
`0001`	ADD	hedef ← hedef + kaynak
`0010`	SUB	hedef ← hedef − kaynak
`0011`	AND	hedef ← hedef VE kaynak (bit bit)
`0100`	OR	hedef ← hedef VEYA kaynak (bit bit)

Tablo 3.1.2: Örnek işlemcinin işlem kodları.

İşlem kodu alanı 4 bit olduğu için en çok $2^4 = 16$ farklı işlem tanımlanabilir. Bu tasarım yalnız beşini kullanıyor; kalan on bir kod sonradan eklenecek komutlar için boştur. Bir alanın genişliği, o alanın seçebileceği şey sayısını belirler: k bitlik bir alan 2^k şeyden birini seçer. Tersine, n şeyden birini seçmek

için en az $\lceil \log_2 n \rceil$ bit gerekir.

Örnek bir komut: `0001 10 11`. İlk dört bit 0001 = ADD, sonraki iki bit 10 = R2 (hedef), son iki bit 11 = R3 (kaynak). Komut ADD R2, R3'tür ve R2 ← R2 + R3 işini yapar. Onaltılık yazımda bu bayt 1Bh'dir. Onaltılık yazımın kolaylığı burada görülür: ilk onaltılık basamak doğrudan işlem kodudur, ikinci basamak ise hedef ve kaynak alanlarını birlikte taşır.

3.1.2.3 Komutları kodlamak ve çözmek

Bir assembly satırını makine koduna çevirmeye **kodlama**, bir bayttan assembly satırını geri çıkarmaya **çözme** denir. Assembler birincisini, ters assembler (*disassembler*) ve işlemcinin kod çözücüsü ikincisini yapar. Çözmenin yolu bitleri alanlarına ayırmaktır. Tablo 3.1.3 dört komutluk bir programı alanlarına ayırır.

Makine kodu	Onaltılık	İşlem	Hedef	Kaynak	Anlamı
`0001 0001`	11h	ADD	R0	R1	R0 ← R0 + R1
`0011 1011`	3Bh	AND	R2	R3	R2 ← R2 VE R3
`0100 0110`	46h	OR	R1	R2	R1 ← R1 VEYA R2
`0000 0000`	00h	HALT	—	—	programı durdur

Tablo 3.1.3: Örnek programın alanlara ayrılması.

Biçim, ilk iki bitlik yazmaç alanını hedef olarak tanımlar. Bu yüzden her komutun sonucu **ilk** yazılan yazmaca gider: birinci komutta R0'a, ikincide R2'ye, üçüncüde R1'e. HALT komutunda yazmaç alanları kullanılmaz; sıfır bırakılmıştır.

Örnek 3.1.2

`SUB R2, R0` komutunu kodlayalım ve 4Dh baytını çözelim.

SUB'un işlem kodu 0010, R2 = 10, R0 = 00'dır. Alanlar yan yana yazılınca `0010 1000` elde edilir; onaltılık olarak 28h. R2 = 12 ve R0 = 5 ise komut R2 ← 12 − 5 = 7 yapar, R0 değişmez.

4Dh = `0100 1101`. İşlem kodu 0100 = OR. Kalan dört bit 11 01'dir: hedef R3, kaynak R1. Bayt `OR R3, R1` komutudur ve R3 ← R3 VEYA R1 işini yapar. R3 = 1010 ve R1 = 0011 ise sonuç 1011 = 11 olur.

3.1.2.4 Kod izleme: örnek makine programı

Tablo 3.1.3'teki program bellekte 0, 1, 2 ve 3 adreslerinde durur. İşlemcinin **program sayacı** (PC), sıradaki komutun adresini tutar. 8086'da bu yazmacın adı IP'dir. Her komut getirildiğinde PC bir artar, çünkü her komut tek bayttır.

```
0001 0001   ; ADD R0, R1 → R0 ← R0 + R1
0011 1011   ; AND R2, R3 → R2 ← R2 AND R3
0100 0110   ; OR  R1, R2 → R1 ← R1 OR R2
0000 0000   ; HALT
```

Başlangıç değerleri R0 = 5, R1 = 3, R2 = 12, R3 = 10 olsun. Mantıksal işlemleri izleyebilmek için değerleri ikili de yazmak gerekir. Tablo 3.1.4 her komuttan sonraki durumu verir; değişen yazmaç kalın yazılmıştır.

Adım	Komut	PC	R0	R1	R2	R3
başlangıç	—	0	0000 0101 (5)	0000 0011 (3)	0000 1100 (12)	0000 1010 (10)
1	ADD R0, R1	1	**0000 1000 (8)**	0000 0011 (3)	0000 1100 (12)	0000 1010 (10)
2	AND R2, R3	2	0000 1000 (8)	0000 0011 (3)	**0000 1000 (8)**	0000 1010 (10)
3	OR R1, R2	3	0000 1000 (8)	**0000 1011 (11)**	0000 1000 (8)	0000 1010 (10)
4	HALT	3	0000 1000 (8)	0000 1011 (11)	0000 1000 (8)	0000 1010 (10)

Tablo 3.1.4: Örnek makine programının izlemesi.

Birinci komut ADD R0, R1'dir: 5 + 3 = 8 olur ve sonuç R0'a yazılır. Kaynak R1 değişmez. İkinci komut AND R2, R3'tür. Bit bit VE işleminde yalnız iki tarafta da 1 olan bit 1 kalır: 1100 VE 1010 = 1000, yani R2 = 8. Üçüncü komut OR R1, R2'dir. VEYA işleminde iki taraftan birinde 1 olan bit 1 olur: 0011 VEYA 1000 = 1011, yani R1 = 11. Bu komut R2'nin bir önceki adımda değişmiş değerini kullanır. Komutlar sırayla yürütüldüğünden her komut, öncekilerin bıraktığı durumu görür. Dördüncü komut HALT'tır ve işlemci durur. Bu tasarımda HALT getirildikten sonra PC artırılmaz ve 3'te kalır; gerçek işlemcilerde bu ayrıntı tasarıma göre değişir. Sonuç: R0 = 8, R1 = 11, R2 = 8, R3 = 10.

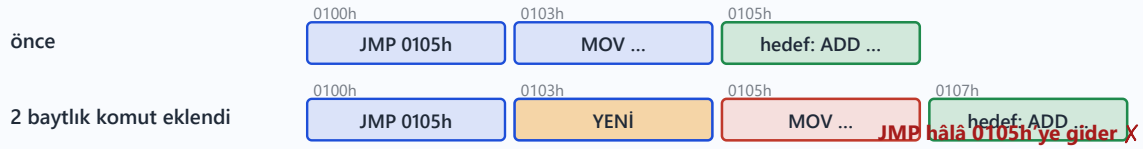
Bu küçük işlemci, 8086'nın da temel çalışma döngüsünü taşır: getir, PC'yi ilerlet, çöz, yürüt, sonucu yaz. İşlemci Simülatörü, benzer bir 8 bitlik işlemcide bu adımları PC, komut yazmacı, bellek adres ve veri yazmaçları üzerinde tek tek gösterir.

3.1.2.5 Makine dilinin artıları ve eksileri

Makine dilinin artıları doğrudan donanıma yakınlığından gelir. Çeviri gerekmez; işlemci kodu olduğu gibi çalıştırır. Programcı donanım üzerinde tam denetime sahiptir: hangi bayt nereye yazılacak, hangi komut hangi sırayla gelecek, hepsi onun kararıdır. Doğru yazılırsa en küçük ve en hızlı kod da böyle elde edilir.

Eksileri ise ağırdır. Makine kodu okunmaz, yazılması ve hata ayıklaması çok zordur. Tek bir bitin yanlış olması komutu başka bir komuta çevirir; 0001 0001 (ADD R0, R1) yerine 0011 0001 yazılırsa toplama, VE işlemine dönüşür. Kod taşınabilir de değildir. Program her işlemci için baştan yazılmak zorundadır.

Belki de en can sıkıcı sorun adreslerin elle hesaplanmasıdır. Atlama komutları hedefin adresini taşır. Programın ortasına bir komut eklenince arkadan gelen bütün komutların adresi kayar, ama atlama komutlarının içindeki adresler eski hâlinde kalır. Şekil 3.1.6 bu durumu gösterir.



Şekil 3.1.6: Makine koduna komut eklemenin sonucu. Üstte `JMP 0105h` doğru hedefe gider. Altta araya iki baytlık bir komut eklenmiştir; hedef 0107h'ye kaymıştır, ama atlama hâlâ 0105h'ye gider.

Programcı bu durumda bütün atlama adreslerini yeniden bulup düzeltmek zorundadır. Büyük bir programda bu iş hem uzun sürer hem de hataya açıktır. Bu sorunları çözmek için önce assembly dilleri (1950'ler), ardından FORTRAN (1957) gibi yüksek düzeyli diller geliştirildi. Assembly dilinde atlama hedefi bir etiketle yazılır; adresi assembler her seferinde yeniden hesaplar.

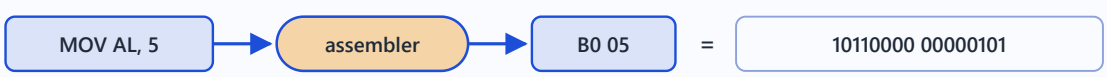
Sık yapılan hata

Komut biçimindeki alanların sırasını karıştırmak. Örnek işlemcide hedef alanı kaynaktan önce gelir: `0010 11 00` SUB R3, R0'dır ve R3 ← R3 − R0 yapar. Alanlar ters okunursa SUB R0, R3 elde edilir; çıkarma değişmeli olmadığı için sonuç da başka bir yazmaca, başka bir değerle yazılır.

3.1.3 Assembly dilleri ve assembler

3.1.3.1 Anımsatıcılar

Assembly dilinin ilk kazancı adlardır. Programcı `10110000 00000101` yerine `MOV AL, 5` yazar. `MOV` bir anımsatıcıdır: "taşı" (*move*) sözcüğünün kısaltması olarak komutun ne yaptığını hatırlatır. `AL` yazmacın adıdır, `5` ise ondalık yazılmış sabittir. Bu satırı makine koduna çeviren programa **assembler** denir. Şekil 3.1.7 çevirinin tek bir satır için ne yaptığını gösterir.



Şekil 3.1.7: Assembler, `MOV AL, 5` satırını `B0 05` makine koduna, yani `10110000 00000101` bitlerine çevirir.

Assembly dili her işlemci mimarisi ve komut kümesi için ayrı tasarlanır. Intel x86, Motorola 68000, ARM, PIC ve AVR işlemcilerinin her birinin kendi assembly dili vardır. Bir dilde geçerli olan yazmaç adları ve komutlar ötekinde yoktur. Bir işlemci ailesi içinde ise uyum korunur. 8086 için yazılmış assembly programı, küçük farklarla 80286, 80386 ve günümüz x86 işlemcilerinde de çevrilip çalıştırılabilir. Mikrodenetleyicilere özel assembly dilleri gömülü sistemlerde ve elektronik uygulamalarda hâlâ sıkça kullanılır.

Assembly dili yüksek düzeyli dillerin yaygınlaştığı bugün de üç nedenle önemlidir. Birincisi, bazı kodlar yalnız assembly ile yazılabilir. Bilgisayar açıldığında ilk çalışan önyükleyici, kesme yordamları, aygıt sürücülerinin donanıma dokunan kısımları ve işletim sistemi çekirdeğinin bağlam değiştiren parçaları bu türdendir. İkincisi, işlemcinin gerçekte nasıl çalıştığını anlamanın yolu assembly'den geçer. Bir programın neden çöktüğü, bir hata ayıklayıcının gösterdiği kodun ne yaptığı ya da tersine mühendislikle bir yazılımın incelenmesi assembly okumayı gerektirir. Üçüncüsü performanstır. Derleyicinin kendiliğinden üretemediği özel komutlar, örneğin SIMD ya da şifreleme komutları, ve zamanı kritik döngüler hâlâ elle yazılır.

3.1.3.2 Kaynak koddan çalışan programa

Programcının yazdığı metin dosyasına **kaynak kod** (*source code*) denir. Kaynak koddan çalışan bir programa giden yolda üç araç görev alır. Şekil 3.1.8 bu araç zincirini gösterir.



Şekil 3.1.8: Kaynak dosya assembler ile nesne dosyasına, bağlayıcı ile çalıştırılabilir dosyaya dönüşür; yükleyici bu dosyayı belleğe yerleştirir.

Assembler
: Kaynak dosyayı (`prog.asm`) okur ve her satırı makine koduna çevirir. Çıktısı bir **nesne dosyasıdır** (*object file*, `prog.obj` ya da `prog.o`). Nesne dosyası makine kodunu içerir, ama henüz tamamlanmamıştır. Başka dosyalarda tanımlı fonksiyonların adresleri boş bırakılmış, bu boşluklar bir listeye yazılmıştır. MASM, NASM, TASM ve GNU as bu işi yapan araçlardır.

Bağlayıcı
: **Bağlayıcı** (*linker*) bir ya da daha çok nesne dosyasını ve kütüphaneleri (`.lib` , `.obj`) birleştirir. Boş bırakılmış adresleri doldurur ve işletim sisteminin anlayacağı biçimde bir **çalıştırılabilir dosya** (*executable file*, `prog.exe`) üretir. Microsoft'un LINK'i ve GNU'nun ld'si yaygın bağlayıcılardır.

Yükleyici
: **Yükleyici** (*loader*) işletim sisteminin bir parçasıdır. Çalıştırılabilir dosyayı diskten belleğe kopyalar, kesim yazmaçlarını ve yığını hazırlar, gerekirse adresleri programın yüklendiği yere göre

düzeltir ve denetimi programın ilk komutuna verir.

Assembler'ın bir başka çıktısı daha vardır: **liste** (*listing*) **çıktısı**. Liste, her kaynak satırının yanına o satırın adresini ve ürettiği makine kodunu yazar. Ekrana “Merhaba!” yazan küçük bir DOS programının listesi şöyledir:

```
Adres   Makine kodu   Kaynak
0100    B4 09         mov ah, 09h
0102    BA 0C 01      mov dx, offset mesaj
0105    CD 21         int 21h
0107    B8 00 4C      mov ax, 4C00h
010A    CD 21         int 21h
010C    4D 65 72 ...  mesaj db 'Merhaba!$'
```

Liste satır satır okunabilir. `mov ah, 09h` iki bayttır: `B4` “AH'ye sabit yükle”, `09` sabittir. `mov dx, offset mesaj` üç bayttır: `BA` “DX'e 16 bitlik sabit yükle” anlamına gelir ve sabit, `mesaj` etiketinin adresi olan 010Ch'dir; little-endian olarak `0C 01` yazılır. `int 21h` iki bayttır: `CD` kesme komutu, `21` kesme numarasıdır. `mov ax, 4C00h` AX'e 4C00h yükler, yani AH = 4Ch ve AL = 00h olur. Son satırdaki `db` (*define byte*) direktifi komut değil, veri üretir: “Merhaba!$” karakterlerinin ASCII kodları bellekte art arda durur. Liste çıktısı hem assembler'ın ne ürettiğini denetlemek hem de bir hata ayıklayıcıda adresleri kaynak satırlarla eşleştirmek için kullanılır. emu8086 bu adımların hepsini tek tuşla yapar ve COM ya da EXE dosyasını doğrudan üretir.

3.1.3.3 Assembler'ın iki geçişi

Yukarıdaki listede ilginç bir durum vardır. `mov dx, offset mesaj` satırı ikinci satırdadır, ama `mesaj` etiketi en sonda tanımlanır. Assembler bu satıra geldiğinde `mesaj`'ın adresini henüz bilmez. Bir etikete tanımından önce başvurulmasına **ileri başvuru** (*forward reference*) denir. Assembler'lar bu sorunu kaynak kodu iki kez okuyarak çözer.

> **Tanım 3.1.3**
>
> **İki geçişli assembler**, kaynak kodu iki kez baştan sona okur. Birinci geçişte her satırın kaç bayt tuttuğunu hesaplayıp **konum sayacını** (*location counter*, LC) ilerletir. Her etiketi de o anki LC değeriyle **sembol tablosuna** (*symbol table*) yazar. İkinci geçişte sembol tablosundaki adresleri kullanarak makine kodunu üretir.

Konum sayacı, sıradaki baytın bellekte hangi adrese yerleşeceğini tutar. Kaynakta `$` işaretiyle okunabilir. Birinci geçiş için önemli bir koşul vardır: her komutun uzunluğu, işlenenin değeri bilinmeden hesaplanabilmelidir. `mov dx, offset mesaj` komutunun 3 bayt olduğu, `mesaj`'ın değeri ne olursa olsun bellidir. İzlenen program şudur:

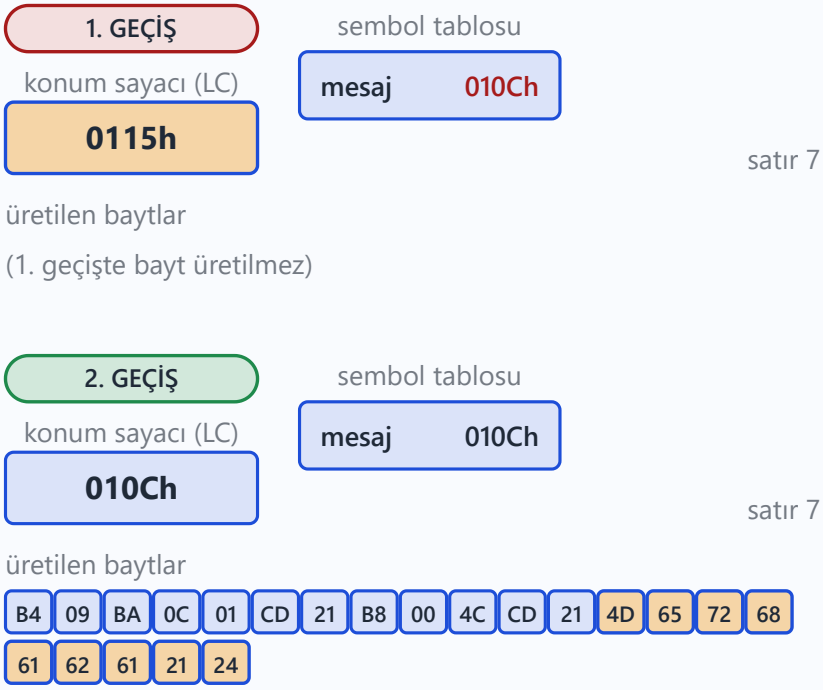
```
org 100h
mov ah, 09h
mov dx, offset mesaj
int 21h
mov ax, 4C00h
int 21h
mesaj db 'Merhaba!$'
```

Geçiş	Satır	Uzunluk	LC (satırdan sonra)	Sembol tablosu	Üretilen baytlar
1	`org 100h`	0	0100h	mesaj = ?	—
1	`mov ah, 09h`	2	0102h	mesaj = ?	—
1	`mov dx, offset mesaj`	3	0105h	mesaj = ?	—
1	`int 21h`, `mov ax, 4C00h`, `int 21h`	2 + 3 + 2	010Ch	mesaj = ?	—
1	`mesaj db 'Merhaba!$'`	9	0115h	**mesaj = 010Ch**	—
2	`mov ah, 09h`	2	0102h	mesaj = 010Ch	`B4 09`
2	`mov dx, offset mesaj`	3	0105h	mesaj = 010Ch	`BA 0C 01`
2	üç satır	7	010Ch	mesaj = 010Ch	`CD 21 B8 00 4C CD 21`
2	`mesaj db …`	9	0115h	mesaj = 010Ch	`4D 65 72 68 61 62 61 21 24`

Tablo 3.1.5: İki geçişli assembler'ın izlemesi.

Birinci geçiş yalnız sayar. `org 100h` LC'yi 0100h yapar; direktif olduğu için bayt üretmez. `mov ah, 09h` 2 bayt tutar ve LC 0102h olur. Üçüncü satırda `mesaj` henüz tanımlanmamıştır, ama komutun 3 bayt olduğu bilinir ve LC 0105h'ye ilerler. Sonraki üç komut 2, 3 ve 2 bayttır; LC 010Ch'ye gelir. Son satırda `mesaj` etiketi görülür ve o anki LC ile sembol tablosuna girer: `mesaj` = 010Ch. Dizgi 9 karakterdir (“Merhaba!” sekiz karakter ve sondaki `$`), LC 0115h olur.

İkinci geçiş LC'yi yeniden 0100h'den başlatır ve bu kez baytları üretir. `mov ah, 09h` için `B4 09` yazılır. `offset mesaj` artık sembol tablosundan okunur: 010Ch, little-endian olarak `0C 01`. Kalan komutlar `CD 21`, `B8 00 4C` ve `CD 21`'dir. En sonda dizginin ASCII kodları yazılır: M = 4Dh, e = 65h, r = 72h, h = 68h, a = 61h, b = 62h, a = 61h, ! = 21h, $ = 24h. Çıktı, 0115h − 0100h = 15h = 21 baytlık bir COM dosyasıdır. Çalıştırılınca ekrana “Merhaba!” yazar. Şekil 3.1.9 iki geçişin sonundaki durumu gösterir.



Şekil 3.1.9: Birinci geçişin sonunda sembol tablosu dolmuş, ama hiç bayt üretilmemiştir (üstte). İkinci geçişin sonunda 21 baytın hepsi üretilmiştir (altta).

İki geçişli yapı, makine dilindeki adres kayması sorununu da kendiliğinden çözer. Programcı araya bir komut eklerse assembler bir sonraki çevirmede bütün etiketlerin adresini yeniden hesaplar ve atlama komutlarına yeni adresleri yazar.

Sık yapılan hata

Direktifleri komut sanmak. `org`, `db`, `equ` ve `offset` işlemcinin komutları değildir; assembler'a verilen bilgilerdir. `org 100h` bayt üretmez, yalnız konum sayacını ayarlar. `db` komut değil veri üretir. İşlemci bir `db` satırının baytlarına komut gibi ulaşırsa onları da komut diye çözmeye çalışır; bu yüzden veri kodun arkasına konur.

3.1.3.4 Assembler'lar ve söz dizimleri

Aynı x86 işlemcisi için birden çok assembler vardır ve hepsi aynı kuralları kullanmaz. Bir assembly satırının nasıl yazılacağını belirleyen kurallara **söz dizimi** (*syntax*) denir. x86 için iki ana söz dizimi kullanılır: Intel ve AT&T. Tablo 3.1.6 yaygın assembler'ları özetler.

Assembler	Söz dizimi	Kullanım
MASM	Intel	Microsoft, Windows
TASM	Intel	Borland, DOS
NASM	Intel	açık kaynak, çok platformlu

Assembler	Söz dizimi	Kullanım
GAS (as)	AT&T	GNU/Linux, gcc'nin kullandığı assembler
emu8086	Intel (MASM benzeri)	8086 öykünücüsü; bu dersin aracı

Tablo 3.1.6: Yaygın x86 assembler'ları.

İki söz dizimindeki en önemli fark işlenenlerin sırasıdır. Intel söz diziminde önce hedef, sonra kaynak yazılır. AT&T söz diziminde sıra terstir: önce kaynak, sonra hedef. Aynı iki komut iki söz diziminde şöyle görünür:

```
; Intel:  hedef, kaynak
mov eax, 5
add eax, [ebx+4]

# AT&T:  kaynak, hedef
movl $5, %eax
addl 4(%ebx), %eax
```

AT&T söz diziminde yazmaç adlarının önüne `%`, sabitlerin önüne `$` konur. İşlemin boyutu komutun sonundaki harfle belirtilir: `b` 8, `w` 16, `l` 32 bit demektir. Bellek işleneni köşeli parantez yerine `uzaklık(taban)` biçiminde yazılır: `4(%ebx)`, Intel'deki `[ebx+4]` ile aynı adrestir. Açıklama işareti de farklıdır: Intel söz dizimli assembler'lar `;`, GNU as ise x86'da `#` kullanır. Her iki yazım da işlemciye aynı baytları verir. Fark yalnız insanın okuduğu metindedir.

> **Sık yapılan hata**
>
> Bir derleyicinin ürettiği AT&T kodunu Intel kuralıyla okumak. `subl %ebx, %eax` komutu EBX'ten EAX'i çıkarmaz; EAX ← EAX − EBX yapar, yani Intel'deki `sub eax, ebx`'tir. gcc ve clang varsayılan olarak AT&T söz dizimi üretir; `-masm=intel` seçeneği çıktıyı Intel söz dizimine çevirir.

3.1.4 Farklı ortamlarda “Merhaba”

Ekrana kısa bir metin yazan program, bir dilin ya da ortamın en küçük tam örneğidir. Bu kesimde aynı iş altı ortamda yapılır: 32 bitlik Linux, DOS, Windows, Motorola 68000 işlemcili Amiga, ARM ve PowerPC. Programların hepsi aynı üç adımı izler: metni bellekte tanımla, işletim sisteminden “şu adresteki şu kadar baytı ekrana yaz” hizmetini iste, programı sonlandır. Farklar, bu üç adımın her ortamda nasıl söylendiğindedir.

3.1.4.1 Linux, x86 (32 bit)

İlk program 32 bitlik x86 işlemcide, Linux işletim sistemi altında çalışır ve NASM ile çevrilir. 8086'nın 16 bitlik AX, BX, CX, DX yazmaçları 80386'dan itibaren 32 bite genişletilmiştir; 32 bitlik adları EAX, EBX, ECX, EDX'tir. AX, EAX'in alt yarısıdır.

```
section .data
hello  db 'Merhaba, Linux!', 10
len    equ $ - hello     ; = 16 bayt
section .text
global _start
_start:
  mov eax, 4     ; sys_write
  mov ebx, 1     ; stdout
  mov ecx, hello ; dizginin adresi
  mov edx, len   ; bayt sayısı
  int 0x80       ; çekirdeği çağır
  mov eax, 1     ; sys_exit
  xor ebx, ebx   ; çıkış kodu 0
  int 0x80
```

Program iki bölümden oluşur. `section .data` verinin, `section .text` kodun bulunduğu bölümü açar. `hello` etiketi metnin başlangıç adresini gösterir; `db` metnin baytlarını ve sonuna 10'u, yani satır sonu karakterini koyar. `len equ $ - hello` satırı bellekte yer kaplamaz. `equ`, bir ada sabit bir değer verir; `$` o andaki konum sayacıdır. Konum sayacından metnin başlangıç adresi çıkarılınca metnin uzunluğu bulunur. “Merhaba, Linux!” 15 bayttır; satır sonuyla 16 bayt olur. Uzunluğu elle saymak yerine assembler'a hesaplatmak, metin değiştiğinde hatayı önler.

`global _start`, `_start` etiketini bağlayıcıya görünür kılar. Linux'ta bağlayıcı programın giriş noktasını bu adla arar. Kodun kendisi iki sistem çağrısıdır. Tablo 3.1.7 her satırdan sonra yazmaçları verir.

Satır	EAX	EBX	ECX	EDX	Ekran
`mov eax, 4`	00000004	????????	????????	????????	
`mov ebx, 1`	00000004	00000001	????????	????????	
`mov ecx, hello`	00000004	00000001	hello (adres)	????????	
`mov edx, len`	00000004	00000001	hello (adres)	00000010	
`int 0x80`	00000010	00000001	hello (adres)	00000010	Merhaba, Linux!
`mov eax, 1`	00000001	00000001	hello (adres)	00000010	Merhaba, Linux!
`xor ebx, ebx`	00000001	00000000	hello (adres)	00000010	Merhaba, Linux!
`int 0x80`	süreç 0 çıkış koduyla sonlanır; çağrı geri dönmez				

Tablo 3.1.7: Linux “Merhaba” programının izlemesi.

Birinci çağrının parametreleri yazmaçlara konur. EAX = 4, Linux'un 32 bitlik çağrı tablosunda `sys_write` hizmetinin numarasıdır. EBX = 1, yazılacak dosya tanıtıcısıdır; 1 numaralı tanıtıcı standart çıktıdır, yani ekran. ECX metnin **adresini** alır, içeriğini değil. EDX yazılacak bayt sayısıdır: 16, onaltılık 10h. `int 0x80` komutu denetimi çekirdeğe verir. Çekirdek metni ekrana yazar ve dönüşte EAX'e yazılan bayt sayısını koyar; EAX bu yüzden 00000010h olur.

İkinci çağrı programı sonlandırır. EAX = 1, `sys_exit`'tir. EBX çıkış kodunu taşır ve 0 başarı demektir. `xor ebx, ebx` bir sayıyı kendisiyle dışlayıcı VEYA işlemine sokar. Her bit kendisiyle aynı olduğundan sonuç sıfırdır. Bu yol `mov ebx, 0`'dan daha kısa bir makine kodu üretir (2 bayta karşı 5 bayt). XOR bayrakları da etkiler: sonuç sıfır olduğu için ZF = 1, mantıksal işlem olduğu için CF = 0 ve OF = 0 olur. Son `int 0x80` süreci sonlandırır ve geri dönmez.

Sık yapılan hata

Uzunluğu elle yazıp sonra metni değiştirmek. `len` 15 yazılsaydı satır sonu yazılmaz ve sonraki çıktı aynı satıra eklenirdi. Metin kısaltılıp uzunluk 16 bırakılsaydı, metnin arkasındaki bellek baytları da ekrana basılırdı. `$ - hello` bu hataların ikisini de önler. Türkçe karakterlerde ayrıca dikkat gerekir: UTF-8'de “ü” gibi harfler 2 bayt tutar; “Merhaba, Dünya!” 15 karakter ama 16 bayttır.

3.1.4.2 Sistem çağrısı

Linux programı ekrana kendisi yazmadı; çekirdekten yazmasını istedi. Modern işletim sistemlerinde uygulama programları donanıma doğrudan erişemez. İşlemcinin koruma düzeneği iki çalışma modu tanımlar. **Kullanıcı modunda** çalışan bir program giriş/çıkış komutlarını ve bazı denetim komutlarını kullanamaz. **Çekirdek modu** ise ayrıcalıklıdır ve yalnız işletim sistemine açıktır. Uygulama bir donanım hizmetine ihtiyaç duyduğunda kontrollü bir kapıdan çekirdeğe geçer.

Tanım 3.1.4

Sistem çağrısı (*system call*), bir programın işletim sisteminden hizmet istediği özel çağrıdır. Program hizmet numarasını ve parametreleri belirli yazmaçlara koyar, ardından yazılım kesmesi ya da özel bir komutla denetimi çekirdeğe verir. Çekirdek hizmeti yapar, sonucu bir yazmaca koyar ve denetimi programa geri verir.



Şekil 3.1.10: Sistem çağrısının yolu. Kullanıcı programı parametreleri yazmaçlara koyar ve `int 0x80` ile çekirdeğe geçer. Çekirdek EAX'teki numaraya göre hizmeti seçer, sürücü üzerinden ekrana yazar ve sonucu EAX'te döndürür.

Şekil 3.1.10 bu yolu gösterir. `int 0x80` bir yazılım kesmesidir. İşlemci bayrakları ve dönüş adresini kaydeder, kesme tablosunun 80h numaralı girişinde yazan adrese atlar ve aynı anda çekirdek moduna geçer. Çekirdekteki giriş yordamı EAX'e bakar ve hizmet tablosundan ilgili fonksiyonu çağırır. Hizmet bitince özel bir dönüş komutu hem kayıtlı adrese döner hem de kullanıcı moduna geri geçer.

8086'da koruma düzeneği ve kullanıcı/çekirdek modu ayrımı yoktur; her program her şeye erişebilir. Yine de kesme düzeneği aynıdır. 8086 `INT n` komutunu yürütürken bayrak yazmacını, CS'yi ve IP'yi yığına iter, IF ve TF bayraklarını sıfırlar. Yeni CS:IP'yi belleğin en başındaki kesme vektör tablosundan okur. Her giriş 4 bayttır; `INT 21h` için giriş 21h × 4 = 84h adresindedir. DOS, kendi hizmet yordamının adresini bu girişe yazar. Tablo 3.1.8 farklı ortamlardaki çağrı yollarını karşılaştırır.

Ortam	Çağrı	Hizmet numarası	Parametreler
Linux 32 bit	`int 0x80`	EAX (4 = write, 1 = exit)	EBX, ECX, EDX
Linux 64 bit	`syscall`	RAX (1 = write, 60 = exit)	RDI, RSI, RDX
MS-DOS	`int 21h`	AH (09h = yaz, 4Ch = çık)	DX, AL …
BIOS	`int 10h`	AH (0Eh = karakter yaz)	AL, BH …

Tablo 3.1.8: Ortamlara göre sistem çağrısı.

Tablo, numaraların ortama özgü olduğunu açıkça gösterir. “Yaz” hizmeti 32 bitlik Linux'ta 4, 64 bitlik Linux'ta 1'dir. Aynı işletim sisteminin iki sürümü bile farklı numara ve farklı yazmaç kullanır. BIOS hizmetleri işletim sisteminden de alttadır. Bilgisayarın kalıcı belleğindeki bu yordamlar, işletim sistemi yüklenmeden önce de çalışır.

3.1.4.3 DOS, 8086 (COM programı)

Üçüncü program bu dersin ortamındadır: emu8086 ile çevrilen, DOS altında çalışan bir COM programı.

```
org 100h          ; COM programı
mov ah, 09h       ; DOS 09h: dizgi yaz
lea dx, mesaj     ; DS:DX = adres
int 21h           ; DOS hizmetini çağır
mov ax, 4C00h     ; DOS 4Ch: sonlandır
int 21h
mesaj db 'Merhaba, DOS!$'
```

`org 100h`, COM programının kesimin 100h adresine yükleneceğini bildirir. DOS, programı yüklerken kesimin ilk 256 baytına (0000h–00FFh) **program kesimi öneki** (PSP, *program segment prefix*) adlı bir yapı koyar. Program bu yapının hemen arkasından başlar. COM programında dört kesim yazmacının hepsi aynı kesimi gösterir: CS = DS = ES = SS. Kod, veri ve yığın aynı 64 KB içindedir.

`mov ah, 09h`, DOS'un 09h numaralı hizmetini seçer: "DS:DX adresinden başlayan ve `$` ile biten dizgiyi ekrana yaz". `lea dx, mesaj`, `mesaj` etiketinin bağıl konumunu DX'e yükler (*load effective address*). DS zaten programın kesimini gösterdiği için DS:DX dizginin tam adresidir. `int 21h` DOS'u çağırır. DOS karakterleri `$` görene kadar ekrana basar; `$`'ın kendisi yazılmaz. `mov ax, 4C00h` AH'ye 4Ch'yi, yani "programı sonlandır" hizmetini, AL'ye 00h çıkış kodunu yükler. Son `int 21h` denetimi DOS'a geri verir.

Satır	Makine kodu	IP (sonra)	AH	AL	DX	Ekran
`org 100h`	—	0100h	??	??	????	
`mov ah, 09h`	`B4 09`	0102h	09h	??	????	
`lea dx, mesaj`	`BA 0C 01`	0105h	09h	??	010Ch	
`int 21h`	`CD 21`	0107h	09h	??	010Ch	Merhaba, DOS!
`mov ax, 4C00h`	`B8 00 4C`	010Ah	4Ch	00h	010Ch	Merhaba, DOS!
`int 21h`	`CD 21`	010Ch	4Ch	00h	010Ch	program biter

Tablo 3.1.9: DOS "Merhaba" programının izlemesi.

Tablo 3.1.9'daki adresler komut uzunluklarından gelir. `lea dx, mesaj` üç bayttır: emu8086 doğrudan adresli LEA'yı `mov dx, offset mesaj` ile aynı biçimde kodlar, yani `BA` işlem kodu ve `0C 01` adresi (MASM aynı komutu `8D 16` ile başlayan dört baytlık gerçek LEA olarak kodlar). Kod 0100h–010Bh aralığında 2 + 3 + 2 + 3 + 2 = 12 bayt tutar; bu yüzden `mesaj` 010Ch'dedir. "Merhaba, DOS!" 13 karakterdir ve `$` ile birlikte 14 bayt yer kaplar.

İkinci `int 21h` satırında IP'nin 010Ch'yi göstermesi dikkat çekicidir: bu adres `mesaj`'ın başıdır. Program sonlandığı için işlemci bu adrese hiç ulaşmaz. Veri **kodun arkasına** konduğu için işlemci veriyi komut sanıp çalıştırmaz. Veri kodun önüne konsaydı, program 0100h'den başlarken "M" harfinin kodu olan 4Dh'yi bir komut olarak çözerdi. 4Dh, 8086'da `DEC BP` komutudur. Arkadan gelen harfler de rastgele komutlara dönüşür ve program öngörülemez biçimde davranırdı.

Sık yapılan hata

09h hizmetinde dizginin sonundaki `$`'ı unutmak. DOS, dizginin nerede bittiğini başka bir yoldan bilemez. `$` yoksa bellekte ilk `24h` baytına rastlayana kadar kodun ve verinin arkasındaki ne varsa ekrana basar. Ekranda anlamsız karakterler, bip sesleri ve imleç sıçramaları görülür.

3.1.4.4 Windows, Win32 API

Windows programları sistem çağrısını doğrudan yapmaz. İşletim sisteminin hizmetleri **DLL** (*dynamic-link library*, devingen bağlantılı kütüphane) dosyalarındaki fonksiyonlar olarak sunulur. Program bu fonksiyonları sıradan bir alt program gibi `call` komutuyla çağırır. Aşağıdaki program ekranda bir ileti kutusu açar ve sonra süreci sonlandırır.

```
extern MessageBoxA    ; user32.dll
extern ExitProcess    ; kernel32.dll

section .data
mesaj db 'Merhaba, Windows!', 0

section .text
global _start
_start:
    push 0          ; uType = MB_OK
    push mesaj      ; lpCaption (başlık)
    push mesaj      ; lpText (metin)
    push 0          ; hWnd = NULL
    call MessageBoxA
    push 0          ; uExitCode = 0
    call ExitProcess
```

`extern` satırları, `MessageBoxA` ve `ExitProcess` adlarının başka bir yerde tanımlı olduğunu söyler. Assembler bu adların adresini bilmez; nesne dosyasına "burayı bağlayıcı doldursun" notu düşer. Bağlayıcı, `user32.dll` ve `kernel32.dll` kütüphanelerinin içe aktarma bilgisini kullanarak çağrıları bu fonksiyonlara yönlendirir. Dizgi bu kez `$` ile değil 0 ile biter. Windows API'si C diliyle tasarlanmıştır ve C dizgileri sıfır baytıyla sonlanır.

Parametreler yazmaçlarla değil yığınla geçirilir. C'de çağrı `MessageBoxA(hWnd, lpText, lpCaption, uType)` biçimindedir. Assembly'de parametreler **sağdan sola** itilir: önce `uType`, en son `hWnd`. Böylece ilk parametre yığının tepesine, dönüş adresinin hemen üstüne gelir ve fonksiyon onu parametre sayısından bağımsız olarak hep aynı yerde bulur. Bu programda başlık ve metin için aynı dizgi kullanılmıştır; iki `push mesaj` aynı adresi iter. Tablo 3.1.10 yığının durumunu izler; ESP'nin başlangıç değeri olarak 0012FF80h örnek alınmıştır.

Komut	ESP (sonra)	Yazılan adres ← değer	EAX
başlangıç	0012FF80h	—	????????
`push 0` (uType)	0012FF7Ch	0012FF7Ch ← 0 (MB_OK)	????????
`push mesaj` (lpCaption)	0012FF78h	0012FF78h ← &mesaj	????????
`push mesaj` (lpText)	0012FF74h	0012FF74h ← &mesaj	????????
`push 0` (hWnd)	0012FF70h	0012FF70h ← 0 (NULL)	????????
`call MessageBoxA`	0012FF6Ch	0012FF6Ch ← dönüş adresi	????????
fonksiyon `ret 16` ile döner	0012FF80h	—	00000001h (IDOK)
`push 0` (uExitCode)	0012FF7Ch	0012FF7Ch ← 0	00000001h
`call ExitProcess`	süreç 0 çıkış koduyla sonlanır; çağrı geri dönmez		

Tablo 3.1.10: Windows programında yığının izlemesi.

Yığın düşük adreslere doğru büyür. 32 bitlik kipte her `push` ESP'yi 4 azaltır ve değeri ESP'nin yeni gösterdiği adrese yazar. Dört parametre 16 bayt tutar. `call` da dönüş adresini iter; fonksiyona girildiği anda ESP = 0012FF80h − 20 = 0012FF6Ch'dir. Şekil 3.1.11 bu anı gösterir.



Şekil 3.1.11: `call MessageBoxA` yürütüldükten hemen sonra yığın. En üstte dönüş adresi, onun üstünde ilk parametre `hWnd` vardır.

Kullanıcı ileti kutusundaki Tamam düğmesine basınca fonksiyon döner. Dönüş değeri EAX'e konur: 1, yani IDOK, "Tamam'a basıldı" demektir. Windows API fonksiyonları **stdcall** çağrı kuralını kullanır. Bu kuralda parametreleri yığından **çağrılan** fonksiyon temizler. `MessageBoxA`, `ret 16` komutuyla döner: önce dönüş adresini alır (ESP + 4), sonra 16 bayt parametreyi atar (ESP + 16). ESP yeniden 0012FF80h olur. C'nin kendi fonksiyonlarında kullanılan cdecl kuralında ise bu temizliği çağıran yapar.

Bağlarken bir ayrıntıyla daha karşılaşılabilir. Bazı 32 bitlik araç zincirlerinde stdcall fonksiyonlarının adları **süslenmiş** biçimde aranır: `_MessageBoxA@16`. Baştaki alt çizgi C adlandırma geleneğinden, sondaki 16 parametrelerin bayt sayısından gelir. 8086 düzeyinde aynı yığın düzeneği Yığın, CALL ve RET simülatöründe adım adım izlenebilir; tek fark 16 bitlik kipte her itmenin 2 bayt tutmasıdır.

3.1.4.5 Motorola 68000, Amiga

Amiga bilgisayarları Motorola 68000 işlemcisini kullanır. 68000, 8086 ile aynı dönemin (1979) işlemcisidir, ama mimarisi farklıdır. Sekiz veri yazmacı (`d0` – `d7`) ve sekiz adres yazmacı (`a0` – `a7`) vardır; hepsi 32 bittir. Bellek kesimlere bölünmez, adresler doğrusaldır.

```
    move.l  4.w,a6          ; a6 = ExecBase
    lea     dosname(pc),a1  ; kütüphane adı
    moveq   #0,d0           ; herhangi bir sürüm
    jsr     -552(a6)        ; OpenLibrary()
    tst.l   d0
    beq.s   cik             ; açılamadı → çık
    move.l  d0,a6           ; a6 = DOSBase
    jsr     -60(a6)         ; Output() → d0
    move.l  d0,d1           ; dosya tanıtıcısı
    move.l  #mesaj,d2       ; dizginin adresi
    moveq   #16,d3          ; bayt sayısı
    jsr     -48(a6)         ; Write(d1,d2,d3)
    move.l  a6,a1
    move.l  4.w,a6
    jsr     -414(a6)        ; CloseLibrary()
cik moveq   #0,d0           ; dönüş kodu 0
    rts
dosname dc.b 'dos.library',0
mesaj   dc.b 'Merhaba, Amiga!',10
```

AmigaOS'ta sistem hizmetleri kütüphaneler hâlinde sunulur ve her kütüphanenin bir **taban adresi** vardır. Fonksiyonlar bu tabana göre negatif uzaklıklarda duran bir atlama tablosundan çağrılır. Bellekteki 4 numaralı adreste, bütün sistemin çekirdek kütüphanesi olan `exec.library` 'nin taban adresi bulunur. İlk satır bu adresi `a6` 'ya yükler. `lea dosname(pc),a1` kütüphane adının adresini `a1` 'e koyar. `moveq #0,d0` "herhangi bir sürüm olur" demektir. `jsr -552(a6)` , `a6` 'nın 552 bayt altındaki girişe atlar; bu giriş `OpenLibrary` 'dir.

Dönüşte `d0` açılan `dos.library` 'nin tabanını taşır. `tst.l d0` ve `beq.s cik` , kütüphane açılamadıysa (`d0` = 0) programdan çıkar. Açıldıysa taban `a6` 'ya alınır. `jsr -60(a6)` , `Output()` fonksiyonudur ve standart çıktının tanıtıcısını `d0` 'da döndürür. `Write(d1, d2, d3)` çağrısının parametreleri yazmaçlarla verilir: tanıtıcı `d1` 'e, metnin adresi `d2` 'ye, uzunluk 16 `d3` 'e konur. "Merhaba, Amiga!" 15 bayttır; satır sonuyla 16 eder. `jsr -48(a6)` yazmayı yapar. Son olarak kütüphane `CloseLibrary` ile kapatılır (`-414` uzaklığı) ve `d0` = 0 dönüş koduyla `rts` çalıştırılır.

68000 söz diziminde işlenen sırası AT&T gibidir: önce kaynak, sonra hedef. Boyut, komutun sonundaki `.b` , `.w` ya da `.l` sonekiyle belirtilir. Sabitler `#` ile yazılır. Kütüphane önce açılır, kullanılır, sonra kapatılır. İşletim sisteminin kaynakları bu düzende yönetilir.

3.1.4.6 ARM

ARM işlemcileri bugün akıllı telefonlarda, gömülü sistemlerde ve birçok dizüstü bilgisayarda kullanılır. ARM bir RISC (*reduced instruction set computer*, indirgenmiş komut kümeli bilgisayar) mimarisidir: komutlar az sayıda, basit ve sabit uzunluktadır.

```
    .data
mesaj:  .ascii "Merhaba, ARM!\n"
uzun = . - mesaj       @ 14 bayt

    .text
    .global _start
_start:
    mov  r7, #4        @ sys_write
    mov  r0, #1        @ stdout
    ldr  r1, =mesaj    @ dizginin adresi
    mov  r2, #uzun     @ bayt sayısı
    svc  #0            @ çekirdeği çağır
    mov  r7, #1        @ sys_exit
    mov  r0, #0        @ çıkış kodu
    svc  #0
```

Program GNU as ile çevrilir. GNU as'ta veri `db` ile değil `.ascii` direktifiyle tanımlanır; `\n` satır sonudur. `uzun = . - mesaj` satırındaki nokta konum sayacıdır, NASM'deki `$`'ın karşılığıdır. “Merhaba, ARM!” 13 bayttır; satır sonuyla 14 eder. Açıklama işareti ARM için GNU as'ta `@`'dir.

32 bitlik ARM Linux'ta hizmet numarası `r7`'ye, parametreler `r0`, `r1`, `r2`'ye konur. Yazma hizmeti burada da 4, çıkış 1'dir; numaralar 32 bitlik x86 Linux ile aynı olsa da yazmaçlar farklıdır. `ldr r1, =mesaj` özel bir yazımdır. ARM komutları 32 bit olduğundan 32 bitlik bir adres tek bir komutun içine sığmaz. Assembler adresi kodun yakınındaki bir sabit havuzuna yazar ve `ldr` komutunu oradan okuyacak biçimde üretir. `svc #0` (eski adı `swi`) denetimi çekirdeğe verir. ARM'da bellekle yalnız `ldr` (oku) ve `str` (yaz) komutları çalışır. Toplama gibi işlemler yalnız yazmaçlar üzerinde yapılır. 8086'daki `add ax, [0202h]` gibi bellekten doğrudan toplayan bir komut ARM'da yoktur.

3.1.4.7 PowerPC

PowerPC, IBM, Motorola ve Apple'ın ortak geliştirdiği bir RISC mimarisidir. Eski Macintosh bilgisayarlarında, oyun konsollarında ve otomotiv denetleyicilerinde kullanılmıştır.

```
    .data
mesaj:  .ascii "Merhaba, PowerPC!\n"
uzun = . - mesaj        # 18 bayt

    .text
    .globl _start
_start:
    li   0, 4           # r0 = sys_write
    li   3, 1           # r3 = stdout
    lis  4, mesaj@ha    # r4 = adres (üst)
    addi 4, 4, mesaj@l  #    + alt yarı
    li   5, uzun        # r5 = bayt sayısı
    sc                  # çekirdeği çağır
    li   0, 1           # r0 = sys_exit
    li   3, 0           # r3 = çıkış kodu
    sc
```

PowerPC'de 32 genel amaçlı yazmaç vardır. GNU as'ta bu yazmaçlar düz sayılarla yazılır: `li 3, 1` komutundaki `3`, r3 yazmacıdır. `li` (*load immediate*) yazmaca sabit yükler. Hizmet numarası `r0`'a, parametreler `r3`, `r4`, `r5`'e konur; çekirdeğe geçiş `sc` komutuyla yapılır. Açıklama işareti `#`'dir. "Merhaba, PowerPC!" 17 bayttır; satır sonuyla 18 eder.

Adres yükleme burada da iki adımdır. PowerPC komutları da 32 bittir ve sabit alanı 16 bit tutar. 32 bitlik adres iki 16 bitlik parçaya bölünür. `lis 4, mesaj@ha` üst yarıyı yazmacın üst 16 bitine yükler; `addi 4, 4, mesaj@l` alt yarıyı ekler. `@ha`, "ayarlanmış üst yarı" demektir. `addi` alt yarıyı işaretli bir sayı olarak eklediği için, alt yarının en yüksek biti 1 ise üst yarı önceden 1 artırılır. Bu ayrıntı assembler'a bırakılmıştır.

3.1.4.8 Altı ortamın karşılaştırması

Tablo 3.1.11 altı programı yan yana koyar.

Ortam	İşlemci	İşletim sistemine geçiş	Hizmet no.	Parametreler	Dizgi sonu
Linux	x86 (32)	`int 0x80`	EAX	EBX, ECX, EDX	uzunluk verilir
DOS	8086	`int 21h`	AH	DX (adres)	`'$'`
Windows	x86 (32)	`call` ile DLL	fonksiyon adı	yığın	`0`
AmigaOS	68000	`jsr -48(a6)`	kütüphane uzaklığı	d1, d2, d3	uzunluk verilir
Linux	ARM	`svc #0`	r7	r0, r1, r2	uzunluk verilir
Linux	PowerPC	`sc`	r0	r3, r4, r5	uzunluk verilir

Tablo 3.1.11: Aynı "Merhaba", altı ortam.

Tablodan çıkan sonuç şudur: assembly kodu iki şeye birden bağlıdır. Birincisi **işlemcidir**: komut kümesi, yazmaç adları, adres yükleme yöntemi ve söz dizimi işlemciyle değişir. Açıklama işareti bile değişir: x86'da `;`, ARM için GNU as'ta `@`, PowerPC'de `#`. İkincisi **işletim sistemidir**: aynı x86 işlemci-

sinde Linux, DOS ve Windows bambaşka çağrı yolları, numaralar ve dizgi kuralları ister. Bir programı yeni bir ortama taşımak, assembly düzeyinde onu yeniden yazmak demektir. Yüksek düzeyli dillerin en büyük kazancı bu bağımlılığı derleyiciye ve yorumlayıcıya devretmesidir. C'deki `printf("Merhaba\n")` satırı altı ortamın hepsinde aynıdır; farkı, her ortamın C kütüphanesi ve derleyicisi karşılar.

3.1.5 Yüksek düzeyli diller

3.1.5.1 Derlenen, yorumlanan, hibrit

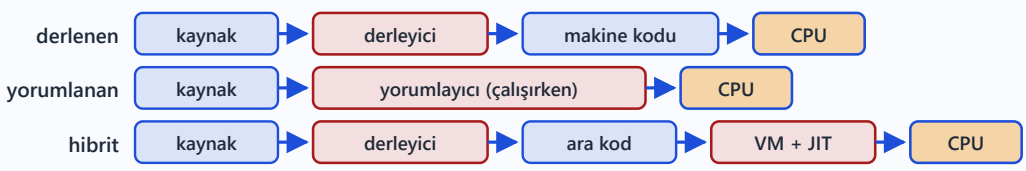
Yüksek düzeyli bir dilde yazılmış program da sonunda işlemcinin makine diliyle çalışmak zorundadır. Diller, bu çeviriyi kimin ve ne zaman yaptığına göre ayrılır.

Tanım 3.1.5

Derleyici (*compiler*), yüksek düzeyli kaynak kodu bütün olarak inceleyip program çalışmadan önce makine koduna çeviren programdır. **Yorumlayıcı** (*interpreter*), kaynak kodu çalışma anında ifade ifade okuyup işleyen programdır; ayrı bir çalıştırılabilir dosya üretmez. **Tam zamanında derleme** (JIT, *just-in-time compilation*), programın parçalarını çalışma sırasında, ihtiyaç duyuldukça makine koduna çevirmektir.

Derlenen dillerde kaynak kod önceden makine diline çevrilir. Derleyici bütün kaynağı gördüğü için tip ve söz dizimi hatalarının çoğu derleme sırasında yakalanır. Çalışma anında çeviri yükü olmadığından program hızlı çalışır. Derleme iki aşamalıdır: derleyici her kaynak dosyayı bir nesne dosyasına çevirir, bağlayıcı da nesne dosyalarını kütüphanelerle birleştirir. Çıktı belirli bir işlemciye ve belirli bir işletim sistemine özgüdür. Windows için derlenmiş bir x86 programı Linux'ta ya da bir ARM işlemcide çalışmaz.

Yorumlanan dillerde kaynak kodun kendisi dağıtılır ve çalışma anında yorumlayıcıda işlenir. Derleme adımı olmadığı için bir değişiklik hemen denenebilir; etkileşimli bir kabukta satır satır çalışmak mümkündür. Kaynak kod platformdan bağımsızdır: yorumlayıcısı olan her bilgisayarda aynı dosya çalışır. Bedeli iki yönlüdür. Bazı hatalar ancak ilgili satır çalışınca ortaya çıkar. Ayrıca her ifadenin çalışırken çözümlenmesi zaman alır ve program genellikle daha yavaştır.



Şekil 3.1.12: Üç çalışma modeli. Derlenen dilde kaynak önceden makine koduna çevrilir. Yorumlanan dilde yorumlayıcı kaynağı çalışırken işler. Hibrit modelde kaynak ara koda derlenir; sanal makine bu ara kodu çalışırken

JIT ile makine koduna çevirir.

İki modelin arasında bir de **hibrit** model vardır (Şekil 3.1.12). Java ve C# programları önce işlemciden bağımsız bir **ara koda** derlenir. Java'da bu koda bytecode, C#'ta IL (*intermediate language*) denir. Ara kodu bir **sanal makine** çalıştırır: Java için JVM, C# için CLR. Sanal makine ara kodu önce yorumlar; sık çalışan kısımları JIT ile gerçek makine koduna çevirip saklar. Böylece hem taşınabilirlik hem de yüksek hız elde edilir. Bugün Python ve JavaScript motorları da içeride bytecode ve JIT kullanır; sınırlar keskin değildir.

3.1.5.2 Derleme süreci

C dilinde yazılmış bir programın çalıştırılabilir dosyaya dönüşmesi beş aşamada olur. Örnek olarak bir sayının tek mi çift mi olduğunu yazan `tekcift.c` programı alınsın. gcc derleyicisi bütün aşamaları tek komutla çalıştırır, ama her aşamada durmak için bir seçenek de sunar.

1. **Önişlemci**: `#include` satırlarının yerine ilgili başlık dosyalarının içeriğini koyar, `#define` ile tanımlanmış adları açar. Çıktı yine C kodudur: `tekcift.i` (`gcc -E`). 15 satırlık program, `stdio.h` eklenince binlerce satıra çıkar; bu not hazırlanırken kullanılan clang tabanlı derleyicide 1.543 satır oldu.
2. **Derleyici**: C kodunu söz dizimi ve tip denetiminden geçirir, sonra **assembly** diline çevirir: `tekcift.s` (`gcc -S`). Hataların çoğu bu aşamada bildirilir.
3. **Assembler**: assembly kodunu makine koduna çevirip nesne dosyası üretir: `tekcift.o` (`gcc -c`). Bu dosyada `printf` ve `scanf` çağrılarının adresleri henüz boştur.
4. **Bağlayıcı**: nesne dosyasını C kütüphanesindeki `printf`, `scanf` gibi kodlarla birleştirir ve çalıştırılabilir dosyayı üretir: Linux'ta `tekcift`, Windows'ta `tekcift.exe`.
5. **Yükleyici**: program çalıştırıldığında işletim sistemi dosyayı belleğe yükler, yığını ve başlangıç ortamını hazırlar ve sonunda `main` fonksiyonuna atlar.

Tek komutla bütün iş `gcc tekcift.c -o tekcift` ile yapılır. Derleyicinin ürettiği ara assembly kodunu Intel söz dizimiyle görmek için `gcc -S -masm=intel tekcift.c` kullanılır. Bu çıktıyı okumak, yüksek düzeyli bir satırın hangi komutlara dönüştüğünü görmenin en doğrudan yoludur. Örneğin tek/çift programının döngüsü, x86-64 için iyileştirmeyle (`-O2`) derlendiğinde `i % 2 == 0` karşılaştırması bir bölme komutuna değil, `test bl, 1` komutuna dönüşür. Derleyici bölme yerine en düşük biti sınar, çünkü bölme komutu çok daha yavaştır. Aynı hile 8086 assembly ile elle de yazılabilir: `test cl, 1`.

Sık yapılan hata

Derleme hatasıyla bağlama hatasını karıştırmak. `printf`'in adını yanlış yazmak derleyicide yalnız bir uyarıya yol açabilir. Asıl hata bağlayıcıda "tanımsız başvuru" (*undefined reference*) olarak çıkar, çünkü o adı taşıyan bir fonksiyon hiçbir kütüphanede bulunamaz. Hata iletisinin

hangi araçtan geldiğine bakmak, sorunun hangi aşamada olduğunu gösterir.

3.1.5.3 Kod izleme: derleyicinin ürettiği döngü

Derleyicinin bir döngüyü nasıl çevirdiğini görmek için küçük bir C parçası alalım:

```
toplam = 0;
i = 10;
while (i > 0) {
    toplam += i;
    i -= 3;
}
```

Bu parçanın 8086 için üretilebilecek bir karşılığı şöyledir:

```
org 100h
mov bx, 0          ; toplam = 0
mov cx, 10         ; i = 10
kosul: cmp cx, 0   ; i > 0 ?
jle bitti
add bx, cx         ; toplam += i
sub cx, 3          ; i -= 3
jmp kosul
bitti: mov toplam, bx
ret
toplam dw ?
```

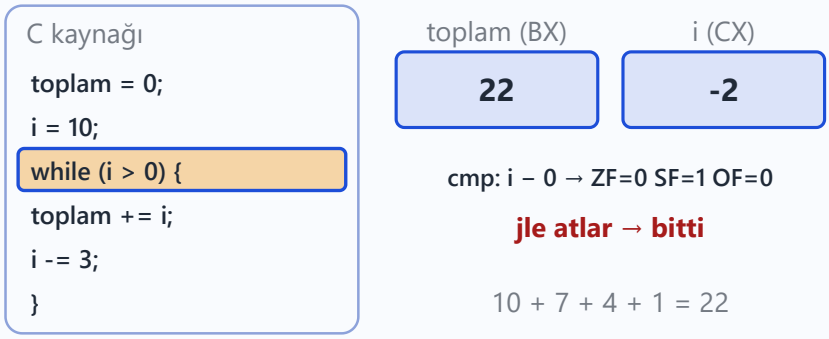
Derleyici değişkenleri yazmaçlara yerleştirmiştir: `toplam` BX'te, `i` CX'te durur. Döngünün sonunda toplam, bellekteki `toplam` değişkenine yazılır. `while` koşulu döngünün başında sınanır. Burada önemli bir dönüşüm vardır. C'deki koşul "`i > 0` ise gövdeye gir" der. Assembly'de ise koşul **tersine çevrilir**: "`i ≤ 0` ise döngüden çık". `cmp cx, 0` komutu CX − 0 çıkarmasını yapar ama sonucu saklamaz; yalnız bayrakları ayarlar. `jle bitti` (*jump if less or equal*), işaretli karşılaştırmada küçük ya da eşitse atlar. Bu koşul ZF = 1 ya da SF ≠ OF olduğunda doğrudur. Gövdeden sonra `jmp kosul` koşulsuz olarak başa döner.

Adım	Satırlar	BX (toplam)	CX (i)	ZF	SF	OF	Karar
1	`mov bx, 0`, `mov cx, 10`	0	10	–	–	–	MOV bayrakları değiştirmez
2	`cmp cx, 0`, `jle`	0	10	0	0	0	10 > 0, atlamaz
3	gövde, `jmp`	10	7	0	0	0	koşula dön
4	`cmp`, `jle`	10	7	0	0	0	atlamaz
5	gövde, `jmp`	17	4	0	0	0	koşula dön
6	`cmp`, `jle`	17	4	0	0	0	atlamaz

Adım	Satırlar	BX (toplam)	CX (i)	ZF	SF	OF	Karar
7	gövde, `jmp`	21	1	0	0	0	koşula dön
8	`cmp` , `jle`	21	1	0	0	0	atlamaz
9	gövde, `jmp`	22	−2 (FFFEh)	0	1	0	koşula dön
10	`cmp` , `jle`	22	−2 (FFFEh)	0	1	0	SF ≠ OF, **atlar**
11	`mov toplam, bx`	22	−2 (FFFEh)	0	1	0	toplam = 22

Tablo 3.1.12: `while` döngüsünün 8086 karşılığının izlemesi.

Tablo 3.1.12'deki bayraklar gövdenin son komutundan, yani `sub cx, 3` 'ten gelir; `jmp` bayrakları değiştirmez. Döngü dört tur döner ve `i` sırasıyla 10, 7, 4, 1 değerlerini alır. Toplam 10 + 7 + 4 + 1 = 22'dir. Dördüncü turun sonunda `sub cx, 3` 1 − 3 = −2 üretir. 16 bitlik ikiye tümleyen gösterimde −2, FFFEh'dir; en yüksek biti 1 olduğu için SF = 1 olur. Bu çıkarma işaretli olarak da doğrudur, yani taşma yoktur ve OF = 0'dır. Onuncu adımda `cmp cx, 0` aynı bayrakları yeniden üretir. SF = 1 ve OF = 0, SF ≠ OF demektir; işaretli olarak CX < 0'dır ve `jle` atlar. Şekil 3.1.13 bu anı C kaynağıyla yan yana gösterir.



Şekil 3.1.13: i = −2 olduğunda `cmp cx, 0` SF = 1, OF = 0 üretir ve `jle bitti` atlar. Döngüden çıkılmıştır; toplam 22'dir.

Bu örnekte koşulun işaretli bir atlamayla sınanması önemlidir. `i` bir `int` 'tir ve negatif olabilir. Derleyici işaretsiz bir atlama, örneğin `jbe` seçseydi, FFFEh değerini 65534 diye okur ve 0'dan büyük bulurdu; döngü yanlış yerde devam ederdi. C'deki değişken tipi, assembly'de hangi koşullu atlamanın seçileceğini belirler. Tek bir C döngüsü üç komut türüne dönüşmüştür: karşılaştırma, koşullu atlama ve koşulsuz atlama. Aynı C parçası bilgisayarda derlenip çalıştırıldığında da `toplam` 22 çıkar ve ara değerler (10, 7), (17, 4), (21, 1), (22, −2) aynıdır.

Sık yapılan hata

Döngü koşulunu assembly'ye çevirirken tersine çevirmeyi unutmak. C'deki `while (i > 0)` doğrudan `jg gövde` diye yazılırsa, gövdeye atlanmayan durumda akış gövdenin hemen altındaki komuta düşer. Doğru kalıp, koşul yanlışken döngünün sonuna atlayan tersine çevrilmiş bir atlama ve gövde sonunda başa dönen bir `jmp` 'tir.

3.1.6 Derlenen diller

Derlenen diller, kaynak kodu çalışmadan önce bütünüyle inceleyen bir derleyiciyle çalışır. Derlenmiş kod doğrudan işlemcide çalıştığı için hızlıdır ve tip ile söz dizimi hataları derleme aşamasında yakalanır. Her dil farklı bir ihtiyaçtan doğmuştur:

C ve C++
: Sistem programlaması, işletim sistemleri, gömülü yazılım ve yüksek performanslı uygulamalar. Mikrodenetleyici yazılımının büyük çoğunluğu C ile yazılır.

Rust
: Güvenli ve hızlı sistem programlaması; bellek güvenliği derleyicide denetlenir.

Fortran
: Bilimsel ve sayısal hesap; ilk yaygın yüksek düzeyli dil (1957).

Go
: Sunucu ve ağ yazılımı; hızlı derleme ve eşzamanlılık.

Ada
: Havacılık, savunma, demiryolu gibi güvenliğin kritik olduğu sistemler.

Java ve C#
: Hibrit: ara koda derlenir, sanal makinede JIT ile çalışır.

3.1.6.1 Ortak görev ve C

Diller arasındaki farkları adil biçimde görmek için hepsine aynı görev verilir. Program kullanıcıdan bir sayı okur ve 1'den o sayıya kadar her sayının tek mi çift mi olduğunu yazar. Görev küçüktür, ama bir programın temel parçalarının hepsini içerir: girdi, tip dönüşümü, sayaçlı döngü, kalan işlemi, koşul ve çıktı. C sürümü şöyledir:

```
#include <stdio.h>

int main() {
    int number;
    printf("Bir sayi girin: ");
    scanf("%d", &number);
    for (int i = 1; i <= number; i++) {
        if (i % 2 == 0) {
            printf("%d sayisi bir cift sayidir.\n", i);
        } else {
            printf("%d sayisi bir tek sayidir.\n", i);
        }
    }
    return 0;
}
```

`#include <stdio.h>`, standart giriş/çıkış fonksiyonlarının bildirimlerini ekler. Program `main` fonksiyonundan başlar. `int number;` bir tam sayı değişkeni ayırır; değeri henüz belirsizdir. `printf` istemi yazar. `scanf("%d", &number)` klavyeden bir ondalık tam sayı okur. `scanf`'e değişkenin kendisi değil **adresi** verilir (`&number`). Fonksiyon okuduğu değeri o adrese yazar. Bu, assembly'de DOS'a dizginin içeriğini değil adresini vermekle aynı düşüncedir.

`for (int i = 1; i <= number; i++)` üç parçalıdır: başlangıç (`i = 1`), her turdan önce sınanan koşul (`i <= number`) ve her turdan sonra yapılan adım (`i++`). `i % 2`, i'nin 2'ye bölümünden kalandır; kalan 0 ise sayı çifttir. `printf`'teki `%d` yerine i'nin değeri yazılır, `\n` satır sonudur. `return 0` işletim sistemine başarılı çıkış kodu döndürür. Tablo 3.1.13, girdi 3 iken programın izlemesidir.

Adım	Satır	number	i	i % 2	Eklenen çıktı
1	`printf`	?	–	–	Bir sayi girin:
2	`scanf`	3	–	–	(kullanıcı 3 yazar)
3	`for`: 1 ≤ 3	3	1	–	
4	`if` → `else`	3	1	1	1 sayisi bir tek sayidir.
5	`i++`; 2 ≤ 3	3	2	–	
6	`if` doğru	3	2	0	2 sayisi bir cift sayidir.
7	`i++`; 3 ≤ 3	3	3	–	
8	`if` → `else`	3	3	1	3 sayisi bir tek sayidir.
9	`i++`; 4 ≤ 3 yanlış	3	4	–	döngü biter
10	`return 0`	3	–	–	çıkış kodu 0

Tablo 3.1.13: C programının izlemesi (girdi 3).

Döngü değişkeni `i`, `for` satırında tanımlandığı için yalnız döngünün içinde geçerlidir; döngü bittikten sonra kullanılamaz. Program bu not hazırlanırken derlenip 3 ve 5 girdileriyle çalıştırılmıştır; çıktı tablodakiyle aynıdır. Metinlerde Türkçe karakter kullanılmamasının nedeni de pratiktir: bazı uçbirimler UTF-8 karakterleri doğru göstermez. Bu dizgilerin her karakteri tek bayttır.

3.1.6.2 C++

```
#include <iostream>

int main() {
    int number;
    std::cout << "Bir sayi girin: ";
    std::cin >> number;
    for (int i = 1; i <= number; i++) {
        if (i % 2 == 0) {
            std::cout << i << " sayisi bir cift sayidir."
                      << std::endl;
        } else {
            std::cout << i << " sayisi bir tek sayidir."
                      << std::endl;
        }
    }
    return 0;
}
```

C++ (1985, Bjarne Stroustrup), C'nin üst kümesi olarak tasarlanmıştır; geçerli C kodunun çoğu C++'ta da geçerlidir. Sınıflar, şablonlar ve istisnalar gibi yapılar ekler. Giriş ve çıkış için `printf` / `scanf` yerine **akışlar** kullanılır. `std::cout <<` yazar, `std::cin >>` okur. `<<` işleci art arda kullanılabilir: önce `i`, sonra metin, sonra satır sonu akışa gönderilir. Okumada adres işareti gerekmez; C++'ın başvuru (*reference*) düzeneği bunu kendisi halleder.

`std::endl` satır sonu ekler ve ayrıca çıktı tamponunu boşaltır. Tamponu her satırda boşaltmak çok satırlı çıktılarda yavaştır; bu durumda `'\n'` daha hızlıdır. Derleme `g++ tekcift.cpp -o tekcift` komutuyla yapılır. Çıktı C sürümüyle birebir aynıdır.

3.1.6.3 C# ve Java

```
using System;

class Program {
  static void Main() {
    Console.Write("Bir sayi girin: ");
    int number = int.Parse(Console.ReadLine());
    for (int i = 1; i <= number; i++) {
      if (i % 2 == 0) {
        Console.WriteLine($"{i} sayisi bir cift sayidir.");
      } else {
        Console.WriteLine($"{i} sayisi bir tek sayidir.");
      }
    }
  }
}
```

C# (2000, Microsoft), .NET platformunun dilidir. Kaynak kod IL ara diline derlenir; .NET'in sanal makinesi CLR (*common language runtime*), çalışırken bu kodu JIT ile makine koduna çevirir. Program bir sınıfın içindeki `Main` metoduyla başlar. `Console.ReadLine()` bir satır metin okur, `int.Parse` bunu tam sayıya çevirir. Girdi sayı değilse `int.Parse` istisna fırlatır ve program durur; `int.TryParse` bu durumda yanlış döndürdüğü için daha güvenlidir. `$"{i} …"` yazımı, dizginin içine bir değişkenin değerini yerleştirir (*string interpolation*). Program `dotnet run` ile çalıştırılır.

```
import java.util.Scanner;

public class Main {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Bir sayi girin: ");
    int number = scanner.nextInt();
    for (int i = 1; i <= number; i++) {
      if (i % 2 == 0) {
        System.out.println(i + " sayisi bir cift sayidir.");
      } else {
        System.out.println(i + " sayisi bir tek sayidir.");
      }
    }
  }
}
```

Java (1995, Sun Microsystems) da hibrit bir dildir. `javac Main.java` komutu, işlemciden bağımsız **bytecode** içeren `Main.class` dosyasını üretir. `java Main` komutu JVM'yi başlatır; JVM bytecode'u önce yorumlar, sık çalışan kısımları JIT ile derler. Aynı `.class` dosyası JVM'si olan her platformda çalışır. Java'nın “bir kez yaz, her yerde çalıştır” sloganı buradan gelir.

Java'da her şey bir sınıfın içindedir ve `public` sınıfın adı dosya adıyla aynı olmak zorundadır: `Main` sınıfı `Main.java` dosyasında durur. Girdi `Scanner` sınıfıyla okunur; `nextInt()` sıradaki tam sayıyı döndürür. `+` işleci bir sayıyı dizgiyle birleştirirken sayıyı metne çevirir. Program bu not hazırlanırken JDK 21 ile derlenip çalıştırılmış ve C ile aynı çıktıyı vermiştir.

3.1.6.4 Go ve Rust

```
package main

import "fmt"

func main() {
    var number int
    fmt.Print("Bir sayi girin: ")
    fmt.Scan(&number)
    for i := 1; i <= number; i++ {
        if i%2 == 0 {
            fmt.Printf("%d sayisi bir cift sayidir.\n", i)
        } else {
            fmt.Printf("%d sayisi bir tek sayidir.\n", i)
        }
    }
}
```

Go (2009, Google), doğrudan makine koduna derlenir ve bütün bağımlılıklarını içeren tek, bağımsız bir çalıştırılabilir dosya üretir. Belleği bir **çöp toplayıcı** (*garbage collector*) yönetir; artık kullanılmayan bellek kendiliğinden geri alınır. Eşzamanlı işler için çok hafif iş parçacıkları olan **goroutine**'ler vardır. Söz dizimi C'ye benzer, ama koşul ve döngülerde parantez kullanılmaz. `i := 1` yazımı değişkeni tanımlar ve tipini sağdaki değerden çıkarır. `fmt.Scan(&number)` C'deki gibi adres alır. Program `go run tekcift.go` ile çalıştırılır.

```
use std::io;

fn main() {
    let mut input = String::new();
    println!("Bir sayi girin:");
    io::stdin().read_line(&mut input)
        .expect("okuma hatasi");
    let number: i32 = input.trim().parse()
        .expect("cevirme hatasi");
    for i in 1..=number {
        if i % 2 == 0 {
            println!("{} sayisi bir cift sayidir.", i);
        } else {
            println!("{} sayisi bir tek sayidir.", i);
        }
    }
}
```

Rust (2015, Mozilla), çöp toplayıcı kullanmadan bellek güvenliği sağlar. Her değerin tek bir **sahibi** (*owner*) vardır; sahiplik ve ödünç alma kuralları derleme sırasında denetlenir. Boşaltılmış belleğe erişmek ya da aynı veriyi iki yerden aynı anda değiştirmek gibi hatalar program çalışmadan önce yakalanır. Değişkenler varsayılan olarak değiştirilemez; `input` değişkeni okunan satırla dolacağı için `mut` ile tanımlanmıştır.

Girdi `read_line` ile bir dizgiye okunur. `trim()` satır sonunu atar, `parse()` sayıya çevirir. Bu işlemler başarısız olabilir; Rust bu olasılığı görmezden gelmeye izin vermez. `.expect("…")`, hata olursa iletiyi yazıp programı durdurur. `1..=number` üst sınırı dahil eden bir aralıktır. Program `rustc tekcift.rs` ya da `cargo run` ile derlenir.

3.1.6.5 Fortran ve Ada

```
program Main
    implicit none
    integer :: number, i
    write(*,*) 'Bir sayi girin:'
    read(*,*) number
    do i = 1, number
        if (mod(i, 2) == 0) then
            write(*,*) i, ' sayisi bir cift sayidir.'
        else
            write(*,*) i, ' sayisi bir tek sayidir.'
        end if
    end do
end program Main
```

Fortran (1957, IBM, John Backus), ilk yaygın yüksek düzeyli dildir. Adı **FOR**mula **TRAN**slation sözcüklerinden gelir ve formülleri doğrudan kod olarak yazmayı hedefler. Hava tahmini, akışkanlar dinamiği ve fizik benzetimlerinde bugün de yaygındır; bu alanlardaki büyük sayısal kütüphanelerin çoğu Fortran ile yazılmıştır.

`implicit none`, bütün değişkenlerin açıkça tanımlanmasını zorunlu kılar. Eski Fortran'da adı i–n harfleriyle başlayan değişkenler kendiliğinden tam sayı sayılırdı; bu kural yazım hatalarını gizlediği için kapatılır. `do i = 1, number` döngüsü üst sınırı dahil eder. Kalan işlemi bir işleçle değil, `mod(i, 2)` fonksiyonuyla yapılır. `write(*,*)` serbest biçimde yazar; sayıların önünde boşluk görülebilir. Derleme `gfortran tekcift.f90 -o tekcift` komutuyla yapılır.

```
with Ada.Text_IO;         use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   Number : Integer;
begin
   Put ("Bir sayi girin: ");
   Get (Number);
   for I in 1 .. Number loop
      Put (I, Width => 0);
      if I mod 2 = 0 then
         Put_Line (" sayisi bir cift sayidir.");
      else
         Put_Line (" sayisi bir tek sayidir.");
      end if;
   end loop;
end Main;
```

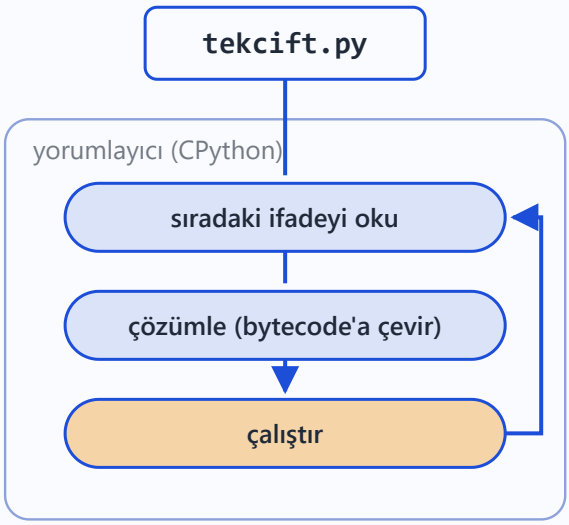
Ada (1983, ABD Savunma Bakanlığı), güvenliğin kritik olduğu büyük sistemler için tasarlanmıştır. Tip denetimi çok güçlüdür; örneğin farklı tipteki tam sayılar açık dönüşüm olmadan karıştırılamaz. Havacılık, savunma ve demiryolu yazılımlarında kullanılır. `with` satırları kullanılan paketleri, `use` satırları bu paketlerdeki adları paket adını yazmadan kullanmayı sağlar. Tam sayı okumak ve yazmak için `Ada.Integer_Text_IO` paketi gerekir. `Put (I, Width => 0)` sayıyı önünde boşluk bırakmadan yazar; `Put_Line` metnin sonuna satır sonu ekler. `I mod 2` kalanı verir. Derleme `gnatmake main.adb` ile yapılır.

Go, Rust, Fortran ve Ada derleyicileri bu notun hazırlandığı bilgisayarda kurulu değildi. Bu dört program çalıştırılmadı; her dilin belgelerine göre satır satır denetlendi.

3.1.7 Yorumlanan diller

3.1.7.1 Yorumlayıcı nasıl çalışır

Yorumlanan bir dilde ayrı bir derleme adımı yoktur. Programcı kaynak dosyayı doğrudan yorumlayıcıya verir: `python tekcift.py`. Yorumlayıcı da işlemcide çalışan sıradan bir programdır; çoğunlukla C ile yazılmıştır. Görevi, kaynak koddaki ifadeleri sırayla okuyup her birinin istediği işi yapmaktır. Bu yaklaşım hızlı geliştirme ve deneme sağlar. Python ve Ruby gibi dillerin etkileşimli kabuğunda (REPL, *read–eval–print loop*) bir satır yazılır, sonucu hemen görülür.



Şekil 3.1.14: Yorumlayıcı döngüsü. CPython sıradaki ifadeyi okur, bytecode'a çevirerek çözümler, çalıştırır ve bir sonraki ifadeye geçer.

Şekil 3.1.14'teki döngü, işlemcinin getir-çöz-yürüt döngüsünün yazılımdaki karşılığıdır. Fark, her adımın donanım yerine bir programla yapılmasıdır. Bir toplama işlemi işlemcide tek bir komut tutarken yorumlayıcıda onlarca, bazen yüzlerce komut tutar. Yorumlayıcı önce işlenenlerin tipine bakmalı, sonra uygun toplama yordamını seçmeli, sonucu yeni bir nesne olarak oluşturmalıdır. Yorumlanan dillerin yavaşlığı buradan gelir. Gerçek yorumlayıcılar kaynak kodu her seferinde metin olarak çözmez; önce bytecode'a çevirir ve bytecode'u yorumlar. CPython, çevirdiği bytecode'u `.pyc` dosyalarında saklayarak bir sonraki çalıştırmada çeviriyi de atlar.

Yaygın yorumlanan dillerin her biri belirli bir alanda öne çıkar. Python genel amaçlıdır; veri bilimi, yapay zekâ ve otomasyon için çok kullanılır. JavaScript web'in dilidir; tarayıcıda ve Node.js ile sunucuda çalışır. Ruby kullanım kolaylığı ve esnekliğiyle, özellikle Ruby on Rails web çatısıyla tanınır.

3.1.7.2 Python

```
# Kullanıcıdan girdi alma
number = int(input("Bir sayi girin: "))

# 1'den number'a kadar sayıları kontrol et
for i in range(1, number + 1):
    if i % 2 == 0:
        print(f"{i} sayisi bir cift sayidir.")
    else:
        print(f"{i} sayisi bir tek sayidir.")
```

Python (1991, Guido van Rossum) okunabilirliği öne koyar. Bloklar süslü parantezle değil **girintiyle** belirlenir. `for` satırının altındaki girintili satırlar döngünün gövdesidir; girinti bitince gövde de biter. Girinti burada bir yazım alışkanlığı değil, dilin kuralıdır. Yanlış girinti ya bir söz dizimi hatası verir ya da programın anlamını değiştirir.

`input` her zaman metin döndürür. Kullanıcı 3 yazsa bile dönen değer `"3"` dizgisidir; `int(...)` onu tam sayıya çevirir. `range(1, number + 1)` 1'den başlayan bir sayı dizisi üretir, ama **üst sınırı dahil etmez**. Bu yüzden `number`'a 1 eklenmiştir. `f"{i} sayisi …"` biçimli dizgidir (*f-string*): süslü parantez içindeki ifadenin değeri metne yerleştirilir. Program `python tekcift.py` ile çalıştırılır.

Python **dinamik tiplidir**: bir değişkenin tipi çalışırken, ona atanan değere göre belirlenir. `number` önceden tanımlanmaz; atama onu yaratır. C sürümünde 15 satır tutan iş burada 6 satır koddur. Yüksek düzeyli dillerin **ifade gücü** budur. Bedeli hızdır: CPython aynı döngüyü derlenmiş C'den onlarca kat yavaş çalıştırır, çünkü her `i % 2` işleminde tipleri denetler ve yeni tam sayı nesneleri oluşturur.

3.1.7.3 JavaScript

```
// Kullanıcıdan girdi alma
let number = Number(prompt("Bir sayi girin:"));

// 1'den number'a kadar sayıları kontrol et
for (let i = 1; i <= number; i++) {
    if (i % 2 === 0) {
        console.log(i + " sayisi bir cift sayidir.");
    } else {
        console.log(i + " sayisi bir tek sayidir.");
    }
}
```

JavaScript (1995, Brendan Eich) tarayıcıların dilidir; Node.js ile sunucuda da çalışır. V8 gibi modern motorlar kodu önce bytecode'a çevirip yorumlar, sık çalışan fonksiyonları JIT ile makine koduna derler. Bu yüzden JavaScript bugün "yorumlanan" ile "hibrit" arasında bir yerdedir.

`prompt` tarayıcıda bir giriş kutusu açar ve kullanıcının yazdığını **metin** olarak döndürür. `Number(...)` bu metni sayıya çevirir. Çevirme yapılmasaydı `i <= number` karşılaştırması yine çalışırdı, çünkü JavaScript sayıyla metni karşılaştırırken metni kendiliğinden sayıya çevirir. Ama `number + 1` gibi bir ifade "3" ile 1'i yan yana yazıp "31" üretirdi. Açık çevirme bu tür sürprizleri önler. `===` işleci tip dönüştürmeden eşitliği sınar: `0 === "0"` yanlıştır, `0 == "0"` doğrudur. `console.log` çıktıyı tarayıcının geliştirici konsoluna ya da uçbirime yazar. Node.js'te `prompt` fonksiyonu bulunmadığından, bu not hazırlanırken program girdiyi veren küçük bir `prompt` tanımıyla Node.js'te çalıştırılmış ve C ile aynı satırları yazmıştır.

3.1.7.4 Ruby, PHP ve Perl

```ruby
# Kullanıcıdan girdi alma
print "Bir sayi girin: "
number = gets.chomp.to_i

# 1'den number'a kadar sayıları kontrol et
for i in 1..number do
  if i % 2 == 0
    puts "#{i} sayisi bir cift sayidir."
  else
    puts "#{i} sayisi bir tek sayidir."
  end
end
```

Ruby (1995, Yukihiro Matsumoto) tamamen nesne yönelimlidir: sayılar dahil her şey bir nesnedir. `gets.chomp.to_i` art arda üç metot çağrısıdır. `gets` bir satır okur, `chomp` sonundaki satır sonunu atar, `to_i` tam sayıya çevirir. `1..number` üst sınırı dahil eden bir aralıktır. Bloklar `end` ile kapanır. `"#{i} …"`, Python'daki biçimli dizginin karşılığıdır: `#{…}` içindeki ifadenin değeri metne yerleşir. `puts` yazdığı metnin sonuna satır sonu ekler. Program `ruby tekcift.rb` ile çalıştırılır.

```php
<?php
// Kullanıcıdan girdi alma
echo "Bir sayi girin: ";
$number = (int) trim(fgets(STDIN));

// 1'den number'a kadar sayıları kontrol et
for ($i = 1; $i <= $number; $i++) {
    if ($i % 2 == 0) {
        echo $i . " sayisi bir cift sayidir.\n";
    } else {
        echo $i . " sayisi bir tek sayidir.\n";
    }
}
?>
```

PHP (1995, Rasmus Lerdorf) sunucu tarafında çalışan bir web dilidir. Kod `<?php` ile `?>` arasına yazılır ve HTML sayfalarının içine gömülebilir. Değişken adları `$` ile başlar. `.` işleci dizgileri birleştirir; `$i . " sayisi …"` önce sayıyı metne çevirir, sonra birleştirir. Komut satırında `fgets(STDIN)` bir satır okur, `trim` satır sonunu atar, `(int)` tam sayıya çevirir. Program `php tekcift.php` ile çalıştırılır.

```
use strict;
use warnings;

# Kullanıcıdan girdi alma
print "Bir sayi girin: ";
my $number = <STDIN>;
chomp $number;

# 1'den number'a kadar sayıları kontrol et
for my $i (1..$number) {
    if ($i % 2 == 0) {
        print "$i sayisi bir cift sayidir.\n";
    } else {
        print "$i sayisi bir tek sayidir.\n";
    }
}
```

Perl (1987, Larry Wall) metin işleme ve sistem yönetimi için tasarlanmıştır ve güçlü düzenli ifadeleriyle tanınır. `<STDIN>` standart girdiden bir satır okur; satır sonu da okunan metne dahildir ve `chomp` onu atar. Çift tırnaklı dizgide `$i` doğrudan değeriyle değiştirilir. `use strict;` ve `use warnings;` tanımlanmamış değişken kullanımı gibi yaygın hataları yakalar. `my` değişkeni tanımlar. Program `perl tekcift.pl` ile çalıştırılır.

Bu üç dilden yalnız Perl bu not hazırlanırken çalıştırılabildi; çıktısı C ile aynıdır. Ruby ve PHP yorumlayıcıları kurulu değildi, iki program dillerin belgelerine göre denetlendi.

3.1.7.5 MATLAB

```
% Kullanıcıdan girdi alma
number = input('Bir sayi girin: ');

% 1'den number'a kadar sayıları kontrol et
for i = 1:number
    if mod(i, 2) == 0
        fprintf('%d sayisi bir cift sayidir.\n', i);
    else
        fprintf('%d sayisi bir tek sayidir.\n', i);
    end
end
```

MATLAB (1984, MathWorks), adını **MAT**rix **LAB**oratory sözcüklerinden alır ve matris tabanlı sayısal hesap için tasarlanmıştır. Mühendislikte, özellikle sinyal işleme ve denetim sistemlerinde çok yaygındır. `input` kullanıcının yazdığını bir ifade olarak değerlendirir; 3 yazılınca sayı olarak 3 döner. `1:number`, 1, 2, …, `number` değerlerinden oluşan bir satır vektörü üretir ve `for` bu vektörün öğelerinde sırayla gezinir. Kalan `mod(i, 2)` ile bulunur. Satır sonundaki `;` sonucun ekrana basılmasını engeller. Açıklamalar `%` ile başlar. MATLAB ticari bir üründür; ücretsiz ve büyük ölçüde uyumlu bir seçenek GNU Octave'dır. MATLAB'in yeni sürümleri de içeride JIT kullanır.

3.1.7.6 On dört dilin karşılaştırması

Tablo 3.1.14, aynı programın on dört dildeki yazılışını üç noktada karşılaştırır: döngü başlığı, kalan işlemi ve yazdırma.

Dil	Model	Döngü başlığı	Kalan	Yazdırma
C	derlenen	`for (int i = 1; i <= n; i++)`	`i % 2`	`printf`
C++	derlenen	`for (int i = 1; i <= n; i++)`	`i % 2`	`std::cout <<`
C#	IL + JIT	`for (int i = 1; i <= n; i++)`	`i % 2`	`Console.WriteLine`
Java	bytecode + JIT	`for (int i = 1; i <= n; i++)`	`i % 2`	`System.out.println`
Go	derlenen	`for i := 1; i <= n; i++`	`i%2`	`fmt.Printf`
Rust	derlenen	`for i in 1..=n`	`i % 2`	`println!`
Fortran	derlenen	`do i = 1, n`	`mod(i, 2)`	`write(*,*)`
Ada	derlenen	`for I in 1 .. N loop`	`I mod 2`	`Put_Line`
Python	yorumlanan	`for i in range(1, n + 1):`	`i % 2`	`print`
JavaScript	yorumlanan/JIT	`for (let i = 1; i <= n; i++)`	`i % 2`	`console.log`
Ruby	yorumlanan	`for i in 1..n do`	`i % 2`	`puts`
PHP	yorumlanan	`for ($i = 1; $i <= $n; $i++)`	`$i % 2`	`echo`
Perl	yorumlanan	`for my $i (1..$n)`	`$i % 2`	`print`
MATLAB	yorumlanan/JIT	`for i = 1:n`	`mod(i, 2)`	`fprintf`

Tablo 3.1.14: Aynı program, on dört dil.

Tabloda iki aile açıkça görülür. C'den türeyen diller (C++, C#, Java, JavaScript, PHP, Go) üç parçalı `for` başlığını ve `%` işlecini korur. Öteki diller döngüyü bir aralık ya da vektör üzerinde yürütür: Rust'ta `1..=n`, Python'da `range`, Ruby'de `1..n`, Perl'de `(1..$n)`, MATLAB'de `1:n`, Fortran ve Ada'da sınır çiftiyle. Aralık yazımında en sık hata kaynağı üst sınırın dahil olup olmadığıdır. Python'un `range`'i üst sınırı dışarıda bırakır; Rust'ta `1..n` dışarıda, `1..=n` içeride bırakır. Ruby'de ise `1..n` üst sınırı dahil eder, `1...n` etmez. Kalan için Fortran, Ada ve MATLAB bir işleç yerine `mod` kullanır.

Sık yapılan hata

Bir dildeki aralık kuralını başka bir dile taşımak. C'deki `for (i = 1; i <= n; i++)` döngüsü Python'a `for i in range(1, n):` diye çevrilirse son değer (`n`) atlanır. Program söz dizimi olarak doğrudur ve hata vermez; eksik satır ancak çıktı incelenince fark edilir.

3.1.8 Derlenen mi, yorumlanan mı?

3.1.8.1 Ölçütler

Üç çalışma modeli beş ölçüt üzerinden karşılaştırılabilir (Tablo 3.1.15).

Ölçüt	Derlenen	Yorumlanan	Hibrit (bytecode + JIT)
Çeviri zamanı	çalışmadan önce, bir kez	her çalıştırmada	önce ara koda, sonra çalışırken
Hız	en yüksek	en düşük	yüksek (ısındıktan sonra)
Dağıtılan	makine kodu	kaynak kod	ara kod (`.class` , `.dll`)
Taşınabilirlik	her platform için yeniden derlenir	yorumlayıcı olan her yerde	sanal makine olan her yerde
Hata yakalama	çoğu derlemede	çoğu çalışırken	tip hataları derlemede
Örnek	C, C++, Go, Rust, Fortran, Ada	Python, Ruby, PHP, Perl, MATLAB	Java, C#, JavaScript (V8)

Tablo 3.1.15: Derlenen, yorumlanan ve hibrit modellerin karşılaştırması.

"Isındıktan sonra" ifadesi JIT'in bir özelliğini anlatır. Program başladığında sanal makine kodu yorumlar ve hangi kısımların sık çalıştığını sayar. Sık çalışan bir döngü ya da fonksiyon belirli bir eşiği aşınca makine koduna derlenir. Bu yüzden hibrit bir program ilk saniyelerde yavaş, sonra derlenmiş koda yakın hızda çalışır. JIT derleyicisi bazı durumlarda önceden derleyen bir derleyiciden de iyi kod üretebilir, çünkü programın gerçekte hangi değerlerle çalıştığını görür.

Tanım 3.1.6

"Derlenen" ya da "yorumlanan" olmak bir dilin değil, o dilin **gerçeklemesinin** (*implementation*) özelliğidir. Aynı dil için hem derleyici hem yorumlayıcı yazılabilir.

C genellikle derlenir, ama C yorumlayıcıları da vardır. Python genellikle CPython ile yorumlanır, ama Cython Python benzeri kodu C'ye çevirip derler, PyPy ise JIT ile çalıştırır. "Python yavaş bir dildir" demek yerine "CPython yorumlayıcısı yavaştır" demek daha doğrudur. Dilin kendisi bir tanımdır; hızı, onu çalıştıran araç belirler.

3.1.8.2 Ara kod ve yığın makinesi

Ara kod, hiçbir gerçek işlemciye ait olmayan, hayalî bir işlemcinin makine dilidir. JVM ve CLR gibi sanal makineler bu hayalî işlemciyi **yığın makinesi** (*stack machine*) olarak tanımlar. Yığın makinesinde komutların işlenen alanı yoktur. İşlenenler bir **işlenen yığınında** durur. Komut, üstteki öğeleri yığından alır, işlemi yapar ve sonucu yığına geri koyar. Yazmaç sayısı işlemciden işlemciye değiştiği için yazmaçsız bir tasarım ara kodu taşınabilir kılar. Bu kodu gerçek işlemcinin yazmaçlarına eşlemek JIT'in işidir.

`x = (a + b) * c` ifadesi için derleyicinin ürettiği ara kod şöyledir:

```
// x = (a + b) * c;
iload a      // a'yı yığına it
iload b      // b'yi yığına it
iadd         // üst iki öğeyi topla
iload c
imul         // üst iki öğeyi çarp
istore x     // yığından x'e yaz
return
```

Yerel değişkenler a = 6, b = 7, c = 3 olsun. Tablo 3.1.16 işlenen yığınını alttan üste doğru yazar.

Komut	İşlem	İşlenen yığını (alt → üst)	x
`iload a`	a'yı it	[6]	?
`iload b`	b'yi it	[6, 7]	?
`iadd`	7 ve 6'yı çek, 13'ü it	[13]	?
`iload c`	c'yi it	[13, 3]	?
`imul`	3 ve 13'ü çek, 39'u it	[39]	?
`istore x`	üstü çek, x'e yaz	[ ]	39
`return`	dön	[ ]	39

Tablo 3.1.16: Yığın makinesinde `x = (a + b) * c` izlemesi (a = 6, b = 7, c = 3).

`iadd` komutunda işlenen adı yoktur; komut hangi değerleri toplayacağını yığının durumundan bilir. Komutların başındaki `i`, işlemin tam sayı (*integer*) üzerinde olduğunu belirtir. Parantezli ifade, ara kodda işlemlerin sırasına dönüşmüştür: toplama çarpmadan önce yapılır, çünkü önce o yazılmıştır. Ara kodun bu sırası, ifadenin sonek gösterimi olan `a b + c *` ile aynıdır.



Şekil 3.1.15: `iadd` 'den sonra yığın makinesi. Üstteki iki öğe (6 ve 7) çekilmiş, toplamları 13 yığına itilmiştir; yerel değişkenler değişmemiştir.

Gerçek Java bytecode'u da aynı yapıdadır. Yalnız yerel değişkenlere adla değil, sıra numarasıyla başvurulur. Aşağıdaki Java metodu JDK 21 ile derlenip `javap -c` ile açıldığında şu kod görülür:

```
static int hesapla(int a, int b, int c) {
    int x = (a + b) * c;
    return x;
}

 0: iload_0      // a (0 numaralı yerel)
 1: iload_1      // b
 2: iadd
 3: iload_2      // c
 4: imul
 5: istore_3     // x (3 numaralı yerel)
 6: iload_3
 7: ireturn
```

Soldaki sayılar bytecode içindeki bayt konumlarıdır; her komut burada tek bayttır. `hesapla(6, 7, 3)` çağrısı 39 döndürür. CPython'un ürettiği bytecode da aynı biçimdedir: `x = (a + b) * c` satırı `LOAD_NAME a`, `LOAD_NAME b`, `BINARY_OP +`, `LOAD_NAME c`, `BINARY_OP *`, `STORE_NAME x` komutlarına çevrilir. İki sanal makine de yığın makinesidir.

Bu ara kod hiçbir işlemciye bağlı değildir. Sanal makine her komutu ya yorumlar ya da JIT ile 8086, x86-64 ya da ARM gibi gerçek bir işlemcinin makine koduna çevirir. 8086'da bu ifade, işlenen yığını yerine yazmaçlarla birkaç komut tutar: `mov ax, a`, `add ax, b`, `imul c`, `mov x, ax`. Yığın makinesindeki “yığına it” ve “yığından çek” işlemleri, gerçek işlemcide yazmaçlara yükleme ve yazmaçtan yazmaya dönüşür.

3.1.9 Aynı görev 8086 assembly ile

3.1.9.1 Program ve bellek yerleşimi

Tek/çift görevi son olarak 8086 assembly ile yazılır. Program emu8086'da bir COM dosyası olarak çevrilir. Görevi kısa tutmak için iki basitleştirme yapılmıştır. `n` = 3 sabit olarak koda yazılmıştır; klavyeden okunmaz. Sayılar tek basamaklıdır; bu yüzden bir sayıyı ekrana yazmak tek bir ASCII karakteri yazmak demektir.

```
org 100h
       mov  cx, 1       ; i = 1
dongu: cmp  cx, 3       ; i ile n karşılaştır
       ja   bitti       ; i > n ise bitir
       mov  dl, cl
       add  dl, '0'     ; rakam → ASCII
       mov  ah, 02h
       int  21h         ; DL'yi yaz
       test cl, 1       ; en düşük bit
       jnz  tek         ; 1 ise tek
       lea  dx, cift_m
       jmp  yaz
tek:   lea  dx, tek_m
yaz:   mov  ah, 09h
       int  21h         ; dizgiyi yaz
       inc  cx          ; i++
       jmp  dongu
bitti: ret
tek_m  db ' sayisi bir tek sayidir.',0Dh,0Ah,'$'
cift_m db ' sayisi bir cift sayidir.',0Dh,0Ah,'$'
```

CX yazmacı döngü sayacıdır ve C'deki `i`'nin yerini tutar. `dongu` etiketindeki `cmp cx, 3` ve `ja bitti`, C'deki `i <= n` koşulunun tersine çevrilmiş hâlidir: `i > n` ise döngüden çık. `ja` (*jump if above*) işaretsiz karşılaştırmada "büyükse" atlar; CF = 0 ve ZF = 0 olduğunda atlama yapılır. Sayaç hiçbir zaman negatif olmadığı için işaretsiz atlama burada doğrudur.

Gövdenin ilk dört satırı sayıyı yazar. `mov dl, cl` sayacın alt baytını DL'ye kopyalar. `add dl, '0'` ona '0' karakterinin ASCII kodunu, yani 30h'yi ekler. ASCII tablosunda rakamlar 30h'den 39h'ye sıralıdır; 1 + 30h = 31h, '1' karakteridir. `mov ah, 02h` ve `int 21h`, DOS'un "DL'deki karakteri yaz" hizmetidir.

Sonraki iki satır C'deki `i % 2 == 0` sınamasının karşılığıdır. 2'ye bölümden kalan, sayının en düşük bitidir. `test cl, 1`, CL ile 1'in bit bit VE işlemini yapar, sonucu saklamaz, yalnız bayrakları ayarlar. En düşük bit 1 ise sonuç 1 olur ve ZF = 0'dır; bit 0 ise sonuç 0 olur ve ZF = 1'dir. `jnz tek` (*jump if not zero*) ZF = 0 iken atlar. Bölme komutu (`div`) 8086'da 80 saat turundan fazla sürebilir; `test` ise birkaç turdur. Derleyiciler de aynı yolu seçer.

Çift sayı için akış atlamadan devam eder: `lea dx, cift_m` DX'e çift iletisinin adresini yükler ve `jmp yaz` ortak yazdırma koduna gider. Tek sayıda `tek` etiketine atlanır ve DX'e tek iletisinin adresi yüklenir. İki yol `yaz` etiketinde birleşir: 09h hizmeti DX'teki adresten başlayıp `$`'a kadar yazar. İletilerin sonundaki `0Dh, 0Ah` satır başı ve satır sonu karakterleridir; DOS'ta yeni satıra geçmek için ikisi birlikte gerekir. `inc cx` sayacı artırır, `jmp dongu` koşula döner. `ret`, COM programını DOS'a döndürür.

Program belleğe şöyle yerleşir. Komutların uzunlukları ve adresleri Tablo 3.1.17'de verilmiştir; kod 38 bayttır ve iletiler kodun hemen arkasından başlar.

Adres	Bayt	Satır
0100h	3	`mov cx, 1`
0103h	3	`dongu: cmp cx, 3`
0106h	2	`ja bitti`
0108h	2	`mov dl, cl`
010Ah	3	`add dl, '0'`
010Dh	2 + 2	`mov ah, 02h` , `int 21h`
0111h	3	`test cl, 1`
0114h	2	`jnz tek`
0116h	3	`lea dx, cift_m`
0119h	2	`jmp yaz`
011Bh	3	`tek: lea dx, tek_m`
011Eh	2 + 2	`yaz: mov ah, 09h` , `int 21h`
0122h	1	`inc cx`
0123h	2	`jmp dongu`
0125h	1	`bitti: ret`
0126h	27	`tek_m db …` (24 karakter + 0Dh + 0Ah + '$')
0141h	28	`cift_m db …` (25 karakter + 0Dh + 0Ah + '$')

Tablo 3.1.17: 8086 tek/çift programının bellek yerleşimi.

Koşullu ve koşulsuz atlamaların hepsi iki bayttır. İlk bayt işlem kodu, ikinci bayt işaretli bir uzaklıktır: hedef, bir sonraki komutun adresine bu uzaklık eklenerek bulunur. Örneğin `ja bitti` 0106h'dedir; sonraki komut 0108h'de başlar ve hedef 0125h'dir. Uzaklık 0125h − 0108h = 1Dh olur ve komut `77 1D` diye kodlanır. Bu **bağıl** adresleme sayesinde program belleğin neresine yüklenirse yüklensin atlamalar doğru çalışır.

3.1.9.2 Kod izleme

Tablo 3.1.18 programın tamamını izler. Bayraklar, satır grubunun son bayrak etkileyen komutundan sonraki değerlerdir. MOV, LEA, INT 21h'nin dönüşü ve atlama komutları bayrakları değiştirmez.

Adım	Satırlar	CX	DL / DX	ZF	CF	Olay
1	`mov cx, 1`	0001h	–	–	–	i = 1
2	`cmp cx, 3`, `ja`	0001h	–	0	1	1 − 3 ödünç ister; atlamaz
3	`mov dl, cl` … `int 21h`	0001h	DL = 31h	0	0	“1” yazılır
4	`test cl, 1`, `jnz`	0001h	DL = 31h	0	0	1 VE 1 = 1; atlar (tek)
5	`lea dx, tek_m` … `int 21h`	0001h	DX = 0126h	0	0	“ sayisi bir tek sayidir.”
6	`inc cx`, `jmp`	0002h	0126h	0	0	i = 2
7	`cmp cx, 3`, `ja`	0002h	0126h	0	1	atlamaz
8	`mov dl, cl` … `int 21h`	0002h	DL = 32h	0	0	“2” yazılır
9	`test cl, 1`, `jnz`	0002h	DL = 32h	1	0	2 VE 1 = 0; atlamaz (çift)
10	`lea dx, cift_m`, `jmp yaz`	0002h	DX = 0141h	1	0	ortak yazdırmaya git
11	`mov ah, 09h`, `int 21h`	0002h	DX = 0141h	1	0	“ sayisi bir cift sayidir.”
12	`inc cx`, `jmp`	0003h	0141h	0	0	i = 3
13	`cmp cx, 3`, `ja`	0003h	0141h	1	0	3 − 3 = 0; ZF = 1, atlamaz
14	`mov dl, cl` … `jnz`	0003h	DL = 33h	0	0	“3” yazılır; tek
15	`lea dx, tek_m` … `int 21h`	0003h	DX = 0126h	0	0	“ sayisi bir tek sayidir.”
16	`inc cx`, `jmp`	0004h	0126h	0	0	i = 4
17	`cmp cx, 3`, `ja`	0004h	0126h	0	0	4 − 3 = 1; **atlar**
18	`bitti: ret`	0004h	0126h	0	0	DOS'a dönülür

Tablo 3.1.18: 8086 tek/çift programının izlemesi (n = 3).

İkinci adımda `cmp cx, 3`, 1 − 3 çıkarmasını yapar. İşaretsiz olarak 1, 3'ten küçüktür; çıkarma ödünç ister ve CF = 1 olur. `ja` yalnız CF = 0 ve ZF = 0 iken atlar, bu yüzden atlamaz. Üçüncü adımda `add dl, '0'` 01h + 30h = 31h üretir; elde yoktur, CF = 0 olur. Dördüncü adımda `test` ZF = 0 bırakır ve `jnz` atlar. `test` komutu CF ve OF'yi her zaman sıfırlar.

Altıncı adımdaki `inc cx` bir ayrıntı taşır: INC komutu ZF, SF ve OF'yi günceller ama **CF'yi değiştirmez**. Burada CF zaten 0'dır, ama döngü sayaçlarında bu özellik önemlidir; bir önceki toplamanın eldesi INC'ten geçerek korunabilir. On üçüncü adımda 3 − 3 = 0 olur: ZF = 1, CF = 0. `ja`'nın koşulu ZF = 0'ı da istediği için atlama olmaz; eşitlik “büyük” sayılmaz. On yedinci adımda 4 − 3 = 1 olur: ZF = 0, CF = 0 ve `ja bitti` atlar. Ekranda sonunda şu üç satır vardır:

```
1 sayisi bir tek sayidir.
2 sayisi bir cift sayidir.
3 sayisi bir tek sayidir.
```

Çıktı, C programının girdi 3 ile verdiği çıktının istem satırı dışında aynısıdır.

3.1.9.3 C ile karşılaştırma

C'deki `for` ve `if` yapıları 8086'da yoktur. Hepsi **karşılaştırma + koşullu atlama** çiftleriyle kurulur. Tablo 3.1.19 bu eşlemeyi satır satır verir.

C	8086
`int i = 1`	`mov cx, 1` (değişken yerine yazmaç)
`i <= number`	`cmp cx, 3` + `ja bitti` (tersine çevrilmiş koşul)
`i++`	`inc cx` + `jmp dongu`
`i % 2 == 0`	`test cl, 1` + `jnz tek` (en düşük bit)
`if … else`	iki dal: `jmp yaz` ile birleşen iki `lea`
`printf("%d …", i)`	`add dl, '0'` + 02h hizmeti, ardından 09h hizmeti
`return 0`	`ret`

Tablo 3.1.19: C yapılarının 8086 karşılıkları.

C sürümü 15 satırdır ve her sayıyı yazabilir. 8086 sürümü 20 satırdır, ama yalnız tek basamaklı sayıları yazar ve `n`'yi okumaz. Çok basamaklı bir sayıyı yazmak için sayıyı art arda 10'a bölüp kalanları rakama çeviren bir döngü, klavyeden okumak için de rakamları birleştiren bir döngü gerekir. C'de `printf` ve `scanf`'in yaptığı iş budur. Yüksek düzeyli dilin bir satırı, assembly'de çoğu zaman bir yordam tutar. Karşılığında assembly programcısı her baytı ve her saat turunu denetler: 8086 programı 93 bayttır ve hiçbir kütüphaneye ihtiyaç duymaz.

3.1.10 Sık yapılan hatalar

Sık yapılan hata

Adres ile içeriği karıştırmak. Bir dizgiyi ekrana yazdıran her hizmet dizginin *adresini* ister: DOS'ta DX, Linux'ta ECX, ARM'da r1, Windows'ta yığına itilen değer. Assembly'de `mov dx, mesaj` yazımı bazı assembler'larda adresi değil, o adresteki sözcüğü yükler. Adres için `lea dx, mesaj` ya da `mov dx, offset mesaj` kullanılır. C'deki `scanf("%d", &number)` çağrısında `&`'ın unutulması aynı hatanın yüksek düzeydeki biçimidir.

Sık yapılan hata

Dizgi sonu kuralını ortama göre seçmemek. DOS'un 09h hizmeti `$`, Windows API'si ve C kütüphanesi 0 baytı bekler; Linux'un `write` çağrısı ise sonlandırıcıya bakmaz, uzunluğu parametre olarak alır. Linux programındaki dizginin sonuna `$` konursa bu karakter ekrana basılır; DOS programında 0 ile biten bir dizgi ise `$` bulunana kadar taşar.

Sık yapılan hata

Veriyi kodun önüne koymak. COM programı ilk baytından çalışmaya başlar. `org 100h`'ın hemen arkasına bir `db` satırı konursa işlemci dizginin baytlarını komut olarak çözer. Veri ya kodun arkasına konur ya da kodun başında bir `jmp` ile verinin üstünden atlanır.

Sık yapılan hata

İşaretli ve işaretsiz atlamaları karıştırmak. Aynı `cmp` komutundan sonra `ja` / `jb` işaretsiz, `jg` / `jl` işaretli karşılaştırma yapar. Negatif olabilen bir sayaçla `ja` kullanılırsa −1 (FFFFh) en büyük işaretsiz sayı olarak görülür ve döngü yanlış dalda devam eder.

Sık yapılan hata

"Yorumlanan dilde hata yoktur" sanmak. Yorumlanan bir dilde program çalışmaya başlar, ama hiç çalışmayan bir daldaki yazım ya da tip hatası ancak o dal çalıştığında ortaya çıkar. Test sırasında her dalı en az bir kez çalıştıran girdiler seçilmelidir.

Sık yapılan hata

Çağrı kuralına uymamak. stdcall bir fonksiyonu çağırdıktan sonra çağıranın da `add esp, 16` ile yığını temizlemesi, yığını iki kez temizlemek demektir. ESP fazladan 16 bayt yukarı kayar ve sonraki `ret` yanlış dönüş adresini alır. Hangi kuralın geçerli olduğu fonksiyonun belgesinden okunmalıdır.

3.1.11 Alıştırmalar ve çözümleri

Sunumdaki alıştırmalar burada ara adımlarıyla çözülmüştür. İlk üç alıştırma Şekil 3.1.5'teki örnek komut biçimini kullanır: 4 bit işlem kodu, 2 bit hedef, 2 bit kaynak; HALT = 0000, ADD = 0001, SUB = 0010, AND = 0011, OR = 0100; R0 = 00, R1 = 01, R2 = 10, R3 = 11.

1. `SUB R3, R0` ve `OR R0, R2` komutlarının ikili ve onaltılık kodları nedir?

SUB'un kodu 0010'dır. Hedef R3 = 11, kaynak R0 = 00. Alanlar sırayla yazılınca `0010 1100` elde edilir. Dört bitlik gruplar onaltılık basamaklara çevrilir: 0010 = 2, 1100 = C. Sonuç 2Ch'dir. OR'un kodu 0100'dür; hedef R0 = 00, kaynak R2 = 10. Komut `0100 0010` = 42h'dir.

2. `0010 0111` hangi komuttur? R1 = 9, R3 = 4 ise sonuç nedir?

İlk dört bit 0010 = SUB. Kalan dört bit 01 11'dir: hedef R1, kaynak R3. Komut `SUB R1, R3`'tür ve R1 ← R1 − R3 yapar. R1 = 9 − 4 = 5 olur; R3 = 4 olarak kalır. İkili sağlama: 1001 − 0100 = 0101.

3. Bu tasarımda kaç farklı işlem kodu tanımlanabilir? Yazmaç sayısını 8'e çıkarmak için komut biçimi nasıl değişmeli?

İşlem kodu alanı 4 bittir: $2^4 = 16$ işlem kodu. Sekiz yazmaçtan birini seçmek için $\log_2 8 = 3$ bit gerekir. Hedef ve kaynak alanları 3'er bite çıkar ve komut 4 + 3 + 3 = 10 bit olur. Komut 8 bitte tutulmak istenirse yazmaç alanları 6 bit kaplar ve işlem koduna 2 bit kalır. 2 bitle yalnız $2^2 = 4$ işlem tanımlanabilir; tasarımdaki beş işlem sığmaz. Bu, komut uzunluğu, yazmaç sayısı ve komut sayısı arasındaki temel ödünleşimdir.

4. DOS programında dizgi sonundaki `'$'` unutulursa ne olur?

09h hizmeti DS:DX'ten başlayarak her baytı ekrana yazar ve 24h değerli bir bayt görünce durur. Sonlandırıcı yoksa dizginin arkasındaki bellek baytları da yazılır. Bu baytlar programın kendi kodu, başka veriler ya da rastgele değerler olabilir. Aralarındaki 07h baytı bip sesi, 0Dh ve 0Ah imleç hareketi üretir. Yazma, bellekte ilk `$` karakterine rastlanana kadar sürer.

5. Linux örneğinde `len` elle 15 yazılsaydı ekrana ne çıkardı?

`sys_write` tam olarak EDX kadar bayt yazar. Dizginin ilk 15 baytı “Merhaba, Linux!” metnidir; 16. bayt satır sonudur. EDX = 15 ise metin yazılır ama satır sonu yazılmaz. Uçbirimin istem satırı metnin hemen arkasına, aynı satıra gelir. Dönüşte EAX = 15 olur.

6. Windows örneğinde parametreler neden ters sırada itilir? `call` 'dan hemen sonra ESP kaçtır?

C çağrı kurallarında ilk parametre yığının tepesinde, dönüş adresinin hemen üstünde bulunmalıdır. Yığın aşağı doğru büyüdüğü için en son itilen değer en üstte kalır; bu yüzden parametreler sondan başa doğru itilir. Fonksiyon ilk parametreyi her zaman [ESP + 4]'te, ikinciyi [ESP + 8]'de bulur. Dört `push` ve `call` 'ın ittiği dönüş adresi 5 × 4 = 20 bayt tutar. ESP = 0012FF80h − 14h = 0012FF6Ch olur.

7. Java neden “derlenen” ya da “yorumlanan” diye tek kelimeyle sınıflanamaz?

Java kaynak kodu `javac` ile derlenir, ama çıktı makine kodu değil bytecode'dur. Bu bytecode JVM'de önce yorumlanır; sık çalışan kısımları çalışma sırasında JIT ile makine koduna derlenir. Program hem önceden derlenir hem de çalışırken yorumlanır ve yeniden derlenir. Bu yüzden Java hibrit modelin örneğidir. Üstelik “derlenen/yorumlanan” ayrımı dilin değil gerçeklemenin özelliğidir: Java'yı doğrudan makine koduna önceden derleyen araçlar da vardır.

3.1.12 Özet

- **Yazılım** donanımı yönetir. **İşletim sistemi** temel yazılımdır; uygulamalar donanıma onun sistem çağrıları üzerinden ulaşır. Programlama dilleri yazılım geliştirmek için kullanılır.
- Diller üç düzeydedir: **makine dili**, **assembly dili** ve **yüksek düzeyli diller**. Düzey yükseldikçe okunabilirlik ve taşınabilirlik artar, donanım üzerindeki doğrudan denetim azalır.

- **Makine dili** işlemcinin tek anladığı dildir: işlem kodu ve işlenen alanlarından oluşan ikili komutlar. Donanıma tamamen bağımlıdır; 8086'da `B0 05`, ARM'da `E3A00005` aynı işi yapar. 8086 çok baytlı sayıları düşük bayt önce saklar.
- Bir komut biçiminde *k* bitlik bir alan 2^k seçenekten birini kodlar. Komut çözmek, bitleri alanlarına ayırıp her alanı tablodan okumaktır.
- **Assembly dili** makine komutlarını anımsatıcılarla yazar. **Assembler** kaynak kodu nesne dosyasına, **bağlayıcı** nesne dosyalarını ve kütüphaneleri çalıştırılabilir dosyaya çevirir; **yükleyici** programı belleğe yerleştirir. İki geçişli assembler birinci geçişte sembol tablosunu doldurur, ikinci geçişte baytları üretir.
- Assembly kodu hem işlemciye hem işletim sistemine bağlıdır: `int 0x80`, `int 21h`, DLL çağrısı, `jsr`, `svc`, `sc`. Hizmet numaraları, parametre yazmaçları ve dizgi sonu kuralları ortama göre değişir.
- **Derlenen diller** (C, C++, Go, Rust, Fortran, Ada) önceden makine koduna çevrilir: hızlıdır ama çıktı platforma özgüdür. **Yorumlanan diller** (Python, Ruby, PHP, Perl, MATLAB) çalışırken işlenir: hızlı geliştirme ve taşınabilir kaynak sağlar, daha yavaş çalışır. Java ve C# **ara kod + JIT** ile ikisinin arasındadır; JavaScript motorları da JIT kullanır.
- Ara kod, yazmaç kullanmayan bir **yığın makinesinin** dilidir; işlenenler yığından alınır, sonuç yığına konur.
- Yüksek düzeydeki `while`, `for` ve `if` yapıları assembly'de **karşılaştırma + koşullu atlama** olarak gerçeklenir; koşul çoğunlukla tersine çevrilir. Değişkenin işaretli ya da işaretsiz olması, hangi atlama komutunun seçileceğini belirler.

3.1.13 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

yazılım *software* · işletim sistemi *operating system* · yüksek / düşük düzeyli dil *high-level / low-level language* · makine dili *machine language* · makine kodu *machine code* · assembly (assembly dili) *assembly language* · anımsatıcı *mnemonic* · komut kümesi *instruction set* · komut biçimi *instruction format* · işlem kodu *opcode* · işlenen *operand* · kodlayıcı / çözücü *encoder / decoder* · kaynak kod *source code* · assembler *assembler* · nesne dosyası *object file* · bağlayıcı *linker* · yükleyici *loader* · çalıştırılabilir dosya *executable file* · liste (listing) çıktısı *listing file* · direktif *directive* · konum sayacı ($) *location counter* · sembol tablosu *symbol table* · söz dizimi (Intel / AT&T) *syntax* · sistem çağrısı *system call* · little-endian (düşük bayt önce) *little-endian* · dizgi *string* · çağrı kuralı *calling convention* · derleyici *compiler* · yorumlayıcı *interpreter* · ara kod *bytecode / intermediate language* · sanal makine *virtual machine* · tam zamanında derleme *just-in-time (JIT) compilation*

yığın makinesi *stack machine* | gerçekleme *implementation*

3.1.14 Kendini Dene

1. Örnek komut biçiminde 0, 1, 2 adreslerinde sırasıyla 13h, 2Ch ve 00h baytları duruyor. Başlangıçta R0 = 2, R1 = 0, R2 = 5, R3 = 9'dur. HALT'ta yazmaçlar kaçtır?

   Cevap

   13h = 0001 00 11: ADD R0, R3 → R0 = 2 + 9 = 11. 2Ch = 0010 11 00: SUB R3, R0 → R3 = 9 − 11 = −2; 8 bitlik yazmaçta ikiye tümleyen olarak 1111 1110 = FEh. 00h: HALT. Sonuç R0 = 11, R1 = 0, R2 = 5, R3 = FEh (−2). İkinci komut R0'ın birinci komuttan sonraki değerini kullanır.

2. 8086'da `mov ax, [0300h]` komutu `A1 00 03` diye kodlanır. 0300h adresinde `34`, 0301h adresinde `12` baytları varsa AX kaç olur? Komut 0301h adresinden bir sözcük okusaydı veri yolu açısından ne değişirdi?

   Cevap

   Düşük bayt önce saklandığı için AX = 1234h olur. 0301h tek bir adrestir. 8086'nın 16 bitlik veri yolu bir çevrimde yalnız çift adresten başlayan bir sözcüğü okuyabilir. Tek adresteki sözcük iki ayrı bayt okumasıyla, yani iki yol döngüsüyle gelir. Sonuç aynıdır, ama erişim en az 4 saat turu uzar.

3. İki geçişli bir assembler'da birinci geçiş neden komutların baytlarını üretmez? Etiketlerin hepsi kullanılmadan önce tanımlansaydı tek geçiş yeterli olur muydu?

   Cevap

   Birinci geçişte ileri başvuruların adresi henüz bilinmez; `offset mesaj` gibi bir işlenen kodlanamaz. Bu geçiş yalnız uzunlukları toplayıp sembol tablosunu doldurur. Bütün etiketler kullanılmadan önce tanımlansaydı her adres gerektiği anda bilinirdi ve tek geçiş yeterdi. Ancak atlama komutlarının çoğu ileriye, örneğin döngü sonuna gider; bu kısıt pratikte uygulanamaz.

4. Bir DOS COM programında `mesaj db 'Selam$'` satırı 0110h adresinde başlıyor. `lea dx, mesaj` komutunun makine kodu ne olur? 09h hizmeti kaç karakter yazar?

   Cevap

   emu8086 doğrudan adresli `lea dx, …` komutunu `mov dx, offset …` gibi kodlar: `BA` ve little-endian adres, yani `BA 10 01` (3 bayt). MASM'da aynı komut `8D 16 10 01` olurdu. 09h hizmeti “Selam” sözcüğünün 5 karakterini yazar; `$` yazılmaz.

5. 32 bitlik Linux'ta `mov eax, 1` / `mov ebx, 7` / `int 0x80` dizisi ne yapar? Aynı iş 64 bitlik Linux'ta hangi yazmaçlarla ve hangi komutla yapılır?

   Cevap

   EAX = 1 `sys_exit`'tir; süreç 7 çıkış koduyla sonlanır. 64 bitlik Linux'ta çıkış hizmetinin numarası 60'tır ve RAX'e, çıkış kodu RDI'ya konur; çekirdeğe `syscall` komutuyla geçilir.

6. stdcall kuralıyla iki parametreli bir fonksiyon çağrılmadan önce ESP = 0012FF40h'dir. Fonksiyona girildiği anda ESP kaçtır ve [ESP + 8] neyi tutar? Fonksiyon hangi komutla dönmelidir?

   Cevap

   İki `push` ve dönüş adresi 12 bayttır: ESP = 0012FF40h − 0Ch = 0012FF34h. [ESP] dönüş adresi, [ESP + 4] ilk parametre, [ESP + 8] ikinci parametredir. Fonksiyon parametrelerin 8 baytını kendisi atmak için `ret 8` ile döner; dönüşte ESP yeniden 0012FF40h olur.

7. Yığın makinesinde `a * (b - c)` ifadesinin ara kodu nasıl olur? a = 4, b = 10, c = 3 için yığının en yüksek derinliği kaç olur?

   Cevap

   `iload a`, `iload b`, `iload c`, `isub`, `imul`, `istore x`. Yığın sırasıyla [4], [4, 10], [4, 10, 3], [4, 7], [28], [ ] olur. En yüksek derinlik 3'tür; sonuç x = 28.

8. 8086 tek/çift programında `cmp cx, 3` yerine `cmp cx, 5` yazılıyor ve `ja` yerine yanlışlıkla `jae` kullanılıyor. Program hangi sayıları yazar?

   Cevap

   `jae` CF = 0 iken, yani CX ≥ 5 olduğunda atlar. CX = 1, 2, 3, 4 için CF = 1'dir ve gövde çalışır; CX = 5'te 5 − 5 = 0, CF = 0 olur ve döngüden çıkılır. Program 1'den 4'e kadar yazar; 5 eksik kalır. Doğrusu `ja`'dır: eşitlikte de gövde çalışmalıdır.

9. Bir Python programı `for i in range(1, n):` ile yazıldığında n = 4 için hangi değerler üzerinde gezinir? Aynı döngü Ruby'de `1..n` ve `1...n` ile yazılırsa ne olur?

   Cevap

   Python 1, 2, 3 değerleri üzerinde gezinir; üst sınır dahil değildir. Ruby'de `1..4` 1, 2, 3, 4 üzerinde, `1...4` ise 1, 2, 3 üzerinde gezinir. İki nokta üst sınırı içerir, üç nokta dışarıda bırakır.

3.1.15 İleri okuma

- K. R. Irvine, *Assembly Language for x86 Processors* (7. baskı), Bölüm 1–3: dil düzeyleri, assembler ve bağlayıcı, liste çıktısı ve x86'da ilk programlar; bu notun assembly kısımlarına en yakın kaynaktır.
- B. B. Brey, *The Intel Microprocessors* (8. baskı), Bölüm 1 ve 4: makine dili, 8086 komutlarının kodlanması, ModR/M baytı ve COM/EXE programlarının yapısı.
- A. S. Tanenbaum ve H. Bos, *Modern Operating Systems* (3. baskı), Bölüm 1.6: sistem çağrılarının işleyişi, kullanıcı ve çekirdek modları.
- J. R. Levine, *Linkers and Loaders*, Morgan Kaufmann: nesne dosyaları, sembol çözümleme, yer değiştirme ve yükleme sürecini ayrıntılı anlatır.

3.1.16 Kaynaklar

1. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 3. baskı, Pearson.
2. B. B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson.
3. K. R. Irvine, *Assembly Language for x86 Processors*, 7. baskı, Pearson.
4. NASM belgeleri (nasm.us) ve GNU Binutils *as* kılavuzu.
5. Ralf Brown's Interrupt List (DOS INT 21h hizmetleri); Microsoft Win32 API belgeleri (`MessageBoxA`, `ExitProcess`).
6. emu8086 belgeleri ve 8086 komut kümesi başvurusu.
7. J. R. Levine, *Linkers and Loaders*, Morgan Kaufmann, 1999.

KISIM I · BÖLÜM 3.2

Assembly

Assembly dili, işlemcinin makine komutlarını insanın okuyabileceği kısaltmalarla yazar. Her komut satırı tek bir makine komutuna karşılık gelir. Bu yüzden assembly ile yazan kişi yazmaçları, belleği ve bayrakları doğrudan yönetir; derleyicinin gizlediği her ayrıntı görünür olur. Not, 8086 assembly dilini emu8086 ortamında ele alır. Önce bir satırın alanları, komut ile direktif arasındaki fark ve ad kuralları verilir. Ardından işlenen türleri, komutların yazılmayan örtük işlenenleri ve bellek erişiminin saat turu cinsinden bedeli incelenir. IP'nin her komutta nasıl ilerlediği ve bayrakları hangi komutların değiştirdiği adım adım izlenir. Son kesimlerde ilk .COM programı yazılır, belleğe yerleşen baytları tek tek okunur, emu8086 penceresinde çalıştırılır ve en sık karşılaşılan hatalar nedenleriyle birlikte sınıflandırılır.

ÖĞRENME ÇIKTILARI

Assembly'nin yerini açıklamak
Assembly dilinin makine diliyle birebir ilişkisini ve yüksek düzeyli dillerden farkını bir komutun üç görünümü üzerinden göstermek.

Satır yapısını okumak
Etiket, komut, işlenen ve açıklama alanlarını ayırmak; komutu direktiften ve makrodan ayırt etmek; geçerli ve geçersiz adları tanımak.

İşlenenleri tanımak
Yazmaç, anlık değer ve bellek işlenenlerini ve MUL, DIV, LOOP, CBW, PUSH, XLAT gibi komutların örtük işlenenlerini belirlemek.

Komut süresini hesaplamak
Yazmaç ve bellek işlenenli komutların süre farkını saat turu ve etkin adres hesabıyla bulmak; farkı yol döngüsüyle açıklamak.

IP ve bayrakları izlemek
IP'nin komut uzunluğu kadar ilerleyişini ve bayrakları değiştiren ya da değiştirmeyen komutların etkisini adım adım izlemek.

İlk programı yazmak
emu8086 ile bir .COM programı yazmak, baytlarını okumak, tek adımla çalıştırmak; söz dizimi hatasını mantık hatasından ayırmak.

3.2.1 Assembly dili

3.2.1.1 Komut kısaltmaları ve makine komutu

Bir mikroişlemci bellekten okuduğu baytları çözer ve her bayt dizisini bir işleme dönüştürür. Bu baytların oluşturduğu dile **makine dili** denir. Makine dili işlemcinin doğrudan anladığı tek dildir, ama insan için okunması zordur. `B8 05 00` baytlarına bakan biri, bunun "AX yazmacına 5 yükle" anlamına geldiğini ancak bir tabloya bakarak çıkarabilir. Program yüzlerce komuta ulaştığında bu yöntem çalışmaz.

Assembly dili (*assembly language*) bu sorunu çözmek için doğdu. Her makine komutuna akılda kalan kısa bir ad verilir: taşıma için `MOV`, toplama için `ADD`, atlama için `JMP`. Bu kısa adlara **anımsatıcı** ya da **komut kısaltması** (*mnemonic*) denir. İşlenenler de sayı yerine adlarıyla yazılır: yazmaçlar AX, BX gibi; bellek konumları ise programcının verdiği adlarla.

> **Tanım 3.2.1**
>
> **Assembly dili**, bir işlemcinin makine komutlarını anımsatıcılar ve simgesel adlarla yazan düşük düzeyli programlama dilidir. Assembly kaynağındaki her komut satırı, işlemcinin tek bir makine komutuna karşılık gelir. Bu birebir ilişki dili işlemciye bağımlı kılar: her işlemci ailesinin kendi assembly dili vardır.

Birebir ilişkiyi görmenin en kolay yolu tek bir satırı üç ayrı biçimde yazmaktır. Tablo 3.2.1 `MOV AX, 5` satırının programcının, assembler'ın ve işlemcinin gözündeki hâlini yan yana verir.

Görünüm	Yazılış	Kim kullanır?
Assembly	`MOV AX, 5`	programcı
Onaltılık makine kodu	`B8 05 00`	liste çıktısı, öykünücü, hata ayıklayıcı
İkili makine kodu	`10111000 00000101 00000000`	işlemci (veri yolundaki gerilim düzeyleri)

Tablo 3.2.1: Tek satır, üç görünüm.

Birinci bayt `B8`, "16 bitlik bir yazmaca anlık değer yükle" işleminin **işlem kodudur** (*opcode*). Bu komut ailesinde hedef yazmacın numarası işlem koduna eklenir. AX'in numarası 0 olduğu için kod B8h + 0 = B8h olur. BX'in numarası 3'tür; `MOV BX, 3` komutunun ilk baytı bu yüzden BBh'dir. Komutun geri kalan iki baytı 16 bitlik sabiti taşır. 0005h değeri bellekte **düşük bayt önce** saklanır: önce 05, sonra 00. 8086'nın bu düzenine *little-endian* denir.

İkili satır, işlemcinin gerçekte gördüğü şeydir. 8086 komut baytlarını adres/veri hatları (AD0–AD15) üzerinden okur. Her bit bir hattaki yüksek ya da düşük gerilimdir. Bir elektronik devre gözüyle bakıldığında `MOV AX, 5`, komut çözücünün girişine gelen 24 bitlik bir desenden başka bir şey değildir.

Çözücü bu deseni ALU'yu, yazmaç dosyasının yazma girişini ve iç yolları süren kontrol sinyallerine çevirir.

Assembly'yi yüksek düzeyli dillerden ayıran özellik de bu yakınlıktır. C'de `x = a * b;` yazan programcı çarpmanın hangi yazmaçta yapılacağını bilmez; bunu derleyici seçer. Assembly'de ise çarpılanın AL'de durması, sonucun AX'e yazılması, AH'deki eski değerin silinmesi programcının sorumluluğundadır. Dilin gücü de zorluğu da buradan gelir. Assembly ile yazmak, yazmaçları, belleği ve sistem yolunu bilmeyi gerektirir. Karşılığında bir programın donanımda nasıl çalıştığı bütün ayrıntısıyla görülür.

Örnek 3.2.1

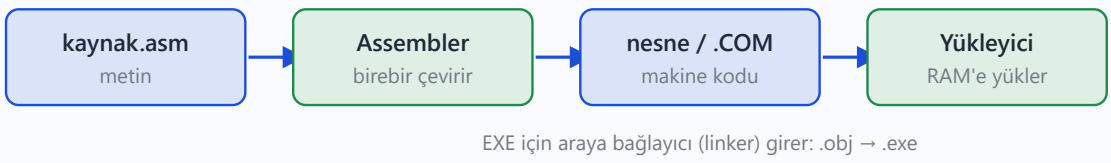
`MOV BX, 1234h` komutunun makine kodu nedir? Bellekte kaç bayt yer kaplar?

Komut ailesi aynıdır: 16 bitlik yazmaca anlık değer. İşlem kodu B8h + yazmaç numarası olur. BX'in numarası 3 olduğundan ilk bayt BBh'dir. Sabit 1234h, düşük bayt önce yazılır: 34h, sonra 12h. Komut `BB 34 12` olarak üç bayt yer kaplar. 8 bitlik yazmaçlar için ayrı bir aile vardır: `MOV AL, 5` komutu `B0 05` diye iki bayta kodlanır, çünkü sabit tek bayttır.

3.2.1.2 Assembler ve emu8086

Assembly kaynağı düz bir metin dosyasıdır. İşlemci bu metni çalıştıramaz; önce makine koduna çevrilmesi gerekir. Bu çeviriyi yapan programa **assembler** denir. Assembler komut kısaltmalarını işlem kodlarına, etiketleri adreslere, değişken adlarını da bağıl konumlara çevirir. Çeviri birebirdir: bir komut satırı bir makine komutu verir. MASM, TASM ve NASM 8086 ailesi için yaygın assembler'lardır.

Çevrilen kod bir dosyaya yazılır. Basit programlar için bu dosya doğrudan çalıştırılabilir bir **.COM** dosyasıdır. Çok kesimli programlar önce bir **nesne dosyasına** (.obj) çevrilir; ardından **bağlayıcı** (*linker*) nesne dosyalarını birleştirip bir **.EXE** dosyası üretir. Programı çalıştırmak için işletim sisteminin **yükleyicisi** dosyayı RAM'e kopyalar, kesim yazmaçlarını ayarlar ve denetimi programın ilk komutuna verir. Şekil 3.2.1 bu zinciri gösterir.



Şekil 3.2.1: Kaynak dosyadan belleğe. Assembler metni birebir makine koduna çevirir, yükleyici sonucu RAM'e yerleştirir. .EXE dosyalarında araya bağlayıcı girer.

Bu derste kullanılan **emu8086**, zincirin bütün halkalarını tek pencerede toplar. İçinde bir metin düzenleyicisi, bir assembler ve bir 8086 **öykünücüsü** (*emulator*) bulunur. Öykünücü, 8086'yı yazılımla taklit eden bir programdır. Gerçek bir işlemcide yazmaçların içi dışarıdan görünmez; öykünücü ise her komuttan sonra yazmaçları, belleği ve bayrakları ekranda gösterir. Program tek komut ilerletilebilir, geri alınabilir, istenen yerde durdurulabilir. Assembly öğrenirken en değerli araç budur, çünkü her komutun etkisi tahmin edilip hemen sınanabilir.

.COM ve .EXE biçimleri arasındaki fark programın yapısını da belirler. Tablo 3.2.2 iki biçimi karşılaştırır.

Özellik	.COM	.EXE
Kesim sayısı	tek kesim; kod, veri ve yığın aynı 64 KB içinde	birden çok kesim (kod, veri, yığın ayrı)
Boyut sınırı	64 KB'tan 256 bayt (PSP) ve yığın payı eksik	64 KB sınırı her kesim için ayrı
Dosya başlığı	yok; dosya belleğe olduğu gibi kopyalanır	var; yer değiştirme tablosu ve başlangıç değerleri içerir
Başlangıç adresi	`ORG 100h`, IP = 0100h	`END` direktifinde verilen etiket
Kesim yazmaçları	CS = DS = ES = SS	DS'yi program kendisi yükler

Tablo 3.2.2: .COM ve .EXE programları.

Bu notun bütün örnekleri .COM biçimindedir ve emu8086 söz dizimiyle yazılmıştır. Örneklerin çoğu MASM ve TASM ile de aynen derlenir. Komutların bayt değerleri, IP ve yazmaç izlemeleri bir 8086 öykünücüsünde çalıştırılarak elde edilmiştir.

3.2.1.3 Assembly ne zaman kullanılır?

Assembly ile yazılan bir program genellikle aynı işi yapan yüksek düzeyli bir programdan küçük ve hızlıdır. Bunun nedeni basittir: programda yalnız programcının yazdığı komutlar bulunur. Derleyicinin eklediği başlatma kodu, kütüphane çağrıları, genel amaçlı yardımcı yordamlar yoktur. Her komutun neden orada olduğu bilinir.

Hızdan daha önemli bir üstünlük, donanıma doğrudan erişimdir. 8086'nın bazı komutlarının standart C'de karşılığı yoktur. `OUT` bir G/Ç portuna bayt yazar, `CLI` kesmeleri yasaklar, `PUSHF` bayrak yazmacını yığına iter. Önyükleyiciler, aygıt sürücüleri, kesme hizmet yordamları ve küçük gömülü sistemler bu yüzden tamamen ya da kısmen assembly ile yazılır. Bilgisayar açıldığında çalışan ilk kod, henüz yığın ve bellek düzeni kurulmadığı için zaten assembly'dir.

Dilin bedeli de ağırdır. Tablo 3.2.3 iki tarafı özetler.

Artılar	Eksiler
Daha az yer kaplar; gereksiz kod yoktur.	Taşınabilir değildir: 8086 kodu ARM işlemcide çalışmaz.
Yalnız gereken işlemleri yaptığı için hızlıdır.	Yazması, okuması ve hata ayıklaması zahmetlidir.

Artılar	Eksiler
Port G/Ç, kesme işleme, bayraklara doğrudan erişim sağlar.	Büyük projelerde verim düşüktür.
Önyükleyici, sürücü ve gömülü sistemlerde vazgeçilmezdir.	Modern derleyiciler çoğu zaman yeterince hızlı kod üretir.

Tablo 3.2.3: Assembly ile programlamanın artıları ve eksileri.

Uygulamada bu iki taraf bir arada kullanılır. Programın büyük bölümü C gibi bir dilde yazılır. Zamanlaması kritik küçük parçalar, örneğin bir kesme yordamının girişi ya da sık çağrılan bir döngünün içi, assembly ile yazılır. Bu **karma** yaklaşım hem geliştirme hızını hem çalışma hızını korur. Assembly bilgisinin bir başka getirisi de yüksek düzeyli kodu okumaktır. Derleyicinin ürettiği assembly çıktısına bakabilen bir mühendis, kodunun neden yavaş çalıştığını ya da bir değişkenin neden beklenmedik bir değer aldığını bulabilir.

Sık yapılan hata

"Assembly her zaman daha hızlıdır" sanmak. Elle yazılmış assembly, ancak programcı işlemciyi derleyiciden iyi tanıyorsa hızlıdır. Bellekte tutulan bir döngü sayacı, gereksiz yere yinelenen bellek okumaları ya da yanlış seçilmiş bir komut, iyi bir derleyicinin ürettiği koddan yavaş bir program verir. Hız, dilden çok yazmaç ve bellek kullanımının doğru seçilmesinden gelir.

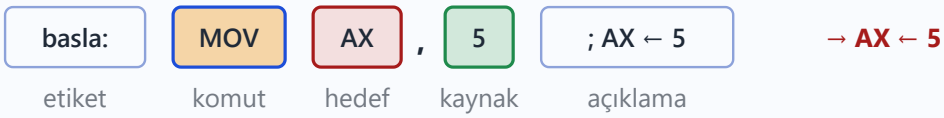
3.2.2 Komut satırının yapısı

3.2.2.1 Satırın alanları

Assembly kaynağı satır satır okunur. Her satır en çok bir komut ya da bir direktif taşır. Bir komut satırının genel biçimi şöyledir; köşeli ayraç içindeki alanlar isteğe bağlıdır:

```
[etiket:]  komut  [hedef], [kaynak]  [; açıklama]
```

Şekil 3.2.2 bu alanları `basla: MOV AX, 5 ; AX ← 5` satırı üzerinde gösterir. **Etiket** satırın adresine verilen addır. **Komut** alanı anımsatıcıyı taşır. Komuttan sonra virgülle ayrılmış en çok iki **işlenen** gelir. Noktalı virgülden satır sonuna kadar olan kısım **açıklamadır**.



Şekil 3.2.2: Komut satırının alanları. `basla` etiketi, `MOV` komutu, `AX` hedef işlenen, `5` kaynak işlenen, noktalı virgülden sonrası açıklamadır. Satırın etkisi AX ← 5 atamasıdır.

Tanım 3.2.2

İşlenen (*operand*), bir komutun üzerinde işlem yaptığı değer, yazmaç ya da bellek konumudur. İki işlenenli komutlarda Intel söz diziminde soldaki işlenen **hedef**, sağdaki **kaynaktır**. Komut kaynağı okur, sonucu hedefe yazar. Kaynak değişmez; hedefin eski değeri kaybolur.

Hedef–kaynak sırası, assembly öğrenen herkesin ilk takıldığı yerdir. `MOV AX, BX` "AX'i BX'e taşı" diye değil, "AX ← BX" diye okunur. Bu sıra C'deki atama ile aynıdır: `ax = bx;`. İki işlenenli aritmetik komutlar da aynı kuralı izler. `SUB CX, AX` çalışınca CX ← CX − AX olur; AX değişmez. Böylece hedef işlenen hem birinci girdi hem de sonucun yazıldığı yerdir.

Her komut iki işlenen almaz. Tablo 3.2.4 farklı sayıda işlenen alan üç satırı alanlarına ayırır.

Satır	Etiket	Komut	İşlenenler	Açıklama
`basla: MOV AX, 5 ; AX ← 5`	`basla`	`MOV`	`AX` , `5`	`; AX ← 5`
`INC AX`	—	`INC`	`AX`	—
`RET`	—	`RET`	yok	—

Tablo 3.2.4: Üç satırın alanları.

`INC AX` tek işlenenlidir: AX hem kaynak hem hedeftir ve AX ← AX + 1 olur. `RET` hiç işlenen yazmaz, ama yine de iş yapar: yığından bir sözcük okuyup IP'ye koyar. Yazılmayan bu işlenenler örtük işlenenler başlığında ayrıntılı ele alınır.

Assembler büyük-küçük harfe duyarsızdır. `mov ax, bx`, `MOV AX, BX` ve `Mov Ax, Bx` aynı makine kodunu verir. Boşluk ve sekme sayısı da önemsizdir; alanları alt alta hizalamak yalnız okunabilirlik içindir. Açıklamalar ise makine koduna hiç girmez. Assembler noktalı virgülü görünce satırın geri kalanını atlar. Bu yüzden açıklama, programın boyutunu ya da hızını etkilemeden istenildiği kadar uzun yazılabilir.

Assembly'de açıklamanın önemi yüksek düzeyli dillerden fazladır. `ADD AX, BX` satırı ne yaptığını zaten söyler; neden yapıldığını söylemez. AX'te bir dizi toplamı mı, bir adres mi, bir sayaç mı durduğu koddan anlaşılmaz. İyi bir açıklama "AX ← AX + BX" diye komutu yinelemez, "toplama bu öğrencinin notunu ekle" gibi amacı yazar.

3.2.2.2 Komut, direktif ve makro

Bir assembly kaynağındaki her satır makine kodu üretmez. Satırlar üç türe ayrılır.

Tanım 3.2.3

Komut (*instruction*), işlemcinin çalışma anında yürüttüğü bir işlemdir ve tek bir makine komutuna çevrilir. **Direktif** (*directive*), assembler'a verilen bir bilgidir: kodun nereye yerleşeceğini, ne kadar yer ayrılacağını, bir adın neyi göstereceğini bildirir. Direktif makine komutu üretmez; veri tanımı direktifleri yalnız veri baytı üretir. **Makro** (*macro*) ise derleme anında yerine birkaç satır yazılan, adı olan bir metin kalıbıdır.

Komut ile direktif arasındaki fark zamanlamadadır. Komutu, program çalışırken işlemci yürütür. Direktif ise yalnız derleme anında anlam taşır; program çalışmaya başladığında ondan geriye hiçbir iz kalmaz. Tablo 3.2.5 sık kullanılan satırları türleri ve bellekte ürettikleriyle verir.

Satır	Türü	Bellekte ne üretir?
`MOV AX, 5`	komut	makine kodu: `B8 05 00`
`ORG 100h`	direktif	hiçbir şey; konum sayacını 0100h yapar
`sayi DB 0`	direktif (veri tanımı)	1 bayt veri: `00`
`k EQU 5`	direktif	hiçbir şey; `k` adı 5 değerini alır
`END`	direktif	hiçbir şey; kaynak burada biter
`PUTC 'A'`	makro çağrısı	makro gövdesindeki komutların kodu

Tablo 3.2.5: Satır türleri ve bellekte ürettikleri.

Direktiflerin işleyişini anlamak için assembler'ın **konum sayacını** (*location counter*) bilmek gerekir. Assembler kaynağı yukarıdan aşağı okurken bir sonraki baytın yerleşeceği bağıl konumu bir sayaçta tutar. Her komut ya da veri tanımı, ürettiği bayt sayısı kadar sayacı ilerletir. `ORG 100h` sayacı doğrudan 0100h'ye kurar; kendisi bayt üretmez. `EQU` ise sayaca hiç dokunmaz: yalnız bir adı bir sabite bağlar. Assembler `k` adını gördüğü her yerde 5 yazar. `sayi DB 0` satırı sayacın o anki değerini `sayi` adına bağlar, bir bayt ayırır ve içine 0 koyar.

Aşağıdaki küçük program bu farkı somutlaştırır. Sağdaki açıklamalar assembler'ın liste çıktısındaki adresleri ve baytları gösterir.

```
ORG 100h            ; bayt yok; konum sayacı = 0100h
k EQU 5             ; bayt yok; k = 5
MOV AX, 5           ; 0100h: B8 05 00
MOV BX, k           ; 0103h: BB 05 00  (k yerine 5)
MOV AL, sayi        ; 0106h: A0 0A 01  (sayi = 010Ah)
RET                 ; 0109h: C3
sayi DB 0           ; 010Ah: 00
END                 ; bayt yok
```

Sekiz satırdan yalnız beşi bayt üretir ve program 11 bayttır. `MOV AX, 5` ile `MOV BX, k` satırlarının kodları yazmaç numarası dışında aynıdır, çünkü assembler `k` yerine 5 koymuştur. `EQU` ile tanımlanan bir sabit bellekte yer kaplamaz. `MOV AL, sayi` satırında ise durum farklıdır. `sayi` bir bellek konumudur; komut o konumdaki baytı okur. Assembler komutun içine konumun adresini yazar: 010Ah, düşük bayt önce `0A 01`. Bu adres, assembler `sayi` satırına gelene kadar bilinmez. Assembler'ın ileriye yapılan bu başvuruyu nasıl çözdüğü iki geçişli çeviriyle açıklanır; burada önemli olan, adresin derleme anında hesaplanıp komutun içine gömülmesidir.

Makro üçüncü türdür. emu8086 ile gelen `emu8086.inc` kütüphanesindeki `PUTC 'A'`, ekrana bir karakter yazan birkaç komutluk bir kalıbın adıdır. Assembler bu satırı gördüğünde makronun gövdesini satırın yerine kopyalar ve kopyayı çevirir. Makro çağrısı bir alt program çağrısı değildir: her kullanımda gövde yeniden yazılır ve program o kadar büyür. Makrolar ayrı bir konudur; burada yalnız bir satırın birden çok makine komutu üretebildiği durumu göstermek için anılır.

Sık yapılan hata

EQU sabitini değişken sanmak. `k EQU 5` tanımından sonra `MOV k, 7` yazmak, "5 ← 7" anlamına gelir ve assembler hata verir. EQU adı bellekte yer kaplamaz; değeri derleme anında sabitlenir ve çalışma anında değiştirilemez. Değeri değişecek bir büyüklük `DB` ya da `DW` ile tanımlanır.

3.2.2.3 Etiketler ve adlar

Assembly programında adresler sayı yerine adlarla yazılır. Bir komutun adresine verilen ada **kod etiketi**, bir veri alanına verilen ada **değişken adı** denir. İkisi de assembler'ın sembol tablosunda bir adrese bağlanır; ama kullanılışları farklıdır.

```
basla:  MOV CX, 3   ; kod etiketi
don:    INC AX
        LOOP don    ; don = adres
        RET
sayi    DB 7        ; veri adı
```

Kod etiketi iki nokta üst üste ile biter. Etiket kendisi makine kodu üretmez; yalnız kendisinden sonra gelen komutun adresini, yani CS içindeki bağıl konumunu temsil eder. Bu programın `ORG 100h` ile başladığı varsayılırsa `basla` = 0100h olur. `MOV CX, 3` üç bayttır (`B9 03 00`); `don` = 0103h olur. `INC AX` tek bayttır (`40`), `LOOP don` iki bayttır (`E2 FD`). `RET` 0106h'de, `sayi` ise 0107h'dedir.

`LOOP don` komutunun ikinci baytı FDh dikkat çekicidir. Komut adresi değil, **uzaklığı** saklar. LOOP yürütülürken IP zaten sonraki komutu, yani 0106h'yi gösterir. Hedef 0103h olduğundan uzaklık 0103h − 0106h = −3'tür. −3, 8 bitlik ikiye tümleyende FDh olarak yazılır. Etiket programcıyı bu hesaptan kurtarır: programcı `don` yazar, assembler uzaklığı hesaplar. Programa yeni bir komut eklenip adresler kaydığında da hesap kendiliğinden yenilenir.

Değişken adı iki nokta üst üste olmadan yazılır. Assembler bir değişken adı için üç bilgi saklar: adresi, kesimi ve **türü**. `DB` ile tanımlanan bir adın türü bayt, `DW` ile tanımlananın türü sözcüktür. Tür bilgisi assembler'ın boyut denetimi yapmasını sağlar. `sayi DB 7` tanımından sonra `MOV AX, sayi` yazılırsa assembler, 8 bitlik bir değişkenin 16 bitlik bir yazmaca taşınmak istendiğini fark eder ve hata verir.

Adlar için kurallar şunlardır:

- Ad bir harfle ya da alt çizgiyle başlar; rakamla başlayamaz. Rakamla başlayan her dizi assembler için bir sayıdır.
- Adın devamında harf, rakam ve alt çizgi bulunabilir.
- Komut, yazmaç ve direktif adları (MOV, AX, DB, END …) ad olarak kullanılamaz; bunlar **ayrılmış sözcüklerdir**.
- Büyük-küçük harf ayrılmaz. `don` ile `DON` aynı etikettir; ikisini birden tanımlamak “iki kez tanımlı” hatası verir.

Ad	Geçerli mi?	Neden?
`_sayac`	**evet**	alt çizgiyle başlar
`toplam2`	**evet**	harfle başlar, rakam sonra gelir
`2toplam`	**hayır**	rakamla başlar; assembler geçersiz bir sayı sanar
`cx`	**hayır**	yazmaç adıdır (CX ile aynı)
`son-deger`	**hayır**	– bir işleçtir; assembler `son` eksi `deger` diye okur

Tablo 3.2.6: Ad örnekleri.

Son satırdaki durum öğreticidir. `son-deger` bir ifade olarak çözümlenir. Programda `son` ve `deger` adları yoksa assembler “tanımsız sembol” hatası verir; varsa iki adresin farkını hesaplar ve programcının hiç istemediği bir sayı üretir. Birden çok sözcükten oluşan adlarda bu yüzden alt çizgi kullanılır: `son_deger` .

Sık yapılan hata

`Sayi` ile `sayi`'yi iki ayrı değişken sanmak. C ve Java büyük-küçük harfi ayırır; 8086 assembler'ları ayırmaz. `sayi DB 7` tanımından sonra `MOV Sayi, 3` yazan programcı yeni bir değişken yarattığını düşünebilir, ama komut aynı bayta 3 yazar ve 7 kaybolur. Assembler bunu hata saymaz, çünkü satır geçerlidir. Farklı büyüklükler farklı adlarla adlandırılır.

3.2.3 İşlenenler

3.2.3.1 Yazmaç, anlık değer ve bellek

Bir komutun işleneni üç yerden gelebilir: işlemcinin içindeki bir yazmaçtan, komutun kendi baytlarından ya da bellekten. Bu üç tür hem söz diziminde hem donanımda farklı davranır.

Tür	Örnek	Değer nereden gelir?	Kısıt
Yazmaç	`MOV AX, BX`	işlemcinin içindeki yazmaç dosyasından	en hızlı tür
Anlık değer	`MOV AX, 5`	komutun baytlarının içinden	yalnız kaynak olabilir
Bellek	`MOV AX, [BX]`	bellekten, bir yol döngüsüyle	bir komutta en çok bir tane

Tablo 3.2.7: Üç işlenen türü.

Yazmaç işleneni işlemcinin içindedir. 8086'nın yürütme biriminde (EU) sekiz genel amaçlı 16 bitlik yazmaç bulunur: AX, BX, CX, DX, SP, BP, SI ve DI. İlk dördü ayrıca 8 bitlik yarılarıyla kullanılabilir: AH ile AL, BH ile BL, CH ile CL, DH ile DL. Yazmaç işlenenli bir komut dış yola hiç çıkmaz. Değer iç yoldan ALU'ya gider, sonuç yine iç yoldan yazmaca döner.

Anlık değer (*immediate value*) komutun içine gömülü bir sabittir. `MOV AX, 5` komutunun `05 00` baytları bu sabittir. Değer komutla birlikte getirildiği için ayrıca okunması gerekmez; komut kuyruğunda zaten hazırdır. Anlık değer yalnız kaynak olabilir, çünkü bir sabite yazmak anlamsızdır: `ADD 5, AX` yazımı “5'in yeni değeri 5 + AX olsun” demektir.

Bellek işleneni bir adresle verilir. `MOV AX, [BX]` komutunda köşeli ayraç “BX'in içindeki sayı bir adrestir; o adresteki sözcüğü al” anlamına gelir. İşlemci önce adresi hesaplar, sonra yol arayüz birimine (BIU) bir okuma isteği verir. BIU adresi adres yoluna koyar, bellekten veriyi okur ve yürütme birimine iletir. Bu dış erişim, yazmaç işlenenine göre en az dört saat turu ek süre getirir.

Üç tür arasındaki temel kurallar şunlardır. İki işlenen aynı boyutta olmalıdır: 8 bitlik bir yazmaca 16 bitlik bir yazmaç taşınamaz. Bir komutta en çok bir bellek işleneni bulunabilir; `MOV [BX], [SI]` geçersizdir. Kesim yazmaçlarına (CS, DS, ES, SS) anlık değer yüklenemez. Bu kuralların hepsi 8086'nın komut kodlamasından gelir. Komut biçiminde bellek adresini tanımlayan alan bir tanedir; ikinci bir bellek işlenenini kodlayacak yer yoktur. Kesim yazmacına sabit yükleyen bir işlem kodu da tasarlanmamıştır.

Örnek 3.2.2

Aşağıdaki komutların işlenen türlerini belirleyin: `ADD CL, 10`, `MOV [BX], AX`, `SUB DX, sayac`.

`ADD CL, 10`: hedef yazmaç (CL), kaynak anlık değer (10). CL ← CL + 10. Bellek erişimi yoktur.

`MOV [BX], AX`: hedef bellek, kaynak yazmaç. AX'in iki baytı, BX'in gösterdiği adrese düşük bayt önce yazılır. Bir yol döngüsü (yazma) gerekir.

`SUB DX, sayac`: hedef yazmaç, kaynak bellek (`sayac` bir değişken adıdır, assembler onu doğrudan adrese çevirir). `sayac` bir `DW` değişkeniyse DX ← DX − [sayac] olur; `DB` değişkeniyse boyut uyuşmazlığı hatası çıkar.

3.2.3.2 Örtük işlenenler

Bazı komutlar yazılan işlenenlerin yanında, komut satırında hiç görünmeyen yazmaçları da kullanır ya da değiştirir. Bu yazmaçların hangileri olduğu komutun tanımına gömülüdür.

Tanım 3.2.4

Örtük işlenen (*implicit operand*), komut satırında yazılmadığı hâlde komutun okuduğu ya da değiştirdiği yazmaç veya bellek konumudur. Örtük işlenenler komutun işlem kodunda sabittir; programcı onları seçemez, yalnız bilmek zorundadır.

Komut	Yazılan işlenen	Örtük işlenen(ler)	İşlem
`MUL BL`	BL	AL (çarpılan), AX (sonuç)	AX ← AL × BL
`DIV BL`	BL	AX (bölünen), AL (bölüm), AH (kalan)	AL ← AX ÷ BL, AH ← AX mod BL
`LOOP don`	don	CX (sayaç)	CX ← CX − 1; CX ≠ 0 ise atla
`CBW`	—	AL → AX	AL'nin işaret biti AH'nin bütün bitlerine kopyalanır
`PUSH AX`	AX	SS:SP (yığın tepesi)	SP ← SP − 2; [SS:SP] ← AX
`XLAT`	—	AL (indis ve sonuç), BX (tablo adresi)	AL ← [DS:BX + AL]

Tablo 3.2.8: Bazı komutların örtük işlenenleri.

Tablodaki örtük kullanımlar 8086 yazmaçlarının adlarının da kaynağıdır. AX **akümülatördür**: çarpma ve bölmenin sonucu her zaman oraya yazılır. CX **sayaç** yazmacıdır: LOOP onu kendiliğinden azaltır, çok bitli kaydırmalar sayıyı onun alt yarısı CL'den okur. BX **taban** yazmacıdır: XLAT tablonun başı-

nı orada arar. SP yığının tepesini gösterir ve her PUSH ile POP'ta kendiliğinden değişir. Yazmaçların "özel görevleri" aslında bu komutların donanımda sabit bağlantılarıdır. Çarpma devresinin bir girişi doğrudan AL'ye, çıkışı doğrudan AX'e bağlıdır.

Örtük işlenen bilgisinin pratik sonucu şudur: bir komut yazılmayan bir yazmacı da değiştirebilir. `MUL BL` satırında AH hiç geçmez, ama komut AH'yi siler, çünkü 16 bitlik sonuç AX'in tamamına yazılır. AH'de saklanan bir değer bu satırdan sonra kaybolur. Bir yazmacın değerini korumak isteyen program, kullandığı komutların örtük işlenenlerini bilmek zorundadır.

Örnek 3.2.3

AX = 100 (0064h) ve BL = 7 iken `DIV BL` çalışıyor. AL, AH ve AX ne olur?

8 bitlik bölmede bölünen örtük olarak AX'in tamamıdır, bölen yazılan BL'dir. 100 ÷ 7 işleminin bölümü 14, kalanı 2'dir (7 × 14 = 98, 100 − 98 = 2). Bölüm AL'ye, kalan AH'ye yazılır: AL = 0Eh, AH = 02h. İki yarı birlikte okunduğunda AX = **020Eh** olur. AX'in bölmeden önceki değeri kaybolmuştur. Bölüm 8 bite sığmasaydı, örneğin AX = 1000 ve BL = 2 olsaydı, işlemci bir bölme hatası kesmesi (INT 0) üretirdi.

Örnek 3.2.4

AL = F6h iken `CBW` çalışıyor. AX ne olur? Aynı işi `MOV AH, 0` yapsaydı fark ne olurdu?

F6h ikiye tümleyende −10'dur; işaret biti (bit 7) 1'dir. CBW bu biti AH'nin sekiz bitine kopyalar: AH = FFh, AX = **FFF6h**. FFF6h de 16 bitte −10'dur; işaretli değer korunmuştur. `MOV AH, 0` ise AX = 00F6h = 246 verir. Bu sonuç, AL işaretsiz bir sayıysa doğrudur, işaretliyse yanlıştır. CBW'nin yazılan hiçbir işleneni yoktur; hem kaynağı (AL) hem hedefi (AX) örtüktür.

3.2.3.3 Kod izleme: MUL, SHR ve LOOP

Aşağıdaki program üç örtük işlenenli komutu art arda kullanır. Programda CX yalnız bir kez yazılır, AX ise yalnız ADD satırında yazılıyor gibi görünür. İzleme, bu yazmaçların gerçekte kaç kez değiştiğini gösterir.

```
ORG 100h
MOV AL, 12
MOV BL, 10
MUL BL          ; AX = AL × BL
MOV CL, 2
SHR AX, CL      ; sayı CL'de
MOV CX, 3       ; LOOP sayacı
don:
ADD AX, 1
LOOP don        ; CX azalır
RET
```

Programın başında .COM programlarının başlangıç durumu geçerlidir: genel yazmaçlar 0, durum bayrakları 0'dır. Tablo 3.2.9 her komuttan sonraki durumu verir. Adres sütunu komutun bağıl konumu, bayt sütunu makine kodudur.

Adres	Baytlar	Komut	AX	BX	CX	CF	ZF	Açıklama
0100h	`B0 0C`	`MOV AL, 12`	000Ch	0000h	0000h	0	0	AL = 12
0102h	`B3 0A`	`MOV BL, 10`	000Ch	000Ah	0000h	0	0	BL = 10
0104h	`F6 E3`	`MUL BL`	0078h	000Ah	0000h	0	?	AX ← AL × BL = 120
0106h	`B1 02`	`MOV CL, 2`	0078h	000Ah	0002h	0	?	kaydırma sayısı
0108h	`D3 E8`	`SHR AX, CL`	001Eh	000Ah	0002h	0	0	120 ÷ 4 = 30
010Ah	`B9 03 00`	`MOV CX, 3`	001Eh	000Ah	0003h	0	0	CL'deki 2 silindi
010Dh	`05 01 00`	`ADD AX, 1`	001Fh	000Ah	0003h	0	0	1. tur
0110h	`E2 FB`	`LOOP don`	001Fh	000Ah	0002h	0	0	CX ≠ 0 → don
010Dh		`ADD AX, 1`	0020h	000Ah	0002h	0	0	2. tur
0110h		`LOOP don`	0020h	000Ah	0001h	0	0	CX ≠ 0 → don
010Dh		`ADD AX, 1`	0021h	000Ah	0001h	0	0	3. tur
0110h		`LOOP don`	0021h	000Ah	0000h	0	0	CX = 0 → düşer
0112h	`C3`	`RET`	0021h	000Ah	0000h	0	0	sonuç 33

Tablo 3.2.9: Örtük işlenen programının izlemesi (her satır, komut yürütüldükten sonraki durum).

MUL adımı. `MUL BL` satırında yalnız BL yazılıdır. Çarpılan örtük olarak AL'dir (12), sonuç örtük olarak AX'tir. 12 × 10 = 120 = 0078h. Sonuç 8 bite sığdığı için AH = 00h olur ve CF = OF = 0 kalır. 8086'da MUL bu iki bayrağı "sonucun üst yarısı anlamlı mı" sorusuna göre kurar. SF, ZF, AF ve PF ise MUL'den sonra **tanımsızdır**: işlemci onlara bir değer yazar, ama Intel bu değeri belgelemez. Tabloda ZF için soru işareti bu yüzden konmuştur. Programın ZF'ye güvenmesi için ardından bayrağı tanımlı biçimde kuran bir komut gelmesi gerekir.

SHR adımı. 8086'da bir yazmacı birden fazla bit kaydırmak için kaydırma sayısı yalnız CL'de verilebilir. `SHR AX, 2` biçimi 80186 ile geldi. Program bu yüzden önce `MOV CL, 2` yazar. Sağa kaydırma her bitte değeri ikiye böler: 0078h = 0111 1000b iki bit sağa kayınca 0001 1110b = 001Eh = 30 olur.

Kaydırmada son dışarı çıkan bit CF'ye yazılır. Çıkan iki bit 0 ve 0 olduğu için CF = 0'dır. Sonuç sıfır olmadığından ZF = 0 olur. Bu komutla ZF yeniden tanımlı hâle gelmiştir.

MOV CX, 3 adımı. Bu satır CX'e 3 yazar. CL, CX'in alt yarısıdır; bir önceki adımda CL'ye yazılan 2 böylece silinir. Kaydırma bittiği için kayıp önemsizdir, ama sıra ters olsaydı, yani CX önce kurulup sonra CL'ye 2 yazılsaydı, döngü 3 kez değil 2 kez dönerdi. Aynı fiziksel yazmacın iki adla anılması, örtük işlenenlerle birleşince sık hata kaynağı olur.

LOOP adımları. `LOOP don` komutu iki işi tek komutta yapar: CX'i 1 azaltır ve CX sıfır değilse `don` etiketine atlar. Programda CX'i azaltan hiçbir satır yoktur, ama izleme CX'in 3 → 2 → 1 → 0 diye üç kez değiştiğini gösterir. LOOP'un azaltması bayrakları etkilemez; LOOP bayrak değiştirmeyen bir komuttur. Her turda `ADD AX, 1` AX'i bir artırır: 30 → 31 → 32 → 33. Üçüncü LOOP'ta CX sıfıra iner, atlama yapılmaz ve akış RET'e düşer.

Programın sonunda AX = 0021h = 33'tür. Sonuç satır satır okununca şaşırtıcı değildir, ama programda AX'in yalnız ADD satırında yazıldığını düşünen biri MUL ve SHR'nin AX'i değiştirdiğini kaçırır. Şekil 3.2.3 izlemenin üç anını sunumdaki çizimlerle gösterir.



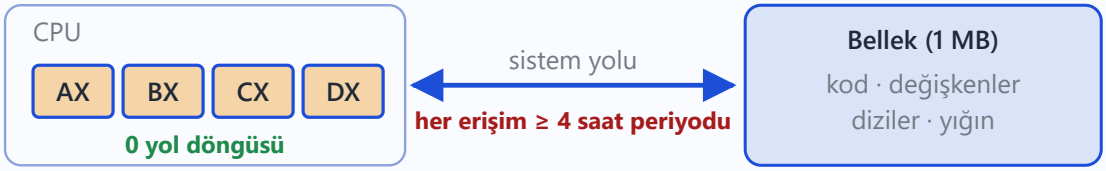
Şekil 3.2.3: Örtük işlenen izlemesinin üç anı. Üstte `MUL BL` sonrası: AX = 0078h, ZF tanımsız. Ortada `SHR AX, CL` sonrası: AX = 001Eh, CX = 0002h. Altta RET'te: AX = 0021h, CX = 0000h. Vurgulu kutular o adımda değişen yazmaçlardır.

Sık yapılan hata

MUL'ün AH'yi koruduğunu sanmak. AH'de bir sayaç ya da ara sonuç tutan bir program `MUL BL` çalıştırınca bu değeri kaybeder. 16 bitlik `MUL BX` ise daha da geniş bir alanı değiştirir: sonuç DX:AX çiftine yazılır ve DX'in eski değeri de silinir. Saklanması gereken yazmaç MUL'den önce başka bir yazmaca ya da yığına alınır.

3.2.3.4 Yazmaç mı, bellek mi?

Bir değer ya yazmaçta ya bellekte tutulur. İkisi arasındaki seçim, programın hızını en çok etkileyen karardır. Şekil 3.2.4 iki yerin işlemciye uzaklığını gösterir.



Şekil 3.2.4: Yazmaçlar CPU'nun içinde, bellek sistem yolunun öbür ucundadır. Yazmaca erişim yol döngüsü gerektirmez; her bellek erişimi en az dört saat turu süren bir yol döngüsüdür.

8086'nın programcıya görünen **14 yazmacı** vardır: 8 genel amaçlı yazmaç (AX, BX, CX, DX, SP, BP, SI, DI), 4 kesim yazmacı (CS, DS, ES, SS), komut işaretçisi IP ve bayrak yazmacı FLAGS. Hepsi 16 bittir. Yazmaçlar flip-flop'lardan yapılmış küçük ve çok hızlı bellek hücreleridir. Bir yazmacın değeri iç yoldan doğrudan ALU'ya ulaşır; dış hatlara hiç çıkmaz.

Bellek ise CPU'nun dışındadır ve 8086'da 1 MB'a kadar büyüyebilir. Program kodu, değişkenler, diziler ve yığın burada durur. Bellekten bir sözcük okumak için işlemcinin bir **yol döngüsü** (*bus cycle*) yürütmesi gerekir.

Tanım 3.2.5

Yol döngüsü, işlemcinin sistem yolu üzerinden bir bellek ya da G/Ç konumunu okuduğu veya yazdığı işlemdir. 8086'da bir yol döngüsü en az dört **T durumundan** (T1–T4) oluşur; her T durumu bir saat turudur. Bellek yavaşsa T3 ile T4 arasına bekleme durumları (Tw) eklenir.

Bir okuma döngüsünde T1'de işlemci 20 bitlik adresi AD0–AD15 ve A16–A19 hatlarına koyar ve ALE sinyaliyle adresin dış kilitlere yakalanmasını sağlar. T2'de aynı hatlar veri için serbest bırakılır ve RD sinyali etkin olur. T3'te bellek veriyi veri yoluna koyar. T4'te işlemci veriyi alır ve sinyaller geri çekilir. Yazmaç işleneninde bu dört adımın hiçbiri yoktur. Bellek erişimi bu yüzden yazmaç erişiminden kat kat uzundur.

Bellek erişiminin bir bedeli daha vardır. 8086'nın veri yolu 16 bittir ve bir yol döngüsünde çift adresten başlayan bir sözcüğü bütün olarak okuyabilir. Sözcük tek adreste başlıyorsa iki ayrı yol döngüsü gerekir: biri düşük bayt, biri yüksek bayt için. Bu durumda erişim dört saat turu daha uzar.

Bu fark iyi assembly kodunun kuralını belirler: sık kullanılan değerler yazmaçlarda tutulur, belleğe yalnız gerektiğinde gidilir. Bir döngü sayacını bellekte değil CX'te tutmak döngüyü hızlandırır. Bir dizinin elemanları bellekte durmak zorundadır, ama elemanların toplamı döngü boyunca bir yazmaçta biriktirilip en sonda belleğe bir kez yazılabilir.

3.2.3.5 Komut süreleri

Intel 8086 veri sayfası her komutun kaç saat turu sürdüğünü verir. Bellek işlenenli komutlarda bu süreye, adresin hesaplanması için bir **etkin adres** (EA) süresi eklenir. Etkin adres, bellek işleneninin kesim içindeki bağıl konumudur; doğrudan bir sabit, bir yazmaç ya da yazmaç ve sabit toplamı olabilir.

Komut	İşlenenler	Saat turu
`MOV AX, BX`	yazmaç, yazmaç	2
`MOV AX, 5`	yazmaç, anlık	4
`MOV AX, sayi`	akümülatör, doğrudan bellek	10
`MOV CX, [BX]`	yazmaç, bellek	8 + 5 = 13
`ADD AX, BX`	yazmaç, yazmaç	3
`ADD AX, sayi`	yazmaç, bellek	9 + 6 = 15
`INC CX`	16 bitlik yazmaç	2
`INC sayac`	bellek (sözcük)	15 + 6 = 21

Tablo 3.2.10: Bazı 8086 komutlarının süreleri (en iyi durum, saat turu).

Adres biçimi	Örnek	EA süresi (tur)
yalnız taban ya da indis	`[BX]`, `[SI]`	5
doğrudan adres	`sayi`, `[0200h]`	6
taban + indis	`[BX+SI]`, `[BP+DI]`	7
taban + indis	`[BX+DI]`, `[BP+SI]`	8
taban ya da indis + sabit	`[BX+4]`	9

Tablo 3.2.11: Etkin adres hesabının süresi.

Tablodaki farkların nedeni donanımdır. `MOV AX, BX` bir iç aktarımdır: 2 tur. `MOV AX, 5` 4 tur sürer, çünkü 16 bitlik sabit komut kuyruğundan çekilmelidir. `MOV CX, [BX]` önce etkin adresi hesaplar (5 tur), sonra bir yol döngüsü yürütür. `MOV AX, sayi` özel bir durumdur. AX'e doğrudan adresten yükle-

me, adresi komutun içinde taşıyan kısa **akümülatör biçimini** (`A1` işlem kodu) kullanır ve EA hesabı gerektirmez: 10 tur. Belleğe yazan komutlar en uzunlarıdır. `INC sayac` önce sözcüğü okur, ALU'da artırır, sonra geri yazar; iki yol döngüsü içerir ve 21 tur sürer.

Tablodaki değerler en iyi durum süreleridir. Tek adresteki sözcük erişimi her erişime 4 tur ekler. Bir atlamadan sonra komut kuyruğu boşalmışsa işlemci baytların gelmesini bekler. Kesim öneki kullanılırsa 2 tur daha eklenir. Gerçek süre bu yüzden tablodakinden uzun olabilir. Oran ise değişmez: bellek işlenenli bir komut, yazmaç işlenenli eşinden birkaç kat yavaştır.

Örnek 3.2.5

5 MHz'lik bir 8086'da 1000 kez dönen bir döngüde sayaç her turda bir kez artırılıyor. Sayaç CX'te ve bellekte tutulduğunda artırma işlemleri toplam ne kadar sürer?

5 MHz'de bir saat turu $1 / (5 \times 10^6)$ s = 0,2 µs'dir. Sayaç CX'teyse her artırma `INC CX` ile 2 tur sürer: 1000 × 2 = 2000 tur = 2000 × 0,2 µs = **0,4 ms**. Sayaç bellekteyse her artırma `INC sayac` ile 21 tur sürer: 1000 × 21 = 21.000 tur = **4,2 ms**. Yalnız sayacın yeri değişmiş, süre on kattan fazla artmıştır. Döngünün öteki komutları aynı kaldığı için toplam süredeki fark da 3,8 ms'dir.

Örnek 3.2.6

`MOV AX, 5` , `MOV BX, 3` , `ADD AX, BX` , `MOV CL, AL` dizisi kaç saat turu sürer? İkinci satır `MOV BX, sayi` (doğrudan adres, sözcük) olsaydı ne değişirdi?

Yazmaça anlık değer 4, yazmaçtan yazmaca toplama 3, 8 bitlik yazmaçtan yazmaca MOV 2 tur sürer: 4 + 4 + 3 + 2 = **13 tur**; 5 MHz'de 2,6 µs. İkinci satır bellekten okursa, hedef AX değil BX olduğu için akümülatör biçimi kullanılamaz; süre 8 + 6 = 14 tur olur. Toplam 4 + 14 + 3 + 2 = **23 tur**, yani 4,6 µs çıkar. Tek bir bellek işleneni dizinin süresini yaklaşık iki katına çıkarmıştır.

Sık yapılan hata

Komut uzunluğu ile süresini karıştırmak. Kısa komut hızlı komut demek değildir. `MUL BL` yalnız iki bayttır (`F6 E3`), ama 70–77 saat turu sürer; üç baytlık `ADD AX, 1234h` (`05 34 12`) ise 4 turdur. Bayt sayısı komutun bellekte kapladığı yeri ve IP'nin ne kadar ilerleyeceğini belirler. Süreyi ise komutun iç işlemi ve yol döngüsü sayısı belirler.

3.2.4 IP ve bayraklar

3.2.4.1 IP nasıl ilerler?

Programlarda IP hiç yazılmaz, ama her komut onu değiştirir. **Komut işaretçisi** (IP, *instruction pointer*), sıradaki komutun kod kesimi içindeki bağıl konumunu tutar. Fiziksel adres CS ile IP'den hesaplanır: CS × 10h + IP. Bu çifte **CS:IP** denir.

> **Tanım 3.2.6**
>
> Sıradan (atlama yapmayan) bir komut yürütüldükten sonra IP, komutun bayt uzunluğu kadar ilerler: $IP_{yeni} = IP_{eski}$ + komut uzunluğu. Atlama, çağrı, dönüş ve kesme komutları ise IP'ye yeni bir değer yazar. IP'ye `MOV` ile yazılamaz.

8086 komutları 1 ile 6 bayt arasında değişen uzunluktadır. IP'nin ne kadar ilerleyeceğini bilmek için her komutun uzunluğunu bilmek gerekir. Aşağıdaki programda uzunluklar açıklama olarak yazılmıştır.

```
ORG 100h
MOV AX, 5      ; 3 bayt
MOV BX, 2      ; 3 bayt
INC AX         ; 1 bayt
ADD AX, BX     ; 2 bayt
RET            ; 1 bayt
```

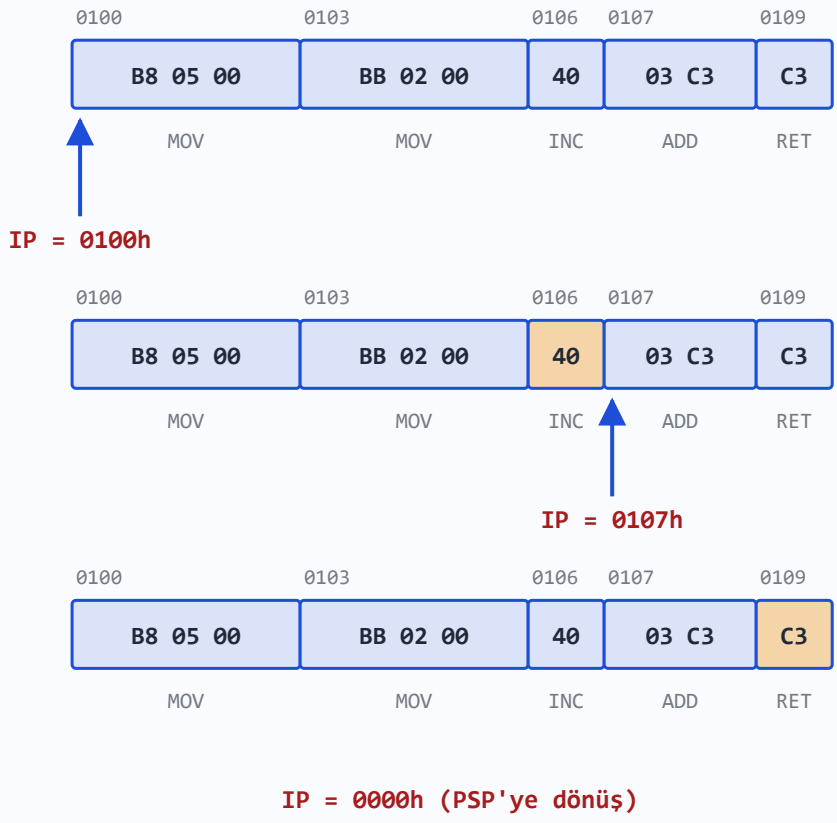
Adım	Komutun adresi	Baytlar	Komut	IP	AX	BX	SP
0	—	—	(yüklendi)	0100h	0000h	0000h	FFFEh
1	0100h	`B8 05 00`	`MOV AX, 5`	0103h	0005h	0000h	FFFEh
2	0103h	`BB 02 00`	`MOV BX, 2`	0106h	0005h	0002h	FFFEh
3	0106h	`40`	`INC AX`	0107h	0006h	0002h	FFFEh
4	0107h	`03 C3`	`ADD AX, BX`	0109h	0008h	0002h	FFFEh
5	0109h	`C3`	`RET`	0000h	0008h	0002h	0000h

Tablo 3.2.12: IP izlemesi (her satır, komut yürütüldükten sonraki durum).

Adım 0'da program kesimin 0100h konumuna yüklenmiştir ve IP = 0100h ilk komutu gösterir. Adım 1'de işlemci `B8` baytını okur, bu işlem kodunun arkasından iki baytlık bir sabit geleceğini anlar ve üç baytı birlikte alır. IP 0103h'ye ilerler; komut yürütülünce AX = 0005h olur. Adım 2'de aynı şey BX için olur: 3 bayt daha, IP = 0106h.

Adım 3'teki `INC AX` tek bayttır. 16 bitlik yazmacı artırma komutlarının kodu 40h + yazmaç numarasıdır; AX için 40h. IP yalnız 1 artar: 0107h. Adım 4'te `ADD AX, BX` iki bayttır: işlem kodu 03h ve işlenenleri tanımlayan bir bayt (C3h). IP = 0109h, AX = 0006h + 0002h = 0008h olur.

Adım 5 sıradan bir komut değildir. `RET` yığının tepesindeki sözcüğü IP'ye alır. .COM programı başladığında yığının tepesinde (SS:FFFEh) DOS'un koyduğu 0000h vardır. RET bu değeri IP'ye yazar ve SP'yi 2 artırır: FFFEh + 2 = 10000h, 16 bite sığmadığı için SP = 0000h olur. IP = 0000h artık programın değil, kesimin başındaki PSP'nin ilk baytını gösterir. Orada `INT 20h` komutu (`CD 20`) durur ve programı bitirir.



Şekil 3.2.5: Komut baytları bellekte ve IP. Üstte program yüklenmiştir, IP = 0100h. Ortada üç komut yürütülmüştür, IP = 0107h. Altta RET yığından 0000h'yi almış ve IP PSP'nin başına dönmüştür. Her kutunun genişliği komutun bayt sayısını gösterir.

IP ile donanım arasındaki ilişki burada bir inceliği gizler. 8086'nın yol arayüz birimi komut baytlarını önceden getirir ve 6 baytlık bir kuyrukta bekletir. BIU'nun bellekten okuduğu adres, yürütülen komutun adresinden birkaç bayt ileridedir. Programcının gördüğü IP ise her zaman **yürütülecek** sıradaki komutun adresidir. Bir atlama olduğunda kuyruk boşaltılır ve BIU yeni IP'den getirmeye başlar. Kuyruk programcıya görünmez; yalnız sürelere yansır.

Örnek 3.2.7

Bir program 0100h'de başlıyor ve uzunlukları 3, 2, 1 ve 4 bayt olan dört komutu sırayla yürütüyor. Komutların hiçbiri atlama değildir. Dördüncü komuttan sonra IP kaçtır? Üçüncü komutun adresi nedir?

IP her komutta uzunluğu kadar ilerler: 0100h → 0103h → 0105h → 0106h → 010Ah. Dört komut toplam 3 + 2 + 1 + 4 = 10 bayttır ve IP = 0100h + 0Ah = **010Ah** olur. Üçüncü komut, ilk iki komutun bittiği yerde, **0105h** adresinde başlar.

Sık yapılan hata

IP'yi komut sayısıyla ilerletmek. "Üç komut yürütüldü, IP 0103h olur" düşüncesi yanlıştır. IP baytları sayar, komutları değil. 8086'da komut uzunlukları değişken olduğu için IP'yi hesaplamak, her komutun uzunluğunu bilmeyi gerektirir. Uzunluklar emu8086'nın bellek görünümünden ya da assembler'ın liste çıktısından okunabilir.

3.2.4.2 Bayrakları hangi komutlar değiştirir?

Bayrak yazmacı (FLAGS), son aritmetik ya da mantık işleminin sonucu hakkında bilgi taşır. Durum bayrakları şunlardır: CF (elde), ZF (sıfır), SF (işaret), OF (taşma), AF (yardımcı elde) ve PF (eşlik). Koşullu atlama komutları bu bayraklara bakarak karar verir. Hangi komutun hangi bayrağı değiştirdiği bu yüzden programın akışını doğrudan belirler.

Temel kural, bayrakların yalnız sonuç *hesaplayan* komutlarca yazılmasıdır. Veri taşıyan komutlar sonuç hesaplamaz; bayraklara dokunmaz. Tablo 3.2.13 komutları bayraklara etkilerine göre gruplar.

Grup	Komutlar	Etki
Veri taşıma	`MOV`, `XCHG`, `LEA`, `PUSH`, `POP`, `CBW`	hiçbir bayrak değişmez
Döngü ve atlama	`JMP`, `Jcc`, `LOOP`, `CALL`, `RET`	hiçbir bayrak değişmez
Değil	`NOT`	hiçbir bayrak değişmez
Artırma, azaltma	`INC`, `DEC`	ZF, SF, OF, AF, PF yeniden yazılır; **CF değişmez**
Toplama, çıkarma, karşılaştırma	`ADD`, `ADC`, `SUB`, `SBB`, `CMP`, `NEG`	altı durum bayrağı yeniden yazılır
Mantık	`AND`, `OR`, `XOR`, `TEST`	CF = OF = 0; ZF, SF, PF sonuca göre
Çarpma	`MUL`, `IMUL`	CF, OF tanımlı; ZF, SF, AF, PF tanımsız
Doğrudan kurma	`STC`, `CLC`, `CMC`	yalnız CF

Tablo 3.2.13: Komutların durum bayraklarına etkisi.

Tablonun iki satırı şaşırtıcıdır. Birincisi NOT'tur. NOT bir mantık işlemidir ve yeni bir değer hesaplar, ama 8086'da hiçbir bayrağı değiştirmez. NOT sonrası sonucu sınamak isteyen program ardından `TEST AL, AL` ya da `CMP` yazmalıdır. İkincisi INC ve DEC'tir. Bu iki komut CF'ye dokunmaz. Tasarımın amacı çok baytlı sayılar üzerinde çalışan döngülerdir: döngü sayacı ya da indis INC/DEC ile değiştirilirken ADC'nin taşıdığı elde kaybolmaz. INC'in bir sayıyı FFFFh'den 0000h'ye geçirdiğini anlamak için CF'ye değil ZF'ye bakılır.

Tanım 3.2.7

Bir koşullu atlamanın baktığı bayrak, ondan önce yürütülen ve o bayrağı **yazan son komutun** sonucunu gösterir. Aradaki MOV, NOT, LOOP gibi bayrak değiştirmeyen komutlar bu ilişkiyi bozmaz; ama aradaki bir INC, ADD ya da CMP bozar.

3.2.4.3 Kod izleme: bayraklar

Aşağıdaki program bayrak kurallarını tek tek sınar. Her satırdaki açıklama o satırın bayraklara etkisini önceden söyler; izleme bu tahminleri doğrular.

```
ORG 100h
STC              ; CF = 1
MOV AL, 0FFh     ; bayraklar aynı
INC AL           ; CF'ye dokunmaz
MOV CX, 0        ; ZF aynı kalır
DEC AL
ADD AL, 1        ; CF yeni
CLC
NOT AL           ; bayraklar aynı
RET
```

Adres	Baytlar	Komut	AL	CF	ZF	SF	OF	AF	PF	Hangi bayraklar yazıldı?
0100h	`F9`	`STC`	00h	1	0	0	0	0	0	yalnız CF
0101h	`B0 FF`	`MOV AL, 0FFh`	FFh	1	0	0	0	0	0	hiçbiri
0103h	`FE C0`	`INC AL`	00h	1	1	0	0	1	1	CF dışındakiler
0105h	`B9 00 00`	`MOV CX, 0`	00h	1	1	0	0	1	1	hiçbiri
0108h	`FE C8`	`DEC AL`	FFh	1	0	1	0	1	1	CF dışındakiler
010Ah	`04 01`	`ADD AL, 1`	00h	1	1	0	0	1	1	hepsi
010Ch	`F8`	`CLC`	00h	0	1	0	0	1	1	yalnız CF
010Dh	`F6 D0`	`NOT AL`	FFh	0	1	0	0	1	1	hiçbiri

Tablo 3.2.14: Bayrak izlemesi (her satır, komut yürütüldükten sonraki durum).

STC ve MOV. `STC` (*set carry*) CF'yi doğrudan 1 yapar. Bayrağı bir işlem sonucu olmadan değiştiren az sayıdaki komuttan biridir. `MOV AL, 0FFh` AL'ye 255 yazar; bayrakların hiçbiri değişmez. MOV bir sonuç hesaplamaz, yalnız kopyalar.

INC AL. FFh + 1 = 100h, 8 bite sığmaz ve AL = 00h olur. Toplama bir ADD ile yapılsaydı bu elde CF'ye yazılırdı. INC ise CF'ye dokunmaz; CF, STC'den kalan 1'dir. Sonuç sıfır olduğu için ZF = 1, bit 3'ten bit 4'e elde geçtiği için AF = 1 olur. 00h'nin bir bitlerinin sayısı sıfırdır, yani çifttir; PF = 1. İşaretli yorumda −1 + 1 = 0 doğru bir sonuçtur, taşma yoktur: OF = 0.

MOV CX, 0. CX'e sıfır yazılır. Sonuç sıfır olduğu hâlde ZF yeniden hesaplanmaz; ZF'deki 1 bir önceki INC'ten kalmıştır. Bu satırdan sonra bir `JZ` gelseydi atlama yapılırdı, ama nedeni CX değil AL olurdu.

DEC AL. 00h − 1 = FFh. En üst bit 1 olduğu için SF = 1, sonuç sıfır olmadığı için ZF = 0 olur. Bit 4'ten ödünç alındığı için AF = 1, FFh'nin sekiz bir biti çift sayıda olduğu için PF = 1'dir. İşaretsiz yorumda 0'dan 1 çıkarmak bir ödünç doğurur. SUB bunu CF'ye yazardı; DEC yazmaz ve CF yine 1 kalır.

ADD AL, 1. FFh + 1 = 100h, AL = 00h. Bu kez toplamayı ADD yapar ve ADD altı durum bayrağının hepsini yeniden yazar. CF = 1 olur; tablodaki değer önceki satırla aynı görünür, ama artık STC'den kalan 1 değil, ADD'in ürettiği eldedir. ZF = 1, SF = 0, AF = 1, PF = 1, OF = 0.

CLC ve NOT. `CLC` CF'yi 0 yapar. `NOT AL` AL'nin bütün bitlerini ters çevirir: 00h → FFh. AL sıfır olmaktan çıktığı hâlde ZF hâlâ 1'dir, çünkü NOT bayrak yazmaz. Bu satırdan sonra AL'nin sıfır olup olmadığını sınamak için `TEST AL, AL` gerekir.



Şekil 3.2.6: Bayrak izlemesinin üç anı. Üstte `INC AL` sonrası: AL = 00h, ZF = 1, CF korunmuştur. Ortada `DEC AL` sonrası: AL = FFh, SF = 1, CF yine korunmuştur. Altta `NOT AL` sonrası: AL = FFh olduğu hâlde ZF = 1 kalmıştır.

Sık yapılan hata

Koşullu atlamadan önce bayrağı kimin kurduğuna bakmamak. `DEC CX` ile `JZ son` arasına bir `MOV AX, [BX]` eklemek akışı bozmaz; MOV bayrak yazmaz. Araya bir `ADD SI, 2` eklemek ise bozar: JZ artık CX'e değil SI'nin toplamına bakar. Atlamadan önce "bu bayrağı en son hangi komut yazdı?" sorusu sorulmalıdır.

3.2.5 İlk program

3.2.5.1 .COM program şablonu

emu8086'da yeni bir dosya açılırken **COM template** seçilirse düzenleyici boş bir .COM şablonuyla açılır. Bir .COM programının iskeleti yalnız üç parçadan oluşur: başlangıç konumunu bildiren `ORG 100h`, komutlar ve programı bitiren `RET`. Değişkenler, varsa, RET'ten sonra gelir.

```
ORG 100h          ; .COM: kod 0100h'den başlar

MOV AX, 5         ; AX = 5
MOV BX, 3         ; BX = 3
ADD AX, BX        ; AX = AX + BX = 8
MOV CL, AL        ; CL = 08h

RET               ; DOS'a dön

sayi DB 0         ; değişkenler RET'ten sonra
```

ORG 100h. DOS bir .COM programını yüklerken önce bir kesim seçer ve bu kesimin ilk 256 baytına (0000h–00FFh) **PSP** (*program segment prefix*, program kesimi öneki) adlı bir başlık yazar. PSP'de programın komut satırı, bitiş adresi ve DOS'un kullandığı başka bilgiler bulunur. Programın baytları PSP'nin hemen arkasına, 0100h'den başlayarak kopyalanır. `ORG 100h` assembler'a bu gerçeği bildirir. Direktif olmasaydı assembler ilk baytı 0000h'de sanar ve bütün değişken adreslerini 256 bayt eksik hesaplardı.

Sıralı yürütme. Komutlar yukarıdan aşağı sırayla yürütülür. Bir atlama komutu yoksa işlemci her komuttan sonra bellekte hemen arkasındaki baytlara geçer. Boş satırlar ve açıklamalar bu sıraya hiçbir şey eklemez.

RET. Program bitince denetimin işletim sistemine dönmesi gerekir. RET bu işi dolaylı yoldan yapar: yığından 0000h'yi IP'ye alır, işlemci PSP'nin başındaki `INT 20h` komutunu yürütür ve DOS programı sonlandırır. RET unutulursa işlemci programın bittiğini bilemez. Komutların arkasındaki baytları, yani değişkenleri ve onların arkasındaki rastgele bellek içeriğini, komut gibi çözmeye devam eder.

Değişkenlerin yeri. Bu yüzden değişkenler kodun sonuna, RET'in arkasına konur. RET'e gelen işlemci geri döner ve değişkenlerin baytlarına hiç ulaşmaz. Değişkenler kodun önüne konmak istenirse ilk komut, verinin üstünden atlayan bir `JMP` olmalıdır.

Başlangıç durumu. .COM programı tek kesimlidir: kod, veri ve yığın aynı 64 KB'tadır. Yükleyici bu yüzden dört kesim yazmacına aynı değeri koyar: CS = DS = ES = SS. emu8086 bu değer olarak 0700h kullanır. IP = 0100h, SP = FFFEh olur; yığın kesimin en sonundan başlayıp aşağı doğru büyür. Yığının tepesinde, FFFEh adresinde, RET'in dönüşünü sağlayan 0000h sözcüğü durur. Genel yazmaçlar sıfırdır.

Yazmaç	Değer	Anlamı
CS, DS, ES, SS	0700h	tek kesim; kesim 07000h fiziksel adresinde başlar
IP	0100h	PSP'nin arkasındaki ilk komut
SP	FFFEh	yığının tepesi; [FFFEh] = 0000h
AX, BX, CX, DX, SI, DI, BP	0000h	—
FLAGS	IF = 1, ötekiler 0	kesmeler açık

Tablo 3.2.15: emu8086'da .COM programının başlangıç durumu.

Şablondaki dört komutun işi basittir: AX'e 5, BX'e 3 yazılır, ikisi toplanır ve sonucun alt baytı CL'ye kopyalanır. Programın amacı bir hesap değil, bir .COM programının bellekte nasıl durduğunu ve nasıl çalıştığını göstermektir.

3.2.5.2 Kod izleme: .COM programı bellekte

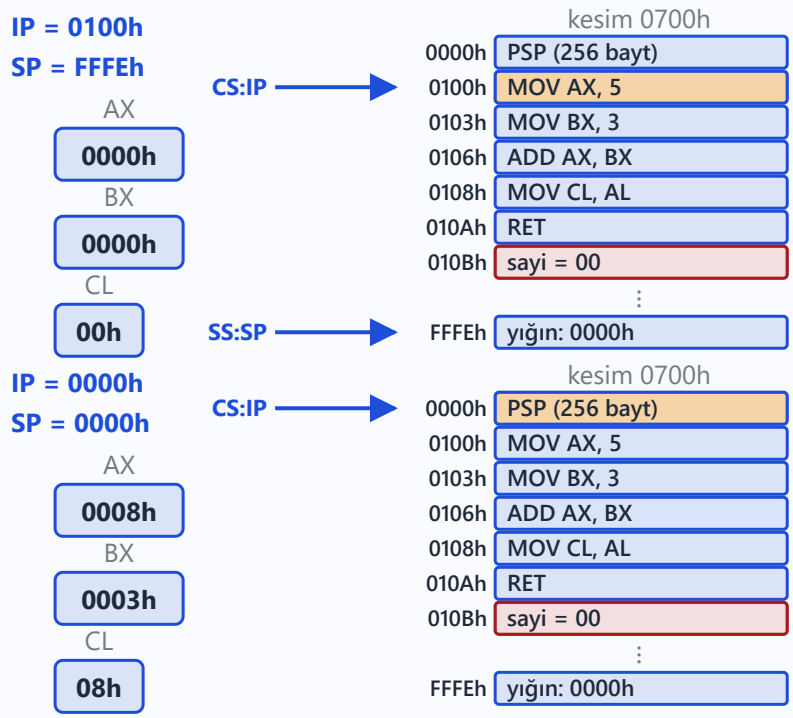
Tablo 3.2.16 şablon programın her komuttan sonraki durumunu verir. Fiziksel adres sütunu, kesim 0700h için 07000h + bağıl konumdur.

Adım	Komut (adres)	Fiziksel adres	AX	BX	CL	IP	SP
0	(yüklendi)	—	0000h	0000h	00h	0100h	FFFEh
1	`MOV AX, 5` (0100h)	07100h	0005h	0000h	00h	0103h	FFFEh
2	`MOV BX, 3` (0103h)	07103h	0005h	0003h	00h	0106h	FFFEh
3	`ADD AX, BX` (0106h)	07106h	0008h	0003h	00h	0108h	FFFEh
4	`MOV CL, AL` (0108h)	07108h	0008h	0003h	08h	010Ah	FFFEh
5	`RET` (010Ah)	0710Ah	0008h	0003h	08h	0000h	0000h
6	`INT 20h` (PSP, 0000h)	07000h	program biter				

Tablo 3.2.16: .COM programının izlemesi (her satır, komut yürütüldükten sonraki durum).

Adım 1–4 sıradan komutlardır ve IP her birinde komut uzunluğu kadar ilerler: 3, 3, 2 ve 2 bayt. Adım 3'teki toplama bayrakları da yazar: 5 + 3 = 8, sonuç sıfır değil (ZF = 0), işaret biti 0 (SF = 0), elde ve taşma yok (CF = OF = 0). 08h'de tek bir 1 biti olduğu için eşlik tektir ve PF = 0'dır. Adım 4'teki `MOV CL, AL` AX'in alt baytını CL'ye kopyalar; CH değişmez ve CX = 0008h olur.

Adım 5'te RET yığından sözcük okur. SS:SP = 0700:FFFEh, yani fiziksel 16FFEh adresindeki sözcük 0000h'dir. Bu değer IP'ye yazılır, SP 2 artarak 0000h'ye döner. Adım 6'da işlemci 0700:0000'daki iki baytı, `CD 20`'yi, yani `INT 20h`'yi yürütür. DOS'un sonlandırma hizmeti programı bellekten kaldırır. 010Bh'deki `sayi` baytı hiçbir adımda yürütülmemiştir; yalnız veridir.



Şekil 3.2.7: .COM programının kesim içindeki yerleşimi. Üstte program yüklenmiştir: PSP 0000h–00FFh, kod 0100h'den, `sayi` 010Bh'de, yığının tepesi FFFEh'dedir; CS:IP ilk komutu, SS:SP yığının tepesini gösterir. Altta RET yürütülmüş, IP = 0000h ile PSP'ye dönülmüştür.

Bu izleme, sistem yolu açısından da okunabilir. Her komutun baytlarını BIU kod kesiminden getirir. Getirme için kullanılan fiziksel adres CS × 10h + IP'dir: 07000h + 0100h = 07100h ilk bayt için. Programın dört aritmetik komutu bellekten veri okumaz; işlenenlerin hepsi yazmaç ya da anlık değerdir. Veri için yol döngüsü yalnız RET'te yapılır: yığından bir sözcük okunur. RET aynı zamanda bir atlamadır; komut kuyruğu boşaltılır ve BIU getirmeye 07000h'den yeniden başlar.

3.2.5.3 Program bellekte: bayt bayt

Assembler'ın şablon programdan ürettiği baytlar Tablo 3.2.17'deki gibidir. Bir .COM dosyası tam olarak bu baytlardan oluşur.

Bağıl konum	Baytlar	Kaynak satırı	Çözümleme
0100h	`B8 05 00`	`MOV AX, 5`	B8 + yazmaç no (AX = 0); sabit düşük bayt önce
0103h	`BB 03 00`	`MOV BX, 3`	B8 + 3 (BX)
0106h	`03 C3`	`ADD AX, BX`	C3 = 11 000 011: yazmaç–yazmaç, AX, BX

Bağıl konum	Baytlar	Kaynak satırı	Çözümleme
0108h	`8A C8`	`MOV CL, AL`	C8 = 11 001 000: CL, AL
010Ah	`C3`	`RET`	tek baytlık komut
010Bh	`00`	`sayi DB 0`	veri; komut değil

Tablo 3.2.17: Şablon programın makine kodu.

İlk iki komut "yazmaca anlık değer" ailesindendir ve işlem kodu yazmaç numarasını içerir. 8086'nın 16 bitlik yazmaç numaraları şöyledir: AX = 0, CX = 1, DX = 2, BX = 3, SP = 4, BP = 5, SI = 6, DI = 7. Sıranın alfabetik olmadığına dikkat edilmelidir; BX'in numarası 3'tür. 8 bitlik yazmaçlarda aynı numaralar AL, CL, DL, BL, AH, CH, DH, BH sırasıyla kullanılır.

Üçüncü ve dördüncü komut iki yazmaç arasında çalışır. Bu komutlarda ilk bayt işlemi, ikinci bayt işlenenleri tanımlar. İkinci bayta **ModR/M baytı** denir ve üç alana bölünür: 2 bitlik **MOD**, 3 bitlik **REG** ve 3 bitlik **R/M**. MOD = 11 "iki işlenen de yazmaç" demektir. `ADD AX, BX` için işlem kodu 03h "REG ← REG + R/M" biçimindedir. REG = 000 (AX) hedefi, R/M = 011 (BX) kaynağı verir. Bitler birleşince 11 000 011 = C3h olur. `MOV CL, AL` için işlem kodu 8Ah "8 bitlik REG ← R/M" biçimidir. REG = 001 (CL), R/M = 000 (AL) olur ve 11 001 000 = C8h çıkar.

Örnek 3.2.8

`ADD DX, CX` ve `MOV BL, AH` komutlarının makine kodları nedir?

`ADD DX, CX` : işlem kodu 03h (REG ← REG + R/M), MOD = 11, REG = DX = 010, R/M = CX = 001. ModR/M = 11 010 001 = D1h. Komut `03 D1` olur.

`MOV BL, AH` : işlem kodu 8Ah, MOD = 11, REG = BL = 011, R/M = AH = 100. ModR/M = 11 011 100 = DCh. Komut `8A DC` olur. İki komut da iki bayttır; IP her birinde 2 ilerler.

Tablodan çıkan dört sonuç önemlidir. Birincisi, .COM dosyası tam olarak bu 12 bayttır. Başlık yoktur; DOS dosyanın baytlarını olduğu gibi 0100h'den itibaren belleğe kopyalar. İkincisi, `ORG 100h` satırının karşılığında hiçbir bayt yoktur; direktif yalnız ilk baytın konumunu belirler.

Üçüncüsü, aynı komutun birden çok kodlaması olabilir. 8086'nın yazmaç–yazmaç komutlarında yön biti vardır: 03h "REG hedeftir", 01h "R/M hedeftir" anlamına gelir. `ADD AX, BX` 01h ile de yazılabilir; bu durumda REG = BX, R/M = AX olur ve kod `01 D8` çıkar. MASM 03 C3'ü seçer, başka bir assembler 01 D8'i seçebilir. İkisi de aynı işi aynı sürede yapar. Liste çıktısı farklı assembler'larda bu yüzden farklı baytlar gösterebilir.

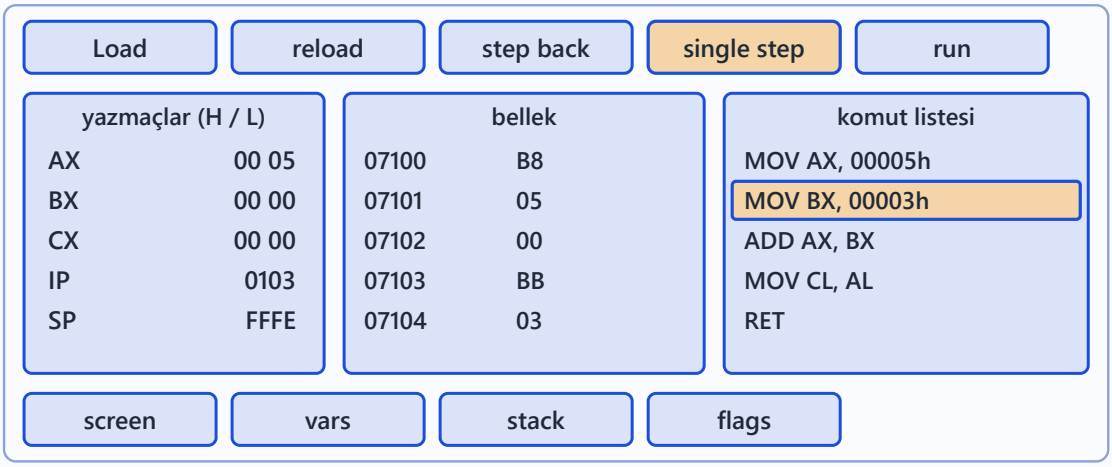
Dördüncüsü ve en önemlisi, bellekte komut ile veri arasında hiçbir işaret yoktur. 010Bh'deki `00` baytı bir değişkendir, ama işlemci IP = 010Bh ile oraya gelseydi onu bir komut olarak çözerdi. 8086'da 00h, bir `ADD` komutunun işlem kodudur ve arkasından gelen bayt ModR/M olarak okunurdu. Bir baytın komut mu veri mi olduğunu yalnız programın akışı belirler. Değişkenlerin RET'ten sonra konmasının nedeni budur.

Sık yapılan hata

Değişkenleri `ORG 100h`'ın hemen altına yazmak. Yüksek düzeyli dillerde değişkenler programın başında tanımlanır; bu alışkanlık assembly'ye taşınınca program çalışmaz. `sayi DB 0` 0100h'ye yerleşir ve işlemci ilk komut olarak 00h baytını, yani bir ADD'in başını yürütür. Sonuç rastgele bir bellek yazması ve kaymış komut sınırlarıdır. Veri ya RET'ten sonra konur ya da kodun başında bir JMP ile üstünden atlanır.

3.2.5.4 emu8086 öykünücü penceresi

Kaynak yazıldıktan sonra emu8086'da **emulate** düğmesi programı derler ve öykünücü penceresine yükler. Şekil 3.2.8 bu pencerenin ana bölümlerini sade bir çizimle gösterir.



Şekil 3.2.8: emu8086 öykünücü penceresinin sade çizimi. Üstte denetim düğmeleri; solda yazmaçlar yüksek ve düşük baytlarıyla; ortada fiziksel adresleriyle bellek; sağda komut listesi. Çizim, şablon programın ilk komutu yürütüldükten sonraki anı gösterir: AX = 0005h, IP = 0103h ve vurgulu satır sıradaki komut olan `MOV BX, 00003h`'tür.

Pencerenin bölümleri şunlardır:

- **Denetim düğmeleri.** *single step* bir komut yürütür ve durur; değişen yazmaçlar ve bellek hemen görünür. *step back* son adımı geri alır; bir hatanın nedenine geri dönmeyi kolaylaştırır. *run* programı sonuna kadar çalıştırır, *reload* programı baştan yükler.
- **Yazmaçlar.** Her yazmaç yüksek (H) ve düşük (L) baytı ayrı ayrı gösterilir: AX = 0005h, “00 05” diye görünür. Bu düzen AH ve AL'nin aynı yazmacın iki yarısı olduğunu açıkça gösterir.
- **Bellek.** Adres sütunu fiziksel adresleri gösterir: 0700:0100 = 07000h + 0100h = 07100h. Program baytları bu adresten başlar: B8, 05, 00, BB, 03 …
- **Komut listesi.** Bellekteki baytların komutlara geri çevrilmiş hâlidir. Vurgulu satır **sıradaki** komuttur, yani CS:IP'nin gösterdiği komut; az önce yürütülen değil. Liste sabitleri onaltılık ve 5 basamakla gösterir: `MOV AX, 00005h`.

- **Ek pencereler.** *screen* ekran çıktısını, *vars* değişkenleri, *stack* yığını, *flags* bayrakları ayrı pencerelerde gösterir.

Komut listesindeki vurgunun anlamı yeni başlayanları sık yanıltır. Şekildeki durumda vurgulu satır `MOV BX, 0003h`'tür, ama BX hâlâ 0000h'dir. Bu bir hata değildir: vurgulu komut henüz yürütülmemiştir. Bir sonraki single step onu yürütür; BX = 0003h olur ve vurgu `ADD AX, BX` satırına geçer. Yazmaç değerleri her zaman vurgulu satırdan *önceki* komutların sonucudur.

Öykünücüde bir programı incelemenin yöntemi basittir. Her single step'ten önce sıradaki komutun hangi yazmacı ve bayrağı nasıl değiştireceği tahmin edilir. Adım atılır ve tahmin pencerenin gösterdiği değerle karşılaştırılır. Bu notun izleme tabloları tam olarak bu tahminlerin yazılı hâlidir.

> **Sık yapılan hata**
>
> **Bellek sütunundaki adresi bağıl konumla karıştırmak.** Öykünücünün bellek sütunu 07100h gösterirken program 0100h'yi kullanır. 07100h fiziksel adrestir; 0100h ise CS = 0700h kesimi içindeki bağıl konumdur. `MOV AL, [0100h]` komutu 07100h'deki baytı okur, 00100h'dekini değil. Aradaki ilişki her zaman kesim × 10h + bağıl konumdur.

3.2.6 Yazım kuralları

Assembler sayıları, karakterleri ve adları yazılışlarından tanır. Kurallar azdır, ama birinin unutulması ya bir derleme hatası ya da fark edilmesi zor bir yanlış değer üretir.

Sayı tabanları. Bir sayının tabanı sonuna eklenen harfle belirtilir. Harf yoksa sayı onluktur. Tablo 3.2.18 aynı değeri, 25'i, dört tabanda verir. Assembler hepsini aynı bayta, 19h'ye çevirir; `MOV AL, …` satırlarının hepsi `B0 19` olur.

Taban	Sonek	Yazılış	`MOV AL, …` kodu
onluk	yok ya da `d`	`25`, `25d`	`B0 19`
onaltılık	`h`	`19h`	`B0 19`
ikili	`b`	`00011001b`	`B0 19`
sekizli	`o`	`31o`	`B0 19`

Tablo 3.2.18: Sayı tabanlarının yazılışı (hepsi 25).

Taban seçimi yalnız okunabilirlik içindir; işlemci farkı görmez. Bir port maskesi ya da bayrak deseni ikili yazıldığında hangi bitin kurulduğu hemen anlaşılır: `00100000b` bit 5'tir. Bir adres ise onaltılık yazılır, çünkü kesim ve bağıl konum hesapları onaltılıkta yapılır. Döngü sayıları ve günlük büyüklükler onluk yazılır.

Harfle başlayan onaltılık sayılar. Onaltılık rakamlar A–F harflerini içerir. `FFh` yazıldığında assembler bunun bir sayı mı yoksa `FFh` adlı bir etiket mi olduğunu ayırt edemez; kurala göre harfle başlayan her dizi bir addır. Bu yüzden harfle başlayan onaltılık sayıların önüne `0` konur: `0FFh`, `0B800h`. Baştaki sıfır değeri değiştirmez, yalnız dizinin bir sayı olduğunu bildirir. `MOV AX, 0B800h` komutu `B8 00 B8` olarak kodlanır.

Karakterler. Tırnak içindeki bir karakter, ASCII kodunun yerine geçer. `'A'` = 41h olduğundan `MOV CL, 'A'` ile `MOV CL, 41h` aynı makine kodunu üretir: `B1 41`. Karakter yazımı, kodu ezberleme yükünü kaldırır ve programın ne yaptığını açıkça gösterir. Ekrana bir rakam yazdıran program `ADD DL, '0'` yazdığında 0–9 arası bir değeri '0'–'9' karakterine çevirdiği hemen anlaşılır.

İşaretli sayılar. Eksi işaretli bir sabit ikiye tümleyen olarak saklanır. `-1` 8 bitte FFh, 16 bitte FFFFh olur: `MOV AL, -1` = `B0 FF`, `MOV AX, -1` = `B8 FF FF`. 8 bitlik bir yazmaca yazılabilecek sabitler −128 ile 255 arasındadır: negatif değerler işaretli, büyük pozitif değerler işaretsiz yorumla sığar. `MOV AL, 256` ya da `MOV AL, -129` hata verir.

Büyük-küçük harf. Assembler duyarsızdır: `mov`, `MOV`, `Mov` aynıdır; `var1` ile `VAR1` aynı addır. Okunabilirlik için tutarlı bir düzen seçilir. Bu notta komutlar ve yazmaçlar büyük, değişken ve etiket adları küçük harfle yazılmıştır.

Açıklamalar. `;` işaretinden sonrası açıklamadır. Assembly'de neredeyse her satıra açıklama yazılır, çünkü komutların kendisi amacı anlatmaz. İyi alışkanlık, açıklamaya komutun "ne" yaptığını değil "neden" yapıldığını yazmaktır. `INC SI ; SI = SI + 1` bilgi vermez; `INC SI ; dizinin sonraki elemanı` verir.

> **Sık yapılan hata**
>
> **Sonek unutmak.** `MOV AL, 10` ile `MOV AL, 10h` farklı değerlerdir: 10 ve 16. Onaltılık bir değeri soneksiz yazmak assembler'ın hata vermeyeceği, ama programın yanlış çalışacağı bir hatadır. Ekran belleğinin kesimi B800h'yi `MOV AX, B800` diye yazan programcı ise hem soneki hem baştaki sıfırı unutmuş olur; assembler `B800` 'ü tanımsız bir ad sayar.

3.2.7 Hatalar

3.2.7.1 MOV'un işlenen kısıtları

MOV en sık kullanılan komuttur ve en sık hata da onunla yapılır. Hataların neredeyse hepsi, 8086'nın komut kodlamasında var olmayan bir işlenen birleşimini yazmaktan gelir. Tablo 3.2.19 en yaygın altısını verir.

Hatalı	Neden?	Doğrusu
`MOV AL, BX`	boyutlar farklı (8 ↔ 16 bit)	`MOV AL, BL` ya da `MOV AX, BX`

Hatalı	Neden?	Doğrusu
`MOV DS, 1234h`	kesim yazmacına sabit atanamaz	`MOV AX, 1234h` · `MOV DS, AX`
`MOV AL, [AX]`	adreslemede yalnız BX, SI, DI, BP	`MOV BX, AX` · `MOV AL, [BX]`
`MOV AL, FFh`	`FFh` bir ad sanılır	`MOV AL, 0FFh`
`MOV IP, 200h`	IP'ye MOV ile yazılamaz	`JMP` ile atlamak
`MOV AL, 300`	300 > 255, 8 bite sığmaz	`MOV AX, 300`

Tablo 3.2.19: Sık yapılan MOV hataları.

Boyut uyuşmazlığı. MOV'un iki işleneni aynı genişlikte olmalıdır, çünkü işlem kodu tek bir genişlik biti taşır: ya 8 bit ya 16 bit. `MOV AL, BX` komutunda 16 bitlik bir değerin hangi yarısının alınacağı belirsizdir. Programcı alt baytı istiyorsa `MOV AL, BL` yazar. Tersi yöndeki `MOV CX, AL` de aynı nedenle hatalıdır; 8 bitlik bir değeri 16 bite genişletmek için üst yarı ayrıca kurulur: `MOV CL, AL` ve `MOV CH, 0`.

Kesim yazmacına sabit. 8086'da kesim yazmacına yazan MOV yalnız genel yazmaçtan ya da bellekten okur. Anlık değer yükleyen bir biçimi yoktur. Bu yüzden ekran belleğine erişmek için kesim kurmak iki satır ister: `MOV AX, 0B800h` ve `MOV DS, AX`.

Adres yazmacı. 8086'da köşeli ayraç içinde yalnız BX, BP, SI ve DI kullanılabilir. Bunun nedeni ModR/M baytının R/M alanıdır: bellek modlarında bu üç bit yalnız bu dört yazmacın belirli birleşimlerini kodlar. AX'i adres olarak kullanmanın bir kodu yoktur.

IP'ye yazmak. IP hiçbir veri komutunun işleneni olamaz. IP'yi değiştirmenin tek yolu akış komutlarıdır: JMP, Jcc, CALL, RET, INT ve IRET. Assembler `IP`'yi yazmaç adı olarak tanımaz; satırı "hedef sabit olamaz" ya da "tanımsız ad" hatasıyla reddeder.

Sığmayan sabit. 8 bitlik bir yazmaç işaretsiz yorumda en çok 255'i, işaretli yorumda −128 ile 127 arasını tutar. 300 dokuz bit ister. Assembler satırı reddeder; değer 16 bitlik bir yazmaca yazılır.

3.2.7.2 Öteki söz dizimi hataları

Hatalı	Neden?	Doğrusu
`MOV CS, AX`	CS yalnız JMP, CALL, RET, INT ile değişir	uzak atlama
`MOV ES, DS`	kesimden kesime MOV yok	`MOV AX, DS` · `MOV ES, AX`
`MOV [BX], [SI]`	iki bellek işleneni	`MOV AL, [SI]` · `MOV [BX], AL`
`INC [BX]`	boyut belirsiz	`INC BYTE PTR [BX]`
`MOV AL, [BX+BP]`	iki taban yazmacı birlikte olmaz	`[BX+SI]` ya da `[BP+DI]`
`ADD 5, AX`	hedef sabit olamaz	`ADD AX, 5`
`PUSH 5`	80186'dan önce yok	`MOV AX, 5` · `PUSH AX`
`SHL AX, 2`	8086'da sayı yalnız 1 ya da CL	`MOV CL, 2` · `SHL AX, CL`

Tablo 3.2.20: Öteki söz dizimi hataları.

CS'ye yazmak. CS'yi MOV ile değiştirmek, kodun ortasında başka bir kesime geçmek demektir; ama IP aynı kalır ve işlemci rastgele bir adresten devam eder. CS ile IP'nin birlikte değişmesi gerekir. Bunu uzak JMP, uzak CALL, uzak RET, INT ve IRET yapar.

Kesimden kesime. İki kesim yazmacı arasında doğrudan aktarım yoktur. Değer bir genel yazmaç üzerinden iki adımda taşınır. Bellekten belleğe aktarım da aynı biçimde yapılır: bir komutta en çok bir bellek işleneni bulunur, değer önce bir yazmaca alınır.

Boyut belirsizliği. `INC [BX]` satırında işlenenin bir bayt mı bir sözcük mü olduğu belli değildir; hiçbir işlenen bir yazmaç değildir ki boyut ondan çıkarılsın. Assembler bu yüzden boyutun açıkça yazılmasını ister: `INC BYTE PTR [BX]` ya da `INC WORD PTR [BX]`. Aynı kural `MOV [BX], 5` için de geçerlidir. emu8086 `b.[bx]` ve `w.[bx]` kısaltmalarını da kabul eder.

Adres birleşimleri. 8086'nın kabul ettiği adres biçimleri bir taban yazmacı (BX ya da BP), bir indis yazmacı (SI ya da DI) ve bir sabitin birleşimidir. İki taban (`[BX+BP]`) ya da iki indis (`[SI+DI]`) birlikte kullanılamaz; ModR/M baytında bunların kodu yoktur.

Sonraki işlemcilerin komutları. `PUSH 5` ve `SHL AX, 2` 80186 ile geldi. MASM 8086 için derlerken bu satırlarda hata verir. emu8086 ise bu biçimleri kabul eder; derlenen kod öykünücüde çalışır, ama gerçek bir 8086'da işlemci bu baytları tanımaz. 8086 için yazılan bir programda sabit yığına bir yazmaç üzerinden itilir, çok bitli kaydırmada sayı CL'ye konur.

Örnek 3.2.9

`sozcuk DW 1234h` ve `bayt DB 5` tanımlı. `MOV sozcuk, bayt` satırı neden hatalıdır ve nasıl düzeltilir?

Satırda iki hata birden vardır. İki işlenen de bellektir ve boyutları farklıdır (16 ↔ 8 bit). Değer bir yazmaç üzerinden taşınır ve üst bayt ayrıca sıfırlanır:

```
MOV AL, bayt      ; AL = 05h
MOV AH, 0         ; AX = 0005h
MOV sozcuk, AX    ; sozcuk = 0005h
```

`bayt` işaretli bir sayı olsaydı ikinci satır yerine `CBW` kullanılırdı.

3.2.7.3 Derleme hatası ve mantık hatası

Bir assembly programında iki tür hata bulunur. Ayrımı bilmek, hatanın nasıl aranacağını belirler.

Tanım 3.2.8

Derleme hatası (söz dizimi hatası), assembler'ın kaynağı çevirirken bulduğu hatadır. Assembler satır numarasını ve nedeni bildirir; program hiç oluşmaz. **Mantık hatası** ise programın derlenip çalıştığı, ama yanlış sonuç verdiği durumdur. Assembler bunu bilemez, çünkü her satır kendi başına geçerlidir.

Derleme hatası	Mantık hatası
`MOV AL, BX` : boyut uyuşmazlığı	`RET` 'i unutmak: veri komut sanılır
`JMP bsla` : tanımsız etiket	işaretli sayı için `JA` kullanmak
`MOV AL, 300` : sığmayan sabit	`ADC` yerine `ADD` : elde kaybolur
`MOV DS, 1234h` : kesime sabit	.EXE programında DS'yi yüklememek

Tablo 3.2.21: Derleme ve mantık hatası örnekleri.

Derleme hatası kolay hatadır: assembler yerini gösterir, düzeltme çoğunlukla tek satırdır. Mantık hatası ise sessizdir. Program çalışır, belki çoğu girdi için doğru sonuç verir, yalnız bazı durumlarda yanılır. Tablodaki örnekler bu türün tipik hâlleridir. `JA` işaretsiz karşılaştırmanın atlamasıdır; −1 ile 1 karşılaştırılırken −1'i FFFFh, yani büyük bir sayı sayar. `ADD` ile yapılan çok baytlı toplama, düşük baytlardan gelen eldeyi yüksek baytlara aktarmaz. .EXE programında DS yüklenmezse değişkenler PSP'nin bulunduğu kesimde aranır ve bambaşka baytlar okunur.

Mantık hatasını bulmanın yolu tek adım çalıştırmadır. Program öykünücüde komut komut ilerletilir. Her adımdan önce beklenen yazmaç ve bayrak değerleri hesaplanır, adımdan sonra pencerede görünenle karşılaştırılır. Değerlerin ilk ayrıldığı komut, hatanın bulunduğu yerdir. Hata bu komutun kendisinde olabilir. Komut doğru da olabilir; o zaman daha önceki bir satır ona yanlış bir değer bırakmıştır. emu8086'nın *step back* düğmesi bu aramayı kolaylaştırır: hatalı değer görülünce birkaç adım geri gidilip değerin nereden geldiği izlenir.

Örnek 3.2.10

Bir öğrenci AL'deki sayıyı 3 ile çarpıp sonucu AX'te istiyor ve şu kodu yazıyor: `MOV BL, 3` · `MUL BL` . Program AL = 7 için AX = 0015h, yani 21 veriyor. Aynı kod bir başka yerde AH = 12h iken çalışınca öğrenci AH'nin neden silindiğini soruyor. Bu bir hata mı?

Kod doğrudur; yanılgı beklentidedir. MUL'ün sonucu örtük olarak AX'in tamamına yazılır: 7 × 3 = 21 = 0015h, AH = 00h. AH'deki 12h'nin korunması beklentisi bir mantık hatasıdır, ama hata kodda değil, programın AH'yi saklayacağı varsayımındadır. AH'deki değer gerekiyorsa MUL'den önce başka bir yazmaca alınır. Tek adım çalıştırma bu durumu hemen gösterir: MUL adımından sonra AH sütunu 12'den 00'a döner.

Sık yapılan hata

Programı bir bütün olarak çalıştırıp yalnız sonuca bakmak. Yanlış sonuç veren bir programı *run* ile çalıştırıp yeniden yazmaya başlamak, hatayı başka bir yere taşır. Önce tek adımla ilerlenir ve ilk sapma bulunur. Uzun bir döngüde tek adım sıkıcıysa döngüden önceki ve sonraki durum karşılaştırılarak hata önce bir bölgeye, sonra bir satıra daraltılır.

3.2.8 Alıştırmalar ve çözümleri

Sunumdaki altı alıştırma aşağıda ara adımlarıyla çözülmüştür. Bütün yazmaç ve bayrak değerleri öykünücüde çalıştırılarak denetlenmiştir.

Alıştırma 1

AX = 0ABCDh iken `MOV AH, 0` sonrası AX, AL ve BX değerleri nedir?

AX = ABCDh'nin yüksek yarısı AH = ABh, düşük yarısı AL = CDh'dir. `MOV AH, 0` yalnız AH'ye yazar: AH = 00h. AL dokunulmadan kalır. Sonuç AX = **00CDh**, AL = **CDh** olur. BX komutta ne yazılı ne örtük olarak geçer; değeri ne ise o kalır. Bu komut, işaretsiz bir 8 bitlik değeri 16 bite genişletmenin en kısa yoludur.

Alıştırma 2

AX = 1234h, BL = 03h iken `MUL BL` sonrası AX kaçtır? AH'deki 12h'ye ne olur?

8 bitlik MUL'de çarpılan AX değil, örtük olarak AL'dir: AL = 34h = 52. 52 × 3 = 156 = **009Ch**. Sonuç örtük olarak AX'in tamamına yazılır; AH = 00h, AL = 9Ch olur. AH'deki 12h silinir. Üst yarı sıfır olduğu için CF = OF = 0'dır. AX'in tamamının 3 ile çarpıldığını düşünen biri 1234h × 3 = 369Ch bulur; bu sonuç için `MUL BX` gerekir ve sonuç DX:AX'e yazılır.

Alıştırma 3

8 MHz'lik bir 8086'da 2000 kez dönen bir döngüde sayaç `INC sayac` (21 tur) yerine `INC CX` (2 tur) ile artırılırsa kaç saat turu ve kaç milisaniye kazanılır?

Her turda kazanç 21 − 2 = 19 saat turudur. 2000 turda 2000 × 19 = **38.000** saat turu kazanılır. 8 MHz'de bir saat turu 1 / (8 × 10^6) s = 0,125 µs'dir. Kazanç 38.000 × 0,125 µs = 4750 µs = **4,75 ms** olur.

Alıştırma 4

AL = 80h iken `ADD AL, 80h` sonrası AL, CF, ZF, SF ve OF ne olur?

80h + 80h = 100h. Sonuç dokuz bit ister; AL'ye alt sekiz bit yazılır: AL = **00h**. Dokuzuncu bit CF'ye gider: CF = 1. Sonuç sıfırdır: ZF = 1. En üst bit 0'dır: SF = 0. İşaretli yorumda 80h = −128'dir ve −128 + (−128) = −256 sekiz bitlik aralığa (−128 … 127) sığmaz: OF = 1. Ayrıca 00h'nin bir bitleri çift sayıda olduğu için PF = 1, bit 3'ten elde olmadığı için AF = 0'dır.

Alıştırma 5

Program 0100h'de başlıyor: 3 + 2 + 1 + 4 baytlık dört komut yürütülünce IP ne olur?

Atlama olmadığı için IP her komutta uzunluğu kadar ilerler: 0100h → 0103h → 0105h → 0106h → 010Ah. Toplam 10 bayt = 0Ah olduğundan IP = 0100h + 0Ah = **010Ah** olur.

Alıştırma 6

`MOV CX, AL` ve `MOV ES, 0B800h` neden hatalıdır? Doğru yazılışları nedir?

`MOV CX, AL` : işlenenler farklı boyuttadır (16 ↔ 8 bit). Amaç yalnız alt baytı kopyalamaksa `MOV CL, AL` yazılır. AL'nin değeri CX'in tamamında işaretsiz olarak istenirse üst yarı da sıfırlanır: `MOV CL, AL` · `MOV CH, 0` .

`MOV ES, 0B800h` : kesim yazmacına anlık değer yüklenemez. Değer bir genel yazmaç üzerinden taşınır: `MOV AX, 0B800h` · `MOV ES, AX` .

3.2.9 Özet

- **Assembly dili** makine koduyla birebir eşleşen düşük düzeyli bir dildir. Her komut satırı tek bir makine komutuna çevrilir: `MOV AX, 5` = `B8 05 00` . Küçük ve hızlı kod üretir, donanıma doğrudan erişir, ama taşınabilir değildir.
- **Assembler** anımsatıcıları işlem kodlarına, adları adreslere çevirir. emu8086 düzenleyici, assembler ve öykünücüyü birleştirir; yazmaçları, belleği ve bayrakları adım adım gösterir.
- Satır yapısı `[etiket:] komut hedef, kaynak ; açıklama` biçimindedir. Hedef soldadır; kaynak değişmez. Assembler büyük-küçük harfe duyarsızdır; açıklamalar makine koduna girmez.
- **Direktifler** (ORG, EQU, END) makine komutu üretmez; assembler'a bilgi verir. DB ve DW yalnız veri baytı üretir. Makro çağrısı birden çok komutun yerine geçer.
- Ad harfle ya da alt çizgiyle başlar; ayrılmış sözcükler ad olamaz. Kod etiketi iki nokta ile biter ve bir adresi gösterir; değişken adı adres ile türü birlikte taşır.

- İşlenen türleri: **yazmaç** (en hızlı), **anlık değer** (yalnız kaynak), **bellek** (yol döngüsü ister, komutta en çok bir tane). İki işlenen aynı boyutta olmalıdır.
- **Örtük işlenenler** komutta yazılmaz ama değişir: MUL AX'e, DIV AL ile AH'ye yazar; LOOP CX'i azaltır; CBW AH'yi doldurur; PUSH SP'yi değiştirir.
- Bellek erişimi en az dört T durumluk bir yol döngüsüdür. Bellek işlenenli komutlar etkin adres süresiyle birlikte yazmaç eşlerinden birkaç kat uzundur: `INC CX` 2, `INC sayac` 21 tur.
- **IP** sıradaki komutun bağıl konumudur ve komut uzunluğu kadar ilerler. Yalnız akış komutları IP'ye yeni değer yazar.
- MOV, XCHG, LEA, PUSH, POP, NOT ve LOOP bayrakları değiştirmez. INC ve DEC CF'ye dokunmaz. ADD, SUB ve CMP altı durum bayrağını yeniden yazar. MUL sonrası ZF, SF, AF, PF tanımsızdır.
- .COM programı `ORG 100h` ile 0100h'den başlar; ilk 256 bayt PSP'dir. Başlangıçta CS = DS = ES = SS, IP = 0100h, SP = FFFEh'dir. RET yığındaki 0000h ile PSP'deki `INT 20h`'ye döner. Veriler RET'ten sonra durur.
- Bellekte komut ile veri arasında işaret yoktur; bir baytın anlamını programın akışı belirler. Aynı komutun birden çok geçerli kodlaması olabilir.
- **Derleme hatasını** assembler bulur; **mantık hatası** ancak tek adım çalıştırıp beklenen değerleri öykünücüdekiyle karşılaştırarak bulunur.

3.2.10 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

assembly (assembly dili) *assembly language* · anımsatıcı *mnemonic* · komut *instruction* · işlem kodu *opcode* · işlenen *operand* · örtük işlenen *implicit operand* · anlık değer *immediate value* · açıklama *comment* · etiket *label* · değişken *variable* · direktif *directive* · sabit (EQU) *constant* · konum sayacı ($) *location counter* · makro *macro* · assembler *assembler* · öykünücü *emulator* · .COM programı *.COM program* · PSP (program kesimi öneki) *program segment prefix* · akümülatör *accumulator* · sayaç yazmacı *counter register* · yol döngüsü *bus cycle* · T durumu *T state* · saat turu (tur) *clock cycle* · etkin adres (EA) *effective address* · komut işaretçisi (IP) *instruction pointer* · bayrak yazmacı (FLAGS) *flags register* · ModR/M baytı *ModR/M byte* · little-endian *little-endian* · tek adım *single-step* · derleme hatası / mantık hatası *syntax error / logic error*

3.2.11 Kendini Dene

1. `STC`, `MOV AX, 0FFFFh`, `INC AX` komutları sırayla çalışıyor. Sonunda AX, CF ve ZF kaçtır?

   Cevap

   FFFFh + 1 = 10000h, 16 bite sığmaz ve AX = 0000h olur. Sonuç sıfırdır: ZF = 1. INC CF'ye dokunmaz; CF, STC'den kalan 1'dir. MOV da bayrak değiştirmediği için CF = 1 hiç bozulmamıştır. Aynı toplama `ADD AX, 1` ile yapılsaydı CF yine 1 olurdu, ama bu kez taşan eldeden dolayı.

2. `ORG 100h` ile başlayan bir programda sırayla `MOV AL, 5`, `MOV BX, 1234h`, `INC BX`, `RET` ve `veri DB 9` satırları var. Programın baytları nelerdir, `veri`'nin bağıl konumu nedir?

   Cevap

   `MOV AL, 5` = `B0 05` (0100h), `MOV BX, 1234h` = `BB 34 12` (0102h), `INC BX` = `43` (0105h; 40h + 3), `RET` = `C3` (0106h), `veri` = `09` (0107h). Komutlar 2 + 3 + 1 + 1 = 7 bayttır; `veri` 0100h + 7 = **0107h**'dedir. Program toplam 8 bayttır ve sonunda BX = 1235h olur.

3. `MOV AX, [BX+SI]` ve `ADD [BX], AX` komutları 5 MHz'lik bir 8086'da en iyi durumda kaç saat turu ve kaç mikrosaniye sürer? (Bellekten yazmaca okuma 8 + EA, belleğe toplama 16 + EA tur.)

   Cevap

   `[BX+SI]` için EA = 7 tur: MOV 8 + 7 = 15 tur. `[BX]` için EA = 5 tur: ADD 16 + 5 = 21 tur. Toplam 36 tur; 5 MHz'de 36 × 0,2 µs = **7,2 µs**. ADD'in uzun sürmesinin nedeni belleğe hem okuma hem yazma yapmasıdır: iki yol döngüsü.

4. AX = 25 iken `MOV BL, 4` ve `DIV BL` çalışıyor. AL, AH ve AX kaçtır?

   Cevap

   Bölünen örtük olarak AX = 25'tir. 25 ÷ 4 = 6, kalan 1. Bölüm AL'ye, kalan AH'ye yazılır: AL = 06h, AH = 01h, AX = **0106h**. AX'i bir sayı olarak okuyan biri 262 görür; DIV'den sonra AX'in iki yarısı ayrı anlam taşır.

5. Şu programın kaç satırı bayt üretir, program kaç bayttır? `ORG 100h` · `k EQU 5` · `MOV AX, 5` · `MOV BX, k` · `MOV AL, sayi` · `RET` · `sayi DB 0` · `END`. `MOV AL, sayi` satırının baytları nedir?

   Cevap

   ORG, EQU ve END bayt üretmez. Üç MOV 3'er bayt, RET 1 bayt, DB 1 bayt: beş satır, **11 bayt**. `sayi` 0100h + 10 = 010Ah'dedir. `MOV AL, sayi` AL'ye doğrudan adresten okuyan akümülatör biçimidir: `A0 0A 01`.

6. `MOV AL, 0F0h`, `ADD AL, 20h`, `MOV BL, AL`, `NOT BL` çalışıyor. Sonunda AL, BL, CF ve ZF kaçtır?

   Cevap

   F0h + 20h = 110h: AL = 10h, CF = 1, ZF = 0. MOV ile BL = 10h olur; bayraklar değişmez. NOT BL = EFh; NOT da bayrak değiştirmez. Sonuç AL = 10h, BL = EFh, CF = 1, ZF = 0. Bayraklar ADD'in bıraktığı gibidir.

7. emu8086'da şablon programda (`MOV AX, 5`, `MOV BX, 3`, `ADD AX, BX`, `MOV CL, AL`, `RET`) üç kez single step'e basılıyor. Komut listesinde hangi satır vurgulu, IP ve AX kaçtır? Bellek sütununda bu komutun ilk baytı hangi fiziksel adreste görünür?

   Cevap

   Üç komut yürütülmüştür: AX = 0008h, BX = 0003h. IP = 0100h + 3 + 3 + 2 = **0108h**. Vurgulu satır sıradaki komut olan `MOV CL, AL`'dir; CL henüz 00h'dir. Fiziksel adres 07000h + 0108h = **07108h**; orada `8A` baytı görünür.

8. `MOV DX, 0Ah` ve `INC CX` komutlarının makine kodları nedir? İkisi ardışık olarak 0103h'den başlıyorsa sonraki komut hangi adresten başlar?

   Cevap

   DX'in yazmaç numarası 2'dir. `MOV DX, 0Ah` = B8h + 2 = BAh ve sabit düşük bayt önce: `BA 0A 00` (3 bayt). `INC CX` = 40h + 1 = `41` (1 bayt). İki komut 4 bayttır; sonraki komut 0103h + 4 = **0107h**'den başlar.

9. Bir program `MOV CL, 3` ile kaydırma sayısını kuruyor, ardından `MOV CX, 10` ile bir döngü sayacı kuruyor ve en sonda `SHL AX, CL` yazıyor. Program derlenir mi? AX kaç bit kayar?

   Cevap

   Program derlenir; her satır geçerlidir. Ancak CL, CX'in alt yarısıdır. `MOV CX, 10` CL'yi 0Ah yapar ve kaydırma 3 değil **10 bit** olur. Bu bir mantık hatasıdır. Kaydırma sayısı CL'ye, kaydırmadan hemen önce yazılmalıdır.

3.2.12 İleri okuma

- emu8086 belgeleri, *8086 assembler tutorial for beginners*: sayı sistemleri, bellek erişimi, değişkenler ve komut kümesi başvurusu; emu8086 söz dizimine en yakın kaynaktır.
- B. B. Brey, *The Intel Microprocessors* (8. baskı), Bölüm 4 ve 6: veri taşıma komutları, ModR/M kodlaması, .COM ve .EXE programlarının yapısı.

- M. A. Mazidi, J. G. Mazidi, *The 80x86 IBM PC and Compatible Computers*, Bölüm 1–2: assembly programının yapısı, direktifler, örnek programlar ve hata ayıklayıcıyla çalışma.
- Intel, *8086 16-Bit HMOS Microprocessor* veri sayfası: yol döngüsünün T durumları ve komut süreleri tablosu.

3.2.13 Kaynaklar

1. emu8086 belgeleri, *8086 assembler tutorial for beginners* (bölüm 1: *numbering systems*, bölüm 2: *memory access*).
2. Intel, *8086 16-Bit HMOS Microprocessor* veri sayfası.
3. B. B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson.
4. M. A. Mazidi, J. G. Mazidi, *The 80x86 IBM PC and Compatible Computers*, Pearson.

KISIM II

Bellek ve Sistem Hizmetleri

Bölüm 4–5

KISIM II · BÖLÜM 4

Bellek Erişimi ve Değişkenler

8086 belleğe kesim ve bağıl konum değerlerinin birleşimiyle erişir. Bölümde BX, SI, DI ve BP yazmaçlarıyla kurulan adresleme modları, etkin adres ve fiziksel adres hesabı, byte ptr ve word ptr ile veri boyutunun belirtilmesi anlatılır; MOV ve XCHG komutlarının geçerli işlenen bileşimleri örneklerle işlenir. Değişkenler kısmında DB, DW ve DD tanımları, bellek yerleşimi, ORG 100h ve .COM programları, DUP ile diziler, LEA ve OFFSET ile adres alma ve EQU sabitleri ele alınır. Değişkenler üzerinde taşıma, toplama ve karşılaştırma yapan örnek programlarla bölüm tamamlanır.

Kod (8086): `github.com/sercankulcu/microprocessors-8086/Bolum04`

BU BÖLÜMÜN NOTLARI

4.1 Bellek Erişimi

4.2 Değişkenler

KISIM II · BÖLÜM 4.1

Bellek Erişimi

Bir 8086 programının yaptığı işlerin çoğu, bellekten bir değeri yazmaca almak ya da yazmaçtaki bir değeri belleğe geri koymaktır. Toplama, karşılaştırma, ekrana yazı yazma ve bir diziyi gezme bu taşımalar olmadan yapılamaz. İşlemcinin bellekte hangi bayta ulaşacağını komutun içindeki köşeli parantez belirler. Bu not, köşeli parantez içine neyin yazılabileceğini, yani adresleme modlarını anlatır. Taban ve indis yazmaçlarından, yer değiştirmeden bir etkin adres hesaplanır. Etkin adres bir kesim yazmacıyla birleşir ve adres yoluna çıkan 20 bitlik fiziksel adres olur. Hangi kesimin seçildiği, aynı bağıl konumun bambaşka bir bayta gitmesine yol açabilir. Not bu hesabı her iki yönde de yapar ve kesim önekiyle varsayılan kesimin nasıl değiştirildiğini gösterir. Ardından erişimin boyutu (bayt mı sözcük mü), bayt sırası ve MOV ile XCHG komutlarının kuralları gelir. Örnek programlar adım adım, yazmaç ve bellek düzeyinde izlenir; son örnek, renkli metin ekranının belleğine doğrudan bir harf yazar.

ÖĞRENME ÇIKTILARI

Geçerli bellek işleneni kurmak
BX, BP, SI, DI ve yer değiştirmeyle 8086'nın 24 bellek adresleme biçimini oluşturmak; geçersiz birleşimleri nedeniyle ayırt etmek.

Adresleme modlarını adlandırmak
Anlık, yazmaç, doğrudan, yazmaç dolaylı, tabanlı, indisli ve tabanlı-indisli modları, ayrıca dizgi, G/Ç ve göreceli modları örnekleriyle tanımak; bir veri yapısına uygun modu seçmek.

Adres hesaplamak
Etkin adresi 16 bitlik sarmayla hesaplamak; kesim:bağıl konumdan fiziksel adrese ve fiziksel adresten olası kesim:bağıl konum çiftlerine geçmek; adres hattı sayısından bellek kapasitesini bulmak.

Kesimi doğru seçmek
Varsayılan kesim kuralını (BP varsa SS, yoksa DS) uygulamak; kesim önekinin makine koduna etkisini ve BP ile veriye erişmenin tuzağını açıklamak.

Boyut ve bayt sırası
byte ptr ve word ptr ile erişim boyutunu belirtmek; little-endian düzende sözcüklerin bellekteki yerleşimini okumak; sözcük dizisinde indisi bayt uzaklığına çevirmek.

MOV ve XCHG ile program izlemek
Geçerli işlenen birleşimlerini ayırt etmek; kopyalama, takas ve video belleğine yazma programlarını yazmaç, bellek ve IP düzeyinde adım adım izlemek.

4.1.1 Bellek işlenenleri

4.1.1.1 Köşeli parantez ve adres yazmaçları

8086'nın on dört yazmacı vardır ve bunların yalnız sekizi genel işlerde kullanılır. Bir programın verisi ise çoğunlukla bellekte durur: değişkenler, diziler, tablolar, ekranın içeriği, yığın. İşlemci bu verilerle çalışmak için her seferinde belleğe gider, istediği baytı ya da sözcüğü okur, işini bitirince sonucu belleğe geri yazar. Bu yüzden bir komut kümesini öğrenmenin ilk adımı, komutun bellekteki hangi hücreyi kastettiğini nasıl söylediğini öğrenmektir.

Assembly dilinde bir bellek hücresi **köşeli parantezle** gösterilir. `MOV AX, BX` komutu BX'in içindeki sayıyı AX'e kopyalar. `MOV AX, [BX]` komutu ise BX'teki sayıyı bir adres olarak kullanır; o adresteki sözcüğü bellekten okuyup AX'e koyar. Köşeli parantez, “içindekinin gösterdiği yer” anlamına gelir. C dilindeki işaretçi gösterimiyle karşılaştırılırsa `BX` bir işaretçinin kendisi, `[BX]` ise `*p` gibidir.

8086'da köşeli parantez içine her yazmaç yazılamaz. Bellek adresi tutabilen yalnız dört yazmaç vardır: **BX**, **BP**, **SI** ve **DI**. Bunlardan BX ile BP **taban yazmacı** (*base register*), SI ile DI **indis yazmacı** (*index register*) diye anılır. Taban yazmacı çoğunlukla bir veri bloğunun, örneğin bir dizinin ya da bir kaydın başlangıç adresini tutar. İndis yazmacı bu bloğun içindeki konumu sayar. Bu dört yazmaca bir de komutun içinde yazılı sabit bir sayı eklenebilir. Sabit sayıya **yer değiştirme** (*displacement*) denir.

Bu kısıtın kaynağı donanımdır. Komutun makine kodunda bellek işleneninin nasıl hesaplanacağını yalnız üç bitlik bir alan anlatır. Üç bit sekiz seçenek demektir ve 8086'nın tasarımcıları bu sekiz seçeneği BX, BP, SI ve DI'nin en kullanışlı birleşimlerine ayırmıştır. AX, CX, DX ve SP bu listede yoktur. Bu yüzden `MOV AL, [CX]` gibi bir yazım ne kadar doğal görünse de assembler onu reddeder; işlemcinin böyle bir adres üretecek devresi yoktur.

Tanım 4.1.1

Bir komutun işleneninin nerede bulunduğunu ve adresinin nasıl hesaplanacağını belirleyen kurala **adresleme modu** (*addressing mode*) denir. 8086'da bir bellek işleneninin adresi köşeli parantez içinde yazılır. Adres üç parçanın toplamıdır: bir taban yazmacı (BX ya da BP), bir indis yazmacı (SI ya da DI) ve bir yer değiştirme. Bu üç parçanın her biri isteğe bağlıdır; ama en az biri bulunmalıdır.

4.1.1.2 Yirmi dört birleşim

Taban, indis ve yer değiştirme parçalarının izin verilen bütün birleşimleri Tablo 4.1.1'de toplanmıştır. Tablonun satırları yazmaç kısmını, sütunları yer değiştirmenin boyutunu gösterir. **d8** 8 bitlik, **d16** 16 bitlik bir yer değiştirme demektir.

Yalnız yazmaç(lar)	+ d8	+ d16
`[BX + SI]`	`[BX + SI + d8]`	`[BX + SI + d16]`
`[BX + DI]`	`[BX + DI + d8]`	`[BX + DI + d16]`
`[BP + SI]`	`[BP + SI + d8]`	`[BP + SI + d16]`
`[BP + DI]`	`[BP + DI + d8]`	`[BP + DI + d16]`
`[SI]`	`[SI + d8]`	`[SI + d16]`
`[DI]`	`[DI + d8]`	`[DI + d16]`
`d16` (doğrudan adres)	`[BP + d8]`	`[BP + d16]`
`[BX]`	`[BX + d8]`	`[BX + d16]`

Tablo 4.1.1: 8086'nın 24 bellek adresleme biçimi. Satır yazmaç kısmını, sütun yer değiştirmenin boyutunu belirler.

Tablodaki sekiz satır, makine kodundaki üç bitlik alanın sekiz değerine karşılık gelir. Bu alanın adı **R/M**'dir ve komutun ikinci baytı olan **ModR/M baytının** en düşük üç bitinde durur. Aynı baytın en yüksek iki biti **MOD** alanıdır; tablonun hangi sütununun kullanılacağını o seçer. MOD = 00 yer değiştirme yok, MOD = 01 d8 var, MOD = 10 d16 var demektir. MOD = 11 ise işlenenin bellek değil yazmaç olduğunu gösterir. Üç sütun ile sekiz satırın çarpımı 24 eder; 8086'nın bütün bellek adresleme biçimleri bu 24 hücrededir.

Bir satırda tuhaflık göze çarpar. Yedinci satırın ilk hücresinde `[BP]` yoktur; yerine `d16`, yani yalnız sabit bir adres vardır. Tasarımcılar R/M = 110 ve MOD = 00 birleşimini **doğrudan adreslemeye** ayırmıştır: `MOV AL, [1234h]` gibi bir komutun adresi bu kodla yazılır. Yer değiştirmesiz `[BP]` için kod kalmamıştır. Programcı `MOV AL, [BP]` yazarsa assembler bunu sessizce `[BP + 0]` olarak, yani MOD = 01 ve d8 = 00 ile kodlar. Komut bir bayt uzar ama doğru çalışır. Tablo 4.1.2 birkaç komutun assembler'ın ürettiği makine kodunu gösterir.

Komut	Makine kodu	MOD	R/M	Uzunluk
`MOV AL, [BX+SI]`	`8A 00`	00	000	2 bayt
`MOV AL, [BX+DI]`	`8A 01`	00	001	2 bayt
`MOV AL, [BP+SI]`	`8A 02`	00	010	2 bayt
`MOV AL, [BP+DI]`	`8A 03`	00	011	2 bayt
`MOV AL, [SI]`	`8A 04`	00	100	2 bayt
`MOV AL, [DI]`	`8A 05`	00	101	2 bayt
`MOV CL, [1234h]`	`8A 0E 34 12`	00	110	4 bayt
`MOV AL, [BX]`	`8A 07`	00	111	2 bayt

Komut	Makine kodu	MOD	R/M	Uzunluk
`MOV AL, [BP]`	`8A 46 00`	01	110	3 bayt
`MOV AL, [BX+5]`	`8A 47 05`	01	111	3 bayt
`MOV AL, [BX+128]`	`8A 87 80 00`	10	111	4 bayt

Tablo 4.1.2: Aynı `MOV AL, …` komutunun farklı bellek işlenenleriyle makine kodu. İkinci bayt ModR/M baytıdır.

Tablodaki `8A` işlem kodu "bir baytı bellekten ya da yazmaçtan 8 bitlik bir yazmaca taşı" demektir. ModR/M baytının ortadaki üç biti (REG alanı) hedef yazmacı seçer: AL için 000, CL için 001. Örneğin `0E` = 00 001 110 bitleri MOD = 00, REG = 001 (CL), R/M = 110 (doğrudan adres) olarak okunur. `MOV CL, [1234h]` bu yüzden dört bayttır: işlem kodu, ModR/M ve iki baytlık adres. Adres, 8086'nın bütün çok baytlı sayıları gibi düşük bayt önce yazılmıştır: `34 12`. Doğrudan adresli bir komut AL ya da AX'e yükleme yapıyorsa assembler daha kısa bir özel biçim seçer: `MOV AL, [1234h]` komutu `A0 34 12` diye üç baytla kodlanır.

4.1.1.3 Birleştirme kuralı

Tablonun yapısı birkaç basit kurala indirgenebilir. Bu kurallar ezberlendiğinde tabloya bakmadan bir yazımın geçerli olup olmadığı söylenebilir. Şekil 4.1.1 kuralı üç sütunlu bir seçim olarak gösterir: her sütundan en çok bir öge alınır.



Şekil 4.1.1: Geçerli bir bellek işleneni, taban (BX, BP ya da hiçbiri), indis (SI, DI ya da hiçbiri) ve yer değiştirme (d8, d16 ya da hiçbiri) sütunlarının her birinden en çok bir seçimle kurulur.

Tanım 4.1.2

Birleştirme kuralı. Bir bellek işleneninde en çok bir taban yazmacı (BX ya da BP) ve en çok bir indis yazmacı (SI ya da DI) bulunur. Bunlara en çok bir yer değiştirme eklenir. BX ile BP, SI ile DI aynı adreste bir arada kullanılamaz. Parçaların yazılış sırası önemsizdir.

Varsayılan kesim kuralı. Adreste BP varsa erişim SS (yığın) kesimine, yoksa DS (veri) kesimine yapılır.

Kesim kavramının özü şudur: 8086 belleği 64 KB'lık pencerelerle görür ve her erişimde hangi pencerenin kullanılacağını bir kesim yazmacı söyler. Veri için DS, yığın için SS kullanılır. BP yığındaki verilere erişmek için tasarlandığından onunla kurulan adresler kendiliğinden SS penceresine gider. Sunumdaki dört örnek bu kuralları uygular:

- `[BX+5]` : taban + d8. BP yok, kesim DS.
- `[BX+SI]` : taban + indis. Kesim DS.
- `[DI+BX-4]` : assembler bunu `[BX+DI-4]` olarak okur. Sıra önemsizdir; makine kodu `8A 41 FC` 'dir. Burada `FC` , −4'ün 8 bitlik ikiye tümleyenidir. Kesim DS.
- `[BP+SI+300]` : taban + indis + d16. 300, −128…127 aralığına sığmadığı için 16 bitlik yer değiştirme gerekir. Adreste BP olduğundan kesim SS.

4.1.1.4 Yer değiştirme

Yer değiştirme, adres hesabına eklenen ve komutun makine kodunun içinde duran işaretli bir sabittir. 8 bitlik biçimi (**d8**) −128 ile +127 arasındaki değerleri, 16 bitlik biçimi (**d16**) −32.768 ile +32.767 arasındaki değerleri taşır. 16 bitlik toplamada −1 ile FFFFh aynı sonucu verdiğinden d16 aslında 0000h–FFFFh aralığının tamamını kapsar. Örnek değerler: d8 için 22, 55h, −1; d16 için 300, 5517h, −259.

Yer değiştirmenin kaynağı ya bir sayıdır ya da bir değişkenin **bağıl konumu**dur. `[BX + dizi]` yazıldığında assembler `dizi` adının kesim içindeki adresini, yani bağıl konumunu yer değiştirme olarak koyar. Yazımda birden fazla sabit varsa assembler bunları derleme sırasında tek bir sayıda toplar: `[BX + 2 + 3]` , `[BX + 5]` ile aynı komuttur. Sabit köşeli parantezin içinde ya da dışında yazılabilir; `[BX+5]` , `5[BX]` ve `[BX]+5` yazımlarının üçü de `8A 47 05` baytlarını üretir. `dizi[SI]` gibi C'ye benzeyen yazım bu esneklikten gelir. Sabit negatif de olabilir: `[SI-1]` komutu SI'nın gösterdiği baytın bir öncesine erişir ve `8A 44 FF` olarak kodlanır.

Assembler iki biçimden hangisini seçeceğine değere bakarak karar verir. Değer −128…127 aralığındaysa kısa biçim (d8) seçilir ve komut bir bayt kısalır. Değilse uzun biçim (d16) gerekir. `[BX+127]` üç bayt (`8A 47 7F`), `[BX+128]` dört bayttır (`8A 87 80 00`). 128 sayısı 8 bite sığar ama işaretli olarak −128 diye okunurdu; bu yüzden 16 bitlik biçime geçilir. Değişken adlarının bağıl konumu genellikle 127'den büyük olduğu için `dizi[SI]` gibi yazımlar hemen her zaman d16 kullanır.



Şekil 4.1.2: 8 bitlik yer değiştirme, toplamaya katılmadan önce işaretiyle 16 bite genişletilir: FFh, 00FFh değil FFFFh olur ve −1 gibi davranır.

İşlemci 8 bitlik yer değiştirmeyi 16 bitlik yazmaçlarla toplamadan önce **işaret genişletme** (*sign extension*) yapar. En yüksek bit 1 ise üst bayt FFh, 0 ise 00h ile doldurulur (Şekil 4.1.2). `FC` böylece FFFCh olur. FFFCh'yi bir sayıya eklemek, 16 bitlik sınırın dışına taşan elde atıldığında 4 çıkarmakla aynıdır.

Bu son nokta adres hesabının genel bir özelliğidir. Etkin adres her zaman 16 bitle hesaplanır ve sonuç 16 bite sığmazsa taşan bit atılır; adres **başa sarar**. BX = FFFFh iken `[BX+2]` erişimi 10001h'ye değil, 0001h'ye gider. 8086, kesimin sonundan taşan bir adresi bir sonraki kesime taşımaz; aynı kesimin başına döndürür.

Örnek 4.1.1

SI = 0010h iken `MOV AL, [SI+0FFF8h]` komutu hangi bağıl konumdan okur?

Toplama 0010h + FFF8h = 1 0008h verir. Sonucun 17. biti atılır, etkin adres 0008h olur. FFF8h, 16 bitte −8'in ikiye tümleyenidir; komut aslında `[SI-8]` ile aynı yere gider. Assembler da bunu fark eder ve kısa biçimi seçer: iki yazım da `8A 44 F8` baytlarını üretir. İşlemci `F8` baytını işaret genişletmeyle yeniden FFF8h yapar.

4.1.1.5 Geçerli ve geçersiz yazımlar

Birleştirme kuralı bir yazımın geçerliliğini hızla sınamaya yarar. Tablo 4.1.3 sunumdaki örnekleri, geçersizlerin nedeniyle birlikte toplar. Geçersiz satırların hepsinde assembler hata verir.

Yazım	Durum	Açıklama
`[BX]`	geçerli	yazmaç dolaylı, DS
`[1234h]`	geçerli	doğrudan adres, DS
`[SI+10]`	geçerli	indisli, d8, DS
`[BP+DI-2]`	geçerli	tabanlı-indisli, d8, SS
`ES:[BX+SI]`	geçerli	kesim önekli, ES
`[AX]`, `[CX]`, `[DX]`	geçersiz	bu yazmaçlar adres hesabında kullanılamaz
`[BL]`, `[BH]`	geçersiz	8 bitlik yazmaç 16 bitlik bağıl konum tutamaz
`[BX+BP]`	geçersiz	iki taban yazmacı
`[SI+DI]`	geçersiz	iki indis yazmacı
`[SP]`, `[BX*2]`	geçersiz	8086'da yok; 80386 ve sonrasında 32 bitlik adreslemede vardır

Tablo 4.1.3: Geçerli ve geçersiz bellek işlenenleri.

Son satır sık karışıklık yaratır. Günümüz x86 işlemcilerinde `[ESP+4]` ya da `[EBX+ESI*4]` gibi yazımlar geçerlidir. Bunlar 80386 ile gelen 32 bitlik adreslemenin parçasıdır. 8086'nın 16 bitlik ModR/M tablosunda SP de, ölçekleme çarpanı da yoktur. Yığındaki bir değere ulaşmak için SP önce BP'ye kopyalanır, sonra `[BP+n]` kullanılır.

Sık yapılan hata

İki yazmacın toplamını adres olarak kullanmak isterken yasak birleşime düşmek. `MOV AX, [SI+DI]` geçersizdir. İki indis yazmacından biri önce bir taban yazmacına kopyalanırsa iş görülür: `MOV BX, SI` ardından `MOV AX, [BX+DI]`. Aynı yol `[BX+BP]` için de geçerlidir: BP ya da BX'ten biri SI ya da DI'ye aktarılır. Kopyalamada hangi yazmacın seçildiği varsayılan kesimi de değiştirebilir; BP'li bir adres SS'ye gider.

4.1.2 Adresleme modu türleri

4.1.2.1 İşlenen nerede durur?

Yirmi dört birleşim, işlenenin bellekte bulunduğu durumların hepsini kapsar. Ama bir komutun işleneni bellekte olmak zorunda değildir. Sabit bir sayı doğrudan komutun içine yazılabilir ya da işlenen bir yazmaçta durabilir. Kitaplar adresleme modlarını bu yüzden "işlenen nerede?" sorusuna göre adlandırır. Tablo 4.1.4 sunumdaki sınıflamayı verir.

Mod	Örnek	İşlenen nerede?
Anlık (*immediate*)	`MOV AX, 5`	komutun içinde (sabit)
Yazmaç (*register*)	`MOV AX, BX`	yazmaçta
Doğrudan (*direct*)	`MOV AX, [1000h]`	DS:1000h
Yazmaç dolaylı (*register indirect*)	`MOV AX, [BX]`	DS:BX
Tabanlı (*based*)	`MOV AX, [BP+4]`	SS:BP+4
İndisli (*indexed*)	`MOV AL, [SI+3]`	DS:SI+3
Tabanlı-indisli	`MOV AL, [BX+DI]`	DS:BX+DI
Tabanlı-indisli + yer değiştirme	`MOV AL, [BX+SI+8]`	DS:BX+SI+8

Tablo 4.1.4: Veri işlenenleri için adresleme modu türleri. Son sütun, BP'siz modlarda varsayılan kesimin DS olduğunu gösterir.

Anlık modda işlenen komutun kendisidir. `MOV AX, 5` komutunun makine kodu `B8 05 00`'dır; 5 sayısı komutun son iki baytında durur. İşlemci bu değeri komutu getirirken zaten okumuştur ve belleğe ikinci kez gitmez. Anlık değer yalnız kaynak olabilir. Bir sabitin içine yazı yazılamayacağı için `MOV 5, AX` anlamsızdır.

Yazmaç modunda işlenen işlemcinin içindedir. `MOV AX, BX` komutu yürütme biriminin içinde, veri yoluna hiç çıkmadan tamamlanır. Bu yüzden en hızlı moddur; 8086'da 2 saat turu sürer. ModR/M baytında MOD = 11 yazılarak seçilir: `MOV AX, BX` = `8B C3`, yani `C3` = 11 000 011 bitleri.

Geri kalan bütün modlar belleğe gider. Bunlarda işlemci önce bir **etkin adres** (EA, *effective address*) hesaplar. Etkin adres, işlenenin kesim içindeki konumudur, yani bir bağıl konumdur. Ardından bu bağıl konum kesim yazmacıyla birleştirilir:

> fiziksel adres = kesim × 10h + EA

Modların adları etkin adresin nasıl kurulduğunu anlatır. **Doğrudan modda** EA, komutun içindeki 16 bitlik sabittir. **Yazmaç dolaylı modda** EA bir yazmacın (BX, BP, SI ya da DI) içeriğidir. **Tabanlı modda** EA, BX ya da BP ile bir yer değiştirmenin toplamıdır. **İndisli modda** taban yerine SI ya da DI kullanılır. **Tabanlı-indisli modda** bir taban ile bir indis toplanır ve buna isteğe bağlı bir yer değiştirme eklenir. Kitaplar arasında adlandırma küçük farklar gösterir. Bazıları `[BX+4]`'ü "yazmaç göreli", bazıları "tabanlı" diye anar. Ama hesap her yerde aynıdır.

Tanım 4.1.3

Etkin adres (EA), bir bellek işleneni için köşeli parantezdeki taban yazmacı, indis yazmacı ve yer değiştirmenin 16 bitlik toplamıdır. Toplam FFFFh'yi aşarsa taşan bit atılır. EA kesimin içindeki bağıl konumdur. **Fiziksel adres**, EA'nın seçilen kesim yazmacının 16 katına eklenmesiyle elde edilen 20 bitlik adrestir.

4.1.2.2 Dizgi, G/Ç ve göreceli mod

Üç mod daha vardır. Bunlar köşeli parantezli yazımı kullanmaz. İşlenen ya komutta hiç yazılmaz ya bellekten ayrı bir adres alanında bulunur ya da adres IP'ye göre hesaplanır.

Dizgi modunda işlenenler örtüktür. `MOVSB` komutu yanında hiçbir işlenen yazılmadan bir baytı DS:SI adresinden ES:DI adresine taşır. Kaynak her zaman DS:SI, hedef her zaman ES:DI'dir. Her işlemden sonra SI ve DI kendiliğinden ilerler. Yön bayrağı DF = 0 ise artar, DF = 1 ise azalır. Bayt işleminde adım 1, sözcük işleminde 2'dir. Dizgi komutlarının ayrıntısı ayrı bir konudur. Burada önemli olan, bu komutlarda adres hesabının yazmaç dolaylı modla aynı yapıda olması ve hedefin DS değil ES kesiminde aranmasıdır.

```
MOVSB           ; ES:[DI] ← DS:[SI], sonra SI ve DI bir ilerler
```

G/Ç modunda işlenen bir giriş/çıkış **portudur**. 8086'nın portları bellekten ayrı, 64 K'lık kendi adres alanındadır. İşlemci bir porta erişirken adres yoluna port numarasını çıkarır ve M/IO̅ hattını 0 yapar. Böylece bellek yongaları değil G/Ç aygıtları yanıt verir. Veri yalnız AL ya da AX üzerinden geçer. Port numarası 0–255 aralığındaysa komutun içine yazılabilir. Daha büyük numaralar DX'e konur.

```
IN  AL, 45h      ; 45h numaralı porttan bir bayt oku
OUT DX, AX       ; DX'teki porta AX'i yaz
```

Göreceli modda (*relative addressing*) komutun içinde hedefin kendisi değil, IP'ye eklenecek işaretli bir uzaklık durur. Atlama komutları bu modu kullanır. Yeni IP, atlama komutundan hemen sonraki komutun adresi ile uzaklığın toplamıdır. Kısa atlamalarda uzaklık 8 bittir ve −128...+127 bayt aralığına ulaşır. Örneğin bir döngünün sonundaki `JNZ dongu` komutu `75 FA` diye kodlanabilir. `FA`, −6'nın 8 bitlik ikiye tümleyenidir. Komut iki bayt olduğundan atlama, komutun başından 4 bayt geriye gider. Göreceli adreslemenin büyük bir artısı vardır: program bellekte başka bir yere taşınsa da uzaklıklar değişmez.

```
JNZ dongu       ; 75 FA: IP ← sonraki komutun adresi − 6
```

> **Sık yapılan hata**
>
> Başka işlemcilerin yazımını 8086'ya taşımak. Komut adları bitişik yazılır: `MOVSB` doğrudur, `MOVS B` değildir. `OUT` komutunda önce port, sonra veri yazmacı gelir: `OUT 50h, AL`. `IN A, 45` yazımı 8085 işlemcisine aittir. 8086'da akümülatörün adı AL ya da AX'tir ve onaltılık sayılar `h` sonekiyle yazılır.

4.1.2.3 Kod izleme: altı adresleme modu

Aşağıdaki küçük COM programı, aynı altı baytlık tablodan farklı modlarla değer okur. Her adımın sonucu izleme tablosunda verilmiştir.

```
ORG 100h
MOV BX, OFFSET tablo
MOV SI, 2
MOV AH, 7          ; anlık
MOV AL, [tablo+1]  ; doğrudan
MOV CL, [BX]       ; dolaylı
MOV CH, [BX+4]     ; tabanlı
MOV DL, [BX+SI]    ; taban+indis
MOV DH, [BX+SI+3]  ; + d8
RET
tablo DB 11h,22h,33h,44h,55h,66h
```

`ORG 100h` satırı assembler'a programın 0100h bağıl konumundan başlayacağını söyler. COM programlarında kod, veri ve yığın aynı 64 KB'lık kesimdedir. emu8086 bu kesimi 0700h paragrafına yükler: CS = DS = ES = SS = 0700h. Komutların uzunlukları toplandığında `RET` 0115h'ye, `tablo` 0116h'ye düşer. Tablonun altı baytı 0116h–011Bh bağıl konumlarındadır; fiziksel adresleri 07116h–0711Bh'dir.

`MOV BX, OFFSET tablo` satırındaki `OFFSET` işleci, değişkenin içeriğini değil bağıl konumunu verir. Bu değer derleme sırasında bilinir ve komutun içine anlık değer olarak yazılır: `BB 16 01`. Sonuçta BX = 0116h olur. BX artık tablonun başını gösteren bir işaretçidir. `MOV SI, 2` indis yazmacına 2 koyar.

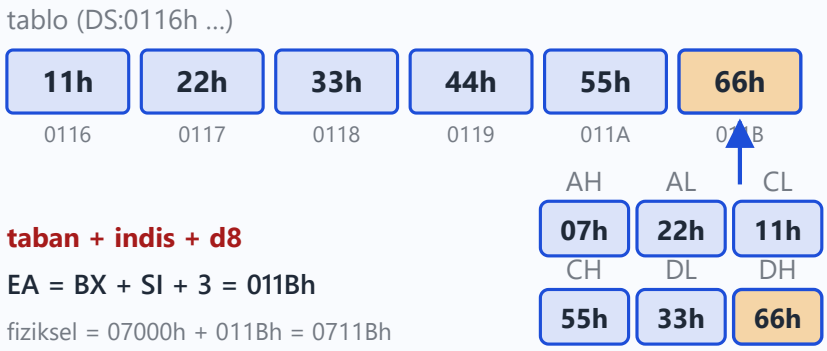
Komut	Makine kodu	Mod	EA	Fiziksel	Sonuç
`MOV AH, 7`	`B4 07`	anlık	—	—	AH = 07h
`MOV AL, [tablo+1]`	`A0 17 01`	doğrudan	0117h	07117h	AL = 22h, AX = 0722h
`MOV CL, [BX]`	`8A 0F`	yazmaç dolaylı	0116h	07116h	CL = 11h
`MOV CH, [BX+4]`	`8A 6F 04`	tabanlı	011Ah	0711Ah	CH = 55h, CX = 5511h
`MOV DL, [BX+SI]`	`8A 10`	tabanlı-indisli	0118h	07118h	DL = 33h
`MOV DH, [BX+SI+3]`	`8A 70 03`	tabanlı-indisli + d8	011Bh	0711Bh	DH = 66h, DX = 6633h

Tablo 4.1.5: Altı adresleme modunun izlemesi (DS = 0700h, BX = 0116h, SI = 0002h; tablo = 11 22 33 44 55 66).

Satırlar teker teker okunduğunda her modun farkı açıkça görülür. `MOV AH, 7` anlık moddur: 7 sayısı `B4 07` komutunun ikinci baytıdır ve belleğe ayrıca gidilmez. `MOV AL, [tablo+1]` doğrudan moddur. Assembler `tablo+1` ifadesini 0116h + 1 = 0117h olarak hesaplar ve bu adresi komuta gömer. Komut, akümülatöre özgü kısa biçimiyle `A0 17 01` olur. 0117h'deki bayt tablonun ikinci elemanı olan 22h'dir. AH daha önce 07h yapıldığından AX = 0722h olur.

`MOV CL, [BX]` yazmaç dolaylı moddur. Adres komutta yazılı değildir; işlemci onu çalışma anında BX'ten alır. EA = 0116h, okunan bayt 11h'dir. `MOV CH, [BX+4]` tabanlı moddur: EA = 0116h + 4 = 011Ah, okunan değer tablonun beşinci elemanı 55h'dir. CX = 5511h olur. Bu iki komut arasındaki fark, sabit bir uzaklığın eklenmesidir. Bir kaydın alanlarına erişirken BX kaydın başını, yer değiştirme alanın kayıt içindeki yerini tutar.

`MOV DL, [BX+SI]` tabanlı-indisli moddur: EA = 0116h + 0002h = 0118h, DL = 33h. Son komut buna bir de 8 bitlik yer değiştirme ekler: EA = 0116h + 2 + 3 = 011Bh, DH = 66h. Bu altı erişimin hiçbirinde BP kullanılmadığı için kesim hep DS'dir. Fiziksel adres DS × 10h + EA = 07000h + EA ile bulunur.



Şekil 4.1.3: Altı adresleme modunun son adımı: `MOV DH, [BX+SI+3]` etkin adresi 011Bh olan baytı, yani tablonun son elemanını (66h) okur.

Programın son durumu AX = 0722h, BX = 0116h, CX = 5511h, DX = 6633h, SI = 0002h'dir. `RET` yığından 0000h değerini çeker ve IP = 0000h olur. COM programının yüklendiği kesimin 0000h bağıl konumunda DOS'un yerleştirdiği `INT 20h` komutu bulunur ve program biter. MOV komutları bayrakları

değiştirmediğinden bütün izleme boyunca bayraklar başlangıçtaki gibi kalır: yalnız IF = 1.

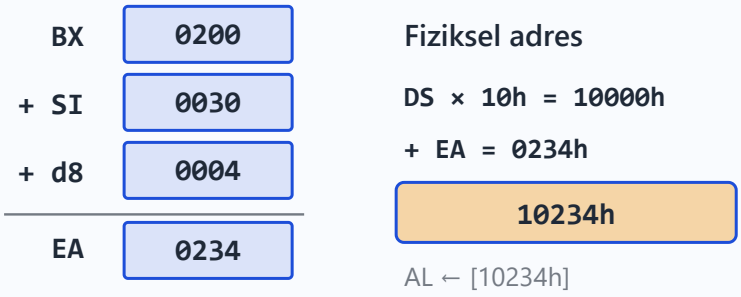
4.1.3 Etkin adres hesabı

4.1.3.1 Adım adım: DS ile

Etkin adres hesabı, yürütme biriminin (EU) içindeki bir toplama ile yapılır. Sonuç, kesim yazmacıyla birleştirilmek üzere yol arayüz birimine (BIU) verilir. BIU'nun kendi toplayıcısı kesimi 4 bit sola kaydırıp EA'yı ekler ve 20 bitlik adresi A19–A0 hatlarına çıkarır. Bu iki aşama bir örnekle izlenebilir:

```
MOV AL, [BX + SI + 4]   ; DS = 1000h, BX = 0200h, SI = 0030h
```

1. Taban yazmacı BX = 0200h alınır.
2. İndis yazmacı SI = 0030h eklenir: 0230h.
3. 8 bitlik yer değiştirme +4 eklenir.
4. EA = 0200h + 0030h + 0004h = 0234h. Toplama 16 bitliktir; burada taşma yoktur.
5. Adreste BP olmadığından kesim DS'dir. Fiziksel adres = 1000h × 10h + 0234h = 10000h + 0234h = 10234h. AL, bu adresteki baytı alır.



Şekil 4.1.4: `MOV AL, [BX + SI + 4]` için etkin adres ve fiziksel adres: BX + SI + 4 = 0234h; DS = 1000h ile 10234h.

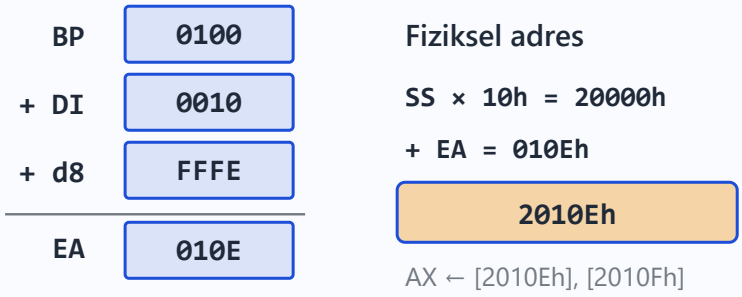
Kesimi 10h ile çarpmak, onaltılık yazımda sağa bir sıfır eklemek demektir: 1000h → 10000h. İkili düzeyde bu, 16 bitlik sayıyı 4 bit sola kaydırıp en düşük dört bite 0 koymaktır. Fiziksel adres bu yüzden kolay hesaplanır: kesimin sonuna bir 0 yazılır, altına EA sağa yaslanarak yazılır ve iki sayı toplanır.

4.1.3.2 Adım adım: BP ile

Adreste BP bulunursa aynı hesap SS kesimiyle yapılır. Sunumdaki ikinci örnek bir sözcük okur ve negatif bir yer değiştirme kullanır:

```
MOV AX, [BP + DI - 2]   ; SS = 2000h, BP = 0100h, DI = 0010h
```

1. Taban yazmacı BP = 0100h alınır.
2. İndis yazmacı DI = 0010h eklenir: 0110h.
3. Yer değiştirme −2'dir; makine kodunda `FE` baytı olarak durur ve işaret genişletmeyle FFFEh olur.
4. EA = 0100h + 0010h + FFFEh = 1 010Eh → 010Eh. Taşan bit atılır; sonuç 0110h − 2 ile aynıdır.
5. BP kullanıldığı için kesim SS'dir: 20000h + 010Eh = 2010Eh. AX bir sözcük olduğundan iki bayt okunur: AL ← [2010Eh], AH ← [2010Fh].



Şekil 4.1.5: `MOV AX, [BP + DI - 2]` : EA = 010Eh; BP kullanıldığı için kesim SS ve fiziksel adres 2010Eh. Sözcüğün ikinci baytı 2010Fh'dedir.

Bu komutun makine kodu `8B 43 FE`'dir. ModR/M baytı `43` = 01 000 011: MOD = 01 (d8 var), REG = 000 (AX), R/M = 011 (`[BP+DI]`). Komutun hiçbir yerinde "SS" yazmaz. Kesim seçimi R/M alanının kendisinden gelir: R/M değeri BP içeren bir birleşimse donanım SS'yi, değilse DS'yi seçer.

Sözcük okumanın veri yolu açısından bir ayrıntısı vardır. 8086'nın veri yolu 16 bittir, ama bellek bayt bayt adreslenir. Çift adresteki bir sözcüğün iki baytı tek bir yol döngüsünde birlikte gelir. Bu örnekteki 2010Eh çift olduğundan okuma tek döngüdür. Sözcük tek numaralı bir adreste başlasaydı, örneğin 2010Fh'de, işlemci önce düşük baytı yolun yüksek yarısından, sonra yüksek baytı ayrı bir döngüde yolun düşük yarısından okurdu. Sonuç aynı olur ama erişim bir yol döngüsü, yani en az 4 saat turu uzar. Bu yüzden assembler'larda sözcük değişkenlerini çift adrese hizalayan `EVEN` direktifi vardır.

4.1.3.3 Etkin adres hesaplayıcısı

Sunumdaki hesaplayıcı ve Adresleme Modları simülatörü aynı hesabı etkileşimli olarak yapar. Taban sütunundan BX, BP ya da hiçbiri; indis sütunundan SI, DI ya da hiçbiri seçilir ve bir yer değiştirme girilir. Yazmaç ve kesim değerleri onaltılık, yer değiştirme onluk (−2) ya da onaltılık (1Ah) yazılabilir. Araç geçerli bir birleşim için EA'yı, varsayılan kesimi ve fiziksel adresi verir. Elle yapılan hesabın denetimi için uygundur. Aynı hesap kâğıt üzerinde üç kurala dayanır: toplamayı 16 bitte yapmak, BP varsa SS'yi seçmek ve kesimi 10h ile çarpmak.

Örnek 4.1.2

DS = 3000h, SS = 4000h, BX = 0010h, BP = FFFEh, SI = 0005h. `[BX+SI+20h]` ve `[BP+4]` işlenenlerinin etkin ve fiziksel adresleri nedir?

Birinci işlenende BP yoktur. EA = 0010h + 0005h + 0020h = 0035h, kesim DS: 30000h + 0035h = 30035h.

İkinci işlenende EA = FFFEh + 0004h = 1 0002h'dir. 17. bit atılır ve EA = 0002h olur. BP olduğundan kesim SS: 40000h + 0002h = 40002h. Sarma olmasaydı adres 50002h çıkardı. Bu değer SS kesiminin dışındadır ve 8086 böyle bir adres üretmez.

4.1.3.4 Veri yapısına göre mod seçimi

Aynı bayta birden çok modla ulaşılabilir. Doğru modu seçmek için verinin hangi parçasının derleme sırasında sabit, hangisinin çalışma sırasında değişken olduğuna bakılır. Sabit parça yer değiştirmeye, değişken parçalar yazmaçlara konur. Tablo 4.1.6 yaygın durumları özetler.

Erişilen veri	Uygun mod	Örnek
Tek bir değişken	doğrudan	`MOV AX, sayac`
İşaretçinin gösterdiği eleman	yazmaç dolaylı	`MOV AL, [SI]`
Dizinin i. elemanı (dizi sabit, indis değişken)	indisli + d16	`MOV AL, dizi[SI]`
Bir kaydın (yapının) alanı	tabanlı + d8	`MOV AX, [BX+2]`
Matris ya da kayıt dizisi	taban + indis + d	`MOV AL, mat[BX+SI]`
Yığındaki parametre ya da yerel değişken	tabanlı, BP ile (SS)	`MOV AX, [BP+4]`
Başka kesimdeki tampon	kesim önekli	`MOV AL, ES:[DI]`

Tablo 4.1.6: Erişilen veriye göre uygun adresleme modu.

Tablonun arkasındaki düşünce şudur. Taban yazmacı hangi kaydın ya da hangi satırın işlendiğini gösterir. Kayıt değiştiğinde yalnız taban yazmacı güncellenir. İndis yazmacı kaydın içindeki elemanı sayar ve döngüde artırılır. Yer değiştirme dizinin ya da alanın sabit konumudur; assembler onu komuta gömer. Örneğin her biri 6 baytlık öğrenci kayıtlarından oluşan bir dizide, kaydın 4. baytındaki not alanı `[BX+4]` ile okunur. Sonraki kayda geçmek için `ADD BX, 6` yeterlidir; okuma komutu aynı kalır.

Yığın satırı ayrı bir öneme sahiptir. Bir alt programa yığın üzerinden parametre geçirildiğinde, alt program SP'yi BP'ye kopyalar ve parametrelere `[BP+4]`, `[BP+6]` gibi sabit uzaklıklarla ulaşır. BP'nin varsayılan kesiminin SS olması bu kullanım için seçilmiştir: yığın SS kesimindedir.

4.1.3.5 Kod izleme: matris elemanına erişim

Taban + indis + yer değiştirme modunun en açık kullanımı iki boyutlu bir dizidir. Aşağıdaki programda `mat`, 3 satır ve 4 sütundan oluşan bir bayt matrisidir. Eleman değerleri bir bakışta tanınsın diye seçilmiştir: onaltılık değerin ilk basamağı satırın, ikinci basamağı sütunun 1'den başlayan sırasıdır. Örneğin 32h üçüncü satırın ikinci elemanıdır. Program satır ve sütunları 0'dan saydığı için bu eleman `mat[2][1]` diye anılır.

```
org 100h
mov bx, 2*4          ; satır 2
mov si, 1            ; sütun 1
mov al, mat[bx+si]   ; mat[2][1]
mov si, 3            ; sütun 3
mov ah, mat[bx+si]   ; mat[2][3]
sub bx, 4            ; satır 1
mov dl, mat[bx+si]   ; mat[1][3]
ret
mat db 11h,12h,13h,14h
    db 21h,22h,23h,24h
    db 31h,32h,33h,34h
```

Matris bellekte **satır öncelikli düzende** (*row-major order*) saklanır: önce 0. satırın dört baytı, ardından 1. satırın, en son 2. satırın baytları gelir. `mat` 0119h'de başlar ve 0124h'de biter. Bir elemanın adresi şu formülle bulunur:

> adres(mat[satır][sütun]) = mat + satır × sütun sayısı + sütun

Programda `satır × sütun sayısı` terimi BX'e, `sütun` terimi SI'ya, `mat`'ın bağıl konumu yer değiştirmeye konmuştur. `mat[bx+si]` yazımı `[bx+si+0119h]` ile aynıdır ve `8A 80 19 01` diye kodlanır. 0119h, 127'den büyük olduğu için d16 kullanılır.

Adım	Komut	BX	SI	EA	AX	DL
1	`mov bx, 2*4`	0008h	0000h	—	0000h	00h
2	`mov si, 1`	0008h	0001h	—	0000h	00h
3	`mov al, mat[bx+si]`	0008h	0001h	0122h	0032h	00h
4	`mov si, 3`	0008h	0003h	—	0032h	00h
5	`mov ah, mat[bx+si]`	0008h	0003h	0124h	3432h	00h
6	`sub bx, 4`	0004h	0003h	—	3432h	00h
7	`mov dl, mat[bx+si]`	0004h	0003h	0120h	3432h	24h

Tablo 4.1.7: Matris erişiminin izlemesi (mat = 0119h, satır uzunluğu 4 bayt).

Adım 1'de BX = 2 × 4 = 8 olur. Çarpmayı assembler derleme sırasında yapar ve komutun içine 0008h yazar. 2. satır matrisin başından 8 bayt ileridedir. Adım 2'de SI = 1 sütunu seçer. Adım 3'te EA = 0119h + 8 + 1 = 0122h olur ve AL = 32h okunur. Bu, matrisin 2. satırının 1. sütunudur (ikisi de 0'dan sayılarak).

Adım 4 ve 5 aynı satırda kalıp sütunu değiştirir. BX dokunulmadan kaldığı için satır sabittir. EA = 0119h + 8 + 3 = 0124h ve AH = 34h olur; AX = 3432h. Okuma komutu öncekinin aynısıdır, yalnız indis yazmacının değeri değişmiştir. Adım 6'daki `sub bx, 4` bir satır yukarı çıkar: BX = 0004h. SI = 3 korunduğu için sütun aynı kalır. Adım 7'de EA = 0119h + 4 + 3 = 0120h, DL = 24h olur. Bu, 1. satırın 3. sütunudur.



Şekil 4.1.6: Matris erişiminin 3. adımı: BX satırı (8 bayt ileri), SI sütunu (1) seçer; EA = 0122h ve AL = 32h.

`sub bx, 4` MOV'dan farklı olarak bayrakları değiştirir. 8 − 4 = 4 sonucu sıfır değildir ve elde gerekmez; ZF = 0, CF = 0, SF = 0 olur. 4 = 0000 0100b'de tek sayıda 1 bulunduğundan eşlik bayrağı PF = 0'dır. Program sonunda AX = 3432h, BX = 0004h, DL = 24h'dir. Üç farklı eleman tek bir adresleme kalıbıyla okunmuştur.

Sözcüklerden oluşan bir matriste formüldeki iki terim de 2 ile çarpılır: adres = mat + (satır × sütun sayısı + sütun) × 2. Bir sözcük 2 bayt yer kaplar ve indis yazmaçları her zaman bayt sayar.

4.1.3.6 Etkin adres hesap süresi

8086'da etkin adres, EU içinde ayrı bir toplama ile hesaplanır. 80286 ve sonraki işlemcilerde bu iş için ayrı bir adres birimi vardır. 8086'da ise EA hesabının süresi komutun kendi süresine eklenir ve moda göre değişir. Tablo 4.1.8, Intel 8086 kullanıcı kılavuzundaki EA süre tablosunu verir.

Etkin adres birleşimi	Saat turu
`[BX]`, `[BP]`, `[SI]`, `[DI]`	5
doğrudan `[d16]`	6
`[BP+DI]`, `[BX+SI]`	7
`[BP+SI]`, `[BX+DI]`	8
tek yazmaç + d8/d16	9

Etkin adres birleşimi	Saat turu
`[BP+DI+d]`, `[BX+SI+d]`	11
`[BP+SI+d]`, `[BX+DI+d]`	12
kesim öneki	+2

Tablo 4.1.8: 8086'da etkin adres hesabının süresi (saat turu).

Bellekten yazmaca taşıyan genel MOV komutu 8 tur artı EA süresi tutar. Yazmaçtan yazmaca MOV 2 turdur. Buna göre:

- `MOV AX, BX` : 2 tur. Belleğe gidilmez.
- `MOV CL, [BX]` : 8 + 5 = 13 tur.
- `MOV CL, [BX+DI+4]` : 8 + 12 = 20 tur.
- `MOV CL, ES:[BX]` : 8 + 5 + 2 = 15 tur.

Tablodaki tuhaf bir ayrıntı dikkat çeker: `[BX+SI]` 7, `[BX+DI]` 8 turdur. Bu fark 8086'nın iç düzeninden gelir ve programcı için küçük bir seçim ölçütü olabilir. Tek adreste başlayan bir sözcük ise ikinci bir yol döngüsü gerektirdiğinden aktarım başına 4 tur daha ekler. 5 MHz'lik bir 8086'da bir saat turu 200 ns'dir. `MOV CL, [BX+DI+4]` böylece 20 × 200 ns = 4 µs sürer; aynı iş yazmaçtan yazmaca 0,4 µs'dir.

Bu farklar, sık okunan bir değeri döngü boyunca bir yazmaçta tutmanın neden hızlı olduğunu açıklar. Döngünün her turunda belleğe gidilmez ve EA hesabı tamamen ortadan kalkar. AL ve AX'e özgü doğrudan adresli kısa biçim (`A0` / `A1`) de ayrıca hızlıdır: EA hesabı içermez ve 10 tur sürer. `MOV CL, [1234h]` ise 8 + 6 = 14 turdur.

4.1.4 Kesim ve bağıl konum

4.1.4.1 Kesim:bağıl konumdan fiziksel adrese

8086'nın adres yolu 20 hattan oluşur: A19–A0. 20 bitle 2^{20} = 1.048.576 farklı bayt, yani 1 MB adreslenebilir. İşlemcinin yazmaçları ise 16 bittir ve tek bir yazmaç ancak 64 KB'lık bir alanı tarayabilir. Tasarımcılar bu uyumsuzluğu iki 16 bitlik sayıyı birleştirerek çözmüştür. Birinci sayı belleğin hangi bloğunun kullanıldığını, ikinci sayı o bloğun içindeki konumu söyler.

Tanım 4.1.4

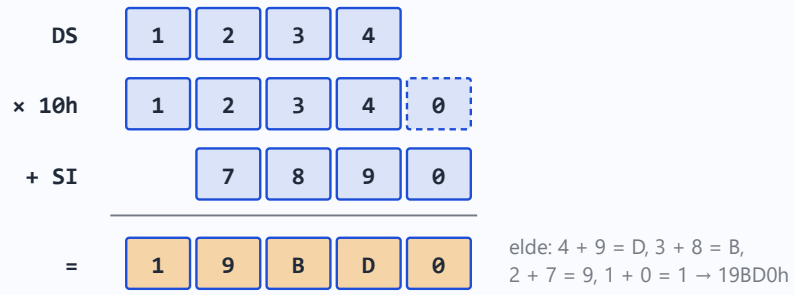
Bir kesim yazmacındaki (CS, DS, SS, ES) değere **kesim** (*segment*) denir. Etkin adres hesabından ya da IP, SP gibi bir yazmaçtan gelen adrese **bağıl konum** (*offset*) denir. Bir adres `kesim:bağıl konum` biçiminde yazılır. Fiziksel adres

$$\text{fiziksel adres} = \text{kesim} \times 10\text{h} + \text{bağıl konum}$$

ile bulunur. Kesim belleğin 16 baytlık bir sınırda başlayan bloğunu, bağıl konum o bloğun içindeki baytı gösterir.

Sunumdaki örnekte DS = 1234h ve SI = 7890h'dir. Adres `1234:7890` diye yazılır. Hesap dört adımda yapılır (Şekil 4.1.7):

1. Kesim ve bağıl konum alınır: 1234h ve 7890h.
2. Kesim 10h ile çarpılır, yani 4 bit sola kaydırılır: 12340h. Onaltılık yazımda bu, sağa bir 0 eklemektir.
3. Bağıl konum, toplamaya girmek üzere sağa yaslanır: 07890h.
4. İki sayı toplanır: 12340h + 07890h = 19BD0h.



Şekil 4.1.7: 1234:7890 adresinin fiziksel karşılığı: kesim 4 bit sola kaydırılır (12340h), bağıl konum (7890h) eklenir; sonuç 19BD0h.

Toplama basamak basamak yapıldığında 0 + 0 = 0, 4 + 9 = 13 = Dh, 3 + 8 = 11 = Bh, 2 + 7 = 9, 1 + 0 = 1 bulunur. Hiçbir basamakta elde oluşmadığından sonuç kolayca okunur. Elde oluşan bir örnek 1A5D:FFF3 çiftidir. 1A5D0h + 0FFF3h hesabında sağdan başlanır: 0 + 3 = 3; D + F = 13 + 15 = 28 = 1Ch, basamağa C yazılır ve 1 elde aktarılır; 5 + F + 1 = 21 = 15h, basamağa 5; A + F + 1 = 26 = 1Ah, basamağa A; 1 + 0 + 1 = 2. Sonuç 2A5C3h'dir. Onaltılık toplamada bir basamak 15'i aşarsa 16 çıkarılır ve bir sonraki basamağa 1 elde aktarılır.

Bu toplama donanımda BIU'nun içindeki ayrı bir 20 bitlik toplayıcıyla yapılır. Kesim yazmacının 16 biti toplayıcının yüksek 16 girişine, en düşük 4 girişe 0 bağlanarak verilir. Bu bağlantı "10h ile çarpma" işinin bütün devresidir; ayrı bir çarpıcı yoktur. Toplayıcının çıkışı adres mandallarına ve oradan

A19–A0 hatlarına gider. Komut getirirken BIU aynı toplayıcıyı CS ve IP ile kullanır.

Kesimin 16 ile çarpılmasının bir sonucu, kesimlerin yalnız 16'nın katı olan adreslerde başlayabilmesidir. Bu 16 baytlık birime **paragraf** denir. Kesim yazmacındaki sayı, kesimin kaçıncı paragrafta başladığını söyler. Bir kesimin bağıl konumu 16 bit olduğundan kesim en çok 2^{16} = 64 KB uzunluğundadır. Ardışık iki kesim değeri yalnız 16 bayt aralıklı başladığından kesimler birbirinin üstüne biner. Aynı fiziksel bayta pek çok farklı `kesim:bağıl konum` çiftiyle ulaşılabilir. Örneğin 0700:0100, 0710:0000 ve 06F0:0200 çiftlerinin üçü de 07100h adresini gösterir.

Toplam 20 bitten taşarsa ne olur? FFFF:0010 çiftinin toplamı FFFF0h + 0010h = 100000h'dir; 21 bit gerekir. 8086'da 21. hat yoktur ve taşan bit atılır. Adres 00000h'ye, belleğin başına sarar. FFFF:0010 ile FFFF:FFFF arasındaki adresler böylece 00000h–0FFEFh bölgesine düşer.

4.1.4.2 Adres hatları ve bellek kapasitesi

Adres hattı sayısı ile bellek büyüklüğü arasındaki ilişki sayısal tasarımdaki kod çözücülerden tanıdıktır. *n* adres hattı 2^n farklı ikili örüntü üretir ve her örüntü bir bellek konumunu seçer. Bir konumda kaç bit bulunduğunu ise veri hattı sayısı belirler:

> kapasite = 2^n × veri genişliği

8086'nın 20 adres hattı 2^{20} = 1.048.576 konum seçer. Her konum bir bayt olduğundan kapasite 1 MB'tır ve adresler 00000h ile FFFFFh arasındadır. Bir kesimin sınırı ise bağıl konumdan gelir: 16 bit → 2^{16} = 65.536 bayt = 64 KB. Tablo 4.1.9 farklı hat sayıları için kapasiteyi verir.

Adres / veri hattı	Konum sayısı	Kapasite
12 / 8	2^{12} = 4096	4096 × 1 B = 4 KB
16 / 4	2^{16} = 64 K	64 K × 4 bit = 256 Kbit = 32 KB
16 / 8	64 K	64 KB (bir kesim)
20 / 8	1 M	1 MB (8086)
24 / 16	16 M	16 M × 2 B = 32 MB
32 / 8	4 G	4 GB (80386)

Tablo 4.1.9: Adres ve veri hattı sayısına göre bellek kapasitesi.

Bir bellek yongasında veri hattı sayısı, yonganın sözcük genişliğidir. Kapasite bayt cinsinden istenirse toplam bit sayısı 8'e bölünür. 16 adres ve 4 veri hattı olan bir yonga 65.536 × 4 = 262.144 bit, yani 32.768 bayt tutar. 8086 sisteminde bu tür yongalar yan yana bağlanarak 8 ya da 16 bitlik bir veri yolu kurulur.

Soru tersine de sorulabilir: belirli bir kapasite için en az kaç adres hattı gerekir? Bunun için kapasiteye eşit ya da ondan büyük ilk 2'nin kuvveti bulunur. Bellek bayt adresli ise 6 GB için 2^{32} = 4 GB yetmez; 2^{33} = 8 GB yeter ve 33 hat gerekir. 4 KB için 2^{12} = 4096 tam olarak yeter; 12 hat gerekir (A0–A11).

Sık yapılan hata

Bit ile baytı karıştırmak. 20 adres ve 16 veri hattı olan bir yonga $2^{20} \times 16$ bit = 16 Mbit tutar. Bu 2 MB'tır, 16 MB değildir. Kapasite yazılırken birimin bit mi bayt mı olduğu açıkça belirtilmelidir. 8086 belleği bayt adreslidir; orada 20 hat 1 MB demektir.

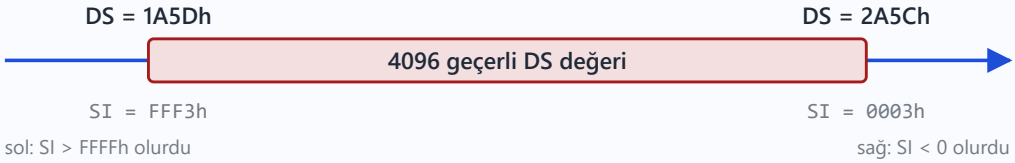
4.1.4.3 Ters hesap: fiziksel adresten kesim:bağıl konuma

Fiziksel adres verilip kesim ve bağıl konum istendiğinde tek bir cevap yoktur. Kesimler üst üste bindiği için pek çok çift aynı adresi verir. Doğru soru, kesim yazmacının alabileceği değerlerin aralığıdır. Sunumdaki örnekte 2A5C3h adresine DS:SI ile erişilmek isteniyor.

En büyük DS. SI'nın en küçük olduğu durumdur. DS × 10h'nin 2A5C3h'yi aşmaması gerekir. Adresin ilk dört onaltılık basamağı alınırsa DS = 2A5Ch olur. DS × 10h = 2A5C0h ve SI = 2A5C3h − 2A5C0h = 0003h kalır. DS 2A5Dh olsaydı 2A5D0h zaten 2A5C3h'den büyük olurdu.

En küçük DS. SI'nın en büyük olduğu durumdur. SI en çok FFFFh olabilir. Bu durumda DS × 10h ≥ 2A5C3h − FFFFh = 1A5C4h olmalıdır. 1A5C4h'yi 10h'ye bölmek 1A5C,4h verir. DS tam sayı olduğu için yukarı yuvarlanır: DS = 1A5Dh. O zaman SI = 2A5C3h − 1A5D0h = FFF3h olur.

Kaç çift? DS her bir arttığında kesimin başı 10h ileri gider ve aynı adres için SI 10h azalır. DS, 1A5Dh ile 2A5Ch arasındaki her değeri alabilir. Çift sayısı 2A5Ch − 1A5Dh + 1 = 0FFFh + 1 = 1000h = 4096'dır.



Şekil 4.1.8: 2A5C3h adresi için DS'nin alabileceği değerler: en küçük 1A5Dh (SI = FFF3h), en büyük 2A5Ch (SI = 0003h); aralıkta 4096 çift vardır.

Böyle bir hesabın denetimi her zaman ileri yönde yapılır. 1A5Dh × 10h + FFF3h = 1A5D0h + FFF3h = 2A5C3h. 2A5Ch × 10h + 0003h = 2A5C0h + 3 = 2A5C3h. İki uç da aynı fiziksel adresi verir.

Aralığın uzunluğu her adres için aynı değildir. Çok küçük adreslerde DS'nin alt sınırı 0'a dayanır. Örneğin 00345h adresi için DS en küçük 0000h, en büyük 0034h'dir ve yalnız 35h = 53 çift vardır. 0FFF0h ve üstündeki adreslerde aralık her zaman 4096 çifttir. Bu sayımda 1 MB sınırındaki başa sarma hesaba katılmamıştır; FFFF:0355 gibi sarmalı çiftler de 00345h'ye gider.

4.1.4.4 Kesim öneki

Varsayılan kesim kuralı çoğu durumda doğru kesimi seçer. Bazen ise program, veri kesiminin dışındaki bir bayta ulaşmak ister. Bunun için köşeli parantezin önüne bir kesim yazmacının adı ve iki nokta yazılır. Bu yazıma **kesim öneki** (*segment override prefix*) denir.

```
MOV AX, 0B800h
MOV ES, AX          ; ES = video belleği
MOV BX, 0
MOV AL, ES:[BX]     ; ES:0000'dan oku
MOV AL, [BX]        ; DS:0000'dan oku
MOV DL, SS:[SI]     ; SS:SI'dan oku
```

Önek makine kodunda komutun önüne eklenen bir baytlık bir koddur: ES: = `26h`, CS: = `2Eh`, SS: = `36h`, DS: = `3Eh`. İşlemci bu baytı okuyunca yalnız arkasından gelen komut için varsayılan kesimi değiştirir. Önek bir sonraki komuta taşınmaz. `MOV AL, ES:[BX]` böylece `26 8A 07` olur. Öneksiz `MOV AL, [BX]` komutunun `8A 07` kodunun önüne `26` eklenmiştir. EA hesabına da 2 saat turu eklenir.

Önek üç durumda gerekir:

- DS'yi değiştirmeden başka bir kesimdeki veriye erişmek. Video belleği en sık örnektir.
- Yığındaki bir veriye BP dışındaki bir yazmaçla erişmek: `SS:[BX]`.
- Kod kesimindeki bir tabloya erişmek: `CS:[SI]`. Bazı programlar sabit tabloları kodun yanına koyar; DS başka bir yeri gösterse de tablo CS ile okunur.

Her önek her erişimi değiştiremez. IP her zaman CS ile, SP her zaman SS ile kullanılır; bu eşleşmeler donanıma bağlıdır. Komut getirme işlemi CS:IP'den, yığın işlemleri (PUSH, POP, CALL, RET) SS:SP'den yapılır. Dizgi komutlarının hedefi olan ES:DI de değiştirilemez. Dizgi kaynağı DS:SI ise önekle başka bir kesime yönlendirilebilir.

> **Tanım 4.1.5**
>
> **Varsayılan kesim** (*default segment*), bir bellek erişiminde önek yoksa kullanılan kesimdir. Komut getirme için CS (IP ile), yığın işlemleri için SS (SP ile), BP içeren adresler için SS, dizgi hedefi için ES (DI ile), diğer bütün veri erişimleri için DS'dir. Kesim öneki, veri erişimlerinin varsayılan kesimini yalnız o komut için değiştirir.

4.1.4.5 Kod izleme: ES ile video belleği

IBM PC uyumlu bilgisayarlarda renkli metin ekranının içeriği B8000h fiziksel adresinden başlayan bellekte durur. Ekran kartı bu belleğe yazılan her baytı hemen ekrana yansıtır. Ekranın her karakter hücresi iki bayttır: önce karakterin ASCII kodu, sonra rengi belirleyen **öznitelik baytı**. Bir satır 80

hücre, yani 160 bayttır. Aşağıdaki program ES ile ekranın 1. satırının ilk hücresine sarı bir K yazar ve aynı bağıl konumu bir de DS ile okur.

```
ORG 100h
MOV AX, 0B800h
MOV ES, AX          ; ES = video
MOV BX, 00A0h       ; satır 1
MOV BYTE PTR ES:[BX], 'K'
MOV BYTE PTR ES:[BX+1], 1Eh
MOV AL, [BX]        ; DS:00A0h
MOV DL, ES:[BX]     ; B800:00A0h
RET
```

IP	Komut	Makine kodu	Erişilen adres	Sonuç
0100h	`MOV AX, 0B800h`	`B8 00 B8`	—	AX = B800h
0103h	`MOV ES, AX`	`8E C0`	—	ES = B800h, DS = 0700h
0105h	`MOV BX, 00A0h`	`BB A0 00`	—	BX = 00A0h
0108h	`MOV BYTE PTR ES:[BX], 'K'`	`26 C6 07 4B`	B80A0h	[B80A0h] = 4Bh
010Ch	`MOV BYTE PTR ES:[BX+1], 1Eh`	`26 C6 47 01 1E`	B80A1h	[B80A1h] = 1Eh
0111h	`MOV AL, [BX]`	`8A 07`	070A0h	AL = 00h
0113h	`MOV DL, ES:[BX]`	`26 8A 17`	B80A0h	DL = 4Bh
0116h	`RET`	`C3`	yığın	IP = 0000h

Tablo 4.1.10: Kesim öneki programının izlemesi (DS = 0700h; video hücresi başlangıçta 20h 07h).

İlk iki komut ES'yi kurar. 8086'da bir kesim yazmacına sabit doğrudan yüklenemez; B800h önce AX'e, oradan ES'ye aktarılır. DS hâlâ programın kendi kesimini (0700h) gösterir. Bu sayede programın değişkenlerine erişim bozulmaz. Üçüncü komut BX'e 00A0h = 160 koyar. Bu sayı (1 × 80 + 0) × 2 hesabından gelir ve 1. satırın ilk hücresinin video belleğindeki bağıl konumudur (satırlar 0'dan sayılır).

Dördüncü komut `ES:` önekiyle başlar. Makine kodundaki `26` baytı, arkasından gelen `C6 07 4B` ("BX'in gösterdiği baytı 4Bh yap") komutunun ES kesimine gitmesini sağlar. Fiziksel adres B8000h + 00A0h = B80A0h'dir ve buraya 'K' harfinin ASCII kodu 4Bh yazılır. Beşinci komut bir sonraki bayta 1Eh yazar. 1Eh = 0001 1110b öznitelik baytıdır: zemin rengi bitleri (6–4) 001 = mavi, yazı rengi bitleri (3–0) 1110 = sarı. Ekranda mavi zemin üzerinde sarı bir K belirir.

Altıncı komut önek taşımaz. `[BX]` varsayılan olarak DS'yi kullanır ve 07000h + 00A0h = 070A0h adresini okur. Bu adres COM programının önündeki 256 baytlık **PSP** (program kesimi öneki) alanına düşer; örnekte orada 00h vardır ve AL = 00h olur. Aynı bağıl konum farklı bir kesimle bambaşka bir bayta gitmiştir. Yedinci komut yeniden `ES:` önekini kullanır ve video belleğine yazılan 4Bh'yi okur: DL = 4Bh.



Şekil 4.1.9: Aynı bağıl konum (00A0h) iki kesimde: önek yokken DS ile 070A0h okunur (AL = 00h); `ES:` önekiyle B80A0h'deki 'K' (4Bh) okunur.

Programın sonunda ES = B800h, DS = 0700h'dir. Önek yalnız yazıldığı komutu etkilemiş, DS'ye hiç dokunmamıştır.

4.1.4.6 Kod izleme: BP ve SS tuzağı

COM programlarında CS, DS, SS ve ES aynı değeri taşır. Bu yüzden bir COM programında yanlış kesimin seçilmesi çoğu zaman fark edilmez. Kesimleri ayrı olan bir EXE programında ise BP ile veriye erişmek yanlış bayt getirir. Aşağıdaki program bu tuzağı gösterir.

```
veri segment
bos db 3 dup(0)
x db 'D'
veri ends
yigin segment stack
dw 64 dup(0)
yigin ends
kod segment
assume cs:kod, ds:veri, ss:yigin
basla:
mov ax, veri
mov ds, ax
mov bp, offset x
mov al, [bp]        ; SS:BP okunur!
mov bl, ds:[bp]     ; DS:BP
mov si, offset x
mov cl, [si]        ; DS:SI
mov ax, 4C00h
int 21h
kod ends
end basla
```

Programda üç kesim tanımlanmıştır. `veri` kesimi dört bayttır: üç sıfır baytı ve 'D' harfini taşıyan `x`. `x`'in bağıl konumu 0003h'dir. `yigin` kesimi 64 sözcüklük, sıfırla dolu bir yığındır. `stack` sözcüğü bağlayıcıya bunun programın yığını olduğunu söyler. Yükleyici programı açarken SS'yi bu kesime, SP'yi onun sonuna (0080h) kurar. `assume` direktifi kod üretmez; assembler'a hangi kesim yazmacının hangi kesimi göstereceğini bildirir.

Programın yükleme yerleşimi şöyledir. DOS, PSP'yi 0700h paragrafına koyar; programın görüntüsü 256 bayt ileride, 0710h'de başlar. Kesimler paragraf sınırlarına hizalanarak tanım sırasıyla dizilir. Dört baytlık `veri` bir paragraf tutar: 0710h. `yigin` hemen arkasında 0711h'de, kod ise 128 baytlık yığından sonra 0719h'de başlar. Bu değerler örnek bir yerleşimdir. Gerçek bir DOS'ta paragraf numaraları belleğin dolu kısmına göre değişir; hesabın mantığı aynı kalır.

Komut	Makine kodu	DS	SS	Erişilen adres	Sonuç
`mov ax, veri`	`B8 10 07`	0700h	0711h	—	AX = 0710h
`mov ds, ax`	`8E D8`	0710h	0711h	—	DS = 0710h
`mov bp, offset x`	`BD 03 00`	0710h	0711h	—	BP = 0003h
`mov al, [bp]`	`8A 46 00`	0710h	0711h	SS:0003 = 07113h	AL = 00h
`mov bl, ds:[bp]`	`3E 8A 5E 00`	0710h	0711h	DS:0003 = 07103h	BL = 44h ('D')
`mov si, offset x`	`BE 03 00`	0710h	0711h	—	SI = 0003h
`mov cl, [si]`	`8A 0C`	0710h	0711h	DS:0003 = 07103h	CL = 44h ('D')
`mov ax, 4C00h` / `int 21h`	`B8 00 4C` / `CD 21`	0710h	0711h	—	DOS'a dönüş, kod 0

Tablo 4.1.11: BP ile veriye erişimin izlemesi (veri = 0710h, yigin = 0711h; x = 0710:0003 = 07103h).

Program başladığında DS veri kesimini göstermez; yükleyici DS'yi PSP'ye (0700h) kurar. `mov ax, veri` komutu `veri` kesiminin paragraf numarasını AX'e yükler. Bu sayı yükleme anında belli olur; yükleyici EXE dosyasındaki yeniden yerleştirme tablosuna (*relocation table*) bakarak komutun içine yazar. `mov ds, ax` ardından DS = 0710h olur. SS = 0711h ayrı bir bloktur; bu programda DS ≠ SS'dir.

`mov bp, offset x` BP'ye `x`'in doğru bağıl konumunu (0003h) koyar. Adres doğrudur; sorun kesimdedir. `mov al, [bp]` komutunda önek yoktur ve adreste BP vardır. İşlemci SS'yi seçer ve 0711h × 10h + 3 = 07113h adresini okur. Orası yığının dördüncü baytıdır ve içinde 00h vardır. AL = 00h olur; 'D' gelmemiştir. Assembler hiçbir hata vermez, çünkü komut geçerlidir.

`mov bl, ds:[bp]` komutunda `DS:` öneki varsayılanı değiştirir. Makine kodunun başına `3E` baytı eklenir ve 07103h okunur: BL = 44h ('D'). Aynı bağıl konum SI'ya alındığında önek gerekmez. `[si]` varsayılan olarak DS kullanır ve CL = 44h olur. Son iki komut DOS'un 4Ch hizmetini çağırır: AH = 4Ch "programı bitir", AL = 0 dönüş kodudur.



Şekil 4.1.10: `mov al, [bp]` adımı: BP = 0003h doğru bağıl konumdur, ama varsayılan kesim SS olduğu için yığının 07113h baytı okunur (AL = 00h). 'D' harfi veri kesiminde 07103h'dedir.

Bu izlemeden çıkan kural basittir. Veriye erişmek için BX, SI ya da DI; yığına erişmek için BP seçilir. BP ile veri kesimine erişmek gerçekten gerekiyorsa `DS:` öneki yazılır. `mov al, [bp]` komutunun makine kodundaki `46 00` ayrıntısı da dikkat çeker. Yer değiştirmesiz `[BP]` kodlanamadığı için assembler `[BP+0]` üretmiştir.

Sık yapılan hata

Bir değişkene BP ile erişip COM programında doğru sonuç aldıktan sonra kodu EXE programına taşımak. COM programında CS = DS = SS olduğundan `[BP]` ile `DS:[BP]` aynı bayta gider ve hata görünmez. Kesimleri ayrı bir EXE programında aynı kod yığından okur. Hata derleme sırasında değil, yanlış değerler ortaya çıktığında fark edilir.

4.1.4.7 Kesim yazmacını değiştirmek

Kesim yazmaçlarının değeri değiştirilebilir, ama her birinin kendine özgü kuralları vardır.

```
PUSH DS            ; DS'yi sakla
MOV  AX, 0B800h
MOV  DS, AX        ; geçici kesim
MOV  BYTE PTR [0], 'Z'
POP  DS            ; değişkenler geri
MOV  AX, yeni_yigin
MOV  SS, AX        ; ardından hemen
MOV  SP, 0400h     ; SP kurulur
```

DS. Assembly'deki değişken adları yalnız bağıl konumdur. Program DS'yi değiştirirse aynı ad başka kesimdeki bir bayta gider. Kodun ilk dört satırı ekranın sol üst köşesine bir Z yazmak için DS'yi geçici olarak B800h yapar. `BYTE PTR [0]` böylece B8000h'ye gider. İş bitince `POP DS` eski değeri geri geti-

rir. PUSH ve POP kullanılmasaydı sonraki bütün değişken erişimleri video belleğine giderdi. Daha güvenli bir yol, DS'ye hiç dokunmadan ES'yi kullanmaktır. ES dizgi hedefi dışında hiçbir erişimin varsayılan kesimi değildir.

SS ve SP. İkisi birlikte değişir. SS değiştirildikten sonra SP'nin yeni kesime uygun değeri alması gerekir. İki komutun arasında bir kesme gelseydi işlemci dönüş adresini yeni SS ile eski SP'nin oluşturduğu yanlış bir yere iterdi. 8086 bunu donanımla önler: bir kesim yazmacına MOV ya da POP yapan komuttan sonraki komut bitene kadar kesme kabul edilmez. Bu yüzden `MOV SS, AX` komutunun hemen arkasına `MOV SP, …` yazılır ve SS:SP çifti hiçbir an yarım kalmaz.

CS. MOV ile yazılamaz. CS değişirse bir sonraki komut tamamen başka bir yerden getirilir ve IP'nin eski değeri anlamsız kalır. CS bu yüzden IP ile birlikte değişir: uzak JMP, uzak CALL, uzak RET, INT ve IRET komutları ikisini aynı anda yükler. `MOV AX, CS` gibi okumalar ise serbesttir.

4.1.5 Veri tipi ve bayt sırası

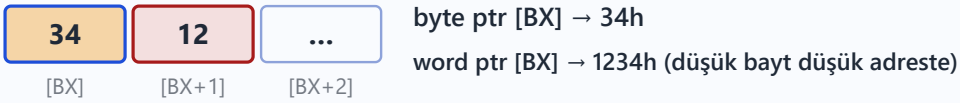
4.1.5.1 byte ptr ve word ptr

Bir bellek adresi tek bir baytı gösterir. 8086 ise aynı adresten başlayarak bir bayt (8 bit) ya da bir sözcük (16 bit, iki bayt) okuyup yazabilir. Komutun hangisini yapacağı işlem kodunun en düşük bitinde, **W bitinde** yazılıdır: W = 0 bayt, W = 1 sözcük demektir. `8A` ile `8B` işlem kodlarının farkı budur. Assembler bu biti doğru koymak için erişimin boyutunu bilmek zorundadır.

Çoğu komutta boyut diğer işlenenden anlaşılır. `MOV [BX], AL` komutunda AL 8 bittir ve bir bayt yazılır. `MOV [BX], AX` komutunda AX 16 bittir ve bir sözcük yazılır. Ama `MOV [BX], 5` komutunda iki işlenenin de kendi boyutu yoktur. Köşeli parantez yalnız bir adres söyler. 5 sayısı ise hem bir bayta (`05`) hem bir sözcüğe (`05 00`) sığar. Assembler bu durumda karar veremez. Bazı assembler'lar hata verir, bazıları bir boyut tahmin eder. Tahmin yanlışsa program sessizce komşu baytı bozar.

> **Tanım 4.1.6**
>
> **Veri tipi belirtici** (*data type specifier*), bir bellek işleneninin bayt mı sözcük mü olduğunu assembler'a bildiren ektir. `byte ptr [BX]` bir baytlık, `word ptr [BX]` iki baytlık erişim demektir. emu8086 bunların kısa biçimlerini de tanır: `b.[SI]` bayt, `w.[DI]` sözcük.

Belirtici yalnız assembler içindir; işlemci onu görmez. `MOV BYTE PTR [BX], 5` komutu `C6 07 05` (3 bayt), `MOV WORD PTR [BX], 5` komutu `C7 07 05 00` (4 bayt) olarak kodlanır. Farkın tamamı işlem kodunun W biti (`C6` → `C7`) ve sabitin uzunluğudur. Şekil 4.1.11 aynı adresten iki farklı boyutta okumayı gösterir.



Şekil 4.1.11: Aynı adreste `byte ptr` bir bayt, `word ptr` iki bayt okur.

Komut	Boyut nereden belli?
`MOV [BX], AL`	AL 8 bit → bayt yazılır
`MOV [BX], AX`	AX 16 bit → sözcük yazılır
`MOV [BX], 5`	belirsiz → `MOV BYTE PTR [BX], 5` ya da `MOV WORD PTR [BX], 5`
`MOV AL, sayi`	AL 8 bit; `sayi` DB ile tanımlıysa uyumlu, DW ile tanımlıysa tür uyuşmazlığı

Tablo 4.1.12: Erişim boyutunun nereden anlaşıldığı.

Son satır, değişken adının da bir tür taşıdığını gösterir. `sayi DB 7` diye tanımlanan bir değişken bayt, `DW` ile tanımlanan sözcük türündedir. Assembler değişken adı geçen bir komutta boyutu bu türden alır ve yazmaçla uyuşmazsa hata verir. Türü bilerek değiştirmek için yine belirtici yazılır: `MOV AL, BYTE PTR sozcuk` bir sözcük değişkeninin yalnız düşük baytını okur.

4.1.5.2 Kod izleme: byte ptr ve word ptr

Aşağıdaki program aynı adrese farklı boyutlarda yazar ve okur. 0200h'den başlayan dört bayt önceden AA BB CC DD değerlerini taşır. Böylece hangi baytların değiştiği açıkça görülür.

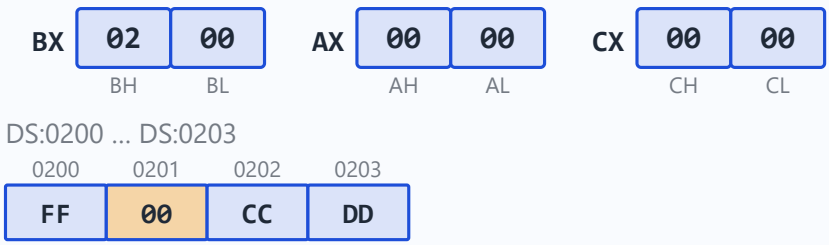
```
MOV BX, 0200h
MOV WORD PTR [BX], 1234h
MOV BYTE PTR [BX], 0FFh
MOV WORD PTR [BX], 0FFh
MOV AX, [BX]        ; sözcük
MOV CL, [BX+1]      ; bayt
```

Adım	Komut	Makine kodu	0200h 0201h 0202h 0203h	Yazmaç
1	`MOV BX, 0200h`	`BB 00 02`	AA BB CC DD	BX = 0200h
2	`MOV WORD PTR [BX], 1234h`	`C7 07 34 12`	**34 12** CC DD	—
3	`MOV BYTE PTR [BX], 0FFh`	`C6 07 FF`	**FF** 12 CC DD	—
4	`MOV WORD PTR [BX], 0FFh`	`C7 07 FF 00`	**FF 00** CC DD	—
5	`MOV AX, [BX]`	`8B 07`	FF 00 CC DD	AX = 00FFh
6	`MOV CL, [BX+1]`	`8A 4F 01`	FF 00 CC DD	CL = 00h

Tablo 4.1.13: byte ptr ve word ptr izlemesi (başlangıçta [0200h...0203h] = AA BB CC DD).

Adım 2'de iki bayt yazılır. Sabitin düşük baytı 34h düşük adrese (0200h), yüksek baytı 12h bir sonraki adrese (0201h) gider. Adım 3'te yalnız 0200h değişir ve FFh olur; 0201h'deki 12h yerinde kalır. Adım 4'te aynı FFh sabiti bu kez sözcük olarak yazılır. Sözcük olarak FFh, 00FFh demektir. 0200h'ye FFh, 0201h'ye 00h gider. Önceki adımda korunan 12h bu kez silinmiştir. Sabitin kendisi aynıdır; değişen yalnız belirticidir.

Adım 5'te belirtici gerekmez; AX 16 bit olduğundan bir sözcük okunur. 0200h'deki FFh AL'ye, 0201h'deki 00h AH'ye gider: AX = 00FFh. Adım 6'da CL 8 bittir ve `[BX+1]` = 0201h'deki tek bayt okunur: CL = 00h. 0202h ve 0203h'deki CC DD baytlarına hiçbir adımda dokunulmamıştır.



Şekil 4.1.12: Adım 4: `WORD PTR` ile yazılan 0FFh iki bayta yayılır (FF 00); bir önceki adımda korunan 12h silinir.

4.1.5.3 Little-endian düzen

Önceki izlemede sözcüklerin bellekte "ters" durduğu görüldü: 1234h, bellekte `34 12` olarak saklandı. Bu bir rastlantı değil, 8086'nın bayt sırası kuralıdır.

> **Tanım 4.1.7**
>
> Çok baytlı bir değerin düşük baytını düşük adrese, yüksek baytını bir sonraki adrese yazan düzene **little-endian** (düşük bayt önce) denir. Tersine, yüksek baytı önce yazan düzen **big-endian**'dır. 8086 ve bütün x86 ailesi little-endian'dır.



Şekil 4.1.13: Little-endian: 1234h sözcüğünün düşük baytı 34h düşük adrese, yüksek baytı 12h bir sonraki adrese yazılır.

Kuralın yazmaç düzeyindeki karşılığı çok düzgündür. `MOV [0200h], AX` komutu AL'yi 0200h'ye, AH'yi 0201h'ye yazar. `MOV AX, [0200h]` tersini yapar: 0200h'deki bayt AL'ye, 0201h'deki AH'ye gelir. Yazmacın düşük yarısı her zaman düşük adresle eşleşir. Little-endian'ın bir yararı da buradan doğar. Bir sözcük değişkeninin adresinden bir bayt okunursa düşük bayt gelir. 0034h gibi küçük bir sayı, sözcük olarak da bayt olarak da aynı adreste aynı değeri verir.

emu8086'nın bellek penceresi baytları adres sırasıyla gösterir. Bu yüzden sözcükler pencerede ters görünür: 1234h → `34 12`. Kural makine kodundaki sabitler için de geçerlidir. `MOV AX, 1234h` komutu `B8 34 12`, `MOV AX, 0B800h` komutu `B8 00 B8` olarak kodlanır. Adresler de aynı kurala uyar: `MOV AX, [1000h]` = `A1 00 10`.

> **Sık yapılan hata**
>
> Bellek dökümünü soldan sağa okuyup sözcüğü öyle kurmak. 0200h'de 34h, 0201h'de 12h varsa `MOV AX, [0200h]` sonrası AX = 1234h olur, 3412h değil. Düşük adres düşük bayttır. Aynı yanılgı makine kodunu okurken de yapılır: `A1 00 10` komutundaki adres 0010h değil, 1000h'dir.

4.1.5.4 Kod izleme: sözcük dizisinde indis

Bayt sırası ve bayt adresleme bir araya geldiğinde sık görülen bir hata ortaya çıkar. Bir sözcük dizisinin i. elemanına erişmek için indis yazmacına i değil, 2i konmalıdır. Aşağıdaki program önce yanlış, sonra doğru yolu izler.

```
org 100h
mov si, 1            ; indis 1
mov ax, dizi[si]     ; HATA!
shl si, 1            ; indis × 2
mov bx, dizi[si]     ; dizi[1]
mov si, 2            ; indis 2
shl si, 1            ; SI = 4
mov cx, dizi[si]     ; dizi[2]
ret
dizi dw 1234h, 5678h, 9ABCh
```

Komutların uzunlukları toplandığında `dizi` 0117h'de başlar. Üç sözcük bellekte little-endian olarak `34 12 78 56 BC 9A` diye 0117h–011Ch arasına yerleşir. dizi[0] = 1234h 0117h'de, dizi[1] = 5678h 0119h'de, dizi[2] = 9ABCh 011Bh'dedir.

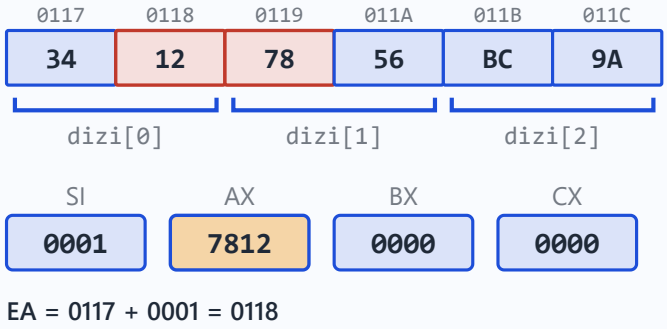
Adım	Komut	SI	EA	Okunan baytlar	Sonuç
1	`mov si, 1`	0001h	—	—	—
2	`mov ax, dizi[si]`	0001h	0118h	12 78	AX = 7812h (yanlış)
3	`shl si, 1`	0002h	—	—	—

Adım	Komut	SI	EA	Okunan baytlar	Sonuç
4	`mov bx, dizi[si]`	0002h	0119h	78 56	BX = 5678h = dizi[1]
5	`mov si, 2`	0002h	—	—	—
6	`shl si, 1`	0004h	—	—	—
7	`mov cx, dizi[si]`	0004h	011Bh	BC 9A	CX = 9ABCh = dizi[2]

Tablo 4.1.14: Sözcük dizisinde indis izlemesi (dizi = 0117h: 34 12 78 56 BC 9A).

Adım 1'de programcı “ikinci eleman” düşüncesiyle SI = 1 verir. Ama indis yazmacı eleman değil bayt sayar. Adım 2'de EA = 0117h + 1 = 0118h olur. Buradan okunan sözcük, dizi[0]'ın yüksek baytı (12h) ile dizi[1]'in düşük baytının (78h) birleşimidir. Little-endian kuralıyla 0118h'deki 12h AL'ye, 0119h'deki 78h AH'ye gider: AX = 7812h. Bu değer dizinin hiçbir elemanı değildir. İşlemci hiçbir uyarı vermez; tek adresten sözcük okumak 8086'da geçerlidir, yalnız bir yol döngüsü fazla sürer.

Adım 3'teki `shl si, 1` SI'yı bir bit sola kaydırır, yani 2 ile çarpar: SI = 2. Sözcük dizisinde bayt uzaklığı indis × 2'dir. Adım 4'te EA = 0119h ve BX = 5678h = dizi[1] doğru gelir. Adım 5 ve 6 aynı işi indis 2 için yapar: SI = 2, sonra 4. Adım 7'de EA = 011Bh ve CX = 9ABCh = dizi[2] olur. Kaydırma, 2 ile çarpmanın en kısa ve hızlı yoludur; `D1 E6` iki baytlık bir komuttur.



Şekil 4.1.14: SI = 1 ile `dizi[si]` iki elemanın ortasından okur: 0118h'deki 12h ve 0119h'deki 78h birleşir, AX = 7812h.

`shl` komutu MOV'dan farklı olarak bayrakları da günceller. 1'i bir bit kaydırmak 2 verir; dışarı çıkan bit 0 olduğundan CF = 0 olur, sonuç sıfır olmadığından ZF = 0'dır. Yardımcı elde bayrağı AF kaydırma komutlarından sonra tanımsızdır. Döngüyle bir sözcük dizisi gezilirken SI her turda `ADD SI, 2` ile ilerletilir. `INC SI` yalnız bayt dizileri içindir.

4.1.6 MOV komutu

4.1.6.1 Tanım ve kurallar

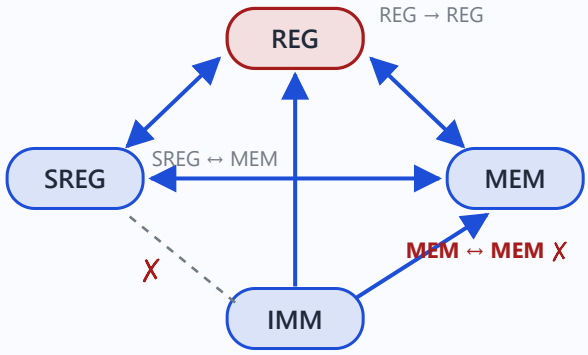
Önceki bütün örneklerde kullanılan MOV, 8086'nın en sık kullanılan komutudur. Derleyicilerin ürettiği kodun önemli bir kısmı yalnız MOV'lardan oluşur.

Tanım 4.1.8

`MOV hedef, kaynak` komutu kaynaktaki değeri hedefe kopyalar. Kaynak değişmez. Kaynak bir anlık değer (sabit), genel amaçlı yazmaç, kesim yazmacı ya da bellek konumu olabilir. Hedef ise genel amaçlı yazmaç, kesim yazmacı (CS hariç) ya da bellek konumudur.

MOV'un kuralları beş maddede toplanır:

1. CS ve IP'ye MOV ile değer atanmaz. CS kaynak olarak okunabilir; IP hiçbir MOV'da yer almaz.
2. İki işlenenin boyutu aynı olmalıdır: ikisi de bayt ya da ikisi de sözcük.
3. İki işlenen birden bellek olamaz.
4. Kesim yazmacına anlık değer yüklenemez ve bir kesim yazmacı diğerine doğrudan kopyalanamaz.
5. MOV bayrakları etkilemez.



Şekil 4.1.15: MOV için izin verilen kaynak → hedef yönleri. Bellek ile bellek, sabit ile kesim yazmacı ve iki kesim yazmacı arasında ok yoktur.

Kuralların çoğu makine kodunun yapısından gelir. Bir MOV komutunda en çok bir ModR/M baytı bulunur ve bu bayt yalnız bir bellek işleneni tanımlayabilir. İkinci işlenen REG alanındaki bir yazmaçtır ya da komutun sonundaki bir sabittir. İki bellek işleneni için yer yoktur. Kesim yazmaçları için ayrı işlem kodları vardır: `8E` kesim yazmacına, `8C` kesim yazmacından taşır. Bunların ikinci işleneni her zaman ModR/M ile seçilen bir genel yazmaç ya da bellek sözcüğüdür. Bu yüzden kesimden kesime ya da sabitten kesime komut yoktur.

Bayrakların değişmemesi de bilinçli bir seçimdir. Bir karşılaştırmanın ardından, koşullu atlamadan önce bir değeri yazmaca almak gerekebilir. MOV bayrakları bozsaydı karşılaştırmanın sonucu kaybolurdu.

Sunumdaki iki örnek kuralları uygular. `MOV AX, BX` BX'teki değeri AX'e kopyalar; BX değişmez. `MOV [SI], 10h` ise SI'nın gösterdiği DS:SI adresine 10h yazmak ister. Hedef bir bellek konumu, kaynak bir sabittir ve boyut belirsizdir. Doğru yazım `MOV BYTE PTR [SI], 10h`'dir.

4.1.6.2 Desteklenen işlenenler

emu8086 belgeleri MOV'un geçerli biçimlerini iki grupta listeler. Tablo 4.1.15 bu listeyi, yazımda kullanılan kısaltmaların anlamlarıyla verir.

Genel amaçlı	Kesim yazmaçlarıyla
`MOV REG, memory`	`MOV SREG, memory`
`MOV memory, REG`	`MOV memory, SREG`
`MOV REG, REG`	`MOV REG, SREG`
`MOV memory, immediate`	`MOV SREG, REG`
`MOV REG, immediate`	—

Tablo 4.1.15: MOV komutunun desteklenen işlenen birleşimleri.

- **REG**: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
- **SREG**: DS, ES, SS ve CS. CS yalnız ikinci işlenen, yani kaynak olabilir.
- **memory**: `[BX]`, `[BX+SI+7]`, bir değişken adı ve 24 biçimin herhangi biri.
- **immediate**: 5, −24, 3Fh, 10001101b gibi sabitler.

`MOV SREG, immediate` ve `MOV SREG, SREG` biçimleri yoktur. Değer önce bir genel yazmaca alınır: `MOV AX, 0B800h` ardından `MOV DS, AX`. Kesimden kesime kopyalama için iki yol vardır: `MOV AX, ES` / `MOV DS, AX` ya da `PUSH ES` / `POP DS`. İkinci yol bir genel yazmacı bozmaz ama yığını kullanır.

4.1.6.3 Geçerli mi?

Kurallar on komut üzerinde sınanabilir. Tablo 4.1.16 her komutun durumunu ve gerekçesini verir. Geçersiz komutların hepsinde assembler hata verir.

No	Komut	Durum	Gerekçe ve doğrusu
1	`MOV [BX], [SI]`	geçersiz	bellekten belleğe MOV yok → `MOV AL, [SI]` · `MOV [BX], AL`
2	`MOV DS, ES`	geçersiz	kesimden kesime yok → `MOV AX, ES` · `MOV DS, AX` ya da `PUSH ES` · `POP DS`

No	Komut	Durum	Gerekçe ve doğrusu
3	`MOV DS, 100h`	geçersiz	kesime sabit atanamaz
4	`MOV CS, AX`	geçersiz	CS hedef olamaz (uzak JMP kullanılır); IP de yazılamaz (`MOV IP, AX` yerine `JMP AX`)
5	`MOV [BX], 5`	geçersiz	boyut belirsiz → `BYTE PTR` / `WORD PTR`
6	`MOV AX, CS`	geçerli	CS kaynak olabilir; kodu `8C C8`
7	`MOV SP, BX`	geçerli	SP genel bir yazmaçtır; ama yığını başka yere taşır
8	`MOV BL, DX`	geçersiz	8 ve 16 bit uyuşmaz
9	`MOV AL, [SI+BP]`	geçerli	`[BP+SI]` ile aynı; kesim SS
10	`MOV 5, AX`	geçersiz	hedef sabit olamaz

Tablo 4.1.16: On MOV komutunun geçerliliği.

7. komut üzerinde biraz durmak gerekir. Komut kurallara uyar ve assembler onu kabul eder. Ama SP'nin değeri değiştiği anda yığının yeri de değişir. Bundan sonraki ilk PUSH, CALL ya da gelen bir kesme veriyi BX'in gösterdiği yere yazar. Bu satır bilerek yazılmadıysa program çoğu zaman `RET` komutunda yanlış bir adrese döner. 9. komut ise yazım sırasının önemsiz olduğunu hatırlatır: `[SI+BP]`, BP içerdiği için SS kesimine gider.

4.1.6.4 Kod izleme: yazmaç üzerinden blok kopyalama

Bellekten belleğe MOV olmadığına göre bir bellek bloğunu başka bir yere kopyalamak için her bayt bir yazmaçtan geçirilir. Aşağıdaki program dört baytlık `kaynak` dizgisini `hedef` tamponuna kopyalar.

```
org 100h
mov si, offset kaynak
mov di, offset hedef
mov cx, 4
kopya:
mov al, [si]         ; bellek → AL
mov [di], al         ; AL → bellek
inc si
inc di
loop kopya
ret
kaynak db 'ABCD'
hedef  db 4 dup('.')
```

Kod 0100h–0111h arasındaki 18 bayttır. `kaynak` 0112h'de, `hedef` 0116h'de başlar. `4 dup('.')` dört nokta karakteri (2Eh) ayırır. `loop kopya` komutu CX'i bir azaltır ve CX sıfır değilse `kopya` etiketine atlar. Makine kodu `E2 F8`'dir: LOOP 010Fh'de, sonraki komut 0111h'dedir ve 0111h + (−8) = 0109h, yani `kopya`'nın adresidir. Atlama göreceli moddadır.

Adım	Komut	SI	DI	CX	AL	hedef
1	`mov si, offset kaynak`	0112h	0000h	0000h	00h	`....`
2	`mov di, offset hedef`	0112h	0116h	0000h	00h	`....`
3	`mov cx, 4`	0112h	0116h	0004h	00h	`....`
4	`mov al, [si]`	0112h	0116h	0004h	41h	`....`
5	`mov [di], al`	0112h	0116h	0004h	41h	`A...`
6	`inc si`	0113h	0116h	0004h	41h	`A...`
7	`inc di`	0113h	0117h	0004h	41h	`A...`
8	`loop kopya`	0113h	0117h	0003h	41h	`A...`
9–13	tur 2	0114h	0118h	0002h	42h	`AB..`
14–18	tur 3	0115h	0119h	0001h	43h	`ABC.`
19–23	tur 4	0116h	011Ah	0000h	44h	`ABCD`
24	`ret`	0116h	011Ah	0000h	44h	`ABCD`

Tablo 4.1.17: Blok kopyalamanın izlemesi (kaynak = 0112h: 'ABCD', hedef = 0116h: '....').

İlk üç adım iki işaretçiyi ve sayacı kurar. SI kaynağın, DI hedefin bağıl konumunu tutar; iki işaretçi iki ayrı tamponu gösterir. CX = 4 kopyalanacak bayt sayısıdır. Adım 4'te `mov al, [si]` bellekten yazmaca taşır ve AL = 'A' = 41h olur. Adım 5'te `mov [di], al` yazmaçtan belleğe taşır ve 0116h'ye 41h yazılır. AL burada bir ara duraktır. Bellekten belleğe MOV olmadığı için başka yolu yoktur.

Adım 6 ve 7'de iki işaretçi birlikte bir bayt ilerler. Adım 8'de LOOP, CX'i 3'e indirir. CX sıfır olmadığından IP = 0109h olur ve döngü başa döner. İkinci, üçüncü ve dördüncü turlar aynı beş adımı tekrarlar; tabloda her tur tek satırda özetlenmiştir. Dördüncü turun sonunda CX = 0 olur ve LOOP atlamaz; program `ret`'e düşer. Hedef artık “ABCD” içerir. SI ve DI tamponların hemen arkasını gösterir: SI = 0116h, DI = 011Ah. SI'nın son değeri rastlantı eseri `hedef`'in başlangıcıdır, çünkü iki tampon bitişiktir.



Şekil 4.1.16: Blok kopyalamanın 5. adımı: AL'deki 'A' (41h), DI'nın gösterdiği 0116h'ye yazılır.

MOV bayrakları değiştirmez, ama INC değiştirir. Bayrak durumu bu programda hiçbir kararı etkilemez. LOOP, bayraklara bakmaz; yalnız CX'e bakar. Yine de izlemede eşlik bayrağının INC'lerle değiştiği görülür. Örneğin `inc di` ile DI = 0117h olduğunda düşük bayt 17h = 0001 0111b'dir. Dört adet 1 biti bulunduğundan PF = 1 olur.

Her bayt için iki bellek erişimi, iki işaretçi artırma ve bir döngü komutu gerekir. Bu iş çok sık yapıldığı için 8086'da ayrı bir komut vardır: `REP MOVSB`, DS:SI'dan ES:DI'ya CX bayt kopyalar ve işaretçileri kendisi ilerletir. Aynı program dizgi komutlarıyla `CLD`, `REP MOVSB` satırlarına iner.

4.1.7 Program örneği: ekrana 'A' yazmak

4.1.7.1 Program ve makine kodu

Buraya kadar anlatılanlar küçük bir programda bir araya gelir. Program, renkli metin ekranının video belleğine doğrudan bir karakter yazar. İşletim sisteminin hiçbir hizmeti kullanılmaz; ekrana yazmak, belirli bir bellek adresine bir sözcük koymaktan ibarettir.

```
ORG 100h          ; .com programı için
MOV AX, 0B800h    ; AX = B800h
MOV DS, AX        ; DS = B800h
MOV CL, 'A'       ; CL = 41h
MOV CH, 11011111b ; CH = DFh
MOV BX, 15Eh      ; BX = 015Eh
MOV [BX], CX      ; B800:015E ← CX
RET               ; DOS'a dön
```

Programın her satırı bir öğeyi tanıtır. `ORG` bir direktiftir; programın başlangıç adresini belirler ve makine koduna çevrilmez. Noktalı virgülden sonrası açıklamadır; assembler onu yok sayar. `MOV` kaynak ile hedef arasında değer kopyalar. DS'ye sabit doğrudan atanamadığı için B800h önce AX'e yüklenir.

`'A'` karakter sabitidir ve ASCII kodu 41h ile aynıdır. `11011111b` ikili yazılmış bir sabittir ve DFh'ye eşittir. `MOV [BX], CX` CX'teki sözcüğü DS:BX adresine yazar.

Adres	Komut	Makine kodu	Açıklama
0100h	`MOV AX, 0B800h`	`B8 00 B8`	B8 = "AX'e anlık sözcük"; sabit little-endian
0103h	`MOV DS, AX`	`8E D8`	8E = kesim yazmacına; D8 = 11 011 000 (DS ← AX)
0105h	`MOV CL, 'A'`	`B1 41`	B0 + 1 (CL); 41h = 'A'
0107h	`MOV CH, 11011111b`	`B5 DF`	B0 + 5 (CH)
0109h	`MOV BX, 15Eh`	`BB 5E 01`	B8 + 3 (BX)
010Ch	`MOV [BX], CX`	`89 0F`	89 = yazmaçtan belleğe sözcük; 0F = 00 001 111
010Eh	`RET`	`C3`	yakın dönüş

Tablo 4.1.18: Programın makine kodu (0100h'den başlayarak 15 bayt).

Tablodaki kodlarda düzenli bir yapı göze çarpar. Bir yazmaca anlık değer yükleyen MOV'un işlem kodu, bayt yazmaçları için B0'dan, sözcük yazmaçları için B8'den başlar ve yazmacın üç bitlik numarası eklenir. Numaralar AL/AX = 0, CL/CX = 1, DL/DX = 2, BL/BX = 3, AH/SP = 4, CH/BP = 5, DH/SI = 6, BH/DI = 7'dir. CL için B1, CH için B5, BX için BB böylece bulunur. Bu komutlarda ModR/M baytı yoktur; yazmaç işlem kodunun içine gömülüdür.

4.1.7.2 Kod izleme

Program emu8086'da bir COM programı olarak çalışır. Başlangıçta CS = DS = ES = SS = 0700h, IP = 0100h ve SP = FFFEh'dir. Ekranın bütün hücreleri boştur: her hücrede 20h (boşluk) ve 07h (siyah zemin üzerine gri yazı) baytları vardır.

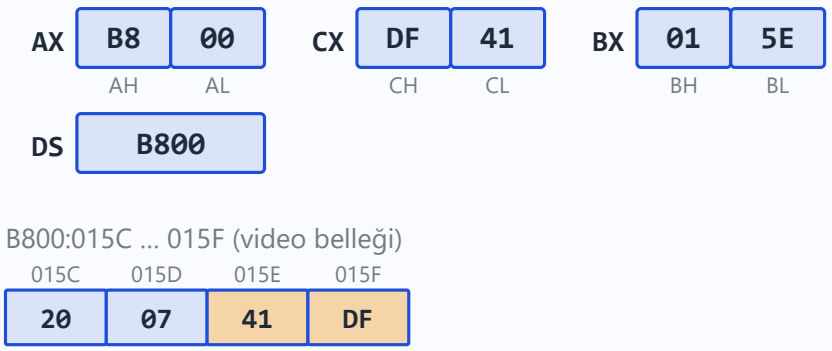
Adım	Komut	AX	DS	CX	BX	[B800:015E] [015F]
0	(başlangıç)	0000h	0700h	0000h	0000h	20h 07h
1	`MOV AX, 0B800h`	B800h	0700h	0000h	0000h	20h 07h
2	`MOV DS, AX`	B800h	B800h	0000h	0000h	20h 07h
3	`MOV CL, 'A'`	B800h	B800h	0041h	0000h	20h 07h
4	`MOV CH, 11011111b`	B800h	B800h	DF41h	0000h	20h 07h
5	`MOV BX, 15Eh`	B800h	B800h	DF41h	015Eh	20h 07h
6	`MOV [BX], CX`	B800h	B800h	DF41h	015Eh	**41h DFh**
7	`RET`	B800h	B800h	DF41h	015Eh	41h DFh

Tablo 4.1.19: Ekrana 'A' yazan programın izlemesi.

Adım 1'de AX = B800h olur. B800h, renkli metin ekranının kesimidir; fiziksel olarak B8000h'de başlar. Adım 2'de DS = B800h olur. Bundan sonra köşeli parantezle yapılan bütün öneksiz veri erişimleri video belleğine gider. Program bu noktadan sonra kendi değişkenlerine DS ile erişemez; bu küçük programın değişkeni olmadığı için sorun çıkmaz.

Adım 3 ve 4 CX'in iki yarısını ayrı ayrı doldurur. CL = 41h karakterin ASCII kodu, CH = DFh renk (öznitelik) baytıdır. İki yarı birlikte okununca CX = DF41h olur. Adım 5'te BX = 015Eh = 350 olur. Bu sayı ekrandaki konumun video belleğindeki bağıl adresidir.

Adım 6 programın asıl işidir. `MOV [BX], CX` bir sözcük yazar. Little-endian kuralıyla CL (41h) düşük adrese, B800:015Eh'ye; CH (DFh) yüksek adrese, B800:015Fh'ye gider. Fiziksel adresler B8000h + 015Eh = B815Eh ve B815Fh'dir. Karakter baytının önce, renk baytının sonra gelmesi video belleğinin düzenine tam uyar. Bu yüzden karakter düşük yarıya, renk yüksek yarıya konmuştur. Ekran kartı bu iki baytı bir sonraki tarama sırasında okur ve hücrede renkli bir 'A' görünür.



Şekil 4.1.17: Adım 6: CX = DF41h sözcük olarak yazılır; [015Eh] = 41h ('A'), [015Fh] = DFh (renk). Fiziksel adres B815Eh.

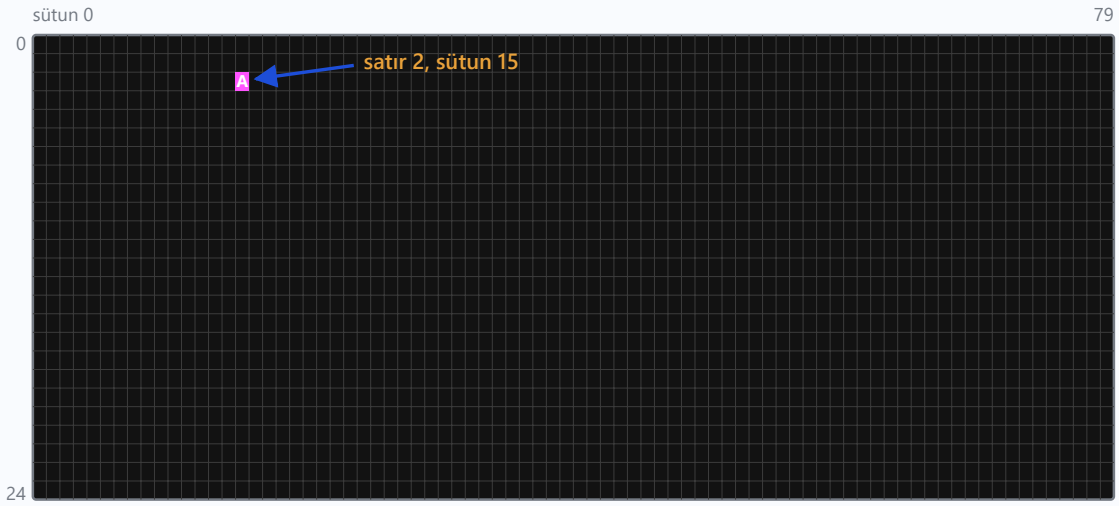
Adım 7'de RET yığının tepesindeki 0000h değerini IP'ye alır ve SP = 0000h olur. 0700:0000'daki `INT 20h` programı bitirir. Harf ekranda kalır, çünkü video belleği programdan bağımsızdır. DS'nin B800h kalması da bir sorun yaratmaz; DOS program bittikten sonra kendi yazmaçlarını yeniden kurar.

4.1.7.3 Metin modu video belleği

80 × 25 metin modunda ekran 25 satır ve 80 sütundan oluşur. Her hücre video belleğinde 2 bayt tutar: önce karakter, sonra renk. Bir satır 80 × 2 = 160 bayttır ve bütün ekran 25 × 160 = 4000 bayt kaplar. Satır ve sütun 0'dan sayılırsa bir hücrenin bağıl adresi şöyle bulunur:

> bağıl adres = (satır × 80 + sütun) × 2

Programdaki 015Eh bu formülden gelir: (2 × 80 + 15) × 2 = 175 × 2 = 350 = 15Eh. Harf 2. satırın 15. sütununa yazılmıştır (Şekil 4.1.18). Ters yönde de hesap yapılabilir. 015Eh = 350 ikiye bölünür: 175. 175 = 2 × 80 + 15 olduğundan satır 2, sütun 15'tir.



Şekil 4.1.18: 80 × 25 metin ekranı. Satır 2, sütun 15'teki hücreye macenta zemin üzerine parlak beyaz bir 'A' yazılmıştır.

Öznitelik baytının bitleri rengi kodlar. DFh = 1101 1111b şöyle okunur:

- Bit 7 = 1: gerçek ekranda karakterin yanıp sönmesi; emu8086'da parlak zemin.
- Bit 6–4 = 101: zemin rengi macenta (5).
- Bit 3–0 = 1111: yazı rengi parlak beyaz (15).

Yazı rengi dört bitle 16 renkten birini, zemin rengi üç bitle 8 renkten birini seçer. Temel renkler 0 siyah, 1 mavi, 2 yeşil, 3 camgöbeği, 4 kırmızı, 5 macenta, 6 kahverengi, 7 açık gridir; 8–15 bunların parlak hâlleridir ve 14 sarıdır. Önceki örnekteki 1Eh böylece mavi zemin (001) üzerine sarı yazı (1110), 07h ise siyah zemin üzerine gri yazı demektir.

Örnek 4.1.3

Ekranın 10. satırının 40. sütununa yeşil zemin üzerine sarı bir 'X' yazan kod nasıl olur? Renk baytı 2Eh'dir.

Bağıl adres (10 × 80 + 40) × 2 = 840 × 2 = 1680 = 690h'dir. 'X' karakterinin ASCII kodu 58h'dir. Karakter düşük, renk yüksek baytta olacağından yazılacak sözcük 2E58h'dir. 2Eh = 0010 1110b: zemin 010 (yeşil), yazı 1110 (sarı).

```
MOV AX, 0B800h
MOV DS, AX
MOV BX, 690h
MOV WORD PTR [BX], 2E58h
```

Son komutta `WORD PTR` gereklidir; aksi hâlde sabitin boyutu belirsiz kalır. Kod B8690h'ye 58h, B8691h'ye 2Eh yazar.

4.1.8 MOV örnekleri

emu8086 belgelerindeki kısa MOV örneklerinin her biri tek bir aktarım yönünü gösterir: bellekten yazmaca, yazmaçtan belleğe, yazmaçtan yazmaca, sabitten belleğe. Aşağıdaki kesimler bunları sırayla izler. Hepsi COM programıdır; başlangıçta CS = DS = 0700h, IP = 0100h'dir.

4.1.8.1 Bellekten yazmaca

```
ORG 100h
MOV AX, [1000h]   ; A1 00 10
RET
```

DS:1000h adresinde önceden `CD AB` baytları bulunsun. Komut doğrudan adreslemeyle bu iki baytı okur. Little-endian kuralıyla 1000h'deki CDh AL'ye, 1001h'deki ABh AH'ye gider: AX = ABCDh. Komut üç bayttır; yürütüldükten sonra IP = 0103h olur. Fiziksel adres 07000h + 1000h = 08000h'dir. Bellek değişmez; MOV yalnız kopyalar.

Adım	Komut	AX	IP (sonra)	[1000h] [1001h]
0	(başlangıç)	0000h	0100h	CD AB
1	`MOV AX, [1000h]`	ABCDh	0103h	CD AB
2	`RET`	ABCDh	0000h	CD AB

Tablo 4.1.20: Bellekten yazmaca izlemesi (DS:1000h = CD AB).

Bu programın emu8086 penceresindeki görünümü üç bölümden oluşur. Solda yazmaçlar vardır; AX, BX, CX ve DX'in yüksek ve düşük yarıları (H, L) ayrı kutularda gösterilir. Başlangıçta CS = DS = 0700h, IP = 0100h, SP = FFFEh'dir. Ortada bellek dökümü bulunur. 07100h fiziksel adresi, yani 0700:0100, komutun üç baytını (`A1 00 10`) taşır. Sağda ayrıştırılmış (*disassembled*) kod yer alır: `MOV AX, [01000h]` ve ardından `RET` (`C3`). *single step* düğmesi bir komutu yürütür ve değişen yazmaçlar renkle vurgulanır. Gerçek emu8086'da DS:1000h bölgesi başlangıçta sıfırdır; bu yüzden pencerede komut çalıştıktan sonra AX = 0000h kalır. ABCDh değeri, belleğe önceden CD AB yazıldığı varsayımıyla elde edilir.

4.1.8.2 Yazmaçtan belleğe

```
ORG 100h
MOV BX, 1234h     ; BB 34 12
MOV [2000h], BX   ; 89 1E 00 20
RET
```

Adım	Komut	BX	IP (sonra)	[2000h] [2001h]
0	(başlangıç)	0000h	0100h	00 00
1	`MOV BX, 1234h`	1234h	0103h	00 00
2	`MOV [2000h], BX`	1234h	0107h	34 12
3	`RET`	1234h	0000h	34 12

Tablo 4.1.21: Yazmaçtan belleğe izlemesi (DS:2000h başlangıçta 00 00).

Birinci komut BX'e anlık değer yükler. İkinci komut BX'i doğrudan adrese yazar: BL (34h) 2000h'ye, BH (12h) 2001h'ye. Bu komut dört bayttır. AX dışındaki bir yazmaç kullanıldığı için kısa `A3` biçimi seçilemez. Genel `89` işlem kodu, ModR/M baytı ve iki adres baytı gerekir. ModR/M baytı `1E` = 00 011 110 olarak okunur: REG = BX, R/M = doğrudan adres. IP 0103h'den 0107h'ye ilerler. BX değişmez. Fiziksel adres 07000h + 2000h = 09000h'dir.

Son adımdaki RET, yığının tepesinden COM programının dönüş adresini (0000h) çeker. Bu yüzden RET'ten sonraki IP 0000h'dir; program bir sonraki komutu 0700:0000'dan, yani PSP'nin başındaki `INT 20h`'den alır. Bütün COM örneklerinde son adım böyledir.

4.1.8.3 XCHG: takas

İki değerin yerini değiştirmek sıralama gibi pek çok algoritmanın temel işlemidir. 8086'da bunun için ayrı bir komut vardır.

Tanım 4.1.9

`XCHG hedef, kaynak` komutu iki işlenenin içeriğini tek bir komutta karşılıklı değiştirir (*exchange*). İşlenenlerden biri yazmaç, diğeri yazmaç ya da bellek olabilir. Bellek ile bellek, kesim yazmaçları ve sabitler kullanılamaz. İki işlenenin boyutu aynı olmalıdır. XCHG bayrakları etkilemez.

```
MOV CX, 10   ; CX = 000Ah
MOV DX, 20   ; DX = 0014h
XCHG CX, DX  ; takas
```

Sayılar sonek taşımadığında onluk kabul edilir. Birinci komuttan sonra CX = 10 = 000Ah, ikinciden sonra DX = 20 = 0014h olur. XCHG ardından CX = 0014h ve DX = 000Ah'dir. Bayraklar değişmez. Komutun makine kodu `87 CA`'dır: `87` sözcük takası, `CA` = 11 001 010 ise REG = CX, R/M = DX'tir.

Adım	Komut	CX	DX
1	`MOV CX, 10`	000Ah	0000h
2	`MOV DX, 20`	000Ah	0014h
3	`XCHG CX, DX`	0014h	000Ah

Tablo 4.1.22: İki yazmacın takası.

4.1.8.4 XCHG kuralları

- Yazmaç ↔ yazmaç: `XCHG AX, BX`.
- Yazmaç ↔ bellek: `XCHG AL, [SI]`. Yazım sırası önemsizdir; `XCHG [SI], AL` aynı komuttur.
- Bellek ↔ bellek yoktur.
- Kesim yazmaçları ve sabitler kullanılamaz. Sabitle takas zaten anlamsızdır: bir sabitin içine değer yazılamaz.
- Boyutlar aynı olmalıdır; bayraklar değişmez.
- `XCHG AX, reg` biçimi tek baytlık özel bir koda sahiptir: `90h` + yazmaç numarası. `XCHG AX, BX` = `93`. `XCHG AX, AX` hiçbir şeyi değiştirmez ve kodu `90`'dır. Bu bayt aynı zamanda `NOP` (işlem yok) komutudur.

XCHG olmasaydı takas için geçici bir yazmaç gerekirdi:

```
MOV AX, CX    ; AX ← CX (yedek)
MOV CX, DX    ; CX ← DX
MOV DX, AX    ; DX ← eski CX
```

Bu yol üç komut ve bir yazmaç daha harcar. AX'in eski değeri de kaybolur. Süre açısından da XCHG öndedir. 8086'da yazmaçtan yazmaca MOV 2 tur, XCHG ise 4 turdur; üç MOV 6 tur tutar. AX ile yapılan tek baytlık takas 3 turdur. Yazmaç ile bellek arasındaki takas ise pahalıdır: 17 tur artı EA süresi. Bunun nedeni, işlemcinin önce bellekten okuyup sonra belleğe yazmasıdır; iki ayrı yol döngüsü gerekir.

XCHG'nin bellekle kullanımının çok işlemcili sistemlerde ayrı bir önemi vardır. Okuma ve yazma tek komutta yapıldığı için arada başka bir işlemcinin aynı bayta erişme fırsatı daralır. `LOCK` önekiyle birlikte kullanıldığında 8086, LOC$\bar{K}$ pinini komut boyunca etkin tutar ve yolu başka bir yol ustasına bırakmaz. Bir kaynağı aynı anda yalnız bir işlemcinin kullanmasını sağlayan kilitler bu düzene dayanır.

4.1.8.5 Kod izleme: XCHG ile takas

Aşağıdaki program XCHG'nin bellekle kullanımını ve aynı takasın XCHG ile ve XCHG olmadan yapılışını yan yana gösterir.

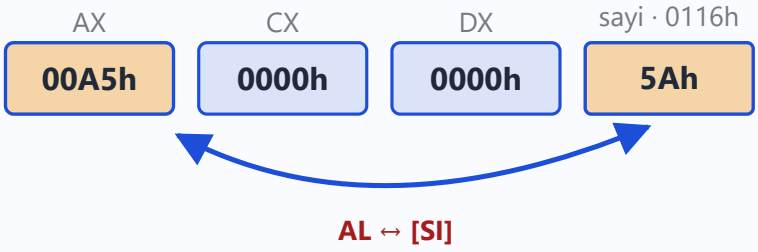
```
ORG 100h
MOV SI, OFFSET sayi
MOV AL, 5Ah
XCHG AL, [SI]      ; AL ⇔ [SI]
MOV CX, 1111h
MOV DX, 2222h
MOV AX, CX         ; yedek
MOV CX, DX
MOV DX, AX         ; takas
XCHG CX, DX        ; tek komut
RET
sayi DB 0A5h
```

Adım	Komut	AX	CX	DX	SI	sayi
1	`MOV SI, OFFSET sayi`	0000h	0000h	0000h	0116h	A5h
2	`MOV AL, 5Ah`	005Ah	0000h	0000h	0116h	A5h
3	`XCHG AL, [SI]`	00A5h	0000h	0000h	0116h	5Ah
4	`MOV CX, 1111h`	00A5h	1111h	0000h	0116h	5Ah
5	`MOV DX, 2222h`	00A5h	1111h	2222h	0116h	5Ah
6	`MOV AX, CX`	1111h	1111h	2222h	0116h	5Ah
7	`MOV CX, DX`	1111h	2222h	2222h	0116h	5Ah
8	`MOV DX, AX`	1111h	2222h	1111h	0116h	5Ah
9	`XCHG CX, DX`	1111h	1111h	2222h	0116h	5Ah

Tablo 4.1.23: XCHG ile takasın izlemesi (sayi = 0116h, başlangıçta A5h).

Adım 1'de SI, `sayi` değişkeninin bağıl konumunu (0116h) alır. Bellekte `sayi` = A5h'dir. Adım 2'de AL = 5Ah olur. Adım 3'teki `XCHG AL, [SI]` (`86 04`) bir yazmaçla bir bellek baytını takas eder: AL = A5h, `sayi` = 5Ah. Ara yazmaç gerekmemiştir. Bu tek komut, belleği okuyan ve belleğe yazan iki yol döngüsünü birlikte içerir.

Adım 4 ve 5 CX ve DX'e tanınabilir değerler koyar. Adım 6–8 aynı takası XCHG olmadan yapar. Adım 6'da CX'in yedeği AX'e alınır; AX'in önceki değeri (00A5h) kaybolur. Adım 7'de CX = 2222h olur. Bu anda CX'in eski değeri yalnız AX'te durur. Adım 8'de DX = 1111h olur ve takas tamamlanır. Üç komut ve bir yazmaç harcanmıştır. Adım 9'daki tek `XCHG CX, DX` takası geri alır: CX = 1111h, DX = 2222h. Bu sırada AX'e dokunulmaz.



Şekil 4.1.19: Adım 3: `XCHG AL, [SI]` yazmaç ile bellek baytını tek komutta takas eder; AL = A5h, sayi = 5Ah.

4.1.8.6 Yazmaçtan yazmaca

```
MOV SI, 3000h  ; SI = adres
MOV DI, SI     ; DI = SI
MOV AL, [SI]   ; DS:3000h
MOV [DI+1], AL ; DS:3001h
```

Bu örnek, adres tutan yazmaçlar arasında kopyalamanın ne anlama geldiğini gösterir. DS:3000h'de 5Ah bulunsun. Birinci komut SI'ya bir bağıl konum (3000h) atar. İkinci komut `MOV DI, SI` ile DI = 3000h olur. Kopyalanan şey adrestir, bellekteki veri değil. Artık iki yazmaç aynı hücreyi gösterir. Üçüncü komut `[SI]` ile 5Ah'yi okur: AL = 5Ah. Dördüncü komut bunu `[DI+1]` = 3001h'ye yazar. Bellekte 3000h–3001h artık `5A 5A`'dır.

Adım	Komut	Makine kodu	SI	DI	AL	[3000h] [3001h]
1	`MOV SI, 3000h`	`BE 00 30`	3000h	0000h	00h	5A ??
2	`MOV DI, SI`	`8B FE`	3000h	3000h	00h	5A ??
3	`MOV AL, [SI]`	`8A 04`	3000h	3000h	5Ah	5A ??
4	`MOV [DI+1], AL`	`88 45 01`	3000h	3000h	5Ah	5A 5A

Tablo 4.1.24: Yazmaçtan yazmaca izlemesi (DS:3000h = 5Ah).

C dilinde bu, iki işaretçiyi eşitlemeye benzer: `q = p;` ardından `q[1] = p[0];`. İşaretçilerin kendisi kopyalanır, gösterdikleri veri değil.

4.1.8.7 Belleğe sabit yazma

```
ORG 100h
MOV BYTE PTR [4000h], 65  ; 'A'
MOV WORD PTR [4002h], 65
RET
```

DS:4000h–4003h başlangıçta `00 99 00 99` olsun. Birinci komut 65'i tek bayt olarak yazar. 65 = 41h, 'A' harfinin ASCII kodudur. Makine kodu `C6 06 00 40 41` 'dir ve beş bayt tutar: işlem kodu, ModR/M, iki adres baytı ve bir veri baytı. IP 0105h olur. 4001h değişmez.

İkinci komut aynı sabiti `WORD PTR` ile yazar. 65 artık 0041h sözcüğüdür; 4002h'ye 41h, 4003h'ye 00h gider. 4003h'deki 99h silinmiştir. Komut `C7 06 02 40 41 00` diye altı bayttır ve IP 010Bh'ye ilerler. Sabitin kendi boyutu yoktur; boyutu PTR belirticisi belirler. Kaynak koddaki 65 sayısı iki komutta da aynıdır ama makine koduna bir kez bir, bir kez iki bayt olarak girer.

Adım	Komut	IP (sonra)	4000h 4001h 4002h 4003h
0	(başlangıç)	0100h	00 99 00 99
1	`MOV BYTE PTR [4000h], 65`	0105h	**41** 99 00 99
2	`MOV WORD PTR [4002h], 65`	010Bh	41 99 **41 00**
3	`RET`	0000h	41 99 41 00

Tablo 4.1.25: Belleğe sabit yazmanın izlemesi (DS:4000h = 00 99 00 99).

4.1.8.8 8 bitlik yarılarla çalışmak

AX, BX, CX ve DX'in her birinin iki 8 bitlik yarısı ayrı yazmaç adlarıyla kullanılabilir: AH ve AL, BH ve BL gibi. Bir yarıya yazmak diğer yarıyı etkilemez. Bu, iki küçük değeri tek bir yazmaçta tutmaya ve bir sözcüğün baytlarını ayrı ayrı işlemeye olanak verir.

Düşük baytı kopyalama. Aşağıdaki kodda AL, BL'ye kopyalanır.

```
MOV AX, 1234h   ; AH = 12h, AL = 34h
MOV BX, 0FFFFh
MOV BL, AL      ; yalnız BL
```

Birinci komut AX = 1234h yapar: AH = 12h, AL = 34h. İkinci komut, etkinin görülebilmesi için BX'i FFFFh ile doldurur. Üçüncü komut (`8A D8`) yalnız BL'yi değiştirir: BL = 34h. BH = FFh yerinde kalır ve BX = FF34h olur. 16 bitlik bir yazmaçtan 8 bitlik bir yazmaca kopyalama (`MOV BL, AX`) geçersizdir; boyutların aynı olması gerekir.

Yüksek bayta ekleme. Yarılar üzerinde aritmetik yapıldığında eldenin bir yarıdan diğerine geçmediği görülür.

```
MOV AX, 00FFh   ; AH = 00h, AL = FFh
ADD AH, 01h     ; yalnız AH
ADD AL, 01h     ; yalnız AL
MOV AX, 01FFh
ADD AX, 1       ; 16 bit
```

Adım	Komut	Makine kodu	AX	CF	ZF	SF	AF	PF
1	`MOV AX, 00FFh`	`B8 FF 00`	00FFh	0	0	0	0	0
2	`ADD AH, 01h`	`80 C4 01`	01FFh	0	0	0	0	0
3	`ADD AL, 01h`	`04 01`	0100h	1	1	0	1	1
4	`MOV AX, 01FFh`	`B8 FF 01`	01FFh	1	1	0	1	1
5	`ADD AX, 1`	`05 01 00`	0200h	0	0	0	1	1

Tablo 4.1.26: Yüksek ve düşük bayta ekleme (bayraklar: CF, ZF, SF, AF, PF).

Adım 2'de AH = 00h + 01h = 01h olur ve AX = 01FFh'dir. Sonuç sıfır değildir ve elde yoktur. 01h'de tek sayıda 1 biti olduğundan PF = 0'dır. Adım 3'te AL = FFh + 01h hesaplanır. 8 bitlik sonuç 00h'dir ve dokuzuncu bit olarak bir elde çıkar: CF = 1, ZF = 1. Düşük dört bitten de elde çıktığı için AF = 1 olur. 00h'de sıfır tane 1 vardır; sıfır çift sayıdır ve PF = 1 olur. Elde AH'ye geçmez: AX = 0100h. Bu bir 8 bitlik toplamadır ve AH işlemin parçası değildir.

Adım 4'te AX yeniden 01FFh yapılır. MOV bayrakları değiştirmediği için önceki adımın bayrakları (CF = 1, ZF = 1) aynen kalır. Adım 5'teki 16 bitlik toplamada 01FFh + 1 = 0200h olur. Bu kez elde AL'den AH'ye geçmiştir, çünkü işlem 16 bittir. 16 bitten taşan elde olmadığından CF = 0'dır. AF yine 1'dir; düşük dört bitte F + 1 elde üretir. PF yalnız sonucun düşük baytına bakar: 00h için PF = 1. `ADD AX, 1` komutu AX'e özgü kısa biçimle `05 01 00` diye kodlanır.

> **Sık yapılan hata**
>
> 8 bitlik bir toplamanın eldesinin yazmacın diğer yarısına geçeceğini sanmak. `ADD AL, 1` AL = FFh iken AH'yi artırmaz; yalnız CF = 1 olur. 16 bitlik bir değere 8 bitlik parçalar hâlinde ekleme yapılacaksa yüksek yarıda `ADC` (eldeyle topla) kullanılır: `ADD AL, 1` ardından `ADC AH, 0`.

4.1.8.9 Sayı tabanlarıyla atama

Assembly kaynak kodunda sabitler farklı tabanlarda yazılabilir. Hangi taban kullanılırsa kullanılsın, assembler sayıyı ikili değere çevirip makine koduna yazar. Aşağıdaki program üç tabanı birlikte kullanır.

```
mov al, 00000101b ; ikili
mov bl, 0ah       ; onaltılık
mov cl, 10o       ; sekizli
add al, bl        ; 5 + 10 = 15
sub al, cl        ; 15 - 8 = 7
```

Adım	Komut	Makine kodu	AL	BL	CL	CF	PF
1	`mov al, 00000101b`	`B0 05`	05h	00h	00h	0	0
2	`mov bl, 0ah`	`B3 0A`	05h	0Ah	00h	0	0
3	`mov cl, 10o`	`B1 08`	05h	0Ah	08h	0	0
4	`add al, bl`	`02 C3`	0Fh	0Ah	08h	0	1
5	`sub al, cl`	`2A C1`	07h	0Ah	08h	0	0

Tablo 4.1.27: Sayı tabanlarıyla atamanın izlemesi.

00000101b ikili sayısı 5'tir ve AL = 05h olur. 0Ah onaltılık sayısı 10'dur; harfle başladığı için önüne 0 yazılmıştır. 10o sekizli sayısı 1 × 8 + 0 = 8'dir ve CL = 08h olur. Makine kodlarında sabitler artık tabandan bağımsızdır: `05`, `0A`, `08`. Adım 4'te AL = 5 + 10 = 15 = 0Fh olur. 0000 1111b'de dört adet 1 biti vardır; dört çift sayı olduğundan PF = 1'dir. Adım 5'te AL = 15 − 8 = 7 = 07h olur. Ödünç gerekmediği için CF = 0 kalır. 0000 0111b'de üç adet 1 vardır ve PF = 0 olur.

4.1.8.10 Sayı yazımı

Tablo 4.1.28 emu8086 ve MASM uyumlu assembler'larda sabitlerin yazım kurallarını özetler. Bütün örneklerin onluk değeri 25'tir; yalnız 0FFh ve karakter satırı farklıdır.

Taban	Sonek	Örnek	Onluk değer
Onluk (*decimal*)	yok ya da `d`	`25`, `25d`	25
İkili (*binary*)	`b`	`00011001b`	25
Sekizli (*octal*)	`o` ya da `q`	`31o`	25
Onaltılık (*hexadecimal*)	`h`	`19h`, `0FFh`	25, 255
Karakter	`' '`	`'A'`	65 (41h)

Tablo 4.1.28: Sayı tabanları ve sonekleri.

Onaltılık bir sayı A–F harflerinden biriyle başlıyorsa önüne 0 eklenir: `0Ah`, `0B800h`. Bu kural assembler'ın sayı ile adı ayırt edebilmesi içindir. `Ah` yazılırsa AH yazmacıyla, `B800h` yazılırsa bir etiket ya da değişken adıyla karışır. Assembler her adın bir harfle, her sayının bir rakamla başladığını varsayar. emu8086 ayrıca C dilindeki `0x1000` biçimini de kabul eder. Sekizli sayılar için `o` harfi sıfıra benzediğinden bazı programcılar `q`'yu tercih eder.

Örnek 4.1.4

200 sayısı ikili, sekizli ve onaltılık olarak nasıl yazılır? `MOV AL, 200` ile `MOV AL, 0C8h` aynı makine kodunu üretir mi?

200 = 128 + 64 + 8 olduğundan ikili 11001000b'dir. Üçer bitlik gruplarla 11 001 000 = 3 1 0, yani 310o; dörder bitlik gruplarla 1100 1000 = C 8, yani 0C8h. C harfle başladığı için önüne 0 konmuştur. İki komut da `B0 C8` üretir; taban yalnız kaynak kodun okunuşunu değiştirir.

4.1.9 Sık yapılan hatalar

Bellek erişimindeki hataların çoğu derleme hatası vermez. Program çalışır, ama yanlış adresten okur ya da yanlış baytı bozar. Bu hatalar emu8086'da bellek penceresinde etkin adres ve fiziksel adres denetlenerek bulunur. Tablo 4.1.29 en sık rastlananları toplar.

Hatalı	Ne olur?	Doğrusu
`MOV SI, dizi`	SI'ya dizinin adresi değil ilk elemanın değeri yüklenir.	`MOV SI, OFFSET dizi` ya da `LEA SI, dizi`
`INC SI` (sözcük dizisi)	Sözcüğün ortasından okunur; iki elemanın yarıları birleşir.	`ADD SI, 2`
`MOV AL, [BP]` ile değişken	SS kesiminden okunur; EXE programında yanlış bayt gelir.	`[BX]`, `[SI]` ya da `DS:[BP]`
`MOV [BX], 5`	Boyut belirsiz; assembler hata verir ya da tahmin eder.	`MOV BYTE PTR [BX], 5`
`MOV AX, B800h`	B800h bir etiket adı sanılır.	`MOV AX, 0B800h`
DS'yi değiştirip bırakmak	Sonraki bütün değişken erişimleri başka kesime gider.	`PUSH DS` … `POP DS` ya da ES
`MOV AX, [0200h]` → 3412h beklemek	Little-endian unutulmuştur: bellekte 34 12 varsa AX = 1234h.	düşük adres = düşük bayt

Tablo 4.1.29: Bellek erişiminde sık yapılan hatalar.

İlk satır assembler'ın söz dizimine bağlıdır. MASM ve emu8086'da veri tanımlı bir ad, bir değişken olarak işlenir. `MOV SI, dizi` yazımı `MOV SI, [dizi]` ile aynı anlama gelir ve dizinin ilk sözcüğünü okur. SI bundan sonra ilgisiz bir yeri gösterir. Adres için `OFFSET` işleci ya da `LEA` (etkin adresi yükle) komutu kullanılır. NASM gibi bazı assembler'larda ise yalın ad her zaman adres demektir ve içerik için köşeli parantez zorunludur. Bir kaynak koddan diğerine geçerken bu fark gözden kaçırılmamalıdır.

Sık yapılan hata

Fiziksel adresi hesaplarken kesimi kaydırmayı unutmak. DS = 2000h, BX = 0150h için fiziksel adres 20000h + 0150h = 20150h'dir; 2000h + 0150h = 2150h değildir. Ters hatada bağıl konum kaydırılır: 2000h + 1500h = 3500h. Kesim her zaman sağına bir 0 eklenerek yazılır,

bağıl konum olduğu gibi kalır.

Sık yapılan hata

Etkin adres taşmasını 20 bite taşımak. BP = FFFEh iken `[BP+4]` için EA 1 0002h değil, 0002h'dir. Etkin adres her zaman 16 bitte sarar; taşan bit kesim değerine eklenmez. Kesim:bağıl konum toplamı ise 20 bitte sarar.

4.1.10 Alıştırmalar ve çözümleri

Sunumdaki altı alıştırma burada ara adımlarıyla çözülmüştür.

1. DS = 3000h, BX = 0010h, SI = 0005h için `[BX+SI+20h]` işleneninin etkin ve fiziksel adresi nedir?

Adreste BP yoktur, kesim DS'dir. EA = 0010h + 0005h + 0020h = 0035h. Fiziksel adres 3000h × 10h + 0035h = 30000h + 0035h = 30035h'dir.

2. SS = 4000h, BP = FFFEh için `[BP+4]` işleneninin etkin ve fiziksel adresi nedir?

EA = FFFEh + 0004h = 1 0002h; 16 bitte sarar ve EA = 0002h olur. BP olduğundan kesim SS'dir: 40000h + 0002h = 40002h.

3. `[BX+BP]`, `[DI+SI]`, `[BX+DI+300]`, `[DX]` yazımlarından hangileri geçerlidir?

Yalnız `[BX+DI+300]` geçerlidir: bir taban, bir indis ve bir d16. `[BX+BP]` iki taban, `[DI+SI]` iki indis yazmacı içerir. DX adres hesabında hiç kullanılamaz.

4. [0500h] = 78h, [0501h] = 56h ise `MOV AX, [0500h]` sonrası AX nedir? `MOV AL, [0501h]` sonrası AL nedir?

Sözcük okumada düşük adres düşük bayttır: AL = 78h, AH = 56h, AX = 5678h. İkinci komut yalnız 0501h'deki baytı okur: AL = 56h.

5. Ekranın 10. satır, 40. sütun konumuna yeşil zemin üzerine sarı 'X' yazan kod nasıl olur? Renk baytı 2Eh'dir.

Çözüm Örnek 4.1.3'te ayrıntılı verilmiştir. Bağıl adres (10 × 80 + 40) × 2 = 1680 = 690h'dir; yazılacak sözcük 2E58h'dir.

```
MOV AX, 0B800h
MOV DS, AX
MOV BX, 690h
MOV WORD PTR [BX], 2E58h
```

6. Ekrana 'A' yazan program DS'yi değiştirmeden ES ile nasıl yazılır?

B800h, DS yerine ES'ye yüklenir ve yazma komutuna `ES:` öneki eklenir. DS programın kesiminde (0700h) kalır.

```
MOV AX, 0B800h
MOV ES, AX
MOV CL, 'A'
MOV CH, 11011111b
MOV BX, 15Eh
MOV ES:[BX], CX      ; 26 89 0F
```

Yazma komutunun makine kodu, öneksiz hâlinin (`89 0F`) önüne `26` eklenmiş biçimidir. Program sonunda B815Eh = 41h, B815Fh = DFh olur ve DS = 0700h kalır.

4.1.11 Özet

- **Adresleme**: bir bellek işleneni `[taban + indis + yer değiştirme]` biçimindedir. Taban BX ya da BP, indis SI ya da DI, yer değiştirme d8 ya da d16'dır; her biri isteğe bağlıdır. BX ile BP, SI ile DI bir arada kullanılamaz. Bu kurallar 24 birleşim verir ve ModR/M baytının MOD ile R/M alanlarında kodlanır.
- **Mod türleri**: anlık ve yazmaç modları belleğe gitmez. Doğrudan, yazmaç dolaylı, tabanlı, indisli ve tabanlı-indisli modlar bir etkin adres hesaplar. Dizgi komutları DS:SI ve ES:DI'yi örtük kullanır; G/Ç komutları ayrı bir port alanına, atlamalar IP'ye göre göreceli adrese gider.
- **Etkin adres** 16 bitte hesaplanır ve sarar: BP = FFFEh iken `[BP+4]` → 0002h. d8 işaret genişletilir. EA hesabı 8086'da 5–12 saat turu sürer; kesim öneki 2 tur ekler.
- **Kesim seçimi**: adreste BP varsa SS, yoksa DS. Kesim öneki (`ES:` = 26h, `CS:` = 2Eh, `SS:` = 36h, `DS:` = 3Eh) bunu tek komut için değiştirir. CS:IP, SS:SP ve dizgi hedefi ES:DI değiştirilemez.
- **Fiziksel adres** = kesim × 10h + EA; 1234:7890 → 19BD0h. Bir fiziksel adrese en çok 4096 kesim:bağıl konum çifti gider. n adres hattı 2^n konum seçer; 8086'nın 20 hattı 1 MB, 16 bitlik bağıl konum 64 KB'lık kesim verir.
- **Boyut**: `byte ptr` / `word ptr` (emu8086'da `b.` / `w.`) erişim boyutunu belirtir; diğer işlenen yazmaçsa boyut ondan anlaşılır. 8086 **little-endian**'dır: düşük bayt düşük adrese. Sözcük dizisinde bayt uzaklığı indis × 2'dir.
- **MOV** kopyalar ve bayrakları etkilemez. Bellekten belleğe, sabitten kesim yazmacına, kesimden kesime MOV yoktur; CS ve IP hedef olamaz.
- **XCHG** iki işleneni tek komutta takas eder; kesim yazmacı, sabit ve iki bellek işleneni kullanılamaz. Bayrakları etkilemez.
- **Video belleği** B800h kesimindedir. Her hücre karakter + öznitelik baytıdır; bağıl adres = (satır × 80 + sütun) × 2.

4.1.12 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

adresleme modu *addressing mode* · anlık adresleme *immediate addressing* · yazmaç adresleme *register addressing* · doğrudan adresleme *direct addressing* · yazmaç dolaylı adresleme *register indirect addressing* · tabanlı / indisli adresleme *based / indexed addressing* · göreceli adresleme *relative addressing* · taban yazmacı *base register* · indis yazmacı *index register* · yer değiştirme *displacement* · işaret genişletme *sign extension* · etkin adres (EA) *effective address* · ModR/M baytı *ModR/M byte* · kesim *segment* · bağıl konum *offset* · fiziksel adres *physical address* · paragraf (16 bayt) *paragraph* · adres yolu *address bus* · varsayılan kesim *default segment* · kesim öneki *segment override prefix* · PSP (program kesimi öneki) *program segment prefix* · .COM programı *.COM program* · veri tipi belirtici *data type specifier* · bayt / sözcük *byte / word* · little-endian / big-endian · satır öncelikli düzen *row-major order* · takas (XCHG) *exchange* · dizgi *string* · G/Ç portu *I/O port* · video belleği (metin modu) *video memory* · öznitelik baytı *attribute byte* · saat turu *clock cycle* · yol döngüsü *bus cycle*

4.1.13 Kendini Dene

1. ES = 5000h ve DI = FFF0h iken `MOV AL, ES:[DI+20h]` komutu hangi fiziksel adresten okur?

   Cevap

   EA = FFF0h + 0020h = 1 0010h; 16 bitte sarar ve EA = 0010h olur. Önek ES'yi seçer: 50000h + 0010h = 50010h. Sarma olmasaydı 60010h bulunurdu; bu adres ES kesiminin dışındadır.

2. `MOV AL, [BP+SI+5]` komutu kaç bayttır ve hangi kesime erişir? ModR/M baytı nedir?

   Cevap

   Kesim SS'dir, çünkü adreste BP vardır. Yer değiştirme 5, −128…127 aralığında olduğundan d8 seçilir: MOD = 01. REG = 000 (AL), R/M = 010 (`[BP+SI]`). ModR/M = 01 000 010 = 42h. Komut `8A 42 05` , yani 3 bayttır.

3. DS:0300h'den başlayarak bellekte `78 56 34 12` baytları var. `MOV AX, [0301h]` ve `MOV BL,`

`[0303h]` sonrası AX ve BL nedir? Birinci komutta veri yolu açısından ne olur?

Cevap

AX: 0301h'deki 56h AL'ye, 0302h'deki 34h AH'ye gelir; AX = 3456h. BL = 12h. 0301h tek numaralı bir adrestir. 8086 tek adresten başlayan sözcüğü iki ayrı yol döngüsüyle okur; sonuç aynıdır ama erişim 4 saat turu uzar.

4. Aşağıdaki program bitince AX nedir?

```
org 100h
mov bx, offset t
mov si, 3
mov al, [bx+si]
mov ah, [bx+si-2]
ret
t db 10h,20h,30h,40h,50h
```

Cevap

`[bx+si]` t'nin 3 numaralı baytıdır (0'dan sayılarak): AL = 40h. `[bx+si-2]` 1 numaralı bayttır: AH = 20h. AX = 2040h. Programın uzunluğu 12 bayt olduğundan t 010Ch'dedir; sonuç bu adresten bağımsızdır.

5. 45678h fiziksel adresine DS:SI ile ulaşmak isteniyor. DS en küçük ve en büyük kaç olabilir? Kaç farklı çift vardır?

Cevap

En büyük DS: ilk dört basamak, 4567h; SI = 45678h − 45670h = 0008h. En küçük DS: DS × 10h ≥ 45678h − FFFFh = 35679h, yani DS ≥ 3567,9h → 3568h; SI = 45678h − 35680h = FFF8h. Çift sayısı 4567h − 3568h + 1 = 1000h = 4096.

6. 18 adres hattı ve 8 veri hattı olan bir bellek yongasının kapasitesi nedir? 18 adres ve 4 veri hattı olsaydı?

Cevap

2^{18} = 262.144 konum; her konum 1 bayt → 256 KB. 4 veri hattıyla her konum 4 bittir: 262.144 × 4 bit = 1.048.576 bit = 128 KB.

7. Aşağıdakilerden hangileri 8086 assembler'ında kabul edilir? `MOV ES, [BX]` · `MOV [SI], DS` ·

`MOV AX, [DX]` · `XCHG [SI], [DI]` · `MOV BYTE PTR [DI], 300`

Cevap

İlk ikisi geçerlidir: kesim yazmacı bellekten yüklenebilir (`8E 07`) ve belleğe yazılabilir (`8C 1C`). `[DX]` geçersizdir; DX adres yazmacı değildir. XCHG iki bellek işleneni alamaz. 300 bir bayta sığmaz (en çok 255); assembler hata verir.

8. 80 × 25 metin ekranının en son hücresinin (satır 24, sütun 79) video belleğindeki bağıl adresi nedir? Bu hücrenin renk baytı hangi fiziksel adrestedir?

Cevap

(24 × 80 + 79) × 2 = 1999 × 2 = 3998 = 0F9Eh. Karakter B8000h + 0F9Eh = B8F9Eh'de, renk baytı bir sonraki adreste, B8F9Fh'dedir. Ekranın tamamı 4000 bayt olduğundan son bayt 0F9Fh'dir.

9. `MOV AX, [BX+SI+6]` ile `MOV AX, [BX+DI+6]` komutları 8086'da kaçar saat turu sürer? EA tek bir adres çıkarsa ne değişir?

Cevap

Bellekten yazmaca MOV 8 tur artı EA süresidir. `[BX+SI+d]` için EA 11, `[BX+DI+d]` için 12 turdur: 19 ve 20 tur. EA tek ise sözcük iki yol döngüsünde okunur ve süreye 4 tur eklenir: 23 ve 24 tur.

4.1.14 İleri okuma

- B. B. Brey, *The Intel Microprocessors* (8. baskı), Bölüm 3 (Addressing Modes): veri, program ve yığın adresleme modları, 8086'dan Pentium'a kadar örneklerle; bu notun adresleme kısmına en yakın kaynaktır.
- Intel, *8086/8088 User's Manual: Programmer's and Hardware Reference*: ModR/M kodlaması, kesim öneki ve komut süre tablosu (EA süreleri dahil) birinci elden buradadır.
- K. R. Irvine, *Assembly Language for x86 Processors* (7. baskı), Bölüm 4: veri aktarımı, little-endian, OFFSET, PTR ve dolaylı adresleme; gerçek mod örnekleri.
- emu8086 belgeleri, *8086 assembler tutorial for beginners*, Part 2: Memory Access ve MOV instruction; sunumdaki MOV örneklerinin kaynağıdır.

4.1.15 Kaynaklar

1. emu8086 belgeleri, *8086 assembler tutorial for beginners*, bölüm 2: memory access, MOV instruction.

2. Intel, *8086/8088 User's Manual*, adresleme modları ve komut kodlaması.

3. B. B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson, bölüm 3 (Addressing Modes).

4. IBM PC metin modu video belleği (B800h) ve öznitelik baytı belgeleri.

5. K. R. Irvine, *Assembly Language for x86 Processors*, 7. baskı, Pearson.

KISIM II · BÖLÜM 4.2

Değişkenler

Bir programın verisi, sonuçta bellekteki baytlardan ibarettir. 8086 bu baytlara 20 bitlik fiziksel adreslerle ulaşır; ama programcı için `5A73:235B` gibi bir adresle çalışmak hem zor hem de hataya açıktır. Assembly dili bu yüzden bellek konumlarına ad vermeye izin verir: değişken. Bu not önce değişkenlerin DB, DW ve DD direktifleriyle nasıl tanımlandığını anlatır. Ardından assembler'ın onlara bağıl konum atarken ne yaptığını ve çok baytlı değerlerin belleğe hangi sırayla yazıldığını gösterir. Sonra aynı fikri dizilere, dizgilere ve tablolara genişletir; bir değişkenin değeri ile adresi arasındaki farkı LEA ve OFFSET üzerinden işler; bellekte hiç yer kaplamayan EQU sabitlerini ve konum sayacını tanıtır. Son kesim, değişkenlerle veri taşıyan, toplayan, karşılaştıran ve 32 bitlik değer işleyen küçük programları satır satır izler. Her adımda komutun yazmaçları, bayrakları ve belleği nasıl değiştirdiği ve 8086'nın yollarında neler olduğu birlikte verilir.

ÖĞRENME ÇIKTILARI

Değişken tanımlamak
DB, DW, DD, `?` ve DUP ile bayt, sözcük, çift sözcük değişkenleri ve dizileri tanımlamak; adlandırma kurallarını ve değer aralıklarını uygulamak.

Bellek yerleşimini çıkarmak
Bir kaynak programın her komutunun ve her değişkeninin hangi bağıl konuma, hangi bayt sırasıyla yerleştiğini bulmak; ORG 100h'nin, PSP'nin ve hizalamanın etkisini açıklamak.

Dizilerle çalışmak
Bayt ve sözcük dizilerinde, dizgilerde ve iki boyutlu tablolarda bir elemanın bayt uzaklığını hesaplamak ve elemana indisle erişmek.

Adres ile değeri ayırmak
LEA, OFFSET ve değişken adının yazmaca neyi yüklediğini söylemek; işaretçi değişkenleri kullanmak.

Sabitleri kullanmak
EQU sabitlerini ve konum sayacını (`$`) kullanarak bellekte yer kaplamayan, kendiliğinden güncellenen değerler yazmak.

Programları izlemek
Değişkenler arasında veri taşıyan, toplayan, karşılaştıran ve 32 bitlik değer işleyen programları yazmaç, bayrak ve bellek düzeyinde adım adım izlemek.

4.2.1 Değişkenler

8086'nın belleği, 00000h ile FFFFFh arasında numaralanmış bir milyon baytlık uzun bir şerittir. İşlemci bu şeritteki bir bayta ulaşmak için bir kesim yazmacı ve bir bağıl konumdan 20 bitlik fiziksel adres üretir. Makine dili yalnız bu sayıları tanır. Bir sayacı, bir toplamı ya da bir metni bellekte saklayan program her kullanımda verinin adresini yazmak zorunda kalabilir. Böyle bir programı okumak da değiştirmek de çok güçtür. Bir veriyi bir bayt öteye kaydırmak, o adresin geçtiği bütün komutları tek tek düzeltmek anlamına gelirdi.

Assembly dili bu sorunu adlarla çözer. Programcı bir bellek konumuna anlamlı bir ad verir; adresi hesaplama ve her kullanımda yerine koyma işini assembler üstlenir. Bu kesim, adlandırılmış bellek konumlarının, yani değişkenlerin nasıl tanımlandığını ve assembler ile işlemcinin onları nasıl gördüğünü anlatır.

4.2.1.1 Değişken nedir?

Tanım 4.2.1

Değişken (*variable*), bellekte bir ada bağlanmış ve değeri program çalışırken değişebilen veri alanıdır. Değişkenin üç özelliği vardır: **adı**, bellekteki **adresi** (kesim içindeki bağıl konumu) ve kapladığı bayt sayısını belirleyen **türü**.

Programcı için `var1` adını kullanmak, `5A73:235B` adresini kullanmaktan çok daha kolaydır. Ad yalnız kaynak kodda vardır. Assembler kaynak dosyayı makine koduna çevirirken her adı, o değişkene ayırdığı bağıl konumla değiştirir. Çalışan programda ad kalmaz; işlemci yalnız komutun içine gömülmüş 16 bitlik bağıl konumu görür. Bu yüzden değişken kavramı donanımda değil, assembler'da yaşar.

Assembler iki temel tür destekler. **BYTE** türü 8 bittir ve `DB` (*define byte*) ile tanımlanır. **WORD** türü 16 bittir, 8086'nın bir sözcüğüdür ve `DW` (*define word*) ile tanımlanır. Üçüncü bir direktif olan `DD` (*define double word*) 32 bitlik, yani 4 baytlık çift sözcük ayırır. Tanımın genel biçimi şöyledir:

```
ad  DB  değer
ad  DW  değer
ad  DD  değer
```

Aşağıdaki iki satır, birer baytlık ve iki baytlık iki değişken tanımlar:

```
counter DB 10h     ; BYTE
sum     DW 0FFFFh  ; WORD
```



Şekil 4.2.1: `counter` tek bayt kaplar ve 10h'yi tutar. `sum` iki bayt kaplar: `sum` ve `sum+1` adreslerinde FFh ve FFh.

Şekil 4.2.1 bu iki tanımın bellekteki karşılığını gösterir. Assembler `counter` için bir bayt ayırır ve içine 10h yazar. Hemen arkasından `sum` için iki bayt ayırır. `sum` adı bu iki baytın ilkinin adresini temsil eder. İkinci bayta `sum+1` ile ulaşılır. Değişkenler arasında boşluk bırakılmaz: tanımlar kaynakta hangi sırayla yazıldıysa bellekte de o sırayla, ardışık olarak yerleşir.

Tanım 4.2.2

Direktif (*directive*), assembler'a bilgi veren ve kendisi işlemci komutuna çevrilmeyen kaynak satırıdır. `ORG`, `END`, `EQU` hiçbir bayt üretmez. `DB`, `DW` ve `DD` ise komut değil **veri** baytı üretir: verilen ilk değerleri belleğe, programın içine yazar.

Bu ayrım önemlidir. `MOV AL, counter` gibi bir satır, işlemcinin yürüteceği bir makine komutudur. `counter DB 10h` ise yürütülecek bir şey değildir; yalnız programın bayt görüntüsüne 10h değerini koyar. Program diskten belleğe yüklendiğinde bu bayt da yüklenir ve değişkenin ilk değeri olur. Değişkene ayrıca bir başlangıç ataması yapmaya gerek kalmaz.

4.2.1.2 Adlandırma kuralları ve ilk değer

Değişken adı bir etikettir ve assembler'ın sembol tablosuna yazılır. Bu yüzden bazı kurallara uymak zorundadır:

- Ad **harfle başlar**; ardından harf ve rakam gelebilir. MASM ve emu8086 alt çizgiyi ve birkaç özel karakteri de (`_`, `@`, `$`, `?`) kabul eder.
- Yazmaç adları (`AX`, `SI` …) ve komut adları (`MOV`, `ADD` …) ayrılmış sözcüklerdir; değişken adı olamaz.
- Assembler büyük ve küçük harfi ayırmaz: `VAR1`, `var1` ve `Var1` aynı addır.
- Aynı ad iki kez tanımlanamaz. Bu kural sabitler ve etiketler için de geçerlidir.

Bir veri tanımının adı olmak zorunda değildir. `DB 0Ah` satırı bellekte bir bayt ayırır ve içine 0Ah yazar; bu baytın adı yoktur ama adresi vardır. Bir önceki adlandırılmış değişkenin adresine bir sayı eklenerek ona ulaşılabilir. Çok satırlı tablolarda yalnız ilk satıra ad verilmesi bu yüzden yeterlidir.

İlk değer herhangi bir sayı biçiminde yazılabilir: onluk (`10`), onaltılık (`0Ah`), ikili (`00001010b`), sekizli (`12o`) ya da tırnak içinde karakter (`'A'`). Assembler hepsini aynı ikili desene çevirir. İlk değeri önemsiz olan değişkenler için `?` yazılır. `?` "bu alanın başlangıç değeri tanımsızdır" demektir; emu8086 bu baytlara 00 koyar, ama programın bu değere güvenmesi yanlıştır.

Tanım	Sonuç	Açıklama
`sayi1 DB 5`	geçerli	harfle başlayan ad, bir bayt, değer 05h
`1sayi DB 5`	geçersiz	ad rakamla başlıyor
`ax DW 0`	geçersiz	yazmaç adı değişken adı olamaz
`DB 0Ah`	geçerli	adsız; bir bayt ayırır, adresi vardır
`sonuc DW ?`	geçerli	ilk değersiz sözcük; emu8086'da 00 00
`harf DB 'A'`	geçerli	karakterin ASCII kodu: 41h

Tablo 4.2.1: Geçerli ve geçersiz tanımlar.

4.2.1.3 Değer aralıkları

Bir baytta 8 bit, yani $2^8 = 256$ farklı desen vardır. Bu desenler iki biçimde yorumlanabilir. **İşaretsiz** yorumda desenler 0 ile 255 arasındaki sayılardır. **İşaretli** yorumda ikiye tümleyen gösterimi kullanılır: en üst bit 1 ise sayı negatiftir ve aralık −128 ile 127 arasındadır. Sözcük ve çift sözcük için aynı mantık 16 ve 32 bite uygulanır.

Direktif	Boyut	İşaretsiz	İşaretli	Örnek
`DB`	1 bayt	0 … 255	−128 … 127	`x DB -1` → FF
`DW`	2 bayt	0 … 65.535	−32.768 … 32.767	`y DW -2` → FE FF
`DD`	4 bayt	0 … 4.294.967.295	-2^{31} … $2^{31} - 1$	`z DD 1` → 01 00 00 00

Tablo 4.2.2: Veri tanımlama direktiflerinin boyutları ve değer aralıkları.

Assembler bir ilk değeri kabul ederken iki aralığın birleşimine bakar. `DB` için −128 ile 255 arasındaki her sayı bir bayta sığar. `x DB -1` yazıldığında assembler −1'in 8 bitlik ikiye tümleyenini hesaplar: 1'in tersi 1111 1110, artı 1, 1111 1111 = FFh. `x DB 255` yazılsaydı bellekte yine FFh dururdu. İki tanım bellekte aynı baytı üretir. `DB 300` ise bir bayta sığmaz, çünkü 300 = 12Ch dokuz bit ister; assembler bu satırda hata verir ve değeri sessizce kesmez.

Tanım 4.2.3

Bellekte yalnız **bit deseni** saklanır; bir baytın işaretli mi işaretsiz mi olduğu bilgisi bellekte yoktur. Desenin anlamını, onu kullanan **komut** belirler: `JAE`, `JB` ve CF işaretsiz; `JGE`, `JL`, `CBW` ve OF işaretli yorum yapar.

Bu tanım, değişkenlerle çalışırken sık karşılaşılan bir karışıklığı önler. `DB -1` yazmak, assembler'a "bu değişken işaretlidir" demek değildir. Programcı yalnız ilk deseni seçer. Aynı FFh baytı bir komutta 255, bir sonrakinde −1 olarak işlenebilir. Aşağıdaki izleme bunu adım adım gösterir.

4.2.1.4 Kod izleme: FFh 255 mi, −1 mi?

```
ORG 100h
MOV AL, x        ; FFh
ADD AL, 1        ; 255 + 1 ?
MOV AL, x
CBW              ; işaretli: -1
MOV BX, y        ; FFFEh
NEG BX           ; -(-2)
RET
x DB -1
y DW -2
```

Program 0100h'den başlar. Yedi komut toplam 16 bayt tutar (`A0 10 01`, `04 01`, `A0 10 01`, `98`, `8B 1E 11 01`, `F7 DB`, `C3`); bu yüzden `x` 0110h'ye, `y` 0111h'ye yerleşir. Bellekte `x` = FFh, `y` ise little-endian sırayla FE FF baytlarıdır, yani FFFEh.

`MOV AL, x` satırı, assembler'ın gözünde `MOV AL, [0110h]`'dir. Makine kodu `A0 10 01`: A0 işlem kodu "AL'ye, ardından gelen 16 bitlik bağıl konumdaki baytı yükle" anlamına gelir; 10 01 baytları da 0110h'dir. Yürütme biriminin (EU) belleğe kendisi erişmediğini hatırlamak yararlıdır. EU bağıl konumu kuyruktan alır ve okuma işini yol arayüz birimine (BIU) verir. BIU fiziksel adresi DS × 10h + 0110h olarak hesaplar ve bir okuma yol döngüsü yapar. emu8086'nın yükleme kesimi 0700h ise fiziksel adres 07000h + 0110h = 07110h'dir. Bu adres çift olduğu için bayt veri yolunun düşük yarısından (D0–D7) gelir. Komut 10 saat turu sürer. Sonuçta AL = FFh, AX = 00FFh olur. `MOV` bayrakları değiştirmez.

`ADD AL, 1` deseni işaretsiz bakışla işler. 1111 1111 + 0000 0001 = 1 0000 0000; dokuzuncu bit bir bayta sığmaz. AL = 00h kalır ve dışarı taşan bit elde bayrağına yazılır: CF = 1. Sonuç sıfır olduğu için ZF = 1'dir. Aynı toplama işaretli bakışla −1 + 1 = 0'dır ve doğrudur; işaretli taşma olmadığı için OF = 0 kalır. Tek bir toplama iki yorumun sonucunu aynı anda bayraklara yazar: CF işaretsiz taşmayı, OF işaretli taşmayı bildirir. Hangisine bakılacağına program karar verir.

`MOV AL, x` AL'yi yeniden FFh yapar; bayraklar önceki değerlerini korur. `CBW` (*convert byte to word*) AL'yi **işaretli** sayı kabul eder ve 16 bite genişletir: AL'nin en üst bitini AH'nin sekiz bitine kopyalar. AL = FFh'nin en üst biti 1'dir; AH = FFh olur. AX = FFFFh, 16 bitlik −1'dir. Bu işleme işaret genişletme denir. AL işaretsiz 255 olarak düşünülseydi doğru 16 bitlik karşılık 00FFh olurdu; `CBW` bu seçimi yapmaz, her zaman işaretli genişletir. `CBW` yalnız 2 saat turu sürer ve bayrakları değiştirmez.

`MOV BX, y` iki baytı birlikte okur: 0111h'deki FEh düşük bayta (BL), 0112h'deki FFh yüksek bayta (BH) gider. BX = FFFEh, işaretsiz bakışla 65.534, işaretli bakışla −2'dir. `y` tek adreste durduğu için 8086 bu sözcüğü iki ayrı yol döngüsüyle okur. `NEG BX` sayının ikiye tümleyenini alır, yani 0 − BX hesaplar: 0000h − FFFEh = 0002h. İşlenen sıfırdan farklı olduğu için `NEG` CF = 1 yapar. Sonuç sıfır değildir, ZF = 0 olur.

Satır	AX	BX	CF	ZF	OF	Yorum
2 `MOV AL, x`	00FFh	0000h	0	0	0	AL: 255 ya da −1
3 `ADD AL, 1`	0000h	0000h	1	1	0	işaretsiz taşma: CF
4 `MOV AL, x`	00FFh	0000h	1	1	0	bayraklar korunur
5 `CBW`	FFFFh	0000h	1	1	0	işaret genişletme: −1
6 `MOV BX, y`	FFFFh	FFFEh	1	1	0	65.534 ya da −2
7 `NEG BX`	FFFFh	0002h	1	0	0	0 − (−2) = 2

Tablo 4.2.3: Değer aralıkları programının izlemesi (her satır yürütüldükten sonraki durum).



Şekil 4.2.2: Solda `ADD AL, 1` sonrası: AL = 00h, işaretsiz taşma CF = 1 ile bildirilir. Sağda `CBW` sonrası: AL = FFh işaretli −1 kabul edilerek AX = FFFFh'ye genişletilir.

Tablo 4.2.3 ve Şekil 4.2.2 aynı programın iki yüzünü gösterir. Bellekteki iki değişken hiç değişmedi; değişen, desenleri okuyan komutların yorumudur. `ADD` ve CF deseni işaretsiz, `CBW` ve `NEG` işaretli sayı olarak ele aldı.

Sık yapılan hata

`x DB -1` ile tanımlanan bir değişkenin bundan sonra hep negatif sayı gibi davranacağını sanmak yanlıştır. Assembler `-1`'i yalnız FFh desenine çevirir ve türü BYTE olarak kaydeder; işaret bilgisini saklamaz. Bu değişken `JAE` ile karşılaştırılırsa 255 gibi davranır.

4.2.1.5 Değişkenin türü ve boyut denetimi

Assembler her değişken adının yanına bir **tür** de yazar. DB ile tanımlanan ad BYTE (1 bayt), DW ile tanımlanan WORD (2 bayt), DD ile tanımlanan DWORD (4 bayt) türündedir. Tür, bir komutta değişkenin kaç baytının okunacağını ya da yazılacağını belirler ve derleme anında bir güvenlik denetimi sağlar.

Kural basittir: komuttaki diğer işlenen bir yazmaçsa, yazmacın boyutu ile değişkenin türü eşleşmelidir. `MOV AX, w` geçerlidir, çünkü AX de w de 16 bittir. `MOV AL, w` ise geçersizdir; assembler "1 bayt / 2 bayt" uyuşmazlığını bildirir ve derlemeyi durdurur. Diğer işlenen bir sabitse, sabitin kaç bit yazılacağını değişkenin türü belirler: `MOV w, 5` belleğe 05 00 diye iki bayt yazar.

Değişken adı içermeyen bellek işlenenlerinin türü yoktur. `[BX]` yalnız bir adres söyler; o adreste bir bayt mı yoksa bir sözcük mü olduğunu söylemez. Bu yüzden `MOV [BX], 5` belirsizdir ve hata verir. Programcı boyutu `BYTE PTR` ya da `WORD PTR` ile açıkça yazar: `MOV BYTE PTR [BX], 5`. Aynı belirtici, bir değişkenin türünü **yalnız o komut için** değiştirmeye de yarar: `MOV AL, BYTE PTR w` sözcük değişkeninin düşük baytını okur. Değişkenin tanımı ve türü değişmez.

Komut	Sonuç
`MOV AL, w`	hata: işlenen boyutları uyuşmuyor (1 bayt / 2 bayt)
`MOV AL, BYTE PTR w`	AL = 34h (düşük bayt)
`MOV AL, BYTE PTR w+1`	AL = 12h (yüksek bayt)
`MOV w, 5`	w = 0005h; belleğe 05 00 yazılır
`MOV AX, b`	hata: işlenen boyutları uyuşmuyor (2 bayt / 1 bayt)
`MOV [BX], 5`	hata: boyut belirsiz; BYTE PTR ya da WORD PTR gerekir
`MOV b, w`	hata: bellekten belleğe MOV yoktur
`INC b`	b = 06h; yalnız bir bayt artırılır

Tablo 4.2.4: Tür denetimi örnekleri (`w DW 1234h` , `b DB 5`).

Tür denetimi yalnız derleme anında yapılır. İşlemci çalışırken tür bilmez. Komutun boyutu makine kodunun içinde bir bitle kodlanır: 8086 işlem kodlarının çoğunda en düşük bit (**W biti**) 0 ise işlem bayt, 1 ise sözcük üzerindedir. Örneğin `MOV AL, [adres]` A0, `MOV AX, [adres]` A1 ile kodlanır. Assembler değişkenin türüne bakarak doğru W bitini seçer. Yanlış boyutlu bir erişim derlemeden geçebilseydi işlemci onu itirazsız yürütürdü. Bu yüzden assembler'ın verdiği boyut hatası programcının yararınadır: yanlış değer üreten ve fark edilmesi zor bir hatayı daha baştan yakalar.

`MOV b, w` satırındaki hata ise tür hatası değil, bir donanım kuralıdır. 8086'nın hiçbir veri taşıma komutu iki bellek işleneni almaz. Komut biçiminde yalnız bir ModR/M alanı vardır ve bu alan en çok bir bellek işleneni tanımlayabilir. Bellekten belleğe taşıma her zaman bir yazmaç üzerinden iki komutla yapılır.

4.2.1.6 LABEL ile aynı belleğe iki ad

Bazen aynı bellek alanı hem bayt bayt hem de sözcük olarak işlenmek istenir. Her erişimde `BYTE PTR` ya da `WORD PTR` yazmak mümkündür ama okunaklı değildir. `LABEL` direktifi bunun yerine aynı adrese ikinci bir ad ve tür verir.

> **Tanım 4.2.4**
>
> `ad LABEL tür` direktifi bellekte yer ayırmaz. Kendisinden sonra gelen ilk baytın adresine, belirtilen türde (`BYTE` , `WORD` , `DWORD`) yeni bir ad verir.

```
ORG 100h
MOV AX, sozcuk     ; WORD
MOV BL, baytlar    ; düşük
MOV BH, baytlar+1  ; yüksek
MOV baytlar, 0FFh  ; tek bayt
MOV CX, sozcuk     ; sözcük
RET
sozcuk LABEL WORD
baytlar DB 34h, 12h
```

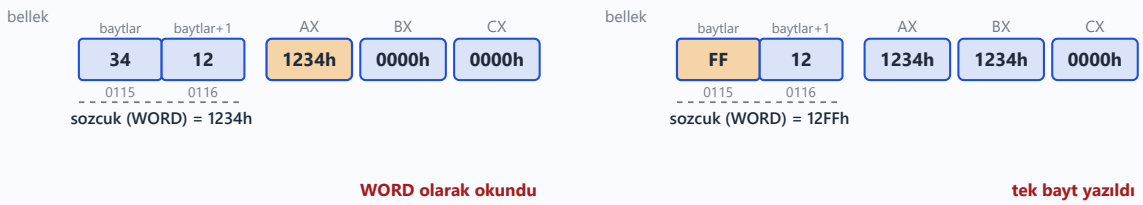
Altı komut 0100h ile 0114h arasını kaplar; `RET` 0114h'dedir. `sozcuk LABEL WORD` hiç bayt üretmediği için hem `sozcuk` hem de `baytlar` 0115h adresini gösterir. Bellekte 0115h'de 34h, 0116h'da 12h durur.

`MOV AX, sozcuk` WORD türünde bir ad kullandığı için iki baytı birden okur: AX = 1234h. Assembler bu satırı `A1 15 01` olarak kodlar. `MOV BL, baytlar` aynı adresin BYTE türündeki adını kullanır ve yalnız 34h'yi okur; PTR yazmaya gerek kalmaz. `MOV BH, baytlar+1` ikinci baytı okur: BH = 12h. BX = 1234h olur. AX ile aynı sonuç, little-endian sıranın doğal sonucudur: düşük adresteki bayt düşük yarıya gider.

`MOV baytlar, 0FFh` satırında hedefin türü BYTE'tır. Assembler 5 baytlık `C6 06 15 01 FF` komutunu üretir ve işlemci yalnız 0115h adresine yazar. Bellek artık FF 12'dir. Son okuma `MOV CX, sozcuk` aynı iki baytı sözcük olarak alır: CX = 12FFh. Bayt adıyla yapılan değişiklik sözcük adıyla görülür, çünkü iki ad tek bir bellek konumunu gösterir.

Satır	AX	BX	CX	[0115h] [0116h]
2 `MOV AX, sozcuk`	1234h	0000h	0000h	34 12
3 `MOV BL, baytlar`	1234h	0034h	0000h	34 12
4 `MOV BH, baytlar+1`	1234h	1234h	0000h	34 12
5 `MOV baytlar, 0FFh`	1234h	1234h	0000h	FF 12
6 `MOV CX, sozcuk`	1234h	1234h	12FFh	FF 12

Tablo 4.2.5: LABEL programının izlemesi.



Şekil 4.2.3: Solda `MOV AX, sozcuk` : iki bayt WORD olarak okunur. Sağda `MOV baytlar, 0FFh` : BYTE türü nedeniyle yalnız 0115h değişir; sozcuk artık 12FFh'dir.

`LABEL` bir tamponu hem bayt hem sözcük olarak işleyen programlarda kullanılır. Örneğin bir seri port tamponu bayt bayt doldurulur ama iki bayt birden karşılaştırılır. Program boyutu yalnız `DB` satırının iki baytı kadar artar; `LABEL` satırı bellekte iz bırakmaz.

4.2.2 Bellek yerleşimi

Bir değişkeni tanımlamak, assembler'dan bellekte yer istemektir. Bu yerin tam olarak neresi olduğunu bilmek, programı izlerken ve hata ayıklarken vazgeçilmezdir. Bir bellek dökümünde hangi baytın hangi değişkene ait olduğunu, bir komutun makine kodunda hangi sayının bir adres olduğunu ancak yerleşimi bilen biri okuyabilir. Bu kesim küçük bir programın baştan sona nasıl bayta dönüştüğünü izler.

4.2.2.1 Örnek program ve bayt görüntüsü

```
ORG 100h
MOV AL, var1   ; AL ← var1
MOV BX, var2   ; BX ← var2
RET            ; programı bitir
VAR1 DB 7      ; BYTE, 7
var2 DW 1234h  ; WORD, 1234h
```

Programın düzeni bilinçli seçilmiştir: önce kod, sonra veriler. Değişkenler `RET`'ten sonra durduğu için işlemci onlara hiç ulaşmaz. Yürütme `RET`'te biter; arkadaki baytlar yalnız veri olarak okunur. Kaynakta değişkenin adı bir yerde `VAR1`, bir yerde `var1` diye yazılmıştır. Assembler büyük-küçük harfe duyarsız olduğu için ikisi aynı değişkendir.



Şekil 4.2.4: Programın bellek görüntüsü. 0100h–0107h arası makine kodu, 0108h `VAR1`, 0109h–010Ah `var2`. Değişken adları komutların içinde bağıl konumlara dönüşmüştür.

Assembler kaynak dosyayı Şekil 4.2.4'teki 11 baytlık diziye çevirir. İlk komut `MOV AL, var1` üç bayttır: `A0 08 01`. A0, AL'ye doğrudan adresten bayt yükleyen kısa işlem kodudur ve ardından gelen iki bayt adresin kendisidir: 0108h. İkinci komut `MOV BX, var2` dört bayttır: `8B 1E 09 01`. 8B işlem kodu "sözcük yazmacına bellekten yükle" anlamındadır, 1E baytı (ModR/M) hedefin BX olduğunu ve adresin komutta doğrudan verildiğini söyler, 09 01 de 0109h'dir. Üçüncü komut tek baytlık `C3`'tür (`RET`). Ardından veri gelir: 0108h'de 07, 0109h'de 34 ve 010Ah'de 12.

Bu görüntüde dikkat çeken iki nokta vardır. Birincisi, makine kodunda hiçbir ad yoktur; `var1` ve `var2` yerini 0108h ve 0109h sayılarına bırakmıştır. İkincisi, adresler bile little-endian yazılır: 0108h, komutun içinde 08 01 sırasıyla durur. Little-endian düzen yalnız veriye değil, komutun içine gömülen her 16 bitlik sayıya uygulanır.

> **Tanım 4.2.5**
>
> **Little-endian** düzende çok baytlı bir değerin **düşük baytı düşük adrese**, yüksek baytı bir sonraki adrese yazılır. 8086 little-endian'dır: `DW 1234h` bellekte 34 12, `DD 12345678h` ise 78 56 34 12 olarak durur.

Bu düzen 8086'nın donanımıyla uyumludur. 16 bitlik veri yolunun düşük yarısı (D0–D7) çift adresli baytı, yüksek yarısı (D8–D15) tek adresli baytı taşır. Çift adresten başlayan bir sözcük okunduğunda düşük bayt kendiliğinden yazmacın düşük yarısına, yüksek bayt yüksek yarısına gelir. Hiçbir bayt takası gerekmez.

4.2.2.2 Kod izleme: var1 ve var2

Programı adım adım izlerken IP'nin her komuttan sonra komutun uzunluğu kadar ilerlediğini görmek, bayt görüntüsünü doğrulamanın en kolay yoludur.

Satır	Makine kodu	IP	AL	BX	Açıklama
1 `ORG 100h`	—	0100h	00h	0000h	yükleme sonrası başlangıç
2 `MOV AL, var1`	A0 08 01	0103h	07h	0000h	[0108h] okunur
3 `MOV BX, var2`	8B 1E 09 01	0107h	07h	1234h	[0109h] ve [010Ah] okunur
4 `RET`	C3	0000h	07h	1234h	yığından 0000h alınır; PSP'deki INT 20h

Tablo 4.2.6: Örnek programın izlemesi. IP, komut yürütüldükten sonraki değerdir.



Şekil 4.2.5: Solda `MOV AL, [0108h]` sonrası AL = 07h, IP = 0103h. Sağda `MOV BX, [0109h]` sonrası: 34h düşük, 12h yüksek bayta gider, BX = 1234h, IP = 0107h.

Birinci adımda işlemci 0100h'deki üç baytı yürütür. BIU fiziksel adresi DS × 10h + 0108h ile hesaplar ve tek bir okuma yol döngüsüyle 07h'yi getirir. AL = 07h, IP = 0103h olur. İkinci adımda adres 0109h'dir ve tektir. Sözcüğün düşük baytı 0109h'de, yüksek baytı 010Ah'dedir. Bu iki bayt veri yolunun aynı yarısına düşmez. 8086 önce 0109h'yi yüksek yarıdan, sonra 010Ah'yi düşük yarıdan okur ve iki baytı BX'te doğru sıraya dizer. Sonuç BX = 1234h'dir; IP dört bayt ilerleyerek 0107h olur.

Son komut `RET`'tir. .COM programı başlarken DOS yığına 0000h değerini koyar: SP = FFFEh, [FFFEh] = 0000h. `RET` bu sözcüğü IP'ye alır ve SP'yi 2 artırır: IP = 0000h, SP = 0000h. Kesimin 0000h bağıl konumunda programın önündeki PSP başlığı vardır ve onun ilk iki baytı `CD 20`, yani `INT 20h`'dir. Bu kesme programı sonlandırır ve denetimi DOS'a geri verir. İzleme tablosunda RET satırında IP'nin 0108h değil 0000h olması bu yüzdendir. 0108h'deki veri baytı hiçbir zaman komut olarak yürütülmez.

Sunumdaki izleme programın 0B56h kesimine yüklendiğini varsayar: DS = CS = 0B56h, `VAR1`'in tam adresi 0B56:0108'dir. emu8086 ise .COM programlarını 0700h kesimine yükler; aynı değişkenin tam adresi orada 0700:0108 olur. İki gösterim çelişmez. Kesim değeri DOS'un programı belleğin neresine yüklediğine bağlıdır ve her çalıştırmada değişebilir. Bağıl konumlar ise derleme anında sabittir. Program yalnız bağıl konumları kullandığı için nereye yüklenirse yüklensin aynı çalışır.

4.2.2.3 Assembler ne yapar?

Assembler kaynak kodu bir bayt dizisine çevirirken iki işi birlikte yürütür. Komutları makine koduna dönüştürür ve değişken adlarını **bağıl konumlarla** değiştirir. Bunu yapabilmek için kaynak dosyayı baştan sona okurken bir sayaç tutar.

Tanım 4.2.6

Konum sayacı (*location counter*), assembler'ın bir sonraki baytın yerleşeceği bağıl konumu tuttuğu sayaçtır. `ORG` onu belirli bir değere ayarlar; her komut ve her veri tanımı onu ürettiği bayt sayısı kadar artırır. Kaynakta `$` simgesiyle okunur.

Örnek programda `ORG 100h` sayacı 0100h yapar. `MOV AL, var1` üç bayt ürettiği için sayaç 0103h olur; `MOV BX, var2` dört bayt üretir, sayaç 0107h'dir; `RET`'ten sonra 0108h'dir. Assembler `VAR1` tanımına geldiğinde sayacın o anki değerini, 0108h'yi, bu adın adresi olarak **sembol tablosuna** yazar.

`DB 7` bir bayt ürettiği için sayaç 0109h olur ve `var2` bu adresi alır.

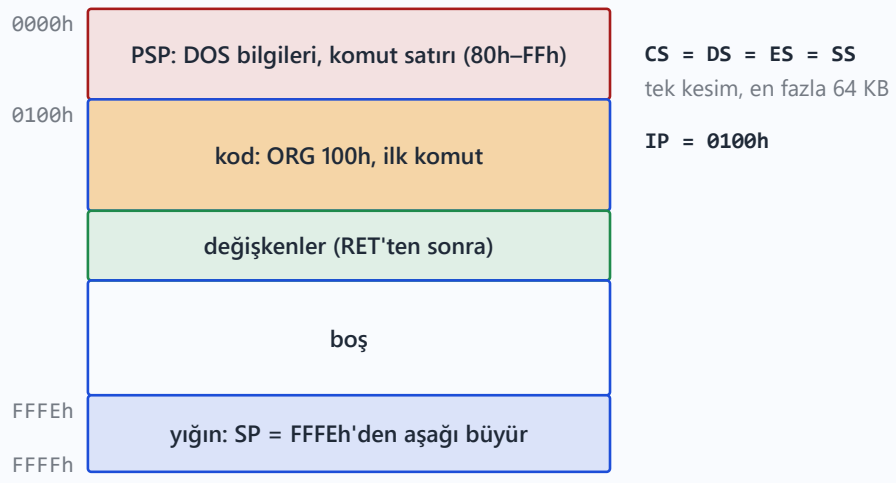
Burada bir sorun vardır. `MOV AL, var1` satırı, `VAR1` tanımından **önce** gelir. Assembler bu satıra ulaştığında `VAR1`'in adresini henüz bilmez. Çözüm kaynağı iki kez okumaktır. Birinci geçişte yalnız her satırın kaç bayt tutacağı hesaplanır ve sembol tablosu doldurulur. İkinci geçişte, bütün adresler bilindiği için makine kodu üretilir. Bu yüzden değişkenler kaynakta kullanıldıkları yerden sonra tanımlanabilir.

Bir değişkenin **bağıl konumu**, kesimin başlangıcından itibaren bayt cinsinden uzaklığıdır. `VAR1`'in bağıl konumu 0108h'dir. Tam (mantıksal) adres kesim değeriyle birlikte yazılır: 0B56:0108. Fiziksel adres ise 0B56h × 10h + 0108h = 0B560h + 0108h = 0B668h'dir. `var2` 0109h'dedir, tam adresi 0B56:0109'dur ve WORD türünde olduğu için 0109h ile 010Ah'yi kaplar.

.COM programları yüklendiğinde DOS dört kesim yazmacının hepsine aynı değeri verir: CS = DS = ES = SS. Bu yüzden kod ile veri aynı kesimde durur ve bir değişkene erişmek için DS'ye ayrıca değer yüklemek gerekmez. Komutun içindeki bağıl konum, DS'nin gösterdiği kesimde doğru yeri bulur.

4.2.2.4 ORG 100h direktifi ve .COM programı

`ORG` (*origin*) direktifi assembler'a konum sayacının başlangıç değerini söyler. `ORG 100h`, programın ilk baytının kesimin 0100h bağıl konumuna yükleneceğini bildirir. Assembler değişken adlarını adreslerle değiştirirken bu değeri hesaba katar. `ORG` unutulursa sayaç 0000h'den başlar ve her değişken adresi gerçek yerinden 100h eksik hesaplanır. Program hata vermeden derlenir ama yanlış baytları okur. emu8086 bu yüzden `ORG` yazılmamış .COM kaynaklarında 100h varsayar. `ORG` bir direktiftir; makine koduna dönüştürülmez ve bayt üretmez.



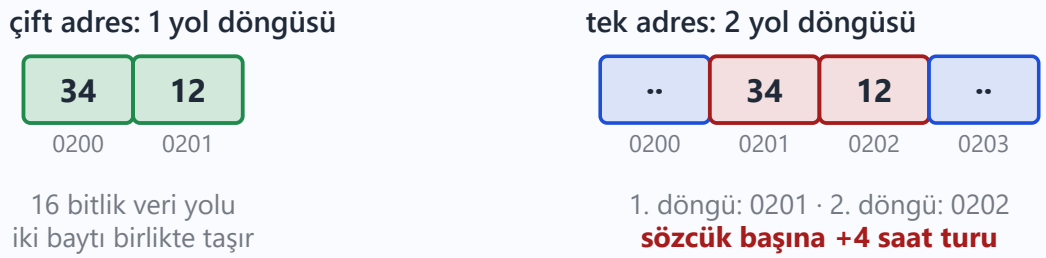
Şekil 4.2.6: .COM programının 64 KB'lık tek kesimdeki yerleşimi. 0000h–00FFh PSP, 0100h'den itibaren kod ve değişkenler, en üstte FFFEh'den aşağı doğru büyüyen yığın. Dört kesim yazmacı aynı değeri taşır.

Neden 100h? DOS bir .COM programı yüklerken kesimin ilk 256 baytına (0000h–00FFh) **PSP** (*program segment prefix*, program kesimi öneki) adlı bir başlık koyar. PSP'de programla ilgili bilgiler tutulur: 0000h'de programı bitiren `INT 20h`, 80h'de komut satırının uzunluğu, 81h–FFh arasında da komut satırı parametreleri bulunur. Program bu başlığın hemen arkasından, 0100h'den başlar. Yığın kesimin en üstündedir: SP = FFFEh ile başlar ve aşağı doğru büyür. Kod, veri ve yığın 64 KB'lık tek bir kesimi paylaşır (Şekil 4.2.6).

.EXE programlarının düzeni farklıdır. .EXE dosyasının kod kesimi 0000h bağıl konumundan başlar ve değişkenler için ayrı bir veri kesimi kullanılır. Kesimlerin her biri 64 KB'a kadar büyüyebildiği için büyük programlar .EXE biçiminde yazılır. Bu derste örneklerin çoğu emu8086'nın .COM şablonuyla yazılır; tek kesim, değişkenlerin adreslerini izlemeyi kolaylaştırır.

4.2.2.5 Hizalama ve EVEN

Bir sözcük değişkeninin hangi adreste başladığı programın doğruluğunu etkilemez, ama hızını etkiler. Nedeni 8086'nın bellek arayüzüdür.



Şekil 4.2.7: Çift adresteki sözcük tek yol döngüsünde, tek adresteki sözcük iki yol döngüsünde okunur. 16 bitlik veri yolu iki baytı ancak çift adresten başlarlarsa birlikte taşır.

8086'nın belleği iki bankaya bölünmüştür. Çift adresli baytlar düşük bankada durur ve veri yolunun D0–D7 hatlarına bağlıdır. Tek adresli baytlar yüksek bankada durur ve D8–D15 hatlarına bağlıdır. İşlemci hangi bankayı seçtiğini iki sinyalle bildirir: A0 adres hattı 0 ise düşük banka, $\overline{\text{BHE}}$ (*bus high enable*) sinyali 0 ise yüksek banka etkindir. Çift adresten başlayan bir sözcük için A0 = 0 ve $\overline{\text{BHE}}$ = 0 aynı anda verilir; iki bayt tek yol döngüsünde birlikte gelir.

Tek adresten başlayan bir sözcükte iki bayt farklı satırlardadır. 0201h'deki bayt yüksek bankada, 0202h'deki bayt bir sonraki satırın düşük bankasındadır. 8086 bunu kendiliğinden fark eder ve iki yol döngüsü yapar: önce 0201h'yi yüksek yarıdan, sonra 0202h'yi düşük yarıdan okur. Program yine doğru değeri alır; yalnız her döngü dört saat turu sürdüğü için sözcük erişimi başına 4 tur kaybedilir.

Örnek 4.2.1

`MOV AX, w` komutu 8086'da 10 saat turu sürer; `w` tek adresteyse 14 tur sürer. Bu komut bir döngüde 1000 kez yürütülürse fark 1000 × 4 = 4000 turdur. 5 MHz'lik bir 8086'da bir tur 1 / 5 MHz = 200 ns'dir. Toplam kayıp 4000 × 200 ns = 800.000 ns = 0,8 ms eder.

Hizalı yerleşimi sağlamak için MASM ve TASM'da `EVEN` direktifi kullanılır. `EVEN`, konum sayacı tekse onu bir artırarak çift sayıya yuvarlar ve araya bir dolgu baytı koyar; sayaç zaten çiftse hiçbir şey yapmaz. Dolgu kod kesiminde genellikle `90h` (`NOP`) baytıdır. Tanımlar yer değiştirmez, yalnız kayar.

```
a  DB 5          ; 0200h
   EVEN          ; 0201h: dolgu
w  DW 1234h      ; 0202h (çift)
```

`a` 0200h'ye yerleşir ve sayaç 0201h olur. `EVEN` sayacı 0202h'ye yuvarlar ve 0201h'ye bir dolgu baytı koyar. `w` çift adreste başlar; bellekte 05, dolgu, 34, 12 sırası oluşur. Bayt değişkenleri için hizalama gerekmez, çünkü tek bir bayt her adreste tek yol döngüsünde okunur. Veri yolu 8 bit olan 8088'de ise her sözcük zaten iki yol döngüsü ister; orada hizalama fark yaratmaz.

4.2.2.6 Değişkenler nereye yazılır?

İşlemci bellekteki baytların kod mu veri mi olduğunu ayırmaz. IP hangi baytı gösteriyorsa onu komut olarak çözer. Değişkenlerin programın neresine yazılacağı bu yüzden önemlidir: yürütme yolunun üzerine düşen veri baytları komut sanılır. Üç yaygın düzen vardır.

Yer	Örnek	Dikkat edilecek nokta
Kodun sonunda	`RET` · `x DB 5`	Yürütme RET'te biter, veriye hiç ulaşmaz. .COM'da en sık kullanılan düzen.
Kodun başında	`JMP basla` · `x DB 5` · `basla:`	JMP veriyi atlar; unutulursa veri baytları komut olarak yürütülür.
Ayrı veri kesimi (.EXE)	`.DATA` · `x DB 5`	Program başında `MOV AX, @data` · `MOV DS, AX` şarttır.

Tablo 4.2.7: Değişkenlerin programdaki yeri.

İlk düzen bu notun programlarının çoğunda kullanılır. İkinci düzende veriler programın başındadır ve ilk komut bir `JMP`'tir. Bu düzenin bir yararı vardır: değişkenlerin adresleri koddaki değişikliklerden etkilenmez. Kod uzayıp kısalsa da `JMP`'in hemen arkasındaki veriler hep aynı bağıl konumda kalır. Örneğin 2 baytlık kısa `JMP`'ten sonra ilk değişken her zaman 0102h'dedir. `JMP` unutulursa işlemci 0100h'deki veri baytlarını komut olarak çözer. Örneğin `DW 10`'un baytları 0A 00, `OR AL, [BX+SI]` olarak yürütülür; program hata vermeden anlamsız işler yapar.

Üçüncü düzen .EXE programlarına özgüdür. Kod CS'nin gösterdiği kesimde, veri DS'nin gösterdiği kesimde durur. Değişkenlerin bağıl konumları veri kesiminin başına göre hesaplanır. Program başladığında DOS, DS'yi programın verisine değil PSP'ye ayarlar. Bu yüzden .EXE programının ilk iki komutu DS'yi yükler. `@data` veri kesiminin kesim değeridir. Kesim yazmacına anlık değer yüklenemediği için değer önce AX'e alınır.

> **Sık yapılan hata**
>
> .EXE programında `MOV AX, @data` · `MOV DS, AX` satırlarını unutmak. `MOV AL, x` doğru bağıl konumu kullanır ama yanlış kesimden, PSP'nin bulunduğu kesimden okur. Assembler da işlemci de hata vermez; program yalnızca yanlış değer alır. .COM programlarında CS = DS olduğu için bu sorun doğmaz.

4.2.3 Diziler

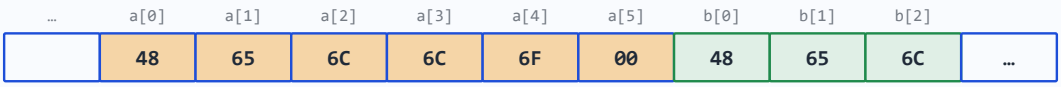
Tek tek değişkenler küçük programlar için yeterlidir. Ama bir sensörden okunan yüz ölçüm, bir metnin harfleri ya da bir dönüşüm tablosu için yüz ayrı ad tanımlamak hem olanaksız hem de anlamsızdır. Bu tür veriler aynı türden ardışık baytlar olarak saklanır ve tek bir adla anılır: dizi.

4.2.3.1 Dizi kavramı

> **Tanım 4.2.7**
>
> **Dizi** (*array*), bellekte ardışık konumlara yerleşmiş aynı türden öğelerdir. Dizinin adı ilk öğenin adresini temsil eder; sonraki öğeler onun hemen ardından gelir. Bir öğeye, dizinin adresine öğenin **bayt uzaklığı** eklenerek ulaşılır.

Bir dizi değişken zinciri gibi düşünülebilir. `DB` ya da `DW` direktifinden sonra virgülle ayrılmış birden çok değer yazılırsa assembler hepsini sırayla, aralarında boşluk bırakmadan yerleştirir. Tırnak içindeki bir metin de kendiliğinden bir bayt dizisine dönüşür: her karakter yerine ASCII kodu yazılır.

```
a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h
b DB 'Hello', 0
```



Şekil 4.2.8: `a` ve `b` dizileri bellekte art arda durur. İki tanım aynı baytları üretir: 48h 'H', 65h 'e', 6Ch 'l', 6Ch 'l', 6Fh 'o' ve dizgi sonu 00h.

Şekil 4.2.8'deki iki tanım farklı yazılmıştır ama bellekte aynı altı baytı üretir. `a` sayıları onaltılık olarak verir; `b` ise metni tırnak içinde yazar ve assembler her harfi ASCII koduna çevirir. Sondaki 0 bir sonlandırıcıdır ve dizginin nerede bittiğini gösterir. Bellek açısından “metin” ile “sayı dizisi” arasında hiçbir fark yoktur; ikisi de bayt dizisidir.

4.2.3.2 Kod izleme: dizi elemanlarına erişim

```
ORG 100h
MOV AL, a[3]   ; a + 3
MOV SI, 0
MOV BL, a[SI]  ; a + 0
INC SI
MOV BH, a[SI]  ; a + 1
MOV a[SI], 'E' ; a[1] ← 45h
RET
a DB 48h,65h,6Ch,6Ch,6Fh,0
```

Program `ORG 100h` ile başlar: kod 0100h'den yüklenir. Yedi komut 21 bayt tutar; `a` dizisi 0115h'den başlar. Bir dizi elemanına iki yolla erişilir: sabit bir indisle ya da bir yazmaçla.

`MOV AL, a[3]` sabit indisli erişimdir. Köşeli parantez içindeki sayı eleman numarası değil, **bayt uzaklığıdır**: `a[3]`, “a'nın adresine 3 ekle” demektir. Uzaklık derleme anında bilindiği için assembler toplamayı kendisi yapar: 0115h + 3 = 0118h. Komut, doğrudan adresli `MOV AL, [0118h]`'e dönüşür ve `A0 18 01` olarak kodlanır. 0118h'deki bayt 6Ch'dir; AL = 6Ch, yani 'l' olur. İşlemci çalışırken hiçbir toplama yapmaz.

`MOV SI, 0` indis yazmacını sıfırlar. `MOV BL, a[SI]` yazmaçlı erişimdir: adres çalışma anında `a + SI` olarak hesaplanır. Bu kez makine kodu `8A 9C 15 01`'dir. ModR/M baytı 9Ch, adresin “SI + 16 bitlik yer değiştirme” olduğunu söyler; yer değiştirme de dizinin adresi 0115h'dir. EU etkin adresi 0115h + SI = 0115h olarak hesaplar ve BIU o baytı okur: BL = 48h ('H').

`INC SI` indisi bir artırır; SI = 1 artık ikinci elemanı gösterir. Aynı biçimdeki `MOV BH, a[SI]` komutu bu kez 0116h'yi okur: BH = 65h ('e'). BX = 6548h olur. Burada döngünün temel fikri görülür: komut aynı kalır, yalnız indis yazmacı değişir. Bir döngü SI'yı her turda bir artırarak bütün diziyi aynı komutla tarayabilir.

`MOV a[SI], 'E'` diziye yazar. Hedef `a` BYTE türünde olduğu için boyut belirsizliği yoktur; assembler `C6 84 15 01 45` komutunu üretir ve tek bayt yazılır: [0116h] = 45h. Dizi artık “HEllo”dur. `'E'` tırnaklı bir karakter sabitidir ve assembler için 45h sayısından farksızdır.

Satır	Etkin adres	AL	SI	BX	Dizi
2 `MOV AL, a[3]`	0118h	6Ch	0000h	0000h	H e l l o 0
3 `MOV SI, 0`	—	6Ch	0000h	0000h	H e l l o 0
4 `MOV BL, a[SI]`	0115h	6Ch	0000h	0048h	H e l l o 0
5 `INC SI`	—	6Ch	0001h	0048h	H e l l o 0
6 `MOV BH, a[SI]`	0116h	6Ch	0001h	6548h	H e l l o 0
7 `MOV a[SI], 'E'`	0116h	6Ch	0001h	6548h	H **E** l l o 0

Tablo 4.2.8: Dizi erişimi programının izlemesi (`a` = 0115h).



Şekil 4.2.9: Solda ilk adım: `a[3]` = a + 3 bayt, AL = 6Ch ('l'). Sağda son yazma: `a[1]` ← 45h; dizi “HEllo” olur.

İki erişim yolu donanımda farklı maliyetlere yol açar. `MOV AL, [0118h]` 10 saat turu sürer. `MOV BL, a[SI]` ise etkin adresi “yer değiştirme + indis” olarak hesaplar; bu hesap 8086'da 9 tur ekler ve komut 8 + 9 = 17 tur sürer. Sabit indisli erişim daha hızlıdır ama esnek değildir; indis yazmacı her turda farklı elemana erişmeyi sağlar.

4.2.3.3 Erişim yolları ve DUP

Bir dizi elemanına erişimin bütün biçimleri aynı etkin adres hesabına iner. 8086'da köşeli parantez içinde yalnız **BX, BP, SI ve DI** kullanılabilir; bu dört yazmaç bellek indis yazmacı olarak çalışabilir. `MOV SI, 3` · `MOV AL, a[SI]` ile `MOV AL, a[3]` aynı baytı okur. Assembler şu yazımları eşdeğer kabul eder:

```
a[SI]  =  [a + SI]  =  [SI + a]  =  [SI]+a
```

Hangisi yazılırsa yazılsın makine kodu aynıdır: adres, indis yazmacı ile 16 bitlik yer değiştirmenin toplamıdır. İndis her zaman **bayt** cinsindendir. Bayt dizisinde i. elemanın uzaklığı i'dir, ama sözcük dizisinde 2 × i'dir. C gibi yüksek düzeyli dillerin aksine assembler eleman boyuyla çarpmayı programcının yerine yapmaz.

Büyük diziler için her değeri tek tek yazmak gerekmez. `DUP` (*duplicate*) işleci bir değer listesini istenen sayıda tekrarlar.

Tanım 4.2.8

`sayı DUP(değer(ler))` ifadesi, parantez içindeki değer listesini `sayı` kez art arda yazar. Parantez içinde `?` yazılırsa ilk değersiz alan ayrılır; `DUP` iç içe kullanılabilir.

- `c DB 5 DUP(9)`, `c DB 9, 9, 9, 9, 9` ile aynıdır: 5 bayt.
- `d DB 5 DUP(1, 2)`, `d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2` ile aynıdır: 10 bayt. Tekrarlanan şey tek bir değer değil, bütün listedir.
- `tampon DB 100 DUP(?)` 100 baytlık ilk değersiz bir alan ayırır. Klavyeden okunan bir satır ya da bir hesap sonucu için yer açmanın olağan yoludur.
- `tablo DW 10 DUP(0)` 10 sözcük, yani 20 bayt ayırır ve hepsini sıfırlar.



Şekil 4.2.10: `DUP` ile üretilen baytlar: `5 DUP(9)` beş tane 09, `5 DUP(1, 2)` on bayt: 01 02 çifti beş kez.

`DW` ile de dizi tanımlanır: `w DW 1000h, 2000h` iki sözcüklük bir sayı dizisidir. 255'ten büyük ya da −128'den küçük değerler bir bayta sığmadığı için bu tür sayılar `DW` ister. `DW`'nin kullanılamadığı yer uzun metinlerdir: `DW 'Hello'` geçersizdir. Bir sözcük en çok iki karakter alabilir; beş karakterli bir metni sözcüklere bölmenin tek anlamlı bir yolu yoktur. Metinler her zaman `DB` ile tanımlanır.

4.2.3.4 Sözcük dizileri

Sözcük dizisinde her eleman iki bayt kaplar. Bu durumda eleman numarası ile bayt uzaklığı birbirinden ayrılır ve en sık yapılan dizi hatası tam bu noktada ortaya çıkar.

```
ORG 100h
MOV AX, w[0]   ; 1. eleman
MOV BX, w[2]   ; 2. eleman
MOV SI, 4
MOV CX, w[SI]  ; 3. eleman
MOV DX, w[1]   ; dikkat!
RET
w DW 1111h, 2222h, 3333h
```

Kod 0100h'den başlar ve 19 bayt tutar; `w` 0113h'dedir. Bellekte 0113h'den itibaren 11 11 22 22 33 33 baytları durur. `MOV AX, w[0]` ilk sözcüğü okur: AX = 1111h. İkinci eleman `w[1]`'de değil `w[2]`'dedir, çünkü ilk eleman iki bayt kaplar: BX = 2222h. Genel kural, i. elemanın (0'dan sayarak)

uzaklığının 2 × i olmasıdır. Üçüncü elemanın indisi 2 × 2 = 4'tür: `MOV SI, 4` · `MOV CX, w[SI]` ile CX = 3333h olur.

Son satır `MOV DX, w[1]` hatalıdır ama assembler bunu fark etmez. Uzaklık 1, birinci elemanın ortasıdır. Komut 0114h'den başlayan sözcüğü okur: düşük bayt 0114h'deki 11h, yüksek bayt 0115h'deki 22h. DX = 2211h olur. Bu değer dizinin hiçbir elemanı değildir; iki komşu elemanın birer yarısından oluşur. Aynı biçimde `w[3]` de 2222h'nin yüksek baytı ile 3333h'nin düşük baytını birleştirir ve 3322h verir. Sözcük dizisinde indisin hep çift olması gerekir.

Satır	Okunan adresler	AX	BX	CX	DX	SI
2 `MOV AX, w[0]`	0113h, 0114h	1111h	0000h	0000h	0000h	0000h
3 `MOV BX, w[2]`	0115h, 0116h	1111h	2222h	0000h	0000h	0000h
4 `MOV SI, 4`	—	1111h	2222h	0000h	0000h	0004h
5 `MOV CX, w[SI]`	0117h, 0118h	1111h	2222h	3333h	0000h	0004h
6 `MOV DX, w[1]`	0114h, 0115h	1111h	2222h	3333h	2211h	0004h

Tablo 4.2.9: Sözcük dizisi programının izlemesi (`w` = 0113h).



Şekil 4.2.11: Solda `w[0]` : AX = 1111h. Sağda `w[1]` : birinci elemanın yüksek baytı ile ikinci elemanın düşük baytı birleşir, DX = 2211h.

Tablo 4.2.9'da `w`'nin tek adreste (0113h) başladığına dikkat etmek gerekir. Bu yerleşimde `w[0]`, `w[2]` ve `w[4]`'ün her biri tek adresli bir sözcüktür ve her okuma iki yol döngüsü ister. `DW` satırından önce bir `EVEN` yazılsaydı dizi 0114h'den başlar ve her eleman tek yol döngüsünde okunurdu. Bu durumda yanlış indisli `w[1]` ise tek adrese düşerdi.

> **Sık yapılan hata**
>
> Sözcük dizisinde i. elemana `w[i]` ile erişmeye çalışmak. Üçüncü eleman `w[2]`'de değil `w[4]`'tedir. Döngülerde indis yazmacı her turda 1 değil 2 artırılır: `ADD SI, 2`.

4.2.3.5 Tanımdan bayta

Bir veri tanımının bellekte hangi baytları ürettiğini kâğıt üzerinde çıkarabilmek, bellek dökümlerini okumak için gereklidir. Kural her zaman aynıdır: değerler soldan sağa sırayla yazılır, her değer türünün boyu kadar yer kaplar ve çok baytlı değerler little-endian düzende durur. Tablo 4.2.10 sık karşıla-

şılan tanımların açılımını verir.

Tanım	Baytlar	Boyut
`b DB 'Hello', 0`	48 65 6C 6C 6F 00	6 bayt
`a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h`	48 65 6C 6C 6F 00	6 bayt
`var2 DW 1234h`	34 12	2 bayt
`d DB 5 DUP(1, 2)`	01 02 01 02 01 02 01 02 01 02	10 bayt
`w DW 3 DUP(?), -1`	?? ?? ?? ?? ?? ?? FF FF	8 bayt
`mydouble DD 12345678h`	78 56 34 12	4 bayt
`x DB 300`	hata: 300 bir bayta sığmaz	—

Tablo 4.2.10: Veri tanımlarının bayt açılımı (?? = ilk değersiz bayt; emu8086 00 yazar).

`w DW 3 DUP(?), -1` satırı iki parçalıdır. Önce üç ilk değersiz sözcük, yani altı bayt ayrılır; emu8086 bunlara 00 yazar. Ardından −1 bir sözcük olarak eklenir. −1'in 16 bitlik ikiye tümleyeni FFFFh'dir ve little-endian sırayla FF FF olarak yazılır. Satırın toplam boyu 8 bayttır. Tanımın tek bir adı olduğu hâlde dört elemanı vardır.

4.2.3.6 Dizgiler ve sonlandırıcılar

Tanım 4.2.9

Dizgi (*string*), bellekte ardışık karakter kodlarından oluşan metindir. Dizginin bittiği yeri bir **sonlandırıcı** bayt (0 ya da '$') ya da başta duran bir uzunluk baytı gösterir.

Bellek bir dizginin nerede bittiğini bilmez. Bir dizgiyi ekrana yazan ya da kopyalayan program, bitiş noktasını bir kurala göre bulmak zorundadır. Üç yaygın kural vardır:

```
s1 DB 'Merhaba', 0            ; sıfırla biten dizgi
s2 DB 'Merhaba$'              ; '$' ile biten (INT 21h / AH = 09h)
s3 DB 13, 10, 'Satir', 13, 10, '$' ; CR LF: satır başı + yeni satır
s4 DB 7, 'Merhaba'            ; uzunluk önekli: ilk bayt = 7
s5 DB "Ali'nin", 0            ; çift tırnak içinde tek tırnak
```

Biçim	Sonu nasıl anlaşılır?	Nerede kullanılır?
0 ile biten	okunan bayt 0 olunca	emu8086.inc `PRINT_STRING`, C dili
'$' ile biten	okunan bayt 24h olunca	DOS dizgi yazdırma hizmeti
uzunluk önekli	ilk bayttaki sayı kadar	DOS satır girişi tamponu (AH = 0Ah)

Tablo 4.2.11: Dizgi biçimleri.

Dizginin bellekte kapladığı yer, karakter sayısı ile sonlandırıcının toplamıdır. `s1` yedi harf ve bir sıfır, yani 8 bayttır. `s3` 'teki 13 ve 10 sayıları karakter değil denetim kodlarıdır: 13 (0Dh) satır başı (CR), 10 (0Ah) yeni satırdır (LF). DOS'ta bir satırı bitirip alttaki satırın başına geçmek için ikisi birlikte yazılır. `s4` uzunluk önekli biçimdir: ilk bayt 7, ardından yedi karakter gelir. Bu biçimde sonlandırıcı gerekmez ve dizginin uzunluğu tek bir okumayla öğrenilir. `s5` tek tırnak içeren bir metni çift tırnakla yazar; assembler iki tırnak türünü de kabul eder.

Sonlandırıcı unutulursa dizgiyi yazan kod durmaz. Dizginin ardındaki baytları da karakter sanır ve bir 0 ya da '$' bulana kadar ekrana anlamsız işaretler basar. '$' ile biten bir dizgi '$' karakterinin kendisini yazdıramaz, çünkü onu görür görmez durur. Türkçe harflerin (ç, ğ, ı, ö, ş, ü) kodu ise kullanılan kod sayfasına bağlıdır. Bu harfler ASCII'nin 0–127 aralığında yer almaz; aynı bayt farklı bir kod sayfasında başka bir harf olarak görünebilir.

4.2.3.7 Karakter sabitleri ve ASCII aritmetiği

Tırnak içindeki karakter assembler için yalnız bir sayıdır. `MOV AL, 'A'` ile `MOV AL, 41h` aynı makine kodunu (`B0 41`) üretir. ASCII tablosunun düzeni, karakterler üzerinde basit aritmetik yapmayı mümkün kılar.

Karakterler	ASCII kodları
'0' … '9'	30h … 39h
'A' … 'Z'	41h … 5Ah
'a' … 'z'	61h … 7Ah
boşluk · '$'	20h · 24h
CR · LF	0Dh · 0Ah

Tablo 4.2.12: Sık kullanılan ASCII kodları.

Rakam karakterleri 30h'den başlar ve sırayla dizilir. Bu yüzden bir rakam karakterinden sayı değerini elde etmek için '0' çıkarmak yeterlidir: '7' = 37h, 37h − 30h = 7. `SUB AL, '0'` bu işi yapar. Ters yönde, 0–9 arasındaki bir sayıyı ekrana yazmak için `ADD AL, '0'` kullanılır. Klavyeden okunan her rakam bir ASCII kodudur; hesaba girmeden önce bu dönüşümden geçer.

Büyük ve küçük harfler arasındaki fark 20h'dir: 'a' = 61h, 'A' = 41h. 20h ikili olarak 0010 0000'dır; yani iki harf yalnız **bit 5**'te ayrılır. Küçük harfte bit 5 = 1, büyük harfte 0'dır. Bu yüzden harf büyütmek için bit 5'i silmek yeter: `SUB AL, 20h` ya da `AND AL, 0DFh` . 0DFh = 1101 1111 maskesi bit 5 dışındaki bütün bitleri korur. Örneğin 'q' = 71h, 71h AND DFh = 51h = 'Q' olur. Harf küçültmek için bit 5 kurulur: `OR AL, 20h` ; 'Q' = 51h, 51h OR 20h = 71h = 'q'.

Sık yapılan hata

`AND AL, 0DFh` 'yi harf olup olmadığını denetlemeden bir dizginin bütün baytlarına uygulamak. Rakamlarda da bit 5 = 1'dir: '1' = 31h, 31h AND DFh = 11h olur ve rakam bir denetim koduna dönüşür. Dönüşümden önce 'a' ≤ AL ≤ 'z' koşulu sınanır.

4.2.3.8 Kod izleme: dizgiyi büyük harfe çevirme

Aşağıdaki program sıfırla biten bir dizgiyi yerinde büyük harfe çevirir. Uzunluğu bilmez; sonlandırıcıyı görünce durur. Harf denetimi rakamların bozulmasını önler.

```
ORG 100h
LEA SI, s
don:
MOV AL, [SI]
CMP AL, 0          ; sonlandırıcı?
JE bitti
CMP AL, 'a'
JB sonraki
CMP AL, 'z'
JA sonraki
SUB AL, 20h        ; 'a' - 'A'
MOV [SI], AL
sonraki:
INC SI
JMP don
bitti:
RET
s DB 'Ab1c', 0
```

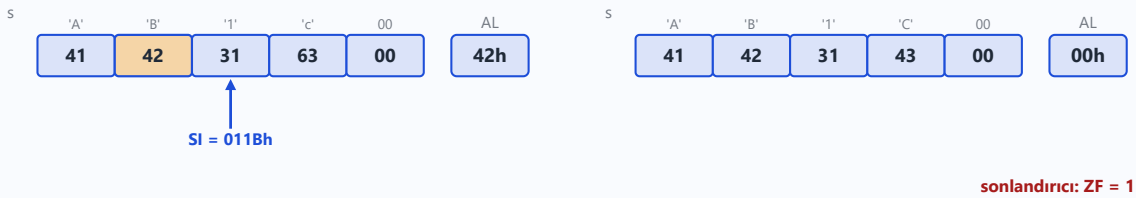
Kod 25 bayttır ve 0100h–0118h arasını kaplar; dizgi 0119h'den başlar. `LEA SI, s` dizginin adresini SI'ya yükler: SI = 0119h. SI bundan sonra işlenen karakteri gösteren bir işaretçidir. Döngünün gövdesi her turda aynı sırayı izler:

1. `MOV AL, [SI]` sıradaki baytı okur.
2. `CMP AL, 0` ve `JE bitti` : bayt 0 ise ZF = 1 olur ve döngü biter.
3. `CMP AL, 'a'` ve `JB sonraki` : AL < 61h ise (CF = 1) bayt küçük harf değildir; yazmadan geçilir.
4. `CMP AL, 'z'` ve `JA sonraki` : AL > 7Ah ise yine geçilir.
5. Bu iki denetimden geçen bayt küçük harftir: `SUB AL, 20h` onu büyütür, `MOV [SI], AL` aynı adrese geri yazar.
6. `INC SI` sonraki bayta geçer, `JMP don` başa döner.

Karşılaştırmalar işaretsiz atlamalarla (`JB` , `JA`) yapılır, çünkü ASCII kodları işaretsiz sayılardır. 'A' = 41h ile 'a' = 61h karşılaştırıldığında 41h − 61h işlemi ödünç ister ve CF = 1 olur; `JB` atlar.

Tur	SI	AL (okunan)	Karar	Yazılan	Dizgi sonra
1	0119h	41h 'A'	41h < 61h: `JB` atlar	—	A b 1 c 0
2	011Ah	62h 'b'	61h ≤ 62h ≤ 7Ah: harf	42h 'B'	A B 1 c 0
3	011Bh	31h '1'	31h < 61h: `JB` atlar	—	A B 1 c 0
4	011Ch	63h 'c'	harf	43h 'C'	A B 1 C 0
5	011Dh	00h	ZF = 1: `JE bitti`	—	A B 1 C 0

Tablo 4.2.13: Büyük harf döngüsünün tur tur izlemesi (`s` = 0119h).



Şekil 4.2.12: Solda ikinci tur: 'b' (62h) − 20h = 42h ('B') aynı adrese yazılır, SI = 011Bh. Sağda son tur: okunan bayt 0, `CMP AL, 0` ZF = 1 yapar ve döngü biter.

Program sonunda SI = 011Dh, AL = 00h ve ZF = 1'dir; dizgi bellekte “AB1C” olmuştur. Döngü beş kez okudu ama yalnız iki kez yazdı. Program dizginin uzunluğunu hiçbir yerde kullanmadı; sonlandırıcı, sayaç yerine durma koşulu oldu. Dizgiye karakter eklense de çıkarılsa da kod değişmeden çalışır. Rakam '1' harf denetimine takıldığı için korundu; denetim olmasaydı 31h − 20h = 11h ile bozulurdu.

4.2.3.9 İki boyutlu diziler

Bir matris ya da bir tablo, satırları ve sütunları olan iki boyutlu bir yapıdır. Bellek ise tek boyutludur. İki boyutlu bir dizi belleğe yerleştirilirken satırlar art arda dizilir.

Tanım 4.2.10

Satır öncelikli düzende (*row-major order*) önce 0. satırın bütün sütunları, sonra 1. satırın bütün sütunları yerleşir. S sütunlu bir tabloda [satır][sütun] elemanının tablonun başından bayt uzaklığı (satır × S + sütun) × eleman boyudur.



Şekil 4.2.13: Üç satır, dört sütunluk bayt tablosu. Mantıksal görünümdeki [2][1] elemanı bellekte 2 × 4 + 1 = 9. bayttır.

```
mat DB 3 DUP(4 DUP(0)) ; 3 × 4
tab DW 3 DUP(4 DUP(0)) ; 24 bayt
```

İç içe `DUP` satır ve sütunu birlikte ayırır: "3 kez (4 kez 0)". `mat` 12 bayt, `tab` 12 sözcük, yani 24 bayttır. Bellekte iç içe yapıdan iz kalmaz; yalnız 12 ya da 24 ardışık bayt vardır. Tabloyu tablo yapan, programın uzaklığı hesaplarken kullandığı formüldür.

Şekil 4.2.13'teki bayt tablosunda [2][1] elemanının uzaklığı 2 × 4 + 1 = 9'dur. Sözcük tablosunda aynı hesap 2 ile çarpılır. Örneğin `tab`'ın [1][2] elemanı (1 × 4 + 2) × 2 = 12. bayttan başlar. Formülde satır sayısı geçmez. Satır sayısı yerleşimi etkilemez, yalnız satır indisinin alabileceği en büyük değeri belirler.

Sütun sayısı 2'nin bir kuvvetiyse çarpma yerine kaydırma kullanılabilir. Dört sütun için satır × 4, satırı iki kez sola kaydırmakla aynıdır: `SHL AL, 1` iki kez. 8086'da 8 bitlik bir `MUL` 70–77 saat turu sürerken her `SHL` 2 tur sürer. Tablo boyutlarının çoğu zaman 2'nin kuvveti seçilmesi bu yüzdendir.

4.2.3.10 Kod izleme: tablo elemanını okuma

```
SUTSAY EQU 4
ORG 100h
MOV AL, satir      ; AL = 2
MOV BL, SUTSAY     ; BL = 4
MUL BL             ; AX = 2 × 4
ADD AL, sutun      ; + 1
MOV SI, AX         ; SI = 9
MOV DL, mat[SI]    ; mat[2][1]
RET
mat DB 11h, 12h, 13h, 14h
    DB 21h, 22h, 23h, 24h
    DB 31h, 32h, 33h, 34h
satir DB 2
sutun DB 1
```

Tablonun elemanları satır ve sütun numarasını taşıyacak biçimde seçilmiştir: 32h, 3. satırın (indis 2) 2. sütunudur (indis 1). Kod 0100h–0111h arasını kaplar; `mat` 0112h'de, `satir` 011Eh'de, `sutun` 011Fh'dedir. Yalnız ilk satır adlandırılmıştır. Sonraki iki `DB` satırı adsızdır ve ilk satırın hemen arkasına yerleşir.

`MOV AL, satir` satır indisini bellekten alır: AL = 2. `MOV BL, SUTSAY` bir sabit kullanır. `SUTSAY EQU 4` bellekte yer kaplamaz; assembler bu satırı `MOV BL, 4` (`B3 04`) olarak kodlar. `MUL BL` AL ile BL'yi işaretsiz çarpar ve 16 bitlik sonucu AX'e yazar: AX = 2 × 4 = 0008h. Bu, 2. satırın tablodaki başlangıç uzaklığıdır. Sonucun üst yarısı (AH) 0 olduğu için CF = OF = 0 olur.

`ADD AL, sutun` sütun indisini ekler: 8 + 1 = 9. AH zaten 0 olduğu için AX = 0009h'dir. Uzaklık bir adres yazmacına alınmalıdır, çünkü 8086'da AL ya da AX adreslemede kullanılamaz; köşeli parantez içinde yalnız BX, SI, DI ve BP olabilir. `MOV SI, AX` ile SI = 0009h olur. Son komut `MOV DL, mat[SI]` etkin adresi 0112h + 9 = 011Bh olarak hesaplar ve oradaki baytı okur: DL = 32h.

Satır	AX	BL	SI	DL	Açıklama
3 `MOV AL, satir`	0002h	00h	0000h	00h	satır indisi
4 `MOV BL, SUTSAY`	0002h	04h	0000h	00h	sabit komuta gömülü
5 `MUL BL`	0008h	04h	0000h	00h	2 × 4: satırın başı
6 `ADD AL, sutun`	0009h	04h	0000h	00h	+ sütun
7 `MOV SI, AX`	0009h	04h	0009h	00h	uzaklık indis yazmacında
8 `MOV DL, mat[SI]`	0009h	04h	0009h	32h	[0112h + 9] = [011Bh]

Tablo 4.2.14: Tablo elemanı programının izlemesi (`mat` = 0112h).



Şekil 4.2.14: Solda `MUL BL` sonrası AX = 0008h: 2. satır tablonun 8. baytından başlar. Sağda `MOV DL, mat[SI]` : 9. bayt okunur, DL = 32h.

Satır ve sütun çalışma anında değişkenlerden okunduğu için aynı kod tablonun her elemanı için çalışır. Sözcük tablosunda son uzaklık ayrıca 2 ile çarpılırdı; bunun için `SHL SI, 1` eklemek yeterlidir.

Sabitin adı bilinçli olarak `SUTSAY` seçilmiştir. `SUTUN EQU 4` yazılsaydı assembler büyük-küçük harfe duyarsız olduğu için bu ad `sutun DB 1` ile çakışırdı. Assembler “sembol iki kez tanımlandı” hatası verir ve derlemeyi durdurur.

4.2.4 Adresler

Bir değişken iki sayıyla ilişkilidir: içinde saklanan **değer** ve bellekteki yeri, yani **adresi**. Assembly'de bir değişkenin adı komutta tek başına yazıldığında değeri temsil eder. `MOV AL, VAR1`, VAR1'in adresindeki baytı AL'ye getirir. Bazen ise değerin değil adresin kendisine ihtiyaç duyulur. Bir diziyi işaretçiyle taramak, yazdırma hizmetine dizginin yerini bildirmek ya da bir prosedüre büyük bir veri bloğunu kopyalamadan vermek bu durumlara örnektir. Bu kesim adresi yazmaca almanın iki yolunu, `LEA` komutunu ve `OFFSET` işlecini anlatır.

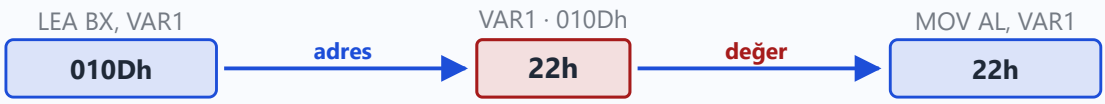
4.2.4.1 Değişkenin adresini alma

Tanım 4.2.11

`LEA` (*load effective address*) bir işlemci komutudur: bellek işleneninin **etkin adresini** (bağıl konumunu) hesaplar ve hedef yazmaca yükler. Belleğe erişmez. `LEA BX, VAR1` BX'e VAR1'in adresini, `LEA BX, a[SI]` ise a + SI toplamını yükler.

Tanım 4.2.12

`OFFSET` bir **assembler işlecidir**: bir değişkenin bağıl konumunu derleme anında bir sabit sayı olarak verir. `MOV BX, OFFSET VAR1`, assembler'ın gözünde `MOV BX, 010Dh` gibi bir anlık değer yüklemesidir.



Şekil 4.2.15: Aynı değişkenin adresi ve değeri. `LEA BX, VAR1` adresi (010Dh), `MOV AL, VAR1` değeri (22h) yükler.

Komut	BX'e yüklenen	Belleğe erişim
`MOV BX, VAR1`	VAR1'in **değeri** (içerik)	var: iki bayt okunur
`LEA BX, VAR1`	VAR1'in **adresi**	yok: yalnız EU'da hesap
`MOV BX, OFFSET VAR1`	VAR1'in **adresi**	yok: adres komutun içinde

Tablo 4.2.15: Yazmaca ne yüklenir? (`VAR1 DW` ile tanımlı)

Bir değişkenin adresi, bir **prosedüre parametre geçirmek** için çok kullanışlıdır. Bin baytlık bir diziyi işleyecek bir alt programa dizinin kendisini kopyalamak hem yavaştır hem de yer harcar. Bunun yerine yalnız 16 bitlik adres, örneğin SI'da, verilir; alt program diziye bu adres üzerinden ulaşır. DOS'un

dizgi yazdırma hizmeti de aynı fikri kullanır: dizginin adresi DX'te beklenir.

4.2.4.2 Kod izleme: LEA ile değişken güncelleme

```
ORG 100h
MOV AL, VAR1          ; değer
LEA BX, VAR1          ; adres
MOV BYTE PTR [BX], 44h
MOV AL, VAR1          ; yeni değer
RET
VAR1 DB 22h
END
```

Programın bayt görüntüsü şöyledir: `A0 0D 01` (3), `BB 0D 01` (3), `C6 07 44` (3), `A0 0D 01` (3) ve `C3` (1). Kod 13 bayttır ve `RET` 010Ch'dedir; `VAR1` 0100h + 13 = 010Dh adresine yerleşir. emu8086, taban ya da indis yazmacı içermeyen `LEA BX, VAR1` komutunu `MOV BX, OFFSET VAR1` ile aynı üç baytlık koda çevirir: `BB` işlem kodu ve 16 bitlik adres. Gerçek 8086 assembler'ı MASM ise aynı satırı `8D 1E 0D 01` biçiminde dört baytlık bir LEA olarak kodlar.

`MOV AL, VAR1` VAR1'in değerini okur: AL = 22h. `LEA BX, VAR1` ise VAR1'in adresini BX'e yükler: BX = 010Dh. BX artık VAR1'i gösteren bir işaretçidir. `MOV BYTE PTR [BX], 44h` bu işaretçi üzerinden yazar. `[BX]` türsüz bir bellek işleneni olduğu için `BYTE PTR` zorunludur. Komut DS:010Dh'ye 44h yazar. Son satırdaki `MOV AL, VAR1` değişkeni adıyla yeniden okur ve yeni değeri görür: AL = 44h. Aynı bellek konumuna hem adıyla hem de bir işaretçiyle erişilmiştir. `END` kaynak dosyanın sonunu bildiren direktiftir ve bayt üretmez.

Satır	AL	BX	[010Dh]	Açıklama
2 `MOV AL, VAR1`	22h	0000h	22h	değer okunur
3 `LEA BX, VAR1`	22h	010Dh	22h	adres yüklenir, bellek okunmaz
4 `MOV BYTE PTR [BX], 44h`	22h	010Dh	44h	işaretçi üzerinden yazma
5 `MOV AL, VAR1`	44h	010Dh	44h	yeni değer okunur

Tablo 4.2.16: LEA programının izlemesi (`VAR1` = 010Dh).



Şekil 4.2.16: Solda `LEA BX, VAR1` sonrası BX = 010Dh, bellekte 22h. Sağda `MOV BYTE PTR [BX], 44h` sonrası VAR1 = 44h.

Donanım açısından `LEA` ile `MOV` arasındaki fark yol döngüsündedir. `MOV AL, VAR1`'de EU adresi BIU'ya verir, BIU bir okuma yol döngüsü yapar ve gelen bayt AL'ye yazılır. `LEA BX, VAR1`'de hiçbir okuma yol döngüsü olmaz: emu8086 adresi komutun içine sabit olarak koyar ve EU onu doğrudan BX'e yazar; bu biçim 4 tur sürer. Yazmaçlı bir `LEA`'da (`LEA BX, a[SI]`) EU adresi çalışma anında hesaplar ama sonucu BIU'ya göndermez, yine doğrudan yazmaca yazar; süresi 2 saat turu artı etkin adres hesabıdır.

4.2.4.3 LEA ve OFFSET aynı mı?

Aynı program `LEA` yerine `OFFSET` ile de yazılabilir:

```
ORG 100h
MOV AL, VAR1
MOV BX, OFFSET VAR1
MOV BYTE PTR [BX], 44h
MOV AL, VAR1
RET
VAR1 DB 22h
END
```

Sonuç aynıdır: BX değişkenin adresini alır, değişken 44h olur, AL son olarak 44h'yi okur. `MOV BX, OFFSET VAR1` sıradan bir anlık değer yüklemesidir: `BB 0D 01`, üç bayt. emu8086 `LEA BX, VAR1` satırını da tam olarak bu koda çevirir; iki programın baytları birebir aynıdır ve `VAR1` ikisinde de 010Dh'dedir. Basit bir değişken için iki yazım arasında hiçbir fark yoktur. MASM'da ise `LEA BX, VAR1` dört baytlık gerçek LEA komutu olarak kodlanır (`8D 1E 0D 01`).

Asıl fark indisli adreslerde ortaya çıkar. `OFFSET` derleme anında çalışır; yalnız assembler'ın bildiği sabit adresleri verebilir. `OFFSET a[SI]` anlamsızdır, çünkü SI'nın değeri derleme anında bilinmez. `LEA BX, a[SI]` ise adresi **çalışma anında**, SI'nın o andaki değeriyle hesaplar. `LEA SI, [BX+DI+4]` gibi karmaşık ifadeler de mümkündür; `LEA` burada bayrakları değiştirmeyen küçük bir toplayıcı gibi çalışır.

	`LEA BX, VAR1`	`MOV BX, OFFSET VAR1`
Ne zaman hesaplanır?	indisli adreste çalışma anında, EU'da	derleme anında, assembler'da
Makine kodu (emu8086)	doğrudan adres: BB imm16 (3 bayt); indisli: 8D ModR/M (+ uzaklık)	BB imm16 (3 bayt)
Süre (8086)	doğrudan adres: 4 tur; indisli: 2 + EA	4 tur
İndisli adres	olur: `LEA BX, a[SI]`	olmaz
Bayraklar	değişmez	değişmez

Tablo 4.2.17: LEA ile OFFSET'in karşılaştırması.

Sabit bir adres için `MOV ... OFFSET` hem kısa hem de hızlıdır. İndisli bir adresin hesaplanması gerektiğinde `LEA` kullanılır. Hangisi seçilirse seçilsin köşeli parantez içinde yalnız BX, SI, DI ve BP bulunabilir.

4.2.4.4 Kod izleme: LEA a[BX] ile OFFSET a

```
ORG 100h
MOV BX, 2
LEA SI, a[BX]      ; SI = a + BX
MOV DI, OFFSET a   ; sabit adres
ADD DI, BX         ; elle ekle
MOV AL, [SI]
INC BX
LEA SI, a[BX]      ; yeni adres
MOV AH, [SI]
RET
a DB 'ABCD'
```

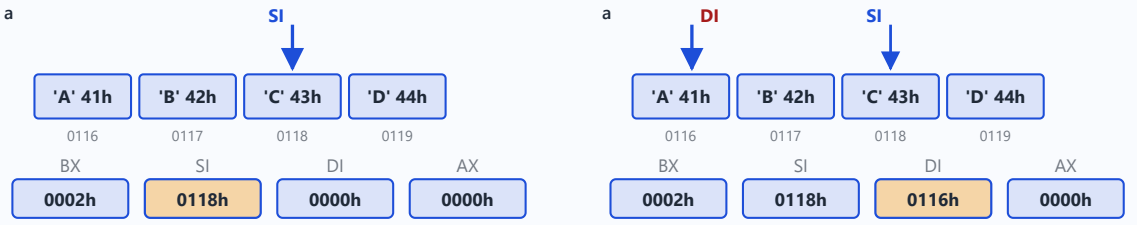
Kod 22 bayttır; `a` dizisi 0116h'de başlar ve 'A' 'B' 'C' 'D' (41h–44h) baytlarını tutar. `MOV BX, 2` indisi hazırlar. `LEA SI, a[BX]` (`8D B7 16 01`) adresi çalışma anında hesaplar: 0116h + 2 = 0118h. Bellek okunmaz; SI yalnız 'C'nin adresini alır.

`MOV DI, OFFSET a` (`BF 16 01`) derleme anında bilinen sabiti yükler: DI = 0116h. Bu sabit BX'i içeremez. Aynı adrese ulaşmak için ayrı bir toplama gerekir: `ADD DI, BX` ile DI = 0118h = SI olur. İki yol aynı adresi verdi; LEA bunu tek komutta yaptı.

`MOV AL, [SI]` 0118h'deki baytı okur: AL = 43h ('C'). `INC BX` indisi 3 yapar. İkinci `LEA SI, a[BX]` aynı makine koduyla yeni sonucu üretir: 0116h + 3 = 0119h. Komut değişmedi, BX değiştiği için adres değişti. `MOV AH, [SI]` AH = 44h ('D') yapar; AX = 4443h olur. DI ise 0118h'de kalır, çünkü OFFSET'ten gelen değer BX'teki değişikliği izlemez.

Satır	BX	SI	DI	AX
2 `MOV BX, 2`	0002h	0000h	0000h	0000h
3 `LEA SI, a[BX]`	0002h	0118h	0000h	0000h
4 `MOV DI, OFFSET a`	0002h	0118h	0116h	0000h
5 `ADD DI, BX`	0002h	0118h	0118h	0000h
6 `MOV AL, [SI]`	0002h	0118h	0118h	0043h
7 `INC BX`	0003h	0118h	0118h	0043h
8 `LEA SI, a[BX]`	0003h	0119h	0118h	0043h
9 `MOV AH, [SI]`	0003h	0119h	0118h	4443h

Tablo 4.2.18: LEA a[BX] ile OFFSET a programının izlemesi (`a` = 0116h).



Şekil 4.2.17: Solda `LEA SI, a[BX]` : SI = 0116h + 2 = 0118h. Sağda `MOV DI, OFFSET a` : DI = 0116h, BX'ten habersiz sabit adres.

4.2.4.5 İşaretçi değişkenleri

Bir adres yalnız yazmaçta değil, bellekte de saklanabilir. `DW`'nin değeri bir değişken adı olursa assembler o değişkenin **adresini** belleğe yazar.

Tanım 4.2.13

İşaretçi değişken (*pointer variable*), içinde başka bir değişkenin adresini saklayan değişkendir. `p DW dizi` 16 bitlik bir yakın işaretçi, `uzak DD dizi` ise bağıl konum ve kesimden oluşan 32 bitlik bir uzak işaretçi tanımlar.

```
ORG 100h
MOV BX, p          ; 010Bh
LEA SI, p          ; 010Fh
MOV DI, OFFSET dizi
RET
dizi DB 'KAYA'     ; 010Bh
p    DW dizi       ; 010Fh, içi 010Bh
uzak DD dizi       ; bağıl konum + kesim
```

Kod 11 bayt tutar (`8B 1E 0F 01`, `BE 0F 01`, `BF 0B 01`, `C3`; emu8086 `LEA SI, p` satırını `MOV SI, OFFSET p` gibi üç bayta çevirir). `dizi` 010Bh'den başlar ve dört bayt kaplar; `p` 010Fh'dedir ve içinde 010Bh, yani 0B 01 baytları durur. `uzak` 0111h'dedir ve dört bayt tutar: önce bağıl konum (0B 01), sonra kesim değeri. emu8086'nın yükleme kesiminde bu baytlar 0B 01 00 07'dir.

Komut	Yazmaca ne yüklenir?	Değer
`MOV BX, p`	p'nin içindeki adres (dizinin adresi)	010Bh
`LEA SI, p`	p'nin kendi adresi	010Fh
`MOV DI, OFFSET dizi`	dizinin adresi, derleme anında sabit	010Bh

Tablo 4.2.19: İşaretçi değişkenle üç farklı yükleme.

Tablo 4.2.19'daki ilk ve üçüncü satır aynı sayıyı verir ama farklı yoldan. `MOV DI, OFFSET dizi` sayıyı komutun içinden alır ve değişmez. `MOV BX, p` ise sayıyı bellekten okur. p bellekte durduğu için program onu değiştirebilir: `ADD p, 2` işaretçiyi dizinin iki bayt ilerisine kaydırır. Bir işaretçiyi kullanmak için önce bir adres yazmacına almak gerekir, çünkü 8086 bellekteki bir adresle doğrudan bellek okuyamaz: `MOV BX, p` · `MOV AL, [BX]`.

Uzak işaretçi farklı bir kesimdeki veriyi göstermek için kullanılır. İçindeki iki sözcük, `LDS` ya da `LES` komutlarıyla tek seferde yüklenir. `LDS SI, uzak` düşük sözcüğü SI'ya, yüksek sözcüğü DS'ye yazar; `LES DI, uzak` aynı işi DI ve ES için yapar. `uzak`'ta 0B 01 00 07 baytları durduğuna göre `LDS SI, uzak` sonrası SI = 010Bh, DS = 0700h olur; ardından `MOV AL, [SI]` 'K'yi (4Bh) okur.

4.2.4.6 Kod izleme: bellekteki işaretçi

```
ORG 100h
MOV BX, p          ; p'nin içi
MOV AL, [BX]       ; dizi[0]
ADD p, 2           ; ilerlet
MOV BX, p
MOV AH, [BX]       ; dizi[2]
RET
dizi DB 'KAYA'
p    DW dizi       ; adres
```

Kod 18 bayttır; `dizi` 0112h'de, `p` 0116h'dedir ve ilk değeri 0112h'dir. `MOV BX, p` p'nin içeriğini okur: BX = 0112h. `MOV AL, [BX]` işaretçinin gösterdiği baytı alır: AL = 4Bh ('K'). `ADD p, 2` doğrudan belleğe yazan bir toplamadır. Assembler sabit küçük olduğu için 5 baytlık `83 06 16 01 02` biçimini seçer; 83h işlem kodu 8 bitlik sabiti işaretle 16 bite genişletir. İşlemci önce p'yi okur, 2 ekler ve sonucu geri yazar. Bu komut bir okuma ve bir yazma yol döngüsü ister. p artık 0114h'dir. BX ise eski adresi tutmaya devam eder; yazmaç ile bellekteki işaretçi birbirinden bağımsızdır.

`MOV BX, p` işaretçiyi yeniden yükler: BX = 0114h. `MOV AH, [BX]` AH = 59h ('Y') yapar; AX = 594Bh olur.

Satır	BX	AX	p
2 `MOV BX, p`	0112h	0000h	0112h
3 `MOV AL, [BX]`	0112h	004Bh	0112h
4 `ADD p, 2`	0112h	004Bh	0114h
5 `MOV BX, p`	0114h	004Bh	0114h
6 `MOV AH, [BX]`	0114h	594Bh	0114h

Tablo 4.2.20: Bellekteki işaretçi programının izlemesi (`dizi` = 0112h, `p` = 0116h).



Şekil 4.2.18: Solda `MOV AL, [BX]` : işaretçinin gösterdiği 'K' okunur. Sağda `ADD p, 2` : bellekteki işaretçi 0114h'yi gösterir; BX eski adreste kalır.

Dizinin adresi kodda hiç yazılmadı; hangi elemanın okunacağına bellekteki işaretçi karar verdi. Yazmaçları az olan 8086'da çok sayıda dizinin konumunu tutmanın yolu budur. Her kullanımda işaretçi önce BX, SI ya da DI'ye alınır.

Sık yapılan hata

Değer ile adresi karıştırmak. `MOV SI, dizi` dizinin adresini değil, ilk sözcüğünü yükler (dizi DB ise boyut hatası verir). Adres için `LEA SI, dizi` ya da `MOV SI, OFFSET dizi` yazılır. İşaretçi değişkende ise tersi doğrudur: `MOV SI, p` adresi verir, `LEA SI, p` işaretçinin kendi yerini.

4.2.5 Sabitler

Programlarda değişmeyen sayılar sık geçer: bir satırdaki karakter sayısı, bir tablonun sütun sayısı, bir portun numarası. Bu sayıları her geçtiği yere elle yazmak iki sorun doğurur. Kod okunaksızlaşır, çünkü `80`'in ne anlama geldiği belli değildir. Değer değişirse bütün kullanımları bulup düzeltmek gerekir ve birinin atlanması kolaydır. Sayıyı bir değişkende tutmak bu sorunları çözer ama bellekte yer kaplar ve her kullanımda bir bellek okuması ister. Sabitler iki sorunu da bedelsiz çözer.

4.2.5.1 EQU ile sabit tanımlama

Tanım 4.2.14

Sabit (*constant*), `ad EQU ifade` biçiminde tanımlanan ve yalnız derleme sırasında var olan addır. Assembler her kullanımda adı ifadenin değeriyle değiştirir. Sabit bellekte yer kaplamaz ve tanımlandıktan sonra değiştirilemez.

```
k      EQU 5
SATIR  EQU 80
MOV AX, k        ; AX = 5
MOV BX, SATIR*2  ; BX = 160
MOV CL, k+1      ; CL = 6
```

Sabitler değişkenlere benzer bir ad taşır ama yalnız assembler'ın sembol tablosunda yaşar. `MOV AX, k` satırını gören assembler onu `MOV AX, 5` olarak kodlar: `B8 05 00`. Sabitin adı da değişken adı gibi büyük-küçük harfe duyarsızdır ve başka bir adla çakışamaz. Bir ifade de verilebilir: `SATIR*2` derleme anında 160 olarak hesaplanır ve komuta 160 gömülür. Programın farklı yerlerinde aynı değer kullanılıyorsa değer tek yerden, EQU satırından güncellenir.

	`k EQU 5`	`k DB 5`
Bellek	kullanmaz	1 bayt
`MOV AL, k`	AL ← 5 (anlık değer, `B0 05`)	AL ← [k] (bellekten, `A0` + adres)
Değiştirilebilir mi?	hayır: `MOV k, 7` hata	evet: `MOV k, 7`
Ne zaman vardır?	yalnız derleme anında	çalışma anında, bellekte

Tablo 4.2.21: Sabit ile değişkenin karşılaştırması.

`MOV k, 7` satırı EQU sabitinde hata verir. Assembler `k`'yi 5 ile değiştirdiğinde ortaya `MOV 5, 7` çıkar ve bir sayının hedef olması anlamsızdır. Değişkende aynı satır geçerlidir, çünkü `k` bir bellek konumunu gösterir.

4.2.5.2 Kod izleme: EQU komuta gömülür

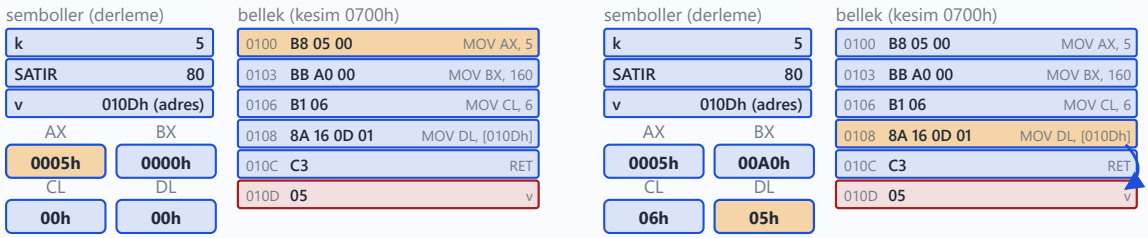
```
k     EQU 5
SATIR EQU 80
ORG 100h
MOV AX, k        ; MOV AX, 5
MOV BX, SATIR*2  ; MOV BX, 160
MOV CL, k+1      ; MOV CL, 6
MOV DL, v        ; bellekten
RET
v DB 5           ; 1 bayt yer
```

İlk iki satır sembol tablosuna k = 5 ve SATIR = 80 yazar; hiçbiri bayt üretmez. Bu yüzden `ORG 100h`'den sonraki ilk bayt doğrudan `MOV AX, k`'nın kodudur. Programın bayt görüntüsü `B8 05 00` · `BB A0 00` · `B1 06` · `8A 16 0D 01` · `C3` · `05`'tir. Kod 13 bayt tutar ve `v` 010Dh'ye yerleşir.

Adres	Makine kodu	Komut	AX	BX	CL	DL
0100h	B8 05 00	`MOV AX, 5`	0005h	0000h	00h	00h
0103h	BB A0 00	`MOV BX, 160`	0005h	00A0h	00h	00h

Adres	Makine kodu	Komut	AX	BX	CL	DL
0106h	B1 06	`MOV CL, 6`	0005h	00A0h	06h	00h
0108h	8A 16 0D 01	`MOV DL, [010Dh]`	0005h	00A0h	06h	05h

Tablo 4.2.22: EQU programının izlemesi.



Şekil 4.2.19: Solda `MOV AX, k` : sabit 5 komutun içindedir (B8 05 00). Sağda `MOV DL, v` : komut v'nin adresini (010Dh) taşır, değer bellekten okunur.

İlk üç komutta değer komutun kendi baytlarındadır. 160 = 00A0h olduğu için `BB A0 00` yazılır; `k+1` = 6 da derleme anında hesaplanır ve tek baytlık `B1 06` olur. İşlemci çalışırken hiçbir toplama ya da çarpma yapmaz. Anlık değer, komutla birlikte önden getirme kuyruğuna girer. EU onu kuyruktan alır ve ayrıca bir veri yol döngüsüne gerek kalmaz. Yazmaca anlık değer yüklemek 8086'da 4 saat turu sürer.

Dördüncü komut farklıdır. `v` bir değişkendir; komut değeri değil adresi taşır (`0D 01`). EU etkin adresi verir, BIU DS:010Dh'den bir okuma yol döngüsü yapar ve DL = 05h olur. Bu komut 8 + 6 = 14 tur sürer. `v` bellekte bir bayt yer kaplar, `k` ve `SATIR` hiç kaplamaz.

MASM ve emu8086'da `=` direktifi de sabit tanımlar: `sayac = 3`. Farkı, `=` ile tanımlanan bir adın kaynakta daha sonra yeniden tanımlanabilmesidir. Bu, derleme anında sayaç gibi kullanılan adlar için uygundur. Program çalışırken ikisi de bellekte yer kaplamaz ve değiştirilemez.

4.2.5.3 Konum sayacı ve boyut işleçleri

Konum sayacı, assembler'ın bir sonraki baytın bağıl konumunu tuttuğu sayaçtır ve kaynakta `$` ile okunur. `$` kullanıldığı satırdaki değerini verir. Bu, bir dizinin boyutunu elle saymadan bulmanın yoludur.

```
s DB 'Merhaba', 0
n EQU $ - s        ; 8
w DW 10 DUP(0)
```

`s` 'nin tanımı bittiğinde konum sayacı dizginin son baytından bir sonraki adresi gösterir. `$ - s` iki adresin farkıdır, yani dizginin bayt sayısı: 7 harf + 1 sonlandırıcı = 8. Bu fark bir EQU sabitine atanır ve derleme anı sabitine dönüşür. Dizgi değiştirilirse `n` kendiliğinden yeni uzunluğu alır. EQU satırının

dizinin **hemen ardından** gelmesi şarttır. Araya başka bir tanım girerse onun baytları da sayılır ve `n` yanlış olur.

MASM aynı bilgiyi değişkenin tanımından okuyan işleçler de sunar. `TYPE` bir elemanın bayt boyunu, `LENGTHOF` eleman sayısını, `SIZEOF` ise toplam bayt sayısını verir (eski MASM sürümlerinde ve emu8086'da `LENGTH` ve `SIZE`).

İfade	Değer	Anlamı
`n`	8	s'nin bayt sayısı
`TYPE s`	1	eleman boyu (DB)
`LENGTHOF s`	8	eleman sayısı
`TYPE w`	2	eleman boyu (DW)
`LENGTHOF w`	10	eleman sayısı
`SIZEOF w`	20	LENGTHOF × TYPE

Tablo 4.2.23: Konum sayacı ve boyut işleçlerinin değerleri.

Bu ifadelerin hepsi derleme anı sabitidir. `MOV CX, LENGTHOF w` komuta 10 sayısını gömer ve `B9 0A 00` olarak kodlanır; bellekte ayrıca yer kaplamaz. Sözcük dizisinde döngü sayacı için LENGTHOF, bayt uzunluğu için SIZEOF kullanılır. Bayt dizisinde ikisi aynıdır.

4.2.5.4 Kod izleme: dizi boyutuyla en büyük eleman

```
ORG 100h
JMP basla
dizi DB 12, 47, 5, 31
n EQU $ - dizi     ; 4
enb DB ?
basla:
MOV CX, n          ; CX = 4
MOV SI, 0
MOV AL, dizi[SI]   ; ilk aday
bak:
CMP AL, dizi[SI]
JAE gec            ; AL >= eleman
MOV AL, dizi[SI]   ; yeni aday
gec:
INC SI
LOOP bak
MOV enb, AL
RET
```

Bu programda veriler başta durur. 0100h'deki iki baytlık `JMP basla` (`EB 05`) veriyi atlar. `dizi` 0102h–0105h'yi, `enb` 0106h'yi kaplar; kod 0107h'den devam eder. `n EQU $ - dizi` satırı dizinin hemen ardındadır: o anda `$` = 0106h, fark 0106h − 0102h = 4'tür. `MOV CX, n` bu yüzden `B9 04 00` olarak kodlanır.

Algoritma "şimdiye kadarki en büyük" fikrine dayanır. AL ilk elemanla başlar ve aday olur. Her turda aday sıradaki elemanla karşılaştırılır. Aday küçükse eleman yeni aday olur. `CMP AL, dizi[SI]` AL − eleman işlemini yapar ve yalnız bayrakları kurar. `JAE` (*jump if above or equal*) işaretsiz karşılaştırır: CF = 0 ise, yani AL ≥ eleman ise atlar. `LOOP bak` CX'i bir azaltır ve CX sıfır değilse `bak`'a döner. `LOOP` ve `INC` CF'yi değiştirmez; tablodaki CF son `CMP`'nin sonucudur.

Tur	Karşılaştırma	CF	JAE	AL	SI	CX
başlangıç	—	0	—	0Ch (12)	0000h	0004h
1	12 − 12	0	atlar	0Ch (12)	0001h	0003h
2	12 − 47	1	atlamaz	2Fh (47)	0002h	0002h
3	47 − 5	0	atlar	2Fh (47)	0003h	0001h
4	47 − 31	0	atlar	2Fh (47)	0004h	0000h

Tablo 4.2.24: En büyük eleman döngüsünün tur tur izlemesi (her tur sonu).



Şekil 4.2.20: Solda ikinci tur: 12 < 47, CF = 1, `JAE` atlamaz ve AL = 47 (2Fh) olur. Sağda döngü sonrası `MOV enb, AL` : en büyük eleman belleğe yazılır.

Dördüncü turdan sonra CX = 0 olur ve `LOOP` düşer. `MOV enb, AL` sonucu 0106h'ye yazar: enb = 2Fh = 47. Döngü sayısı hiçbir yerde elle yazılmadı. Diziye bir eleman eklense `n` 5 olur ve kod değişmeden beş tur döner. Karşılaştırma işaretsizdir, çünkü dizideki değerler 0–255 aralığında düşünülmüştür. Dizi negatif sayılar içerseydi `JAE` yerine işaretli `JGE` kullanılırdı; aksi hâlde −1 (FFh) en büyük sayı olarak seçilirdi.

4.2.6 Örnek programlar

Değişkenlerle ilgili bütün kurallar, birkaç satırlık küçük programlarda bir araya gelir. Bu kesimdeki programlar emu8086'nın örnek programlarına dayanır. Her biri değişkenler arasında veri taşıma, aritmetik, karşılaştırma ya da çok sözcüklü değer gibi temel bir işi yapar. Her programdan sonra bir izleme tablosu yazmaçları, bayrakları ve değişkenleri her adımda verir.

4.2.6.1 Değişkenler arasında veri taşıma

```
ORG 100h
jmp start
value1 DW 10    ; 2 baytlık değişken
value2 DW 0     ; ikinci değişken
start:
MOV AX, value1  ; AX ← value1
MOV value2, AX  ; value2 ← AX
RET
end start
```

Program veriyi başa koyan düzeni kullanır. 0100h'deki `jmp start` iki baytlık kısa bir atlamadır (`EB 04`): 04 sayısı, atlamadan sonraki komutun adresine (0102h) eklenecek uzaklıktır. Hedef 0102h + 4 = 0106h olur. Aradaki dört bayt değişkenlerdir: `value1` 0102h'de (0A 00), `value2` 0104h'te (00 00). `jmp` işlemciyi verilerin üzerinden atlatır. Atlama olmasaydı 0A 00 baytları `OR AL, [BX+SI]` komutu olarak yürütülürdü.

8086'da bellekten belleğe `MOV` olmadığı için taşıma iki adımda yapılır. `MOV AX, value1` (`A1 02 01`) değeri AX'e alır: AX = 000Ah. `MOV value2, AX` (`A3 04 01`) AX'i belleğe yazar: 0104h'te artık 0A 00 vardır. Her iki komut da AX'e özel kısa biçimdir ve 10 saat turu sürer. Ara yazmaç olarak AX seçilmesinin nedeni de budur. Son satırdaki `end start` kaynağın sonunu bildirir ve programın giriş noktasının `start` etiketi olduğunu söyler.

Satır	IP	AX	value1 (0102h)	value2 (0104h)
2 `jmp start`	0106h	0000h	000Ah (0A 00)	0000h (00 00)
6 `MOV AX, value1`	0109h	000Ah	000Ah	0000h
7 `MOV value2, AX`	010Ch	000Ah	000Ah	000Ah (0A 00)

Tablo 4.2.25: Veri taşıma programının izlemesi.



Şekil 4.2.21: Solda `MOV AX, value1` : AX = 000Ah. Sağda `MOV value2, AX` : value2'nin baytları 0A 00 olur; artık value1 = value2 = 10.

Program `ORG 100h` ile başlar ve `RET` ile biter. `ORG` bayt üretmediği için adresler değişmez: `jmp start` yine 0100h'dedir. `MOV value2, AX` 'ten sonra 010Ch'deki `RET` denetimi DOS'a geri verir. Bu satır olmasaydı işlemci 010Ch'den sonraki baytları komut gibi yürütmeye devam ederdi.

4.2.6.2 Değişkenleri toplama

```
ORG 100h
jmp start
value1 DW 105h  ; ilk değer 105h
value2 DW 103h  ; ilk değer 103h
result DW ?     ; belirsiz
start:
MOV AX, value1  ; AX ← value1
ADD AX, value2  ; AX ← AX + value2
MOV result, AX  ; result ← AX
RET
end start
```

Üç değişken 0102h, 0104h ve 0106h'dedir; `jmp start` bu kez altı bayt atlar (`EB 06`). `result ?` ile tanımlanmıştır. emu8086 bu baytlara 00 00 koyar ama değerine güvenilmez; program onu okumadan önce yazar.

`MOV AX, value1` AX = 0105h yapar. `ADD AX, value2` (`03 06 04 01`) ise `MOV` 'dan farklı olarak bellek işlenenini doğrudan kullanabilir: AX ile [0104h]'teki sözcüğü toplar. BIU sözcüğü okur, ALU toplar ve sonuç AX'e yazılır. Komut 9 + 6 = 15 saat turu sürer. Toplam basamak basamak yapılabilir: 5 + 3 = 8, 0 + 0 = 0, 1 + 1 = 2; hiçbir basamakta elde yoktur ve AX = 0208h olur. Onluk sağlama: 261 + 259 = 520 = 208h.

ADD altı durum bayrağını sonuca göre kurar. En üst bitten elde çıkmadığı için CF = 0'dır. Sonuç sıfır değildir, ZF = 0. En üst bit 0'dır, SF = 0. İki pozitif sayının toplamı pozitif kaldığı için işaretli taşma yoktur, OF = 0. Düşük nibble'da 5 + 3 = 8 olur ve bit 3'ten bit 4'e elde geçmez, AF = 0. Eşlik bayrağı yalnız sonucun düşük baytına bakar: 08h = 0000 1000'de tek sayıda (bir) 1 vardır, bu yüzden PF = 0'dır.

Satır	AX	CF	ZF	SF	OF	PF	AF	result (0106h)
başlangıç	0000h	0	0	0	0	0	0	00 00
7 `MOV AX, value1`	0105h	0	0	0	0	0	0	00 00
8 `ADD AX, value2`	0208h	0	0	0	0	0	0	00 00
9 `MOV result, AX`	0208h	0	0	0	0	0	0	08 02

Tablo 4.2.26: Toplama programının izlemesi.



Şekil 4.2.22: Solda `ADD AX, value2` : 0105h + 0103h = 0208h, bayrakların hepsi 0. Sağda `MOV result, AX` :

bellekte little-endian 08 02.

Toplam bir sözcüğe sığmasaydı CF = 1 olurdu. Örneğin value1 = 0F000h, value2 = 2000h için 0F000h + 2000h = 11000h'dir; AX'e yalnız 1000h sığar ve dışarı taşan 1 CF'ye gider. Taşmayı denetleyen bir program `ADD`'den hemen sonra `JNC` ile CF'ye bakar:

```
MOV AX, v1
ADD AX, v2
MOV v3, AX
JNC son
MOV tasma, 1
son:
```

`MOV v3, AX` bayrakları değiştirmediği için `JNC` hâlâ toplamanın CF'sini görür. 105h + 103h için CF = 0'dır ve `tasma` 0 kalır; 0F000h + 2000h için v3 = 1000h, `tasma` = 1 olur.

4.2.6.3 İki değişkeni karşılaştırma

```
ORG 100h
MOV AX, value1     ; AX ← value1
CMP AX, value2     ; AX − value2
JGE greater_or_equal
MOV AX, value2     ; küçükse value2
JMP store_max
greater_or_equal:
MOV AX, value1
store_max:
MOV max_value, AX  ; sonucu sakla
RET
value1 DW 10
value2 DW 5
max_value DW ?
```

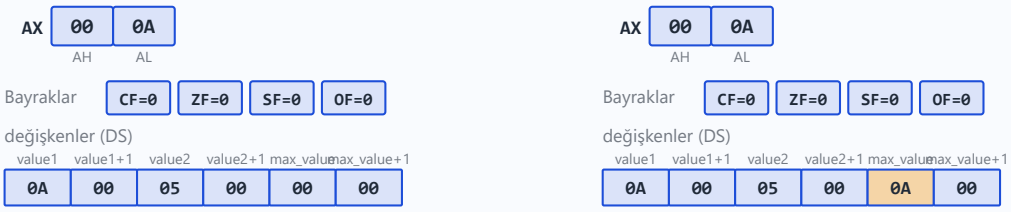
Program iki sayıdan büyük olanı `max_value`'ya yazar. Kod 0100h–0114h arasındadır; `value1` 0115h'de, `value2` 0117h'de, `max_value` 0119h'dedir. Bu düzende veriler `RET`'ten sonra gelir.

`CMP` (*compare*) çıkarma yapar ama sonucu **saklamaz**; yalnız bayrakları kurar. `CMP AX, value2` 000Ah − 0005h = 0005h hesaplar: CF = 0, ZF = 0, SF = 0, OF = 0. AX değişmez. `JGE` (*jump if greater or equal*) işaretli karşılaştırmanın sonucuna bakar ve SF = OF ise atlar. Burada SF = OF = 0 olduğu için atlar ve `greater_or_equal`'a gider. `JGE`'nin makine kodu `7D 05`'tir: hedef, sonraki komutun adresi 0109h'ye 5 eklenerek bulunur, 010Eh.

Atlamanın hedefindeki `MOV AX, value1` AX'i yeniden 000Ah yapar. AX zaten value1'i tuttuğu için bu satır gereksizdir ama zarar vermez. `MOV max_value, AX` sonucu 0119h'ye yazar: 0A 00. `RET` denetimi DOS'a geri verir.

Satır	AX	CF	ZF	SF	OF	max_value	Açıklama
2 `MOV AX, value1`	000Ah	0	0	0	0	0000h	
3 `CMP AX, value2`	000Ah	0	0	0	0	0000h	10 − 5 = 5, sonuç atılır
4 `JGE ...`	000Ah	0	0	0	0	0000h	SF = OF: atlar
8 `MOV AX, value1`	000Ah	0	0	0	0	0000h	gereksiz satır
10 `MOV max_value, AX`	000Ah	0	0	0	0	000Ah	bellekte 0A 00

Tablo 4.2.27: Karşılaştırma programının izlemesi (value1 = 10, value2 = 5).



Şekil 4.2.23: Solda `JGE` : SF = OF olduğu için atlanır. Sağda `MOV max_value, AX` : büyük olan 10 (000Ah) belleğe yazılır.

4.2.6.4 Karşılaştırma: diğer yol

Programın öteki dalını görmek için değişkenlerin değerleri değiştirilir:

```
value1 DW 3
value2 DW 7
max_value DW ?
```

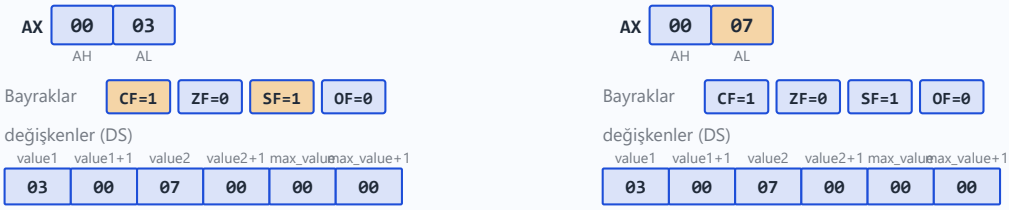
Kodun geri kalanı aynıdır; adresler de değişmez. `MOV AX, value1` AX = 0003h yapar. `CMP AX, value2` 0003h − 0007h hesaplar. İşaretsiz bakışla 3'ten 7 çıkmaz ve ödünç gerekir: CF = 1. 16 bitlik sonuç FFFCh'dir, yani −4; en üst bit 1 olduğu için SF = 1 olur. İşaretli taşma yoktur, çünkü 3 − 7 = −4 aralığın içindedir: OF = 0. Düşük nibble'da 3 − 7 de ödünç ister, AF = 1. Sonucun düşük baytı FCh = 1111 1100'de altı tane 1 vardır, PF = 1.

Bu kez SF = 1, OF = 0'dır ve SF ≠ OF olduğu için `JGE` atlamaz. İşlemci sıradaki komuta geçer: `MOV AX, value2` AX = 0007h yapar. `JMP store_max` koşulsuz atlar ve `greater_or_equal` dalını geçer. `MOV max_value, AX` 0007h'yi yazar. Program iki daldan hangisinden geçerse geçsin `store_max`'ta birleşir.

Satır	AX	CF	ZF	SF	OF	max_value	Açıklama
2 `MOV AX, value1`	0003h	0	0	0	0	0000h	

Satır	AX	CF	ZF	SF	OF	max_value	Açıklama
3 `CMP AX, value2`	0003h	1	0	1	0	0000h	3 − 7 = FFFCh, atılır
4 `JGE ...`	0003h	1	0	1	0	0000h	SF ≠ OF: atlamaz
5 `MOV AX, value2`	0007h	1	0	1	0	0000h	büyük olan
6 `JMP store_max`	0007h	1	0	1	0	0000h	koşulsuz atlama
10 `MOV max_value, AX`	0007h	1	0	1	0	0007h	bellekte 07 00

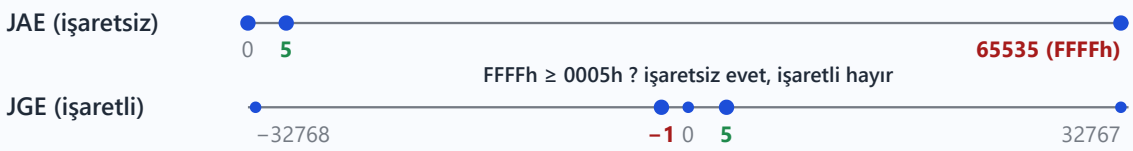
Tablo 4.2.28: Karşılaştırma programının diğer yoldaki izlemesi (value1 = 3, value2 = 7).



Şekil 4.2.24: Solda `CMP AX, value2` : 3 − 7 için CF = 1 ve SF = 1. Sağda `JGE` atlamadığı için `MOV AX, value2` yürütülür: AX = 0007h.

4.2.6.5 İşaretli ve işaretsiz karşılaştırma

Karşılaştırma programındaki atlama komutu bilinçli olarak `JGE` seçilmiştir. 8086'da iki atlama ailesi vardır. `JGE` , `JG` , `JL` ve `JLE` işaretli karşılaştırma yapar ve SF, OF ve ZF'ye bakar. `JAE` , `JA` , `JB` ve `JBE` işaretsiz karşılaştırma yapar ve CF ile ZF'ye bakar. Aynı `CMP` iki ailenin de ihtiyaç duyduğu bayrakları birlikte kurar; programcı yorumu atlama komutuyla seçer.



Şekil 4.2.25: FFFFh ve 0005h iki sayı doğrusu üzerinde. İşaretsiz bakışla FFFFh = 65.535 en büyük sayıdır; işaretli bakışla −1'dir ve 5'ten küçüktür.

Örnek 4.2.2

value1 = −1 (bellekte FFFFh) ve value2 = 5 olsun. `CMP AX, value2` FFFFh − 0005h = FFFAh hesaplar: CF = 0 (işaretsiz olarak 65.535 ≥ 5), SF = 1, OF = 0. `JGE` SF ≠ OF olduğu için atlamaz ve max_value = 0005h olur; bu işaretli bakışla doğrudur, çünkü −1 < 5. `JAE` yazılsaydı CF = 0 olduğu için atlardı ve max_value = FFFFh olurdu; bu da işaretsiz bakışla doğrudur, çünkü 65.535 > 5.

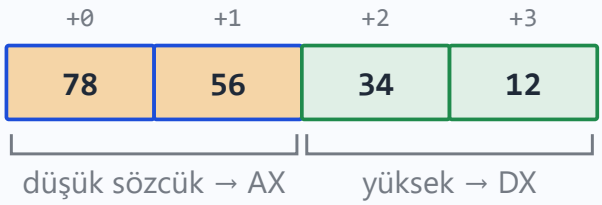
Şekil 4.2.25 aynı desenin iki sayı doğrusundaki yerini gösterir. Hangi atlamanın doğru olduğu, değişkenin neyi temsil ettiğine bağlıdır. Sıcaklık ya da konum gibi negatif olabilen bir değer işaretli, bir sayaç ya da bir adres işaretsiz atlama ister.

Programın sonundaki `RET` zorunludur. Veri tanımları kodun hemen ardından gelir. `RET` olmasaydı işlemci 0115h'deki `0A 00 05 00 …` baytlarını komut olarak yürütürdü. Çözüm ya kodun sonuna `RET` koymak ya da veriyi başa alıp `jmp start` ile atlamaktır. Programın bir iyileştirmesi de mümkündür: `greater_or_equal` altındaki `MOV AX, value1` gereksizdir. `JGE` doğrudan `store_max`'a atlayabilir ve program bir komut kısalır.

4.2.6.6 Çift sözcük değişken

8086 16 bitlik bir işlemcidir; 32 bitlik bir yazmacı yoktur. Yine de 65.535'ten büyük sayılarla çalışmak gerekebilir. Bu tür değerler bellekte `DD` ile 4 bayt olarak saklanır. İşlemcide ise iki 16 bitlik yazmaçta, geleneksel olarak **DX:AX** çiftinde tutulur: DX yüksek sözcüğü, AX düşük sözcüğü alır.

```
org 100h
jmp start
mydouble dd 12345678h  ; double word
mywords  dw 5678h      ; 2 ayrı word
         dw 1234h
mybytes  db 78h        ; 4 ayrı byte
         db 56h
         db 34h
         db 12h
data dd 00010010000110100010101100111100 0b
start:  ; data'yı DX:AX'e yükle
mov ax, word ptr data
mov dx, word ptr [data+2]
ret
```



Şekil 4.2.26: 12345678h bellekte 78 56 34 12 olarak durur. Düşük sözcük (+0, +1) AX'e, yüksek sözcük (+2, +3) DX'e gider.

Programdaki dört tanımın bellekteki karşılığı birebir aynıdır. `mydouble` değeri tek bir çift sözcük olarak verir; assembler onu little-endian sırayla 78 56 34 12 diye yazar. `mywords` aynı baytları iki sözcükle üretir: önce düşük sözcük 5678h (78 56), sonra yüksek sözcük 1234h (34 12). `mybytes` dört

baytı tek tek yazar. `data` değeri ikili tabanda verir: dörder bitlik gruplar 0001 0010 0011 0100 0101 0110 0111 1000, yani 12345678h'dir. Assembler için sayının hangi tabanda ya da hangi parçalarla yazıldığı önemsizdir; bellekte kalan, little-endian dizilmiş dört bayttır.

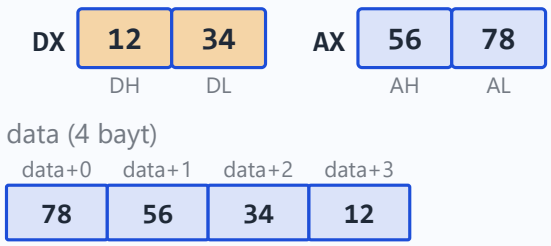
4.2.6.7 Kod izleme: DX:AX'e 32 bitlik yükleme

```
mov ax, word ptr data      ; data+0
mov dx, word ptr [data+2]  ; data+2
ret
data dd 12345678h
```

`data` DD türündedir; bir sözcük yazmacıyla kullanılınca boyutlar eşleşmez. `word ptr` assembler'a bu erişimde yalnız iki bayt okunacağını söyler. Kod 8 bayttır; `data` 0108h'dedir. `mov ax, word ptr data` (`A1 08 01`) 0108h ve 0109h'deki 78 56 baytlarını okur: AX = 5678h. `mov dx, word ptr [data+2]` (`8B 16 0A 01`) 010Ah ve 010Bh'deki 34 12 baytlarını okur: DX = 1234h. DX:AX çifti artık 1234:5678h, yani 12345678h değerini tutar.

Satır	Okunan adresler	DX	AX
1 `mov ax, word ptr data`	0108h, 0109h (78 56)	0000h	5678h
2 `mov dx, word ptr [data+2]`	010Ah, 010Bh (34 12)	1234h	5678h

Tablo 4.2.29: Çift sözcük yükleme izlemesi (`data` = 0108h).



Şekil 4.2.27: İki yüklemeden sonra DX = 1234h, AX = 5678h; DX:AX = 12345678h.

32 bitlik bir değer üzerinde aritmetik de iki parçada yapılır. `data dd 0FFFFFFFFh` değerine 1 eklemek için önce düşük sözcüğe 1 eklenir, sonra düşük sözcükten taşan elde yüksek sözcüğe katılır:

```
ADD AX, 1     ; düşük sözcük
ADC DX, 0     ; yüksek sözcük + elde
```

Başta AX = FFFFh, DX = FFFFh'dir. `ADD AX, 1` AX = 0000h ve CF = 1 yapar. `ADC` (*add with carry*) ikinci işlenene ek olarak CF'yi de toplar: DX = FFFFh + 0 + 1 = 0000h. Sonuç DX:AX = 00000000h'dir ve bu son toplamadan da CF = 1 çıkar, çünkü 32 bitlik sonuç da taşmıştır. Aynı işlem 0001FFFFh için yapılır-

sa DX:AX = 00020000h olur. İkinci satırda `ADC` yerine `ADD DX, 0` yazılsaydı elde kaybolur ve sonuç yanlış olarak 00010000h çıkardı.

4.2.6.8 Değişkenlerle ilgili sık hatalar

Değişkenlerle çalışırken yapılan hataların çoğu assembler'ın yakalayamadığı türdendir: program derlenir, çalışır ve yanlış sonuç verir. Aşağıdaki liste en sık karşılaşılanları toplar.

- **Değer ile adresi karıştırmak.** `MOV SI, dizi` ilk sözcüğü yükler; adres için `LEA SI, dizi` ya da `MOV SI, OFFSET dizi` gerekir.
- **Bellekten belleğe taşımak.** `MOV v1, v2` yoktur; değer bir yazmaçtan geçer. Assembler bunu yakalar.
- **Sözcük dizisinde indis.** 3. eleman `w[4]` 'tedir; indis bayt cinsindendir. `w[1]` iki elemanın yarılarını birleştirir.
- `?` **değerine güvenmek.** `toplam DW ?` sıfır olmayabilir; toplama başlamadan `MOV toplam, 0` yazılır.
- **Soneki unutmak.** `DW 10` onluk 10'dur (000Ah); onaltılık için `10h` yazılır. Harfle başlayan onaltılık sayılar başa 0 alır: `0FFh`, `0ABCDh`.
- **Ad çakışması.** `SUTUN EQU 4` ile `sutun DB 1` aynı addır; assembler "iki kez tanımlı" hatası verir.
- **Veriyi kodun yoluna koymak.** `RET` ya da `JMP` yoksa veri baytları komut sanılır.
- **Sonlandırıcıyı unutmak.** 0 ya da '$' yoksa yazdırma dizginin arkasındaki baytlarla sürer.

4.2.6.9 Veri tanımlama özeti

Yazım	Anlamı	Bellek
`x DB 7`	bir bayt	07
`y DW 1234h`	bir sözcük	34 12
`z DD 12345678h`	çift sözcük	78 56 34 12
`s DB 'Hi', 0`	dizgi + sonlandırıcı	48 69 00
`t DB 3 DUP(0)`	tekrar	00 00 00
`u DW ?`	ilk değersiz	?? ?? (emu8086: 00 00)
`k EQU 10`	sabit	yer kaplamaz

Tablo 4.2.30: Veri tanımlama yazımları ve bellekteki karşılıkları.

Adres almak için `LEA BX, x` ya da `MOV BX, OFFSET x`, değeri okumak için `MOV AL, x` yazılır. Bir erişimin boyutu değişkenin türünden farklı olacaksa `byte ptr` ya da `word ptr` kullanılır.

4.2.7 Özet

- Değişken, bellekte bir ada bağlanmış veri alanıdır. Assembler adı bir bağıl konumla değiştirir; çalışan programda ad yoktur, yalnız komutun içine gömülü adres vardır.
- `DB`, `DW` ve `DD` sırasıyla 1, 2 ve 4 bayt ayırır ve değişkene BYTE, WORD ya da DWORD türü verir. `?` ilk değersiz alan ayırır; değerine güvenilmez.
- Bellekte yalnız bit deseni saklanır. Desenin işaretli mi işaretsiz mi olduğunu onu kullanan komut belirler: CF, `JAE`, `JB` işaretsiz; OF, `JGE`, `JL`, `CBW` işaretli yorum yapar.
- Assembler tür denetimi yapar: yazmaç ile değişkenin boyutu eşleşmelidir. Türsüz işlenenlerde ve bilinçli boyut değişikliğinde `BYTE PTR` / `WORD PTR` kullanılır. `LABEL` aynı adrese ikinci bir ad ve tür verir.
- 8086 little-endian'dır: çok baytlı değerlerin düşük baytı düşük adrestedir. Bu kural komutların içindeki adreslere de uygulanır.
- Assembler konum sayacıyla her baytın bağıl konumunu hesaplar ve iki geçişte adları adreslere çevirir. `ORG 100h` .COM programının PSP'den sonra, 0100h'den başladığını bildirir; .COM'da CS = DS = ES = SS'dir.
- Tek adreste başlayan bir sözcük iki yol döngüsünde okunur ve 4 saat turu kaybettirir. `EVEN` sözcükleri çift adrese hizalar.
- Veriler ya `RET`'ten sonra ya da bir `JMP`'in arkasına yazılır; işlemci yolunun üzerindeki veri baytlarını komut sanır. .EXE programında DS önce yüklenir.
- Dizi, ardışık aynı türden öğelerdir. İndis bayt cinsindendir: sözcük dizisinde i. eleman `2 × i` uzaklıktadır. `DUP` değer listelerini tekrarlar ve iç içe kullanılabilir.
- Dizgiler `DB` ile tanımlanır ve sonları 0, '$' ya da bir uzunluk baytıyla belirlenir. Büyük ve küçük harf yalnız bit 5'te ayrılır; rakam karakterinden '0' çıkarılarak sayı elde edilir.
- İki boyutlu diziler satır öncelikli düzende saklanır: uzaklık = (satır × sütun sayısı + sütun) × eleman boyu.
- `LEA` etkin adresi çalışma anında hesaplar ve belleğe erişmez; `OFFSET` derleme anında sabit bir adres verir. İndisli adres yalnız `LEA` ile alınır. İşaretçi değişkenler adresi bellekte saklar.
- `EQU` sabitleri yalnız derleme anında vardır, komuta anlık değer olarak gömülür ve bellekte yer kaplamaz. `$ - dizi` bir dizinin boyutunu kendiliğinden hesaplar.
- Bellekten belleğe taşıma yoktur; değer bir yazmaçtan geçer. `CMP` sonucu saklamadan bayrakları kurar. 32 bitlik değerler DX:AX çiftinde tutulur ve `ADD` / `ADC` çiftiyle toplanır.

4.2.8 Kavram listesi

değişken *variable* · veri tanımlama (DB / DW / DD) *data definition* · ilk değer *initial value* · direktif *directive* · bayt *byte* · sözcük *word* · çift sözcük *double word*

işaretli / işaretsiz sayı *signed / unsigned number* · ikiye tümleyen *two's complement* · işaret genişletme *sign extension* · veri tipi belirtici *byte ptr / word ptr* · little-endian *little-endian* · bağıl konum *offset* · konum sayacı ($) *location counter* · .COM programı *.COM program* · PSP (program kesimi öneki) *program segment prefix* · hizalı / hizasız erişim *aligned / unaligned access* · yol döngüsü *bus cycle* · dizi *array* · dizgi *string* · ASCII *ASCII* · satır öncelikli düzen *row-major order* · etkin adres (EA) *effective address* · OFFSET işleci *OFFSET operator* · işaretçi değişken *pointer variable* · sabit (EQU) *constant* · anlık değer *immediate value* · elde bayrağı (CF) *carry flag* · taşma bayrağı (OF) *overflow flag*

4.2.9 Kendini Dene

1. `p DB 'Az', 0` · `q DW 300` · `r DD 0Ah` tanımları 0300h'den başlıyor. Her baytın adresi ve değeri nedir? Tanımlar toplam kaç bayt kaplar?

   Cevap

   0300h: 41h ('A'), 0301h: 7Ah ('z'), 0302h: 00h. 300 = 012Ch olduğu için 0303h: 2Ch, 0304h: 01h. `r` 0305h–0308h: 0A 00 00 00. Toplam 3 + 2 + 4 = 9 bayt.

2. `tab DW 5 DUP(4 DUP(?))` kaç bayt kaplar? Satır öncelikli düzende [3][2] elemanının `tab`'a göre bayt uzaklığı nedir?

   Cevap

   5 × 4 = 20 sözcük = 40 bayt. Uzaklık (3 × 4 + 2) × 2 = 28 = 1Ch. Okumak için `MOV SI, 28` · `MOV AX, tab[SI]` yazılabilir.

3. `s DB 'Ders', 13, 10, '$'` satırının hemen ardından `n EQU $ - s` yazılıyor. `n` kaçtır? Araya `t DB 0` girseydi ne olurdu?

   Cevap

   4 harf + CR + LF + '$' = 7. Araya `t DB 0` girseydi `$` bir bayt ilerlemiş olur ve `n` = 8 çıkardı; sabit artık dizginin boyutunu vermezdi.

4. BX = 0100h, DI = 0020h ve CF = 1 iken `LEA SI, [BX+DI+4]` yürütülüyor. SI ne olur? CF değişir mi?

Bellekten okuma yapılır mı?

Cevap

SI = 0100h + 0020h + 4 = 0124h. `LEA` bayrakları değiştirmez, CF = 1 kalır. Yalnız adres hesaplanır; hiçbir okuma yol döngüsü yapılmaz.

5. AL = 80h, BL = 7Fh iken `CMP AL, BL` yürütülüyor. CF, SF ve OF ne olur? Ardından `JA` atlar mı, `JG` atlar mı?

Cevap

80h − 7Fh = 01h. İşaretsiz olarak 128 ≥ 127, ödünç yok: CF = 0, ZF = 0, `JA` atlar. İşaretli olarak −128 − 127 = −255 aralık dışıdır: OF = 1, sonuç 01h olduğu için SF = 0. SF ≠ OF olduğundan −128 < 127 sonucu çıkar ve `JG` atlamaz.

6. `w DW 0102h, 0304h` tanımlı. `MOV AX, w[2]` ve `MOV BL, BYTE PTR w+1` sonrası AX ve BL nedir? `MOV BL, w+1` neden derlenmez?

Cevap

`w[2]` ikinci elemandır: AX = 0304h. Bellek 02 01 04 03 olduğu için `w+1` 'deki bayt 01h'dir: BL = 01h. `w` WORD türünde olduğu için `MOV BL, w+1` boyut uyuşmazlığı verir; `BYTE PTR` gerekir.

7. `MOV DX, OFFSET msg` ile `LEA DX, msg` aynı sonucu verir. Hangisi kaç bayttır ve 8086'da yaklaşık kaç saat turu sürer? Hangisi indisli adres için kullanılabilir?

Cevap

`MOV DX, OFFSET msg` anlık değer yüklemesidir: 3 bayt (BA + 16 bit), 4 tur. emu8086 `LEA DX, msg` satırını aynı koda çevirir; ikisi de 3 bayt ve 4 turdur (MASM'daki gerçek LEA ise 4 bayt, 8D 16 + 16 bit, 2 + 6 = 8 turdur). İndisli adres (`LEA DX, msg[SI]`) yalnız LEA ile alınır.

8. DX:AX = 0001FFFEh iken 3 eklemek için `ADD AX, 3` · `ADC DX, 0` yürütülüyor. Her komuttan sonra AX, DX ve CF nedir?

Cevap

`ADD AX, 3` : FFFEh + 3 = 10001h; AX = 0001h, CF = 1. `ADC DX, 0` : 0001h + 0 + 1 = 0002h, CF = 0. Sonuç DX:AX = 00020001h = 0001FFFEh + 3.

9. Konum sayacı 0105h iken `a DB 1, 2` · `EVEN` · `w DW 5` yazılıyor. `w` hangi adrese yerleşir? `EVEN`

olmasaydı `MOV AX, w` neden daha yavaş olurdu?

Cevap

`a` 0105h–0106h'yi kaplar, sayaç 0107h olur. `EVEN` 0107h'ye bir dolgu baytı koyar ve `w` 0108h'ye yerleşir. `EVEN` olmasaydı `w` 0107h'de, tek adreste başlardı. Sözcük iki yol döngüsünde okunur ve erişim 4 tur uzardı (10 yerine 14).

4.2.10 İleri okuma

- emu8086 belgeleri, *8086 assembler tutorial for beginners*, 3. bölüm (*variables*): DB, DW, diziler, DUP, LEA, OFFSET ve EQU, bu notun programlarının çıkış noktası olan örneklerle anlatılır.
- Kip R. Irvine, *Assembly Language for x86 Processors*, veri tanımlama bölümü: veri direktifleri, little-endian düzen, `$`, TYPE, LENGTHOF ve SIZEOF işleçleri ayrıntılı örneklerle verilir.
- Barry B. Brey, *The Intel Microprocessors*, adresleme modları ve veri taşıma komutları bölümleri: LEA, LDS/LES ve doğrudan adreslemenin makine kodu ve donanım düzeyindeki işleyişi ele alınır.
- Microsoft, *MASM 6.1 Programmer's Guide*: DB/DW/DD, DUP, EQU, `=`, OFFSET, LABEL ve EVEN direktiflerinin kesin tanımları ve kuralları.

4.2.11 Kaynaklar

1. emu8086 belgeleri, *8086 assembler tutorial for beginners*, bölüm 3: variables (DB, DW, DUP, LEA, OFFSET, EQU).
2. Barry B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson.
3. Kip R. Irvine, *Assembly Language for x86 Processors*, Pearson (veri tanımlama direktifleri).
4. Microsoft, *MASM 6.1 Programmer's Guide*: DB/DW/DD, DUP, EQU, OFFSET.
5. Intel, *The 8086 Family User's Manual*, 1979 (komut kodlaması ve saat turu sayıları).

KISIM II · BÖLÜM 5

Kesmeler ve BIOS/DOS Hizmetleri

Kesme, işlemcinin o anki işini geçici olarak bırakıp özel bir hizmet yordamını çalıştırmasını sağlayan olaydır. Bölümde kesme türleri, kesme vektör tablosu, INT ve IRET sırasında FLAGS, CS ve IP'nin yığına yazılıp geri alınması, işlemci istisnaları ve 8259A ile donanım kesmelerinin maskelenmesi anlatılır. BIOS ve DOS hizmetleri kısmında INT 10h ile ekran çıktısı, INT 16h ile klavye girişi, INT 21h ile karakter ve dizgi işlemleri, tarih ve saat okuma ile INT 1Ah işlenir. Programın doğru biçimde sonlandırılması ve hizmetleri bir araya getiren bir uygulama ile bölüm tamamlanır.

Kod (8086): `github.com/sercankulcu/microprocessors-8086/Bolum05`

BU BÖLÜMÜN NOTLARI

5.1 Kesmeler

5.2 Kesme Fonksiyonları

KISIM II · BÖLÜM 5.1

Kesmeler

Bir işlemci yalnız kendi programını yürütseydi dış dünyadan haberi olmazdı. Klavyeye basılır, zamanlayıcı bir darbe üretir, disk sürücüsü veriyi hazırlar; işlemcinin bu olaylara zamanında cevap vermesi gerekir. Kesme, işlemcinin o anki işini bir komut sınırında bırakıp olaya ait yordamı çalıştırması ve sonra kaldığı yerden devam etmesidir. Not, kesmenin ne olduğunu ve yoklamadan farkını anlatarak başlar. Ardından 8086'nın kesme vektör tablosunu ve INT ile IRET komutlarının yığına etkisini ele alır. Sonra işlemcinin kendi ürettiği istisnalara, INTR ve NMI pinlerinden gelen donanım kesmelerine, 8259A kesme denetleyicisine ve kesme gecikmesine geçer. Son kesimde bir kesme hizmet yordamı baştan yazılır, vektörü kurulur ve program biterken eski vektör geri yüklenir. Her kod parçası yazmaçlar, bayraklar, vektör tablosu ve yığınla birlikte adım adım izlenir.

ÖĞRENME ÇIKTILARI

Kesme kavramını açıklamak
Kesmenin ne olduğunu, yoklamadan (*polling*) farkını ve yazılım, donanım ve iç kesmelerin nereden geldiğini açıklamak.

Vektör adresini hesaplamak
Kesme vektör tablosunu kullanarak tip n için vektörün adresini (n × 4), oradaki baytlardan ISR'nin CS:IP değerini ve fiziksel adresini bulmak.

INT ve IRET'i yığında izlemek
INT n ve IRET sırasında FLAGS, CS ve IP'nin yığına hangi sırayla yazıldığını, SP'nin nasıl değiştiğini ve IF ile TF'nin neden sıfırlandığını adım adım göstermek.

İstisnaları tanımak
Bölme hatasının (INT 0) iki nedenini, INTO'nun (INT 4) çalışma koşulunu, tek adım (INT 1) ve kesme noktası (INT 3) düzeneklerini açıklamak.

Donanım kesmelerini izlemek
INTR kabul sürecini, kesme önceliklerini, 8259A'nın IRR, IMR ve ISR yazmaçlarını ve IMR ile tek bir IRQ'yu maskelemeyi izlemek; kesme gecikmesini saat turu olarak hesaplamak.

Kesme hizmet yordamı yazmak
Kendi ISR'sini yazmak, vektörünü güvenle kurmak, eski vektörü saklayıp geri yüklemek ve ISR yazım kurallarına uymak.

5.1.1 Kesme kavramı

5.1.1.1 Kesme nedir?

Bir program çalışırken işlemci komutları sırayla getirir, çözer ve yürütür. IP her komuttan sonra bir sonraki komutu gösterir; atlama ve çağrı komutları dışında bu sıra bozulmaz. Oysa bilgisayarın çevresinde programdan bağımsız olaylar olur. Kullanıcı bir tuşa basar, zamanlayıcı yongası belirli aralıklarla darbe üretir, seri porta bir karakter ulaşır. İşlemcinin bu olayları kaçırmaması ve onlara kısa sürede cevap vermesi gerekir.

Tanım 5.1.1

Kesme (*interrupt*), işlemcinin o anki işini geçici olarak bırakıp özel bir yordamı çalıştırmasını sağlayan olaydır. Bu yordama **kesme hizmet yordamı** (*interrupt service routine*, ISR) denir. ISR bitince işlemci, kesildiği komuttan sonraki komutla programa devam eder.

Tanımdaki en önemli nokta son cümledir. Kesilen program kesildiğini fark etmez. Yazmaçları, bayrakları ve belleği, kesme olmasaydı alacakları değerlerle devam eder. Bunu sağlamak hem işlemcinin hem de ISR'yi yazan programcının görevidir. İşlemci dönüş adresini ve bayrakları kendiliğinden saklar. ISR ise kullandığı yazmaçları saklayıp çıkarken geri yükler.

Günlük hayattan bir benzetme kesmenin adımlarını iyi anlatır. Kitap okurken telefon çalar. Okuyucu önce okuduğu sayfayı işaretler, sonra telefonu açar ve konuşur. Konuşma bitince işaretli sayfaya döner ve okumayı sürdürür. Sayfayı işaretlemek dönüş adresini saklamaktır; telefon konuşması ISR'dir; işaretli sayfaya dönmek IRET komutudur. Okuyucu sayfayı işaretlemeyi unutursa nereden devam edeceğini bilemez. İşlemci de dönüş adresini yığına yazmadan ISR'ye geçseydi aynı duruma düşerdi.



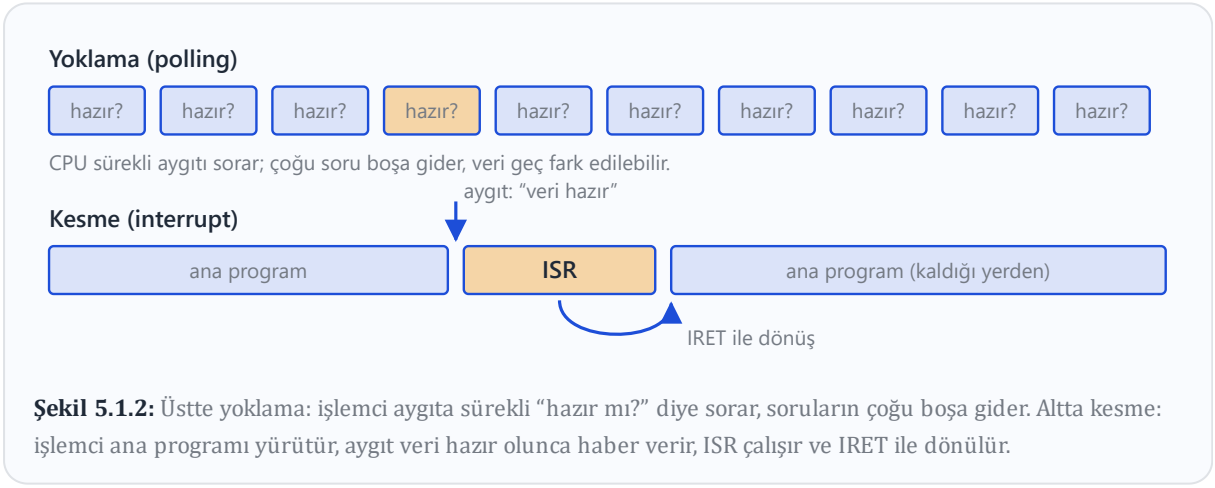
Şekil 5.1.1: Ana program bir istek ya da INT komutuyla kesilir, ISR çalışır ve IRET ile kalınan yere dönülür. Dönüş adresi bu süre boyunca yığında durur.

Şekil 5.1.1 bu akışı zaman ekseninde gösterir. Ana program bir noktada durur, denetim ISR'ye geçer, ISR'nin son komutu IRET denetimi ana programa geri verir. Kesmeyi başlatan şey iki türlü olabilir. Program bir INT komutu yürütür ya da işlemcinin dışından bir istek gelir. İki durumda da işlemcinin sonrasında yaptıkları aynıdır.

Kesmelerin ikinci, daha gündelik bir kullanımı da vardır. 8086 tabanlı bir PC'de ekrana karakter yazmak, klavyeden tuş okumak, diske erişmek ya da programı bitirmek için hazır yordamlar bulunur. Bunlar anakarttaki ROM'da duran BIOS'un ve işletim sistemi DOS'un yordamlarıdır. Program bu yordamları adreslerini bilmeden, bir INT komutuyla çağırır. Örneğin ekrana karakter yazmak için video donanımını süren onlarca komut yazmak gerekmez; `INT 10h` bu işi yapar. Bu yüzden yazılım kesmeleri, programlamayı kolaylaştıran hazır hizmetler olarak da görülür. Ekran, klavye, disk sürücüsü ve bellek yönetimi gibi donanımlarla çalışan kesme işlevlerinin ayrıntısı 5.2 Kesme Fonksiyonları notunun konusudur. Bu not kesmenin kendisini, yani işlemcinin bir kesmeyi nasıl kabul edip nasıl geri döndüğünü anlatır.

5.1.1.2 Yoklama mı, kesme mi?

İşlemcinin bir aygıtın durumunu öğrenmesinin iki yolu vardır. Birincisi aygıta sürekli sormaktır: "Veri hazır mı?" Bu yönteme **yoklama** (*polling*) denir. İşlemci bir döngü içinde aygıtın durum portunu okur, hazır bitine bakar, hazır değilse yeniden okur. İkincisi aygıtın kendisinin haber vermesidir. Aygıt verisi hazır olduğunda işlemcinin bir pinini sürer; işlemci o anki komutu bitirir ve ISR'ye geçer. Bu da kesme yöntemidir.



Şekil 5.1.2: Üstte yoklama: işlemci aygıta sürekli "hazır mı?" diye sorar, soruların çoğu boşa gider. Altta kesme: işlemci ana programı yürütür, aygıt veri hazır olunca haber verir, ISR çalışır ve IRET ile dönülür.

Şekil 5.1.2 iki yöntemi aynı zaman ekseninde karşılaştırır. Yoklamanın bedelini bir sayıyla görmek mümkündür. Klavyenin durumunu soran en kısa yoklama döngüsü üç komuttan oluşur: portu okuyan bir `IN`, hazır bitini sınayan bir `TEST` ve hazır değilse başa dönen bir koşullu atlama.

```
; yoklama döngüsü (port numarası ve bit örnek)
bekle: IN   AL, 64h      ; durum portunu oku (10 tur)
       TEST AL, 01h      ; veri hazır biti (4 tur)
       JZ   bekle         ; hazır değilse yeniden sor (16 tur)
```

Intel'in 8086 kılavuzuna göre sabit portlu `IN` 10, yazmaç ile sabiti sınayan `TEST` 4, atlayan bir `JZ` 16 saat turu sürer. Döngünün bir turu 30 saat turudur; 5 MHz'lik bir 8086'da bir saat turu 200 ns olduğundan döngü 6 µs'de bir döner. İşlemci bir saniyede aygıta yaklaşık 166.000 kez sorar. Hızlı yazan bir kullanıcı saniyede on tuşa basar. Bu durumda 166.000 sorudan yalnız on tanesi “evet” cevabı alır, geri kalanı boşa gider. Daha kötüsü, işlemci bu süre içinde başka bir iş yapamaz.

Yoklamayı daha seyrek yapmak işlemciye zaman kazandırır, ama bu kez olay geç fark edilir. Program her 10 ms'de bir sorarsa bir tuşa basıldıktan sonra cevap 10 ms'ye kadar gecikebilir. Aralık uzadıkça kaybolan veri riski de artar. Örneğin seri porttan 9600 bit/s hızla gelen karakterler arasında yaklaşık 1 ms vardır. Aygıtın tek karakterlik bir tamponu varsa, 1 ms içinde okunmayan karakterin üzerine bir sonraki yazılır.

Kesmede bu ikilem ortadan kalkar. İşlemci olay olmadıkça ana programı yürütür; olay olduğu anda, en geç o anki komut bitince ISR'ye geçer. Bedeli ise düzenektir. Her olay için bir yordam yazılır, yordamın adresi bir tabloya konur, dönüş adresi için yığın kullanılır ve donanımda isteği işlemciye ileten hatlar gerekir. Tablo 5.1.1 iki yöntemi karşılaştırır.

Ölçüt	Yoklama	Kesme
Donanım	basit; ek hat gerekmez	istek hattı, gerekirse kesme denetleyicisi
İşlemci zamanı	boşa harcanır; sorguların çoğu “hayır” döner	yalnız olay olduğunda harcanır
Tepki süresi	sorgu aralığına bağlı; olay geç algılanabilir	kısa; en geç yürüyen komut bitince
Yazılım düzeni	tek bir döngü	vektör tablosu, yığın ve ISR
Uygun olduğu yer	tek işli, basit sistemler; çok hızlı ve sürekli veri	seyrek ve zamanı belirsiz olaylar; çok görevli sistemler

Tablo 5.1.1: Yoklama ile kesmenin karşılaştırılması.

Yoklama her zaman kötü değildir. Mikrodenetleyicili küçük bir devre yalnız bir düğmeyi bekliyorsa yoklama en sade çözümdür. Çok hızlı bir aygıt her birkaç mikrosaniyede veri üretiyorsa, her veri için kesme düzeneğinin yüküne katlanmak yerine sıkı bir yoklama döngüsü daha hızlı olabilir. Genel amaçlı bir bilgisayarda ise klavye, zamanlayıcı, disk ve seri port gibi birbirinden bağımsız ve zamanı önceden bilinmeyen olaylar vardır. Bunların hepsini tek bir döngüde yoklamak hem işlemciyi meşgul eder hem de tepkiyi yavaşlatır. PC mimarisi bu yüzden kesmelere dayanır.

Sık yapılan hata

"Kesmede işlemci aygıtı hiç beklemez" sonucunu "kesme her zaman yoklamadan hızlıdır" biçiminde genellemek. Kesme düzeneğinin kendisi de zaman alır: 8086'da bir donanım kesmesinin kabulü 61, IRET 24 saat turudur. Saniyede binlerce kez gelen bir olayda bu ek yük yoklamanın maliyetini aşabilir. Kesmenin üstünlüğü, seyrek olaylarda işlemcinin boşa beklememesidir.

5.1.1.3 Kesme türleri ve tip numaraları

8086 en çok 256 farklı kesmeyi ayırt eder. Her kesmenin 0 ile 255 (00h–FFh) arasında bir **tip numarası** vardır. İşlemci kesmeyi kabul ettiğinde ilk iş bu numarayı belirler, sonra numaraya göre hangi yordama gideceğini bir tablodan okur. Kesmeler, tip numarasının nereden geldiğine ve kesmeyi neyin başlattığına göre üç gruba ayrılır.

Tanım 5.1.2

Yazılım kesmesi (*software interrupt*), program içindeki `INT n` komutuyla bilerek çağrılan kesmedir; tip numarası n komutun içindedir. **Donanım kesmesi** (*hardware interrupt*), bir dış aygıtın işlemcinin INTR ya da NMI pinini sürmesiyle başlayan kesmedir. **İç kesme** ya da **istisna** (*exception*), işlemcinin bir komutu yürütürken kendisinin ürettiği kesmedir; sıfıra bölme bunun en bilinen örneğidir.

Yazılım kesmeleri. BIOS ve DOS hizmetleri bu yolla kullanılır: `INT 10h` ekran, `INT 16h` klavye, `INT 21h` DOS hizmetleridir. Bu grupta iki özel komut daha vardır. `INT 3` hata ayıklayıcıların kullandığı kesme noktası komutudur ve tek baytla kodlanır. `INTO` ise taşma bayrağı OF = 1 olduğunda INT 4'ü çağırır, OF = 0 ise hiçbir şey yapmaz. Bu iki komut, hata ayıklama ve hata yakalama amacıyla kullanıldıkları için sık sık istisnalarla birlikte anılır.

Donanım kesmeleri. 8086'nın kesme isteği için iki giriş pini vardır. **INTR** (*interrupt request*) maskelenebilir girişidir: IF = 0 iken işlemci bu pini dikkate almaz. Klavye, zamanlayıcı, disk ve seri port istekleri bu pinden gelir. **NMI** (*non-maskable interrupt*) ise maskelenemez girişidir; IF ne olursa olsun kabul edilir. Bellek eşlik hatası ya da güç kaybı gibi, programın kapatıp geçiştiremeyeceği acil olaylar için ayrılmıştır.

İç kesmeler. İşlemci bazı durumları kendisi kesmeye dönüştürür. DIV ya da IDIV sıfıra böldüğünde veya bölüm hedef yazmaca sığmadığında INT 0 üretilir. Tuzak bayrağı TF = 1 iken her komuttan sonra INT 1 üretilir; bu, hata ayıklayıcıların programı tek adım yürütmesini sağlar.

Tablo 5.1.2, ilk beş tip numarasının görevini verir. Bu beş tipi 8086'nın kendisi kullanır; numaraları sabittir ve programcı bunları başka bir iş için kullanmamalıdır.

Tip	Kaynak	Açıklama
00h	iç	Bölme hatası: DIV/IDIV sıfıra bölme ya da bölüm sığmıyor
01h	iç	Tek adım (TF = 1); hata ayıklayıcılar kullanır
02h	donanım	NMI pini (maskelenemez)
03h	yazılım	Kesme noktası; tek baytlık INT 3 (CCh)
04h	yazılım	INTO: OF = 1 ise taşma kesmesi
05h–FFh	—	Intel 05h–1Fh'yi ileriki işlemciler için ayırmıştır; PC'de BIOS, DOS ve kullanıcı programları kullanır

Tablo 5.1.2: 8086'nın ayrılmış kesme tipleri.

Intel, 05h–1Fh aralığını ileriki işlemcileri için ayırdığını belgelerinde açıkça yazmıştı. IBM PC'nin tasarımcıları ise bu uyarıya uymadı ve BIOS hizmetlerini 05h, 08h–1Fh aralığına yerleştirdi. Örneğin ekranı yazıcıya basan BIOS yordamı tip 05h'yi kullanır. 80186 ve 80286 ile birlikte tip 05h, BOUND komutunun sınır aşımı kesmesi oldu. Böylece aynı numara iki farklı anlam kazandı ve PC yazılımı yıllarca bu çakışmayla uğraştı. Bu öykü, ayrılmış numaraların neden ayrılmış kalması gerektiğini iyi gösterir.

5.1.1.4 Üç kaynağın karşılaştırması

Üç kesme türü birbirinden ilk bakışta çok farklı görünür: biri bir komut, biri bir pin, biri bir hata durumudur. Aralarındaki fark ise yalnız iki noktadadır: tip numarasının nereden geldiği ve kesmenin ne zaman oluştuğu. Tablo 5.1.3 bu farkları yan yana koyar.

	Yazılım (INT n)	Donanım (INTR / NMI)	İç (istisna)
Tetikleyen	programdaki komut	dış aygıt, pin	işlemcinin kendisi
Zamanlama	eş zamanlı: hep aynı yerde	eş zamansız: herhangi bir komuttan sonra	eş zamanlı: hatalı komutta
Tip numarası	komutun içinde	INTR: 8259A, INTA ile; NMI: hep 2	sabit: 0, 1
IF = 0 iken	yine çalışır	INTR bekler, NMI çalışır	yine çalışır
Tipik kullanım	BIOS/DOS hizmeti, INT 3, INTO	klavye, zamanlayıcı, disk	bölme hatası, tek adım

Tablo 5.1.3: Yazılım, donanım ve iç kesmelerin karşılaştırılması.

Eş zamanlı ve **eş zamansız** sözcükleri burada program akışına göre kullanılır. Bir yazılım kesmesi programın aynı noktasında, her çalıştırmada aynı biçimde oluşur. Bir istisna da öyledir: aynı veriyle aynı DIV komutu her seferinde aynı hatayı verir. Donanım kesmesi ise programın hangi komutunda geleceği bilinemeyen bir olaydır. Kullanıcı tuşa bir kez döngünün başında, bir kez ortasında basar. Bu yüzden donanım kesmesinin ISR'si, programın herhangi bir noktasında çalışabilecekmiş gibi yazılmalıdır. ISR'lerin yazmaçları saklama zorunluluğu buradan gelir.

Tablonun altında önemli bir ortak nokta yatar. Tip numarası bir kez belirlendikten sonra üç tür de aynı adımları izler: FLAGS, CS ve IP yığına itilir, IF ve TF sıfırlanır, CS:IP vektör tablosundaki değerle yüklenir. Yani işlemcinin içinde tek bir kesme düzeneği vardır; kaynaklar yalnız bu düzeneğe hangi numarayla ve hangi anda girildiğini belirler.

Sık yapılan hata

`CLI` komutunun bütün kesmeleri kapattığını sanmak. CLI yalnız IF'yi sıfırlar ve yalnız INTR girişini etkiler. `INT 21h` gibi yazılım kesmeleri, sıfıra bölme gibi istisnalar ve NMI, IF = 0 iken de çalışır. Kritik bir kod parçasını "kesmelerden korumak" için CLI kullanıldığında NMI'ın yine gelebileceği unutulmamalıdır.

5.1.2 Kesme vektör tablosu

5.1.2.1 Tablonun yapısı

İşlemci bir kesmenin tip numarasını öğrendikten sonra o kesmeye ait ISR'nin nerede olduğunu bulmalıdır. 8086'da bu bilgi belleğin en başındaki bir tabloda durur. Tablonun yeri sabittir; donanım her kesmede aynı adrese bakar.

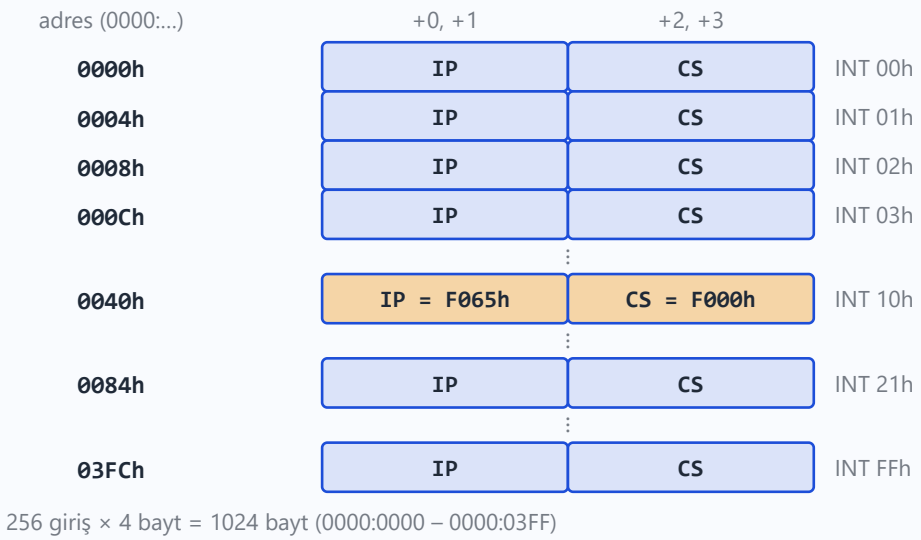
Tanım 5.1.3

Kesme vektörü (*interrupt vector*), bir kesme tipinin hizmet yordamının 4 baytlık uzak adresidir: 2 bayt IP (bağıl konum) ve 2 bayt CS (kesim). **Kesme vektör tablosu** (*interrupt vector table*, IVT), 256 kesme tipinin vektörlerini sırayla tutan ve fiziksel 00000h–003FFh adreslerini kaplayan 1 KB'lık tablodur.

Tablonun boyu kolayca hesaplanır: 256 tip × 4 bayt = 1024 bayt = 1 KB. Tip 0'ın vektörü 00000h'de, tip 1'inki 00004h'te, tip 2'ninki 00008h'de başlar ve böyle sürer. Son tip FFh'nin vektörü 003FCh–003FFh baytlarındadır. Tip n'nin vektörünün başladığı adres bu yüzden basit bir çarpımdır:

$$\text{vektör adresi} = n \times 4 \quad \text{(kesim 0000h içinde)}$$

Vektörün dört baytının dizilişi de sabittir. Şekil 5.1.3 tabloyu gösterir. Düşük adresli sözcük (n × 4 ve n × 4 + 1) IP değerini tutar. Yüksek adresli sözcük (n × 4 + 2 ve n × 4 + 3) ise CS değeridir. İki sözcük de 8086'nın bütün sözcükleri gibi *little-endian* düzende yazılır: önce düşük bayt, sonra yüksek bayt.



Şekil 5.1.3: Kesme vektör tablosu. Her satır bir kesme tipinin 4 baytlık vektörüdür: önce IP, sonra CS. INT 10h satırındaki F000:F065 değeri, orijinal IBM PC BIOS'undaki video yordamının adresidir ve örnek olarak verilmiştir.

Tablonun neden belleğin en başında olduğu sorusunun cevabı 8086'nın tasarımındadır. Donanımın tabloyu bulmak için hiçbir yazmaca bakması gerekmez; adres yalnız tip numarasından üretilir. n × 4 işlemi devrede iki bitlik bir kaydırmadır. Tip numarasının 8 biti adres hatlarının A2–A9 bitlerine konur; A0, A1 ve A10–A19 sıfırdır. Bu yüzden vektör okuma ek bir toplayıcı ya da yazmaç gerektirmez. Bedeli, belleğin ilk 1 KB'ının her zaman RAM olması zorunluluğudur. Programlar tabloyu değiştirebilmelidir; ROM'da olsaydı hiçbir program kendi ISR'sini kuramazdı.

Tablo, belleğin sıradan bir bölgesidir. Özel bir koruması yoktur. Bir kesim yazmacını 0000h yapan her program tablonun herhangi bir baytını okuyup yazabilir. Bu esneklik, programların kendi ISR'lerini kurabilmesini sağlar; aynı zamanda hatalı bir programın tabloyu bozarak bütün sistemi çökertebilmesi anlamına gelir. Korumalı kipteki sonraki işlemciler bu sorunu tabloyu korumalı bir yapıya taşıyarak çözer. 8086'nın gerçek kipinde ise tablonun güvenliği tamamen programcıya kalmıştır.

Sık yapılan hata

Vektörün dört baytını “CS önce, IP sonra” sırasıyla düşünmek. CS:IP yazılışı kesim önce yazıldığı için bu yanılgıyı doğurur. Bellekte ise sıra terstir: n × 4 adresinde IP'nin düşük baytı, n × 4 + 2 adresinde CS'nin düşük baytı vardır.

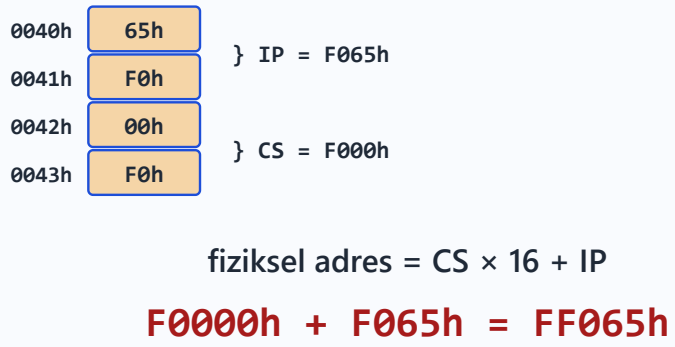
5.1.2.2 Vektör adresi adım adım

Kuralın uygulanışını bir örnek üzerinden izlemek yararlıdır. Program `INT 10h` komutunu yürütüyor olsun. İşlemcinin vektörü bulup ISR'ye geçmesi dört adımda gerçekleşir.

Örnek 5.1.1

INT 10h için vektörün adresini, ISR'nin CS:IP değerini ve fiziksel adresini bulalım. Tablonun 0000:0040h adresinden başlayan dört baytı sırasıyla `65 F0 00 F0` olsun.

1. **Tip numarası.** Komutun ikinci baytı tip numarasıdır: n = 10h = 16.
2. **Vektör adresi.** n × 4 = 10h × 4 = 40h (onlu: 16 × 4 = 64). Vektör 0000:0040h–0000:0043h baytlarındadır. İkili sayıda 0001 0000b iki bit sola kaydırılınca 0100 0000b = 40h olur.
3. **Baytları okuma.** 0040h'te 65h, 0041h'te F0h vardır; düşük sözcük IP = F065h'dir. 0042h'te 00h, 0043h'te F0h vardır; yüksek sözcük CS = F000h'dir.
4. **Fiziksel adres.** CS × 16 + IP = F0000h + F065h = FF065h. ISR, belleğin en üst 64 KB'ındaki BIOS ROM'unun içinde başlar.



Şekil 5.1.4: Üstte 0040h–0043h baytları: düşük sözcük IP = F065h, yüksek sözcük CS = F000h. Altta fiziksel adres hesabı: F0000h + F065h = FF065h.

Örnekte okunan baytlar Şekil 5.1.4'te görülür. İşlemci bu iki sözcüğü iki ayrı bellek okumasıyla alır. 8086 çift adresteki bir sözcüğü tek yol döngüsüyle okur. n × 4 ve n × 4 + 2 adreslerinin ikisi de çift olduğundan her sözcük tek çevrimde gelir. Vektör tablosunun 4 baytlık hizalanması bu açıdan da yerindedir.

Dördüncü adımdaki fiziksel adres hesabı, 8086'nın bütün bellek erişimlerinde kullandığı kesimleme kuralıdır: 16 bitlik kesim değeri dört bit sola kaydırılır (16 ile çarpılır) ve bağıl konum eklenir. CS = F000h kesimi F0000h'de başlar. IP = F065h bu kesimin içinde bir konumdur. Sonuç 20 bitlik FF065h adresidir.

Burada sıranın önemli bir ayrıntısı vardır. İşlemci vektörü okuyup CS:IP'yi yüklemeden önce dönüş bilgisini yığına yazar. Yani ISR'nin ilk komutu FF065h'den getirilirken eski CS, IP ve FLAGS çoktan yığındadır. Bu sıra bozulsaydı, yani CS:IP önce yüklenseydi, dönüş adresi kaybolurdu. Adımların tam sırası Tablo 5.1.5'tedir.

5.1.2.3 Tablodan yordama, yordamdan tabloya

Vektör tablosu üzerinde iki yönlü hesap yapılır. Birinci yönde tip numarasından vektörün adresine gidilir; bu, n × 4 çarpımıdır. İkinci yönde tablonun bir adresinden hangi tipin vektörü olduğu bulunur; bu da adresi 4'e bölmektir. İkinci yön, bellek dökümünde bir bayt dizisine bakıp ne olduğunu anlamak için gerekir.

Örnek 5.1.2

Bir bellek dökümünde 0000:0058h adresinden başlayarak `2E E8 00 F0` baytları görülüyor. a) Hangi kesmenin vektörüdür? b) ISR'nin CS:IP değeri ve fiziksel adresi nedir? c) Aynı tablodaki tip 1Ch'nin vektörü hangi adreslerdedir?

1. a) 58h / 4 = 16h. Adres 4'ün katı olduğu için bir vektörün başıdır: tip 16h, yani BIOS klavye hizmeti INT 16h.
2. b) Düşük sözcük: 2Eh, E8h → IP = E82Eh. Yüksek sözcük: 00h, F0h → CS = F000h. Fiziksel adres F0000h + E82Eh = FE82Eh.
3. c) 1Ch × 4 = 70h. IP 00070h–00071h, CS 00072h–00073h baytlarındadır.

Adres 4'e tam bölünmeseydi, örneğin 005Ah, bu bir vektörün başı değil, tip 16h vektörünün CS yarısı olurdu (5Ah = 4 × 16h + 2).

Hesap bütün tipler için aynıdır: IP n × 4 ve n × 4 + 1'de, CS n × 4 + 2 ve n × 4 + 3'tedir. Tip 00h için adres 0000h, tip FFh için 03FCh çıkar; aradaki bütün adresler 4'ün katıdır.

Sık yapılan hata

Tip numarasını doğrudan adres sanmak (INT 21h → 00021h) ya da 4 yerine 2 ile çarpmak (INT 21h → 00042h). Doğrusu 21h × 4 = 84h'dir. Onaltılık çarpımda yanılmamak için sayıyı önce onluya çevirmek de bir yoldur: 21h = 33, 33 × 4 = 132 = 84h.

5.1.2.4 PC'de tip numaralarının paylaşımı

256 tip numarasının ilk beşi 8086'ya aittir. Geri kalanı, işlemciyi kullanan sistemin tasarımcılarına bırakılmıştır. IBM PC ve DOS'ta numaralar kabaca şöyle paylaşılmıştır (Tablo 5.1.4).

Tip	Vektör adresi	Görev
00h–04h	000h–013h	8086'nın kendi tipleri (bölme hatası, tek adım, NMI, kesme noktası, taşma)
05h	014h	BIOS: ekranı yazıcıya bas
08h–0Fh	020h–03Fh	donanım kesmeleri IRQ0–IRQ7 (08h zamanlayıcı, 09h klavye)
10h	040h	BIOS: video / ekran hizmetleri
13h	04Ch	BIOS: disk hizmetleri

Tip	Vektör adresi	Görev
16h	058h	BIOS: klavye hizmetleri
1Ah	068h	BIOS: saat / tarih
20h	080h	DOS: programı sonlandır
21h	084h	DOS: genel hizmetler (AH ile seçilir)
33h	0CCh	fare sürücüsü
60h–67h	180h–19Fh	kullanıcı programlarına ayrılmış
70h–77h	1C0h–1DFh	PC/AT'de ikinci 8259A'nın IRQ8–IRQ15 kesmeleri

Tablo 5.1.4: IBM PC ve DOS'ta sık kullanılan kesme tipleri.

Bu tablodan çıkan pratik sonuç şudur: kendi ISR'sini kuracak bir program, kullanılmayan bir tip seçmelidir. 60h–67h aralığı bu amaçla ayrılmıştır. Notun kendi ISR'sini kuran örnekleri bu yüzden tip 60h, 61h ve 90h'yi kullanır. Kullanılan bir tipin, örneğin INT 21h'nin vektörünü değiştiren bir program, DOS'un bütün hizmetlerini kendi yordamına yönlendirmiş olur. Bu bazen bilerek yapılır. Örneğin bir program, DOS'un bölme hatası yordamı yerine kendi yordamını çalıştırmak için tip 0'ın vektörünü değiştirir. O zaman program biterken eski vektörü geri yüklemek zorunludur.

5.1.3 INT ve IRET

5.1.3.1 INT n'nin adımları

`INT n` komutu iki bayttır: işlem kodu CDh ve tip numarası n. Örneğin `INT 21h` bellekte `CD 21` olarak durur. İşlemci bu komutu yürütürken programcının göremediği bir dizi iş yapar. Bu işler bir yordam çağrısına benzer, ama ondan daha fazlasını kapsar. Tablo 5.1.5 adımları sırasıyla verir.

Adım	İş	Etkisi	Yol döngüsü
1	FLAGS yığına itilir	SP ← SP − 2, [SS:SP] ← FLAGS	bellek yazma
2	IF ve TF sıfırlanır	IF = 0, TF = 0	—
3	CS yığına itilir	SP ← SP − 2, [SS:SP] ← CS	bellek yazma
4	IP yığına itilir (INT'ten sonraki komutun adresi)	SP ← SP − 2, [SS:SP] ← IP	bellek yazma
5	vektör okunur	IP ← [0000:n×4], CS ← [0000:n×4+2]	iki bellek okuma
6	ISR çalışır, sonunda IRET	IP, CS, FLAGS sırasıyla yığından alınır	üç bellek okuma

Tablo 5.1.5: INT n komutunun adımları ve her adımın yığına, yazmaçlara ve yola etkisi.

Tablonun her satırının bir gerekçesi vardır. **Adım 1** bayrakları saklar. Kesilen programın son karşılaştırmasının sonucu, örneğin ZF, ISR boyunca değişebilir. Program bir `CMP` ile bir `JE` arasında kesilirse ve bayraklar saklanmazsa, JE yanlış yöne atlar. Bayrakların otomatik saklanması bu yüzden zorunludur. CALL komutu bayrakları saklamaz, çünkü bir yordamın ne zaman çağrılacağını program bilir.

Adım 2 IF ve TF'yi sıfırlar. IF = 0, ISR'nin daha ilk komutunda başka bir maskelenebilir kesmeyle kesilmesini önler. TF = 0 ise ISR'nin tek adım kesmesiyle adım adım durdurulmasını engeller. Bu sıfırlamanın FLAGS yığına itildikten *sonra* yapılması önemlidir. Yığındaki kopya eski IF ve TF değerlerini taşır. IRET bu kopyayı geri yüklediğinde IF ve TF de kesmeden önceki değerlerine döner.

Adım 3 ve 4 dönüş adresini saklar. ISR başka bir kesimde olabileceği için yalnız IP yetmez, CS de saklanır. Yığına yazılan IP, INT komutunun kendi adresi değil, INT'ten *sonraki* komutun adresidir. İşlemci INT'i getirirken IP'yi zaten iki bayt ilerletmiştir.

Adım 5 vektörü okur ve CS:IP'ye yükler. Bu andan sonra işlemcinin getireceği ilk komut ISR'nin ilk komutudur. BIU'nun önden getirme kuyruğu da boşaltılır; kuyruktaki baytlar INT'ten sonraki komutlara aitti ve artık geçersizdir.

Yığına toplam 6 bayt, yani 3 sözcük yazılır. INT n'nin tamamı 8086'da 51 saat turu sürer. Bu turların önemli bir kısmı beş bellek erişimine harcanır: yığına üç yazma ve vektör tablosundan iki okuma. Kuyruğun boşaltılıp ISR'nin ilk baytlarının yeniden getirilmesi de ayrıca zaman alır.

Tanım 5.1.4

IRET (*interrupt return*), kesme hizmet yordamından dönüş komutudur. Yığından sırasıyla IP'yi, CS'yi ve FLAGS'i alır; her alışta SP 2 artar. Tek bayttır (CFh) ve 8086'da 24 saat turu sürer.

IRET, INT'in yığına yazdıklarını ters sırada geri okur. Yığın son giren ilk çıkar düzeninde çalıştığı için bu doğaldır: en son itilen IP ilk alınır, en önce itilen FLAGS en son alınır. FLAGS'in geri gelmesiyle IF ve TF de eski değerlerini alır. Böylece kesmeden önce IF = 1 olan bir program, IRET'ten sonra yine IF = 1 ile devam eder.

Tablo 5.1.6 bu notta geçen kesme komutlarının kodlamasını ve süresini toplu olarak verir.

Komut	Makine kodu	Uzunluk	Saat turu
`INT n`	CD n	2 bayt	51
`INT 3`	CC	1 bayt	52
`INTO`	CE	1 bayt	53 (OF = 1) / 4 (OF = 0)
`IRET`	CF	1 bayt	24
`CLI` / `STI`	FA / FB	1 bayt	2

Tablo 5.1.6: 8086 kesme komutlarının makine kodu, uzunluğu ve saat turu (8086 kılavuzu).

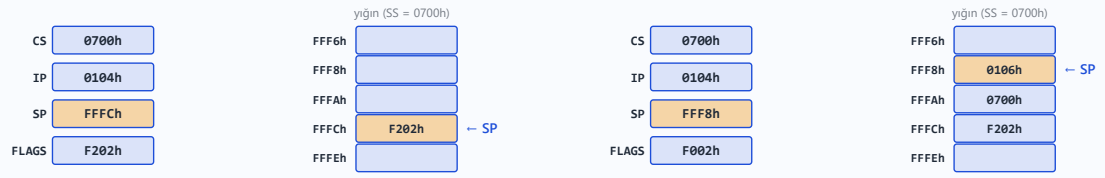
5.1.3.2 Yığın adım adım: INT 10h

Adımların yığına etkisini somut sayılarla izleyelim. Bir .COM programında `INT 10h` komutu 0700:0104 adresinde olsun. emu8086'nın .COM yüklemesinde CS = DS = SS = 0700h ve SP = FFFEh'dir. Başlangıçta yalnız IF = 1'dir; 8086'nın FLAGS yazmacında bit 1 ve bit 12–15 her zaman 1 okunduğu için FLAGS = F202h'dir. INT 10h'nin vektörü Örnek 5.1.1'deki gibi F000:F065 olsun.

Adım	İş	SP	[FFF8h]	[FFFAh]	[FFFCh]	CS:IP	FLAGS
0	başlangıç	FFFEh	—	—	—	0700:0104	F202h
1	FLAGS itilir	**FFFCh**	—	—	**F202h**	0700:0104	F202h
2	IF = TF = 0	FFFCh	—	—	F202h	0700:0104	**F002h**
3	CS itilir	**FFFAh**	—	**0700h**	F202h	0700:0104	F002h
4	IP itilir	**FFF8h**	**0106h**	0700h	F202h	0700:0104	F002h
5	CS:IP ← vektör	FFF8h	0106h	0700h	F202h	**F000:F065**	F002h
6	IRET: IP, CS alınır	**FFFCh**	0106h	0700h	F202h	**0700:0106**	F002h
7	IRET: FLAGS alınır	**FFFEh**	0106h	0700h	F202h	0700:0106	**F202h**

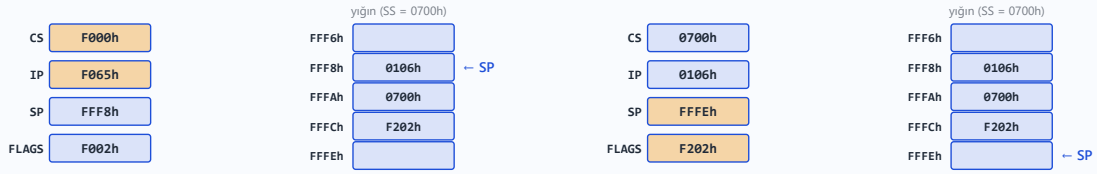
Tablo 5.1.7: 0700:0104'teki INT 10h ve ISR sonundaki IRET sırasında yığın ve yazmaçlar (SS = 0700h).

Tablo 5.1.7'nin satırları şöyle okunur. **Adım 1'de** SP önce 2 azalır ve FFFCh olur, FLAGS'in o anki değeri F202h bu adrese yazılır. **Adım 2'de** yazmaçtaki IF ve TF sıfırlanır; F202h'deki bit 9 (IF) silinince FLAGS = F002h olur. Yığındaki kopya ise F202h olarak kalır. **Adım 3'te** SP = FFFAh olur ve CS = 0700h yazılır. **Adım 4'te** SP = FFF8h olur ve dönüş adresi yazılır. INT 10h iki bayt olduğundan dönüş adresi 0104h + 2 = 0106h'dir. **Adım 5'te** CS:IP vektörden yüklenir ve ISR çalışmaya başlar. Bu örnekte ISR ekrana bir karakter yazar.



Şekil 5.1.5: Solda FLAGS itildikten sonra (SP = FFFCh), sağda dönüş adresi 0106h itildikten sonra (SP = FFF8h). Üç sözcük, yığının tepesine doğru FLAGS, CS, IP sırasıyla dizilir.

ISR bitip IRET çalışınca yığın ters yönde boşalır. **Adım 6'da** [FFF8h]'deki 0106h IP'ye, [FFFAh]'daki 0700h CS'ye alınır ve SP = FFFCh olur. **Adım 7'de** [FFFCh]'deki F202h FLAGS'e alınır ve SP = FFFEh'ye döner. IF yeniden 1'dir. Program 0700:0106'dan, yani INT 10h'den sonraki komuttan devam eder.



Şekil 5.1.6: Solda ISR çalışırken: CS:IP = F000:F065, FLAGS = F002h. Sağda IRET'ten sonra: CS:IP = 0700:0106, SP = FFFEh, FLAGS = F202h. Yığındaki üç sözcük silinmez; yalnız SP geri çekilir.

Son satırdaki bir ayrıntı dikkat ister: IRET'ten sonra yığındaki üç sözcük bellekte durmaya devam eder (Şekil 5.1.6). POP ve IRET belleği silmez, yalnız SP'yi artırır. SP'nin üstündeki bölge artık "boş" sayılır ve bir sonraki PUSH ya da kesme bu değerlerin üzerine yazar. Bir hata ayıklayıcıda yığın penceresine bakarken eski değerlerin görünmesi bu yüzdendir.

Örnek 5.1.3

Aynı program, kendi ISR'si INT 10h vektörüne konarak emu8086'da çalıştırıldığında yığına yazılan sözcükler okunabilir. Kod şudur:

```
org 100h
mov ah, 0Eh      ; 0100h, 2 bayt
mov al, 'A'      ; 0102h, 2 bayt
int 10h          ; 0104h, 2 bayt (CD 10)
ret              ; 0106h: dönüş adresi
```

INT 10h yürütüldükten hemen sonra SP = FFF8h'dir ve yığında [FFF8h] = 0106h, [FFFAh] = 0700h, [FFFCh] = F202h sözcükleri bulunur. IRET'ten sonra SP = FFFEh'dir. Değerler Tablo 5.1.7 ile birebir aynıdır.

Sık yapılan hata

Yığına itilen IP'yi INT komutunun kendi adresi sanmak (bu örnekte 0104h). İtilen değer bir sonraki komutun adresidir (0106h). Aksi hâlde IRET işlemciyi yeniden INT'e götürür ve program sonsuz bir döngüye girer.

5.1.3.3 INT/IRET ile CALL/RET

INT ile bir yordam çağrısı arasındaki benzerlik açıktır: ikisi de dönüş adresini yığına yazar ve başka bir yere atlar. Farklar ise ISR'lerin neden ayrı bir dönüş komutu gerektirdiğini gösterir. Tablo 5.1.8 iki çifti karşılaştırır.

	CALL / RET	INT n / IRET
Hedef adres	komutun içinde (etiket)	vektör tablosundan (n × 4)
Yığına yazılan	IP (yakın) ya da CS + IP (uzak)	FLAGS + CS + IP (6 bayt)
Bayraklar	değişmez	IF = 0, TF = 0 yapılır
Dönüş	RET: IP (ve CS) geri alınır	IRET: IP, CS, FLAGS geri alınır
Komut boyu	3 bayt (yakın CALL)	2 bayt (INT 3: 1 bayt)
Kim çağırır?	yalnız program	program, işlemci (istisna) ya da donanım

Tablo 5.1.8: CALL/RET ile INT n/IRET'in karşılaştırılması.

En önemli satır sonuncusudur. Bir yordamı yalnız program çağırır; program o noktada bayrakların durumunu bilir ve gerekiyorsa kendisi saklar. ISR ise bir donanım olayıyla, programın haberi olmadan çağrılabilir. Bu yüzden bayrakları saklama işi işlemciye verilmiştir.

Hedef adres satırı da önemli bir esneklik sağlar. Assembler CALL'un hedefini komutun içine yazar ve bu değer değişmez. INT'in hedefi ise tablodan çalışma anında okunur. Vektörü değiştirmek, kodu hiç değiştirmeden bütün INT çağrılarını başka bir yordama yönlendirir. DOS'un, sürücülerin ve yerleşik programların BIOS hizmetlerine kendi işlevlerini ekleyebilmesi bu sayededir. Kesme zincirleme (*interrupt chaining*) tekniği bu esnekliğe dayanır. Yeni yordam kendi işini yapar, sonra eski vektöre atlayarak eski yordamı da çalıştırır.

Komut boyu satırında INT'in 2 bayt olması da bir üstünlüktür. Uzak bir CALL 5 bayttır; BIOS'un bir yordamını uzak CALL ile çağırmak hem daha uzun kod hem de BIOS'un sürümüne bağlı sabit bir adres demektir. INT 10h ise her BIOS'ta aynı iki bayttır.

5.1.3.4 RET ile biten ISR: kod izleme

Tablo 5.1.8'deki farkın pratik sonucunu görmek için bilerek hatalı bir program izlenir. Program tip 61h'ye kendi ISR'sini kurar, INT 61h ile çağırır, ama ISR'yi IRET yerine yakın RET ile bitirir.

```
org 100h
mov ax, 0
mov es, ax
mov word ptr es:[61h*4], offset isr
mov word ptr es:[61h*4+2], cs
int 61h
hlt             ; SP = ?
isr:
ret             ; HATA!
```

Kod satır satır şöyle çalışır. `mov ax, 0` ve `mov es, ax` ES'yi 0000h yapar. Kesim yazmacına anlık değer doğrudan yüklenemediği için 0 önce AX'e konur. Bundan sonra `es:[…]` ile yazılan her adres vektör tablosunun içindedir. Üçüncü satır tip 61h'nin vektörünün IP yarısına, 61h × 4 = 184h adresi-

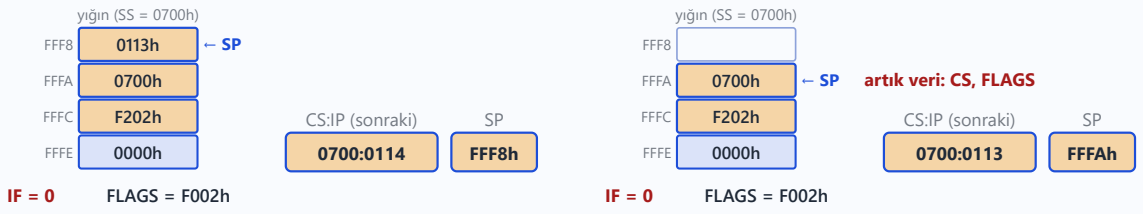
ne, `isr` etiketinin bağıl konumunu yazar. Assembler `61h*4` ifadesini kendisi hesaplar ve komuta 0184h olarak koyar. Dördüncü satır 186h adresine CS'nin değerini yazar. CS ile ISR aynı kesimdedir; böylece vektör 0700:0114'ü gösterir.

Programın baytlarını saymak, izleme tablosundaki IP değerlerini anlamayı kolaylaştırır. Komutların uzunlukları ve adresleri şöyledir: `mov ax, 0` 3 bayt (0100h–0102h), `mov es, ax` 2 bayt (0103h–0104h), ES önekli sözcük yazması 7 bayt (0105h–010Bh), CS yazması 5 bayt (010Ch–0110h), `int 61h` 2 bayt (0111h–0112h), `hlt` 1 bayt (0113h) ve `ret` 1 bayttır (0114h). Bu yüzden `isr` etiketi 0114h'dir ve INT 61h'nin dönüş adresi 0113h'dir.

Satır	Komut	CS:IP (sonraki)	SP	IF	Yığın / bellek
2	`mov ax, 0`	0700:0103	FFFEh	1	[FFFEh] = 0000h
3	`mov es, ax`	0700:0105	FFFEh	1	ES = 0000h
4	`mov … [61h*4], offset isr`	0700:010C	FFFEh	1	[0000:0184h] = 0114h
5	`mov … [61h*4+2], cs`	0700:0111	FFFEh	1	[0000:0186h] = 0700h
6	`int 61h`	**0700:0114**	**FFF8h**	**0**	[FFF8h] = 0113h, [FFFAh] = 0700h, [FFFCh] = F202h
9	`ret` (hatalı)	**0700:0113**	**FFFAh**	0	[FFFAh] = 0700h, [FFFCh] = F202h yığında kaldı
7	`hlt`	0700:0114	FFFAh	0	işlemci durdu, IF = 0

Tablo 5.1.9: RET ile biten ISR programının izlemesi (her satır yürütüldükten sonraki durum; SS = 0700h).

İzlemenin kritik iki satırı vardır (Tablo 5.1.9). **Satır 6'da** `int 61h` yığına FLAGS (F202h), CS (0700h) ve dönüş IP'si (0113h) olmak üzere 6 bayt yazar. SP = FFF8h ve IF = 0 olur, CS:IP vektörden 0700:0114 olarak yüklenir. **Satır 9'da** ISR'deki yakın `ret` yalnız yığının tepesindeki sözcüğü alır: IP = 0113h. CS değişmez; ISR ile program aynı kesimde olduğu için bu sefer tesadüfen doğru kesime dönülür. SP ise yalnız 2 artar ve FFFAh olur. Beklenen değer FFFEh'dir; yığın 4 bayt kaymıştır. FLAGS yığından geri alınmadığı için IF = 0 kalmıştır.



Şekil 5.1.7: Solda `int 61h` 'den sonra: yığında üç sözcük, SP = FFF8h, IF = 0. Sağda hatalı `ret` 'ten sonra: yalnız IP alınmış, CS ve FLAGS artık veri olarak yığında kalmış, SP = FFFAh, IF hâlâ 0.

Program `hlt`'e ulaştığında iki sorun birden vardır (Şekil 5.1.7). Birincisi, kesmeler kapalıdır. HLT işlemciyi bir kesme gelene kadar durdurur. IF = 0 olduğu için INTR'den gelen hiçbir istek işlemciyi uyandıramaz; yalnız NMI ya da RESET bunu yapabilir. Bu küçük programda işlemci sonsuza kadar bekler. İkincisi, yığın kaymıştır. HLT yerine bir `ret` olsaydı yığının tepesindeki 0700h'yi dönüş adresi sanır ve 0700:0700 adresine atlardı. Orada çalıştırılacak anlamlı bir kod yoktur.

Sorunu RETF ile çözmeye çalışmak da yetmez. Uzak dönüş komutu IP ve CS'yi alır, SP'yi 4 artırır. FLAGS'in 2 baytı yine yığında kalır ve IF eski hâline dönmez. Bu yüzden kural kesindir: ISR'ler her zaman IRET ile biter.

Sık yapılan hata

ISR'yi PROC/ENDP içinde yazıp sonuna alışkanlıkla `RET` koymak. Assembler buna itiraz etmez, çünkü RET geçerli bir komuttur. Hata ancak çalışma anında, çoğu zaman ISR'den çok sonra, yanlış bir dönüş adresi ya da bir daha gelmeyen kesmeler biçiminde ortaya çıkar. Belirtinin nedenden uzak olması bu hatayı bulmayı zorlaştırır.

5.1.4 İstisnalar

İstisnalar işlemcinin bir komutu yürütürken karşılaştığı özel durumları programa bildirme yoludur. 8086'nın iki gerçek istisnası vardır: bölme hatası (INT 0) ve tek adım (INT 1). Kesme noktası (INT 3) ve taşma kesmesi (INTO ile INT 4) aslında komutla çağrılan yazılım kesmeleridir. Ancak bu ikisi de hata ayıklama ve hata yakalama işinde kullanıldığı için istisnalarla birlikte ele alınır. Intel'in 8086 kılavuzu bu dört kesmeyi INT n ile birlikte “iç kesmeler” başlığı altında toplar.

5.1.4.1 Bölme hatası (INT 0)

DIV ve IDIV komutları bölümü sabit boyutlu bir yazmaca yazar. 8 bitlik bölmede bölünen AX'tedir, bölüm AL'ye, kalan AH'ye yazılır. 16 bitlik bölmede bölünen DX:AX'tedir, bölüm AX'e, kalan DX'e yazılır. Bölüm hedef yazmaca sığmazsa doğru bir sonuç üretmenin yolu yoktur. İşlemci bu durumda sonucu yazmaz ve tip 0 kesmesini üretir.

Tanım 5.1.5

Bölme hatası (*divide error*), DIV ya da IDIV komutunda bölen sıfır olduğunda ya da bölüm hedef yazmaca sığmadığında işlemcinin ürettiği tip 0 kesmesidir. Hedef 8 bitlik bölmede AL, 16 bitlik bölmede AX'tir.

Tanımdaki ikinci koşul çoğu zaman gözden kaçar. Sıfıra bölme olmadan da bölme hatası oluşabilir.

Örnek 5.1.4

AX = 1000 (03E8h) ve BL = 2 iken `DIV BL` çalışıyor. Ne olur? Aynı bölme nasıl hatasız yapılır?

1. 8 bitlik bölme: AX / BL = 1000 / 2 = 500. Bölüm AL'ye yazılacaktır.
2. AL en çok 255 tutar. 500 > 255 olduğu için bölüm sığmaz. Bölen sıfır olmadığı hâlde işlemci INT 0 üretir.
3. Çözüm 16 bitlik bölmedir: DX = 0, AX = 1000, BX = 2 ve `DIV BX`. Bölünen DX:AX = 0000:03E8h = 1000'dir. Bölüm 500 = 01F4h AX'e, kalan 0 DX'e yazılır. AX en çok 65.535 tutar ve 500 rahatça sığar.

Daha küçük bir sınır örneği: AX = 01FEh (510) / 2 = 255 = FFh sığar; AX = 0200h (512) / 2 = 256 sığmaz ve INT 0 üretir. Bu değerler emu8086'da çalıştırılarak doğrulanmıştır.

Kural kısaca şöyle söylenebilir: 8 bitlik DIV'de AH ≥ BL ise, yani bölünenin yüksek yarısı bölenden küçük değilse, bölüm 255'i aşar ve hata oluşur. Örnekte AH = 03h, BL = 02h'dir ve 03h ≥ 02h olduğu için hata kaçınılmazdır. 16 bitlik DIV için koşul DX ≥ bölendir. Bu sınama bölmeden önce bir CMP ile yapılabilir.

İşaretli bölme IDIV'de sınırlar işaretli aralıktır. 8086'da 8 bitlik IDIV bölümü −127 ile +127 arasında olmalıdır; 16 bitlik IDIV'de aralık −32.767 ile +32.767'dir. Bu sınır, ikiye tümleyen gösterimde mümkün olan en küçük değeri (−128 ya da −32.768) dışarıda bırakır. Örneğin AX = −128 (FF80h) ve BL = 1 iken `IDIV BL` bölümü −128 bulur ve 8086'da yine INT 0 üretir. Sonraki işlemciler bu en küçük değeri kabul eder.

Dönüş adresi. 8086'nın bölme hatasında yığına ittiği IP, DIV'den *sonraki* komutun adresidir. Yani IRET programı bölmeyi atlayarak sürdürür. Bölme hiç yapılmamış gibi devam edilir; hedef yazmaçlar eski değerlerini korur. 80286 ve sonraki işlemcilerde bu davranış değiştirilmiş ve hatalı komutun kendi adresi itilir hâle gelmiştir. Böylece hata yordamı nedeni ortadan kaldırıp komutu yeniden deneyebilir. 8086 için yazılmış bir hata yordamı ise bu imkâna sahip değildir; ancak sonucun geçersiz olduğunu bir yolla programa bildirebilir.

DOS ne yapar? Bir program başlarken INT 0'ın vektörü DOS'un kendi yordamını gösterir. Bu yordam ekrana “Divide overflow” yazar ve programı sonlandırır. Program kendi yordamını kurarak bu davranışı değiştirebilir. Bunun en sade biçimi aşağıdaki izlemede görülür.

Sık yapılan hata

Bölme hatasını yalnız “sıfıra bölme” olarak düşünmek ve yalnız bölenin sıfır olup olmadığını sınamak. Bölünen büyük, bölen küçük olduğunda bölüm hedefe sığmaz ve aynı kesme oluşur. Bölmeden önce hem bölenin sıfır olmadığını hem de bölümün sığacağını (8 bitte AH < bölen, 16 bitte DX < bölen) denetlemek INT 0'ı baştan önler.

5.1.4.2 Kendi INT 0 yordamımız: kod izleme

Aşağıdaki program tip 0'ın vektörünü kendi yordamına yönlendirir ve bilerek sıfıra böler. Yordam AX'e FFFFh yazarak programa "sonuç geçersiz" işareti bırakır.

```
org 100h
mov ax, 0
mov es, ax           ; ES = 0: IVT
mov word ptr es:[0*4], offset hata
mov word ptr es:[0*4+2], cs
mov ax, 100
mov bl, 0
div bl               ; INT 0!
mov dl, al           ; dönüş buraya
ret
hata:
mov ax, 0FFFFh       ; hata işareti
iret
```

İlk iki satır ES'yi 0000h yapar. `es:[0*4]` tip 0'ın vektörünün IP yarısıdır, yani tablonun en başıdır. Üçüncü satır buraya `hata` etiketinin bağıl konumunu, dördüncü satır 0000:0002h'ye CS'yi yazar. Programdan önce vektör, öykünücünün varsayılan yordamını gösteriyordu; bu izlemede eski değer F000:1000h olarak alınmıştır. Bu kısa örnekte eski vektör saklanmaz. Gerçek bir programda saklanır ve çıkıştan önce geri yüklenir.

Bağıl konumları sayalım: `mov ax, 0` 3, `mov es, ax` 2, iki vektör yazması 7 ve 5, `mov ax, 100` 3, `mov bl, 0` 2, `div bl` 2, `mov dl, al` 2 ve `ret` 1 bayttır. Böylece `div bl` 0116h'de, `mov dl, al` 0118h'de ve `hata` etiketi 011Bh'dedir.

Satır	Komut	AX	BL	DL	CS:IP (sonraki)	SP	IF	Vektör 0 / yığın
2	`mov ax, 0`	0000h	00h	00h	0700:0103	FFFEh	1	F000:1000
3	`mov es, ax`	0000h	00h	00h	0700:0105	FFFEh	1	F000:1000
4	`mov … [0*4], offset hata`	0000h	00h	00h	0700:010C	FFFEh	1	**F000:011B**
5	`mov … [0*4+2], cs`	0000h	00h	00h	0700:0111	FFFEh	1	**0700:011B**
6	`mov ax, 100`	**0064h**	00h	00h	0700:0114	FFFEh	1	0700:011B
7	`mov bl, 0`	0064h	00h	00h	0700:0116	FFFEh	1	0700:011B
8	`div bl` → INT 0	0064h	00h	00h	**0700:011B**	**FFF8h**	**0**	[FFF8h] = 0118h, [FFFAh] = 0700h, [FFFCh] = F202h

Satır	Komut	AX	BL	DL	CS:IP (sonraki)	SP	IF	Vektör 0 / yığın
12	`mov ax, 0FFFFh`	**FFFFh**	00h	00h	0700:011E	FFF8h	0	yığın aynı
13	`iret`	FFFFh	00h	00h	**0700:0118**	**FFFEh**	**1**	yığın boşaldı
9	`mov dl, al`	FFFFh	00h	**FFh**	0700:011A	FFFEh	1	—
10	`ret`	FFFFh	00h	FFh	**0700:0000**	**0000h**	1	DOS'a dönüş

Tablo 5.1.10: Kendi INT 0 yordamımız programının izlemesi (her satır yürütüldükten sonraki durum).

Tablo 5.1.10'un satırları üç evrede okunur. **Satır 4–5** vektörü iki parçada değiştirir. Satır 4'ten sonra vektörün yarısı yeni, yarısı eskidir: F000:011B. Bu ara durum hiçbir yordamı göstermez. Bu programda tip 0'ı yalnız programın kendi DIV'i tetikleyebildiği için ara durum zararsızdır. Donanım kesmelerinin vektörleri değiştirilirken ise bu ara durum ciddi bir tehlikedir.

Satır 8'de `div bl` sıfıra böler. İşlemci bölümü yazmaz ve tip 0 kesmesini başlatır. Yığına FLAGS (F202h), CS (0700h) ve dönüş IP'si itilir. İtilen IP, DIV'in adresi 0116h değil, DIV'den sonraki komutun adresi 0118h'dir. SP = FFF8h ve IF = 0 olur, CS:IP ← 0700:011B. AX değişmemiştir; bölünen 0064h olduğu gibi durur.



Şekil 5.1.8: `div bl` INT 0'ı tetikledikten sonra: tip 0'ın vektörü 0700:011Bh, yığında dönüş IP'si 0118h, CS ve FLAGS; IF = 0 ve CS:IP yordamın ilk komutunda.

Satır 12–13 yordamdır. `mov ax, 0FFFFh` sonuç yazmacına geçersiz bir işaret koyar. `iret` IP = 0118h, CS = 0700h ve FLAGS = F202h'yi geri alır; SP = FFFEh ve IF = 1 olur. DIV yeniden çalışmaz. **Satır 9'da** ana program `mov dl, al` ile AL'yi DL'ye alır ve DL = FFh'den hatayı anlar. Son satırdaki `ret`, .COM programlarında yığının tepesindeki 0000h'yi alır ve PSP'nin başındaki INT 20h ile DOS'a döner.

Bu programdaki işaretleme yönteminin bir zayıflığı vardır. FFh, gerçek bir bölmenin de sonucu olabilir: 255 / 1 = 255. Gerçek programlarda hata bilgisi bunun yerine ayrı bir değişkende ya da bir bayrakta, örneğin CF'de döndürülür. CF'yi değiştirmek için ISR'nin yığındaki FLAGS kopyasını değiştirmesi gerekir; çünkü IRET o kopyayı geri yükler.

5.1.4.3 INTO ve taşma kesmesi (INT 4)

İşaretli toplama ve çıkarmada sonuç işaretli aralığa sığmazsa OF = 1 olur. İşlemci bu durumda kendiliğinden bir kesme üretmez; OF'yi sınamak programın işidir. `INTO` (*interrupt on overflow*) bu sınamayı tek baytta yapar.

Tanım 5.1.6

INTO, OF = 1 ise tip 4 kesmesini çağıran, OF = 0 ise hiçbir şey yapmayan tek baytlık (CEh) komuttur. İşaretli bir toplama ya da çıkarmanın hemen ardından yazılır.

Örnek 5.1.5

AL = 64h, BL = 32h iken `ADD AL, BL` ve ardından `INTO` çalışıyor. Sonra aynı işlem BL = F6h ile tekrarlanıyor. Her iki durumda AL, OF, CF nedir ve INT 4 çağrılır mı?

1. 64h + 32h = 96h. İşaretsiz yorumda 100 + 50 = 150 ≤ 255; elde yok, CF = 0. İşaretli yorumda iki pozitif sayının toplamı +150'dir ve +127'yi aşar. 96h'nin en yüksek biti 1'dir; sonuç −106 gibi görünür. OF = 1 olur ve INTO, INT 4'ü çağırır. AL = 96h olarak kalır; INTO sonucu düzeltmez.
2. 64h + F6h = 15Ah; AL = 5Ah ve CF = 1. İşaretli yorumda +100 + (−10) = +90 doğrudur; farklı işaretli iki sayının toplamı taşamaz: OF = 0. INTO hiçbir şey yapmaz. CF = 1 olması INTO'yu etkilemez.

İki durum emu8086'da çalıştırıldığında da aynı sonuç alınır: birincide INT 4 yordamı çalışır, ikincide çalışmaz.

INTO'nun maliyeti küçüktür: OF = 0 iken yalnız 4 saat turu sürer. Bu yüzden işaretli aritmetik yapan bir program her önemli işlemden sonra bir INTO koyarak taşmaları merkezî bir yordamda yakalayabilir. Pratikte ise DOS, INT 4 için anlamlı bir yordam kurmaz. Varsayılan vektör çoğu zaman yalnız IRET'ten oluşan boş bir yordamı gösterir. INTO'yu kullanacak program kendi INT 4 yordamını kurmalıdır.

Sık yapılan hata

INTO'nun CF'ye baktığını sanmak. INTO yalnız OF'ye bakar; işaretsiz taşma (CF = 1) INTO'yu

tetiklemez. İşaretsiz sayılarla çalışan bir program taşmayı `JC` ile yakalar, INTO ile değil.

5.1.4.4 Tek adım (INT 1)

Bir programdaki hatayı bulmanın en etkili yolu onu komut komut yürütüp her komuttan sonra yazmaçlara bakmaktır. 8086 bu işi donanımla destekler. FLAGS yazmacının bit 8'i olan tuzak bayrağı TF = 1 iken işlemci her komuttan sonra tip 1 kesmesini üretir. INT 1 yordamı, yani hata ayıklayıcının bir parçası, yazmaçları ekrana yazar ve kullanıcının bir tuşa basmasını bekler. Sonra IRET ile programa döner, program bir komut daha yürütür ve aynı döngü sürer.

TF'yi doğrudan değiştiren bir komut yoktur. CLC/STC'nin, CLD/STD'nin ve CLI/STI'nin TF için karşılığı bulunmaz. TF yığın üzerinden değiştirilir: FLAGS yığına itilir, bir yazmaca alınır, bit 8 kurulur ve değer yeniden FLAGS'e yüklenir.

```
PUSHF
POP  AX
OR   AH, 01h      ; bit 8 = TF
PUSH AX
POPF              ; TF = 1
```

Başlangıçta FLAGS = F202h ise `PUSHF` ve `POP AX` sonrasında AX = F202h olur. AH'nin bit 0'ı FLAGS'in bit 8'idir; `OR AH, 01h` AX'i F302h yapar. `POPF` bu değeri FLAGS'e yükler ve TF = 1 olur. TF'yi sıfırlamak için aynı dizi `AND AH, 0FEh` ile yazılır.

Burada ince ama önemli bir nokta vardır. INT 1 yordamı da bir komut dizisidir. TF = 1 iken bu yordamın her komutu da INT 1 üretseydi, yordam kendini sonsuza kadar keserdi. Bunu önleyen, INT n adımlarının ikincisidir: işlemci kesmeye girerken FLAGS'i yığına itip TF'yi sıfırlar. INT 1 yordamı TF = 0 ile çalışır ve kendi komutları adım adım kesilmez. Yordam IRET ile döndüğünde yığındaki FLAGS kopyası TF = 1'i geri yükler ve izleme sürer.

Tek adım kesmesinin önceliği bütün kesmelerin en düşüğüdür (Tablo 5.1.12). Aynı komut sınırında başka bir istek de bekliyorsa önce o hizmet görür.

5.1.4.5 Kesme noktası (INT 3)

Tek adım, uzun bir programın her komutunda durmak için yavaştır. Çoğu zaman program belirli bir noktaya kadar tam hızda çalışsın, orada dursun istenir. Bu noktaya **kesme noktası** (*breakpoint*) denir. 8086'da kesme noktaları INT 3 ile kurulur.

INT 3'ün tek baytlık özel bir kodlaması vardır: CCh. Genel `INT n` biçimiyle yazılsaydı iki baytlık `CD 03` olurdu. Hata ayıklayıcı bir kesme noktasını dört adımda kurar ve kaldırır:

1. Durulacak komutun ilk baytını saklar ve yerine CCh yazar.

2. Program tam hızda çalışır. O adrese gelindiğinde CCh, yani INT 3 yürütülür. Yığına FLAGS, CS ve INT 3'ten sonraki adres itilir; hata ayıklayıcının INT 3 yordamı yazmaçları gösterir.
3. Hata ayıklayıcı saklanan baytı geri yazar ve yığındaki dönüş IP'sini 1 azaltır. Böylece dönüş adresi, CCh'nin bulunduğu, yani asıl komutun başladığı adres olur.
4. IRET ile asıl komut baştan, bozulmamış hâliyle yürütülür.

Tek baytlık kodlamanın nedeni birinci adımdadır. 8086'nın en kısa komutları tek bayttır: `INC AX`, `PUSH BX`, `CLC` gibi. İki baytlık `CD 03` böyle bir komutun yerine yazılsaydı, ikinci bayt bir sonraki komutun ilk baytını bozardı. Program o sonraki komuta bir atlamayla ulaşırsa, artık bozuk bir komut yürütülürdü. CCh ise her komutun ilk baytının yerine sığar ve yalnız o komutu etkiler. Dönüş IP'sinin 1 azaltılması da yine INT 3'ün tek bayt olmasına dayanır.

emu8086'nın “single step” düğmesi aynı işi öykünücünün içinde yapar. Gerçek bir 8086 sisteminde ise DOS'un DEBUG programı ve benzeri araçlar INT 1 ve INT 3 vektörlerine kendi yordamlarını kurarak çalışır.

> **Sık yapılan hata**
>
> Hata ayıklayıcı altında çalışan bir programın kodu belleğe döküldüğünde CCh baytlarını programın kendi komutları sanmak. Kesme noktası kurulu adreslerde bellekte geçici olarak CCh durur. Program kendi kodunu okuyor ya da sağlama toplamı hesaplıyorsa, hata ayıklayıcı altında farklı sonuç bulabilir.

5.1.5 Donanım kesmeleri

5.1.5.1 INTR ve NMI pinleri

Dış aygıtlar 8086'ya kesme isteğini iki pin üzerinden iletir. İki pinin davranışı, tetiklenme biçiminden tip numarasının belirlenmesine kadar farklıdır.

INTR (pin 18) maskelenebilir kesme isteği girişidir. Düzey tetiklemelidir: işlemci pini her komutun son saat turunda yoklar ve pin o anda 1 ise isteği görür. IF = 0 ise istek dikkate alınmaz; pin 1'de kalmaya devam ederse IF = 1 olduğu anda görülür. Tip numarası işlemcinin içinde sabit değildir. İşlemci isteği kabul edince INTĀ (pin 24, kesme onayı) çıkışından iki darbe üretir ve tip numarasını veri yolundan okur. Bu düzen, tek bir INTR pininin arkasına birçok aygıt bağlamayı mümkün kılar; her aygıta farklı bir tip numarası verilebilir.

NMI (pin 17) maskelenemez kesme girişidir. Yükselen kenarla tetiklenir: pinin 0'dan 1'e geçişi işlemcinin içindeki bir tutucuya kaydedilir ve istek kaybolmaz. IF'ye bakılmaz; CLI ile kapatılamaz. Tip numarası her zaman 2'dir, bu yüzden INTA döngüsü gerekmez ve vektör doğrudan 00008h'den okunur.

IBM PC'de NMI bellek eşlik hatasını bildirmek için kullanılır. Güç kaybını sezen devreler de NMI ile işlemciye, kaynak tamamen kesilmeden önce önemli verileri saklaması için birkaç milisaniyelik süre tanır.

	INTR	NMI
Pin	18	17
Tetiklenme	düzey (komut sonunda yoklanır)	yükselen kenar (içeride tutulur)
IF = 0 iken	bekletilir	kabul edilir
Tip numarası	INTA ile veri yolundan (PC'de 08h–0Fh)	sabit 2
Kullanım	klavye, zamanlayıcı, disk, seri port	eşlik hatası, güç kaybı

Tablo 5.1.11: INTR ile NMI'ın karşılaştırılması.

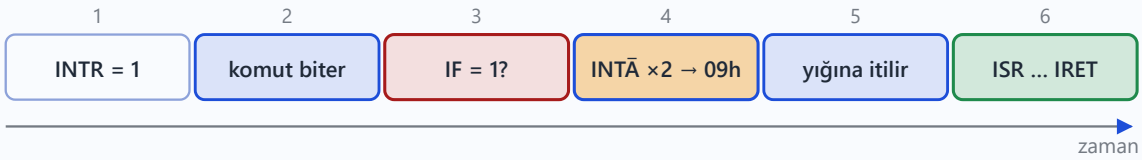
Düzey tetiklemenin bir sonucu vardır. Aygıt, işlemci isteği kabul edene kadar INTR'yi 1'de tutmalıdır. İstek çok kısa bir darbe olarak gelirse ve işlemci o anda IF = 0 ile çalışıyorsa, istek kaybolur. PC'de bu sorunu aygıtlarla işlemci arasındaki 8259A çözer. 8259A gelen istekleri kendi yazmacında saklar ve INTR'yi istek hizmet görene kadar tutar.

> **Sık yapılan hata**
>
> NMI'ın vektörünün de INTA ile okunduğunu düşünmek. INTA döngüsü yalnız INTR içindir. NMI'ın tipi sabittir (2) ve işlemci bunu dışarıya sormaz. Aynı biçimde INT 0, INT 1, INT 3 ve INT 4'ün tipleri de sabittir; dış donanımla ilgisi yoktur.

5.1.5.2 INTR kabul süreci

Bir tuşa basıldığı andan klavye ISR'sinin ilk komutunun yürütülmesine kadar geçen olaylar zincirini adım adım izleyelim. PC'de klavye denetleyicisi 8259A'nın IRQ1 girişine bağlıdır.

1. Klavye denetleyicisi IRQ1 hattını etkinleştirir. 8259A isteği kaydeder ve maske izin veriyorsa INTR pinini 1 yapar.
2. 8086 o anda yürüttüğü komutu bitirir. Kesmeler komut ortasında kabul edilmez; işlemci INTR'yi yalnız komut sınırlarında yoklar.
3. IF = 0 ise istek bekletilir; INTR 1'de kalır. IF = 1 ise istek kabul edilir.
4. İşlemci iki INTĀ darbesi üretir. Birinci darbe 8259A'ya “hazırlan” der. İkinci darbede 8259A tip numarasını veri yolunun D0–D7 hatlarına koyar; PC'de klavye için bu 09h'dir.
5. Bundan sonrası INT n ile aynıdır: FLAGS, CS, IP yığına itilir; IF = TF = 0 yapılır; CS:IP ← [n × 4], yani [0000:0024h].
6. ISR tuşu okuyup işini bitirince 8259A'ya EOI (kesme sonu) gönderir ve IRET ile döner.



Şekil 5.1.9: INTR isteğinden ISR'ye geçişin zaman çizelgesi: istek gelir, yürüyen komut biter, IF sınanır, iki INTA darbesiyle tip numarası okunur, dönüş bilgisi yığına itilir ve ISR IRET ile biter.

Şekil 5.1.9'daki dördüncü adım, donanım kesmesini yazılım kesmesinden ayıran tek adımdır. INTA döngüsü bir okuma yol döngüsüne benzer: işlemci adres yoluna anlamlı bir adres koymaz, RD̄ yerine INTĀ'yı etkinleştirir ve veri yolundaki baytı alır. İki darbe arasında maksimum kipte LOCK̄ sinyali de etkin tutulur; böylece başka bir yol yöneticisi bu sırada yola giremez. Tip numarası alındıktan sonra işlemci, sanki programda bir `INT 09h` komutu varmış gibi davranır.

İkinci adımdaki kural, yani kesmenin yalnız komut sınırında kabul edilmesi, ISR tasarımının temelidir. Bir komut yarım kalmaz. Örneğin `ADD [BX], AX` belleği okuyup yazarken araya kesme girmez; ISR belleği her zaman ya komuttan önceki ya da komuttan sonraki hâliyle görür. Ancak iki ayrı komutla yapılan bir iş, iki komut arasında kesilebilir. ISR'nin ve ana programın paylaştığı veriler için bu durum dikkate alınmalıdır.

5.1.5.3 Öncelik ve PC kesme haritası

Birden çok kesme isteği aynı komut sınırında bekliyorsa işlemci hangisine önce bakacağına sabit bir sırayla karar verir. Tablo 5.1.12 8086'nın öncelik sırasını verir.

Sıra	Kaynak	Gerekçe
1 (en yüksek)	bölme hatası, INT n, INTO	yürütülen komutun kendisinden doğar; o komutun parçasıdır
2	NMI	acil donanım olayı; maskelenemez
3	INTR	sıradan donanım isteği; IF ile bekletilebilir
4 (en düşük)	tek adım (TF)	hata ayıklama; beklemesi bir şey kaybettirmez

Tablo 5.1.12: 8086'nın kesme öncelik sırası.

Sıralamanın mantığı açıktır. Bir komutun kendi ürettiği kesme, o komutun yürütülmesinin parçasıdır ve önce ele alınır. Dış istekler arasında NMI, programın kapatamayacağı acil bir olay olduğu için INTR'nin önündedir. Tek adım ise insanın yazmaçlara bakması içindir; birkaç mikrosaniye beklemesi hiçbir şey kaybettirmez.

Bu öncelik, işlemcinin *kendi* girişleri arasındadır. INTR'nin arkasındaki sekiz aygıtın birbirine göre önceliğini ise 8259A belirler. PC'nin standart ayarında IRQ0 en yüksek, IRQ7 en düşük önceliklidir. Tablo 5.1.13 IBM PC'nin donanım kesmelerini verir. BIOS açılışta 8259A'yı IRQ0'a tip 08h verecek biçimde programlar; böylece IRQ n'nin tipi 08h + n olur.

IRQ	Aygıt	Tip	Vektör adresi
0	zamanlayıcı (18,2 Hz)	08h	00020h
1	klavye	09h	00024h
3 / 4	seri portlar (COM2 / COM1)	0Bh / 0Ch	0002Ch / 00030h
6	disket	0Eh	00038h
7	yazıcı	0Fh	0003Ch

Tablo 5.1.13: IBM PC donanım kesmeleri (tip = 08h + IRQ numarası).

Zamanlayıcının 18,2 Hz'lik frekansı ilginç bir hesaptan gelir. PC'nin 8253 zamanlayıcı yongası 1.193.182 Hz'lik bir saatle beslenir. Sayaç en büyük bölme değeri olan 65.536 ile çalıştırıldığında çıkış frekansı 1.193.182 / 65.536 ≈ 18,2 Hz olur. Bu da yaklaşık her 55 ms'de bir INT 08h demektir. BIOS bu kesmeyi günün saatini saymak için kullanır.

Klavye örneği, donanım kesmesi ile yazılım kesmesi arasındaki iş bölümünü de gösterir. Bir tuşa basılınca önce donanım kesmesi INT 09h çalışır. Bu ISR klavye denetleyicisinden tuşun kodunu okur ve BIOS'un klavye tamponuna koyar. Program ise tuşu okumak istediğinde yazılım kesmesi INT 16h'yi çağırır; INT 16h bu tampondan okur. Böylece kullanıcı program meşgulken de tuşlara basabilir. Tuşlar tamponda bekler ve program hazır olduğunda sırayla alınır. Tamponun ve INT 16h hizmetlerinin ayrıntısı 5.2 Kesme Fonksiyonları notunun konusudur.

5.1.5.4 8259A kesme denetleyicisi

8086'nın tek bir INTR pini vardır, PC'nin ise en az altı kesme kaynağı. Bu kaynakları tek pine toplayan, aralarında öncelik sıralaması yapan ve işlemciye doğru tip numarasını veren yonga 8259A'dır.

Tanım 5.1.7

Programlanabilir kesme denetleyicisi (*programmable interrupt controller*, 8259A), sekiz kesme isteği girişini (IRQ0–IRQ7) öncelik sırasına koyan, maskeleyen ve işlemcinin INTA döngüsünde ilgili tip numarasını veri yoluna koyan yongadır. PC'de port 20h ve 21h adreslerindedir.

8259A'nın içinde, her biri sekiz bitlik ve her biti bir IRQ'ya karşılık gelen üç yazmaç vardır (Şekil 5.1.10).



Şekil 5.1.10: 8259A'nın içi. IRQ0–7 istekleri IRR'ye yazılır, IMR'de maskeli olanlar elenir, öncelik çözücü en yüksek öncelikli isteği seçer ve INTR'yi sürer. INTA sırasında tip numarası (taban + IRQ no) veri yoluna konur ve isteğin biti ISR yazmacına geçer.

- **IRR** (*interrupt request register*, kesme istek yazmacı): gelmiş ama henüz hizmet görmemiş istekleri tutar. Bir IRQ hattı etkinleşince ilgili bit 1 olur.
- **IMR** (*interrupt mask register*, kesme maske yazmacı): her IRQ için bir maske biti tutar. Bit 1 ise o hattın isteği işlemciye iletilmez. IF bütün INTR girişini kapatırken IMR tek tek hatları kapatır. PC'de port 21h'den okunur ve yazılır.
- **ISR** (*in-service register*, hizmette yazmacı): o an hizmet görmekte olan IRQ'yu işaretler. INTA sırasında seçilen isteğin biti IRR'den ISR'ye geçer. EOI gelene kadar bu bit 1 kalır ve aynı ya da daha düşük öncelikli IRQ'lar bekletilir.

Üçüncü yazmacın adı ISR'dir ve kesme hizmet yordamının kısaltmasıyla aynıdır. Karışıklığı önlemek için bu notta yazmaçtan söz edilirken “ISR yazmacı” denir.

EOI. ISR yazmacındaki bit kendiliğinden silinmez. Hizmet yordamı işini bitirince 8259A'ya bunu bildirmelidir. Bu bildirime **kesme sonu** (*end of interrupt*, EOI) denir. PC'de iki komutla yapılır:

```
MOV AL, 20h     ; özel olmayan EOI komutu
OUT 20h, AL     ; 8259A'nın komut portuna
```

Bu komut, ISR yazmacındaki en yüksek öncelikli 1 bitini siler. Port numarası ile komut değerinin ikisinin de 20h olması tesadüftür. EOI unutulursa 8259A o IRQ'nun hâlâ hizmette olduğunu sanır. Aynı ve daha düşük öncelikli IRQ'lar bir daha işlemciye ulaşmaz. Örneğin zamanlayıcı ISR'sinde EOI unutulursa, IRQ0 en yüksek öncelikli olduğu için klavye, disk ve seri port kesmelerinin hepsi durur. Sistem donmuş gibi görünür.

Basamaklama. Sekiz giriş bazen yetmez. 8259A'lar basamaklanabilir (*cascade*): bir ana 8259A'nın her girişine bir yardımcı 8259A bağlanırsa 8 × 8 = 64 IRQ'ya çıkılır. IBM PC/AT ikinci bir 8259A'yı ana yonganın IRQ2 girişine bağlamıştır. İkinci yonganın girişleri IRQ8–IRQ15 adını alır ve tip 70h–77h'ye eşlenir. AT'de IRQ8 gerçek zamanlı saat, IRQ14 sabit diskin kesmesidir.

Sık yapılan hata

IMR'de bir bitin 1 olmasını "kesmeye izin var" diye okumak. IMR bir maske yazmacıdır: 1 engeller, 0 izin verir. IF'de ise durum tersidir: IF = 1 izin verir. İki kuralı karıştırmamak için IMR'deki 1'i "maskeli", IF'deki 1'i "açık" diye okumak yararlıdır.

5.1.5.5 IMR ile maskeleme: kod izleme

Bazen bir programın yalnız tek bir aygıtın kesmesini kısa bir süre engellemesi gerekir. Örneğin klavye tamponunu doğrudan değiştiren bir kod parçası, o sırada klavye ISR'sinin çalışmasını istemez; ama zamanlayıcının saymaya devam etmesini ister. CLI burada fazla kabadır, çünkü bütün INTR girişini kapatır. Doğru araç IMR'dir.

```
org 100h
in  al, 21h        ; IMR'yi oku
or  al, 00000010b  ; IRQ1 maskele
out 21h, al
mov cx, 3          ; kritik iş
in  al, 21h
and al, 11111101b  ; IRQ1 aç
out 21h, al
ret
```

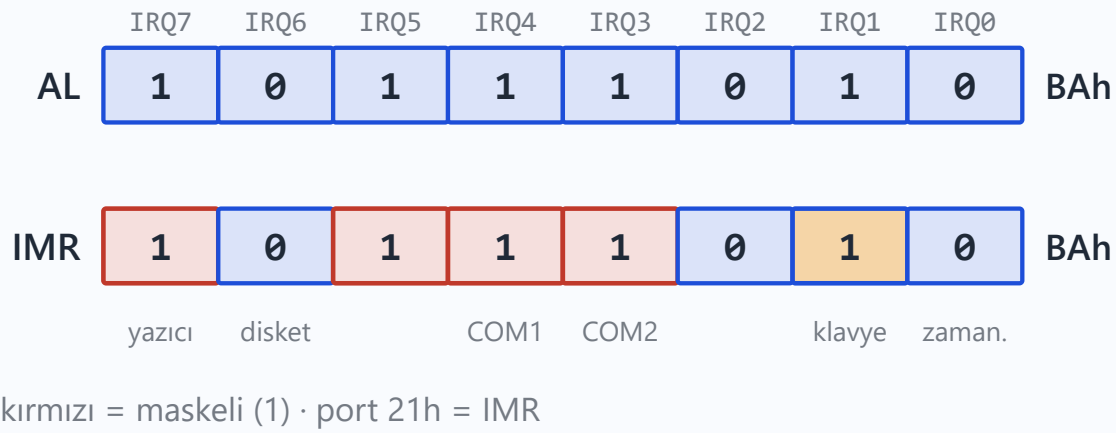
Program **oku–değiştir–yaz** kalıbını iki kez uygular. IMR'nin yalnız bir biti değiştirilmek istenir; öbür yedi bit başka programların, BIOS'un ya da işletim sisteminin ayarıdır. Bu yüzden IMR önce okunur, istenen bit OR ile kurulur ya da AND ile silinir ve sonuç geri yazılır. Doğrudan `MOV AL, 02h / OUT 21h, AL` yazmak IRQ1'i maskelerdi; ama öbür bütün IRQ'ların maskesini kaldırırdı.

İzlemede IMR'nin başlangıç değeri örnek olarak B8h alınmıştır: 1011 1000b. Bu değerde IRQ3, 4, 5 ve 7 maskelidir; IRQ0 (zamanlayıcı), IRQ1 (klavye), IRQ2 ve IRQ6 (disket) açıktır.

Satır	Komut	AL (ikili)	AL	CX	IMR	IRQ1
2	`in al, 21h`	1011 1000	**B8h**	0000h	B8h	açık
3	`or al, 00000010b`	1011 10**1**0	**BAh**	0000h	B8h	açık
4	`out 21h, al`	1011 1010	BAh	0000h	**BAh**	**maskeli**
5	`mov cx, 3`	1011 1010	BAh	**0003h**	BAh	maskeli
6	`in al, 21h`	1011 1010	BAh	0003h	BAh	maskeli
7	`and al, 11111101b`	1011 10**0**0	**B8h**	0003h	BAh	maskeli
8	`out 21h, al`	1011 1000	B8h	0003h	**B8h**	**açık**
9	`ret`	1011 1000	B8h	0003h	B8h	açık

Tablo 5.1.14: IMR ile IRQ1'i maskeleme programının izlemesi (IMR başlangıçta B8h).

Tablo 5.1.14'ün okunuşu şöyledir. **Satır 2'de** IMR okunur: AL = B8h. **Satır 3'te** `or al, 02h` yalnız bit 1'i kurar: 1011 1000 OR 0000 0010 = 1011 1010 = BAh. Diğer bitler korunur. IMR henüz değişmemiştir; değişen yalnız AL'dir. **Satır 4'te** `out` değeri 8259A'ya yazar ve IMR = BAh olur. Bu andan sonra IRQ1 isteği INTR'ye ulaşmaz; IF ve öbür IRQ'lar etkilenmez.



Şekil 5.1.11: `out 21h, al` sonrasında AL ve IMR bitleri: ikisi de BAh. Maskeli bitler (1) IRQ3, 4, 5, 7 ve yeni maskelenen IRQ1'dir; IRQ0, IRQ2 ve IRQ6 açıktır.

Satır 5 kesilmemesi gereken işin yerini tutar. Bu sırada klavyeye basılırsa istek kaybolmaz: 8259A isteği IRR'de saklar. **Satır 6–8** maskeyi aynı kalıpla kaldırır. `and al, 0FDh` ile 1111 1101b maskesi yalnız bit 1'i siler: 1011 1010 AND 1111 1101 = 1011 1000 = B8h. `out` ile IMR yeniden B8h olur. IRR'de bekleyen klavye isteği varsa şimdi INTR'ye ulaşır ve hizmet görür. **Satır 9'da** program döner; maske ilk hâline gelmiştir ve başkasının bitlerine dokunulmamıştır.

IMR'nin CLI'dan farkı bu izlemede açıkça görülür. CLI bütün maskelenebilir kesmeleri bekletir; zamanlayıcının darbeleri de beklerken saat geri kalabilir. IMR ise yalnız seçilen aygıtı bekletir. Bedeli, IMR'ye erişimin iki IN/OUT komutu gerektirmesi ve bir G/Ç portuna yazmak olduğu için CLI'dan daha yavaş olmasıdır.

Sık yapılan hata

IMR'ye okumadan yazmak: `MOV AL, 02h` ve `OUT 21h, AL` . Bu iki satır IRQ1'i maskeler ama IRQ3, 4, 5 ve 7'nin maskesini kaldırır. O hatlara ISR'si kurulmamış bir aygıt bağlıysa, gelen ilk istek işlemciyi anlamsız bir vektöre gönderir.

5.1.5.6 Kesmenin bekletildiği anlar

8086 kesmeyi yalnız komut sınırlarında kabul eder. Bazı durumlarda bu sınır bilerek bir komut ileri kaydırılır. Tablo 5.1.15 bu durumları toplar.

Durum	Davranış	Neden
komut ortası	istek, yürüyen komut bitene kadar bekler	komut yarım kalmasın
önekler (LOCK, REP, kesim öneki)	önekle komut arasında kesme kabul edilmez	önek komutun parçasıdır
`MOV SS, …` ya da `POP SS` sonrası	bir sonraki komut da bitene kadar kesme gelmez	ardından gelen `MOV SP, …` güvenle tamamlansın
`STI` sonrası	bir komut daha yürütülür, sonra kesme kabul edilir	`STI` / `IRET` ya da `STI` / `HLT` arasına kesme girmesin
REP'li dizgi komutu	turlar arasında kesilebilir; IRET'ten sonra kalan CX kadar sürer	uzun bir kopyalama gecikmeyi büyütmesin
`HLT` ve `WAIT`	INTR ya da NMI işlemciyi uyandırır	bekleme kesmeyle biter

Tablo 5.1.15: INTR'nin hemen kabul edilmediği durumlar.

SS yüklendikten sonra. Yığını taşımak için iki komut gerekir: `MOV SS, AX` ve `MOV SP, 0FFFEh`. İki komut arasında bir kesme gelseydi, işlemci FLAGS, CS ve IP'yi yeni SS ile eski SP'nin gösterdiği yere iterdi. Bu adres büyük olasılıkla başka bir verinin üzerindedir. 8086 bu yüzden bir kesim yazmacına yazan her komuttan sonra kesmeyi bir komut erteler. Bunun güvenli çalışması için `MOV SP`, `MOV SS`'nin hemen ardından yazılmalıdır.

STI'den sonra. STI kesmeleri hemen değil, kendisinden sonraki komut bittikten sonra açar. Bu küçük gecikme iki yaygın kalıbı güvenli kılar. Birincisi bir ISR'nin sonundaki `STI` / `IRET` dizisidir: yeni kesme, IRET bitmeden ve yığın boşalmadan gelemez. İkincisi `STI` / `HLT` dizisidir. Program "kesmeleri aç ve bir kesme gelene kadar bekle" demek ister. Kesme STI ile HLT arasına girseydi, ISR çalışıp dönerdi ve HLT bir sonraki kesmeye kadar gereksiz yere beklerdi.

REP'li dizgi komutları. `REP MOVSB` ile 60.000 baytlık bir kopyalama yüz binlerce saat turu sürer. Bütün bu süre boyunca kesmelerin beklemesi kabul edilemez. 8086 bu yüzden REP'li komutu turlar arasında kesebilir. Yığına itilen dönüş adresi, REP önekinin adresidir; SI, DI ve CX o ana kadarki ilerlemeyi tutar. IRET'ten sonra komut yeniden başlar ve kalan CX kadar tur atar. Bu düzenin 8086'ya özgü bir kusuru vardır: işlemci kesmeden dönüşte yalnız bir öneki hatırlar. `REP ES: MOVSB` gibi iki önekli bir komut kesilirse dönüşte kesim öneki kaybolur ve kopyalama yanlış kesimden sürer. Bu yüzden böyle bir satır CLI ile STI arasına alınır ya da kesim öneki gerektirmeyecek biçimde yazılır.

HLT ve WAIT. HLT işlemciyi durdurur. INTR (IF = 1 iken) ya da NMI işlemciyi uyandırır; ISR çalışır ve IRET, HLT'den sonraki komuta döner. Ana programı "bir şey olana kadar uyu" diye yazmanın en ekonomik yolu budur. WAIT, 8087 eş işlemcisinin işini bitirmesini bekler; bu bekleme sırasında gelen kesme de kabul edilir.

Sık yapılan hata

Yığını taşırken önce SP'yi, sonra SS'yi yüklemek ya da iki komut arasına başka bir komut koymak. Erteleme yalnız kesim yazmacına yazan komuttan sonraki *tek* komutu korur. Doğru sıra `MOV SS, AX` ve hemen ardından `MOV SP, …`'dur; en güvenlisi bu iki satırı ayrıca CLI/STI ara-

sına almaktır.

5.1.5.7 Kesme gecikmesi

Bir aygıt kesme istediğinde ISR'nin hemen çalışmaya başlaması beklenir. Gerçekte bu süre sıfır değildir ve bazı aygıtlar için sınırlıdır.

Tanım 5.1.8

Kesme gecikmesi (*interrupt latency*), kesme isteğinin gelmesiyle ISR'nin ilk komutunun başlaması arasında geçen süredir.

Gecikme iki parçadan oluşur. Birincisi, istek geldiğinde yürümekte olan komutun bitmesini beklemektir. İkincisi kabul sürecinin kendisidir: INTA döngüleri, üç yığın yazması, iki vektör okuması ve kuyruğun yeniden doldurulması. Tablo 5.1.16 ilgili süreleri verir.

İşlem	Saat turu	5 MHz'de süre
INT n	51	10,2 µs
INT 3	52	10,4 µs
INTO (OF = 1 / OF = 0)	53 / 4	10,6 / 0,8 µs
INTR kabulü (iki INTA dahil)	61	12,2 µs
IRET	24	4,8 µs
16 bitlik IDIV (yazmaç)	165–184	33–36,8 µs

Tablo 5.1.16: Kesmeyle ilgili işlemlerin 8086'daki saat turu sayıları (8086 kılavuzu).

Örnek 5.1.6

5 MHz'lik bir 8086'da INTR geldiğinde 16 bitlik bir IDIV yeni başlamıştır. En kötü kesme gecikmesi kaç saat turu ve kaç µs'dir? İşlemci 8 MHz'de çalışsaydı ne olurdu?

1. En kötü durumda yürüyen komut bitmelidir: IDIV en çok 184 tur.
2. Ardından kabul süreci: 61 tur. Toplam 184 + 61 = 245 tur.
3. 5 MHz'de bir saat turu 1 / 5 MHz = 200 ns'dir. 245 × 200 ns = 49 µs.
4. 8 MHz'de bir tur 125 ns'dir: 245 × 125 ns ≈ 30,6 µs.

Program o sırada kesmeleri CLI ile kapatmışsa, kapalı geçen süre bu gecikmeye doğrudan eklenir. 100 µs'lik bir CLI–STI aralığı en kötü gecikmeyi 5 MHz'de yaklaşık 149 µs'ye çıkarır.

Bu sayıların anlamı aygıta göre değişir. Saniyede 18,2 kez gelen zamanlayıcı kesmesi için 49 µs önemsizdir; iki kesme arasında 55 ms vardır. Seri port ise farklıdır. 9600 bit/s hızla gelen karakterler arasında yaklaşık 1,04 ms, 115.200 bit/s hızda ise yaklaşık 87 µs vardır. Tek karakterlik tamponu olan bir seri port denetleyicisinde ISR, bir sonraki karakter gelmeden önceki karakteri okumalıdır. Gecikme ile ISR'nin süresinin toplamı bu aralığı aşarsa karakter kaybolur.

Bu hesap, ISR yazımının iki temel kuralını doğrudan verir. Birincisi, CLI–STI arası kısa tutulmalıdır; çünkü bu süre bütün aygıtların gecikmesine eklenir. İkincisi, ISR'nin kendisi de kısa tutulmalıdır. ISR çalışırken IF = 0'dır ve aynı ya da daha düşük öncelikli IRQ'lar EOI'ye kadar bekler. Uzun bir ISR, öbür bütün aygıtlar için uzun bir CLI ile aynı etkiyi yapar.

ISR'nin işlemciden aldığı pay da hesaplanabilir. Zamanlayıcı ISR'si 200 saat turu sürsün. Saniyede 18,2 kez çalıştığında 200 × 18,2 = 3640 tur harcar. Bu, 5 MHz'lik işlemcinin saniyedeki 5.000.000 turunun yalnız %0,07'sidir. Kesmenin yoklamaya göre üstünlüğü bu sayıda açıkça görülür.

5.1.6 Kendi kesme hizmet yordamımız

5.1.6.1 Vektörü kurmak: INT 90h

Bir kesme hizmet yordamı kurmak üç iş gerektirir: yordamı yazmak, kullanılmayan bir tip seçmek ve o tipin vektörüne yordamın adresini yazmak. Aşağıdaki program bu üç işi en sade biçimde yapar ve kendi kesmesini bir kez çağırır.

```
org 100h
mov ax, 0
mov es, ax                          ; ES = 0: IVT kesimi
cli                                 ; kesmeleri kapat
mov word ptr es:[90h*4], offset isr ; IP
mov word ptr es:[90h*4+2], cs       ; CS
sti                                 ; kesmeleri aç
int 90h                             ; kendi kesmemiz
ret
isr:
mov ah, 0Eh
mov al, '*'
int 10h
iret                                ; RET değil!
```

Programdaki her kararın bir gerekçesi vardır.

- **Tip seçimi.** 90h, PC'de BIOS'un ve DOS'un kullanmadığı bir tiptir. Vektörü 90h × 4 = 240h adresindedir; IP 0000:0240h–0241h'de, CS 0000:0242h–0243h'dedir.

- **ES = 0.** Vektör tablosu 0000h kesimindedir. DS programın kendi verisini gösterdiği için tabloya ES üzerinden, `es:` önekiyle erişilir. Kesim yazmacına anlık değer yüklenemediğinden 0 önce AX'e konur.
- **CLI / STI.** Vektör iki ayrı MOV ile, iki sözcük olarak yazılır. İlk MOV'dan sonra vektörün IP yarısı yeni, CS yarısı eskidir. O anda bu tipte bir kesme gelseydi, işlemci anlamsız bir adrese atlardı. CLI bu kısa aralıkta maskelenebilir kesmeleri bekletir. Tip 90h'yi yalnız programın kendisi çağırdığı için bu örnekte tehlike yoktur; ama donanım tiplerinde aynı kalıp zorunludur ve her vektör yazılışında uygulanması iyi bir alışkanlıktır.
- **ISR'nin sonu.** ISR, BIOS'un INT 10h teletype hizmetiyle ekrana bir yıldız yazar ve IRET ile döner. Bir ISR'nin içinden başka bir yazılım kesmesi çağrılabilir; iç içe kesmelerde yığına her düzey için 6 bayt daha eklenir.

Program çalıştırıldığında ekrana `*` yazılır ve program DOS'a döner. Bu kısa örnek eski vektörü saklamaz ve geri yüklemez. Program bittikten sonra tip 90h'nin vektörü, artık başka bir programın yükleneceği bir belleği göstermeye devam eder. Başka bir program INT 90h çağırırsa oradaki rastgele baytlar komut olarak yürütülür. Eski vektörü saklayıp geri yükleyen doğru biçim aşağıda ayrıca izlenir.

DOS altında vektörleri doğrudan bellekten yazmak yerine DOS'un iki hizmeti kullanılabilir: `INT 21h`, AH = 25h vektör atar, AH = 35h vektör okur. Bu hizmetler vektör yazılırken kesmeleri kendileri yönetir ve işletim sisteminin vektörlerden haberdar olmasını sağlar. Bu yüzden daha güvenli yoldur. Hizmetlerin yazmaç düzeni 5.2 Kesme Fonksiyonları notunda verilir; burada vektör tablosunun nasıl çalıştığını görmek için doğrudan yazma yöntemi kullanılmıştır.

5.1.6.2 INT 60h ile kendi ISR'miz: kod izleme

Aynı düzenin biraz farklı bir biçimi, ISR'ye yazmaç üzerinden parametre geçirir. Ana program yazılacak karakteri AL'ye koyar; ISR yalnız AH'yi hazırlar ve INT 10h'yi çağırır. Kullanıcı programlarına ayrılmış 60h tipi kullanılır.

```
org 100h
mov ax, 0
mov es, ax          ; ES = 0: IVT
cli
mov word ptr es:[60h*4], offset isr
mov word ptr es:[60h*4+2], cs
sti
mov al, '#'
int 60h             ; kendi kesmemiz
ret
isr:
mov ah, 0Eh
int 10h
iret
```

Bağıl konumlar şöyledir: `mov ax, 0` 0100h, `mov es, ax` 0103h, `cli` 0105h, iki vektör yazması 0106h ve 010Dh, `sti` 0112h, `mov al, '#'` 0113h, `int 60h` 0115h, `ret` 0117h ve `isr` 0118h. Vektör 60h × 4 = 180h adresindedir.

Satır	Komut	AX	ES	CS:IP (sonraki)	SP	FLAGS	IVT 0180h–0183h / yığın	Ekran
2	`mov ax, 0`	0000h	0700h	0700:0103	FFFEh	F202h	?? ?? ?? ??	
3	`mov es, ax`	0000h	**0000h**	0700:0105	FFFEh	F202h	?? ?? ?? ??	
4	`cli`	0000h	0000h	0700:0106	FFFEh	**F002h**	?? ?? ?? ??	
5	`mov … [60h*4], offset isr`	0000h	0000h	0700:010D	FFFEh	F002h	**18 01 ?? ??**	
6	`mov … [60h*4+2], cs`	0000h	0000h	0700:0112	FFFEh	F002h	**18 01 00 07**	
7	`sti`	0000h	0000h	0700:0113	FFFEh	**F202h**	18 01 00 07	
8	`mov al, '#'`	**0023h**	0000h	0700:0115	FFFEh	F202h	18 01 00 07	
9	`int 60h`	0023h	0000h	**0700:0118**	**FFF8h**	**F002h**	[FFF8h] = 0117h, [FFFAh] = 0700h, [FFFCh] = F202h	
12	`mov ah, 0Eh`	**0E23h**	0000h	0700:011A	FFF8h	F002h	yığın aynı	
13	`int 10h`	0E23h	0000h	0700:011C	FFF8h	F002h	yığın aynı	**#**
14	`iret`	0E23h	0000h	**0700:0117**	**FFFEh**	**F202h**	yığın boşaldı	#
10	`ret`	0E23h	0000h	**0700:0000**	**0000h**	F202h	DOS'a dönüş	#

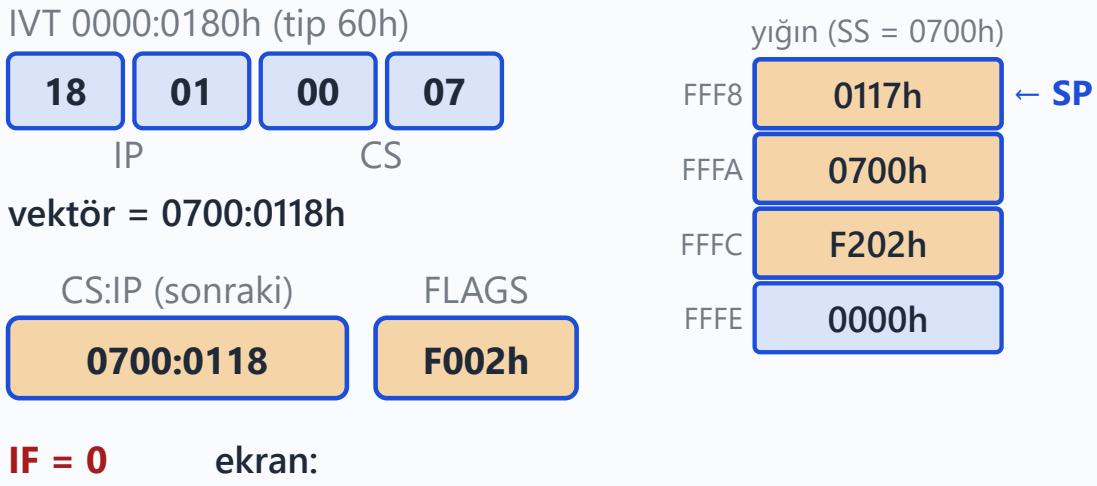
Tablo 5.1.17: INT 60h programının izlemesi (her satır yürütüldükten sonraki durum; SS = 0700h).

Tablo 5.1.17 dört evreye ayrılır. **Hazırlık (satır 2–3).** ES başlangıçta, .COM programının öbür kesim yazmaçları gibi 0700h'dir. `mov es, ax` sonrasında ES = 0000h olur ve `es:[180h]` artık vektör tablosundaki 0000:0180h baytını gösterir. Tablonun bu dört baytının program öncesindeki değeri bilinmez; izlemede "??" ile gösterilmiştir.

Vektörü yazma (satır 4–7). `cli` IF'yi sıfırlar ve FLAGS F202h'den F002h'ye iner. Satır 5 IP yarısına offset isr = 0118h'yi yazar; bellekte little-endian düzende `18 01` görülür. Satır 6 CS yarısına 0700h'yi yazar: `00 07`. Vektör artık 0700:0118'dir. `sti` ile IF = 1 ve FLAGS = F202h olur. 8086'da FLAGS'in bit 1 ve 12–15 bitleri hep 1 olduğu için IF'nin tek başına değeri 0200h fark yaratır.

Kesme (satır 8–9). `mov al, '#'` AL'ye 23h koyar. Bu değer ISR'ye yazmaçla geçirilen parametredir. `int 60h` yığına FLAGS (F202h), CS (0700h) ve dönüş IP'si 0117h'yi iter. INT 60h 0115h'te başlar ve iki bayttır; dönüş adresi 0117h'dir. IF = 0 olur ve CS:IP, 0000:0180h'deki vektörden 0700:0118 olarak

yüklenir.



Şekil 5.1.12: `int 60h` yürütüldükten sonra: tip 60h'nin vektörü 0700:0118h, yığında dönüş IP'si 0117h, CS ve FLAGS; SP = FFF8h, IF = 0 ve CS:IP ISR'nin ilk komutunda.

ISR ve dönüş (satır 12–14 ve 10). ISR'de `mov ah, 0Eh` AX'i 0E23h yapar. AH = 0Eh, BIOS'un teletype hizmetini seçer; AL'deki karakter yazılacaktır. `int 10h` ekrana `#` yazar. Bu, kesmenin içinden çağrılan ikinci bir kesmedir. BIOS kendi yığın kullanımını geri bırakır ve döndüğünde SP yine FFF8h'dir. `iret` IP = 0117h, CS = 0700h ve FLAGS = F202h'yi geri alır; SP = FFFEh ve IF = 1 olur. Ana programdaki `ret` DOS'a döner.

ISR'nin AX'i değiştirdiğine dikkat edilmelidir. Ana program kesmeden sonra AX = 0E23h ile devam eder. Bu programda AL bilerek ISR'ye parametre olarak verildiği için AX'in değişmesi bir sorun yaratmaz. Ama ISR bir donanım kesmesine bağlı olsaydı, ana program AH'nin aniden 0Eh olduğunu görürdü. Bu tür bir ISR kullandığı her yazmacı saklamalıdır.

5.1.6.3 Vektörü sakla, değiştir, geri yükle: kod izleme

Önceki iki programın eksiği, tabloyu değiştirip öyle bırakmalarıdır. Doğru bir program önce eski vektörü saklar, işini bitirince geri yükler. Aşağıdaki program bunu tip 3 için yapar ve ISR'nin ana programın yazmaçlarını nasıl koruduğunu da gösterir.

```
org 100h
xor ax, ax
mov es, ax          ; ES = 0: IVT
mov ax, offset isr3
xchg ax, es:[3*4]   ; yeni IP ⇔ eski
mov eski, ax
mov ax, cs
xchg ax, es:[3*4+2] ; yeni CS ⇔ eski
mov eski+2, ax
mov ax, 1234h
int 3               ; bizim ISR
mov bx, eski
mov es:[3*4], bx    ; geri yükle
mov bx, eski+2
mov es:[3*4+2], bx
ret
isr3: push ax
mov ax, 0E21h       ; AH=0Eh, AL='!'
int 10h
pop ax
iret
eski dw ?, ?
```

Programın yeni öğesi `XCHG` komutudur. `xchg ax, es:[3*4]`, AX ile bellekteki sözcüğü tek komutta değiş tokuş eder: yeni IP belleğe yazılırken eski IP AX'e gelir. Okuma ve yazma iki ayrı MOV ile yapılsaydı, arada iki komutluk bir aralık kalırdı. XCHG ile eski değer okunmadan yeni değerin yazılması da, yazılmadan önce okunan değerin kaybolması da mümkün değildir. Eski vektör `eski` adlı iki sözcüklük değişkene saklanır. `xor ax, ax` AX'i sıfırlar; MOV'dan bir bayt kısadır ama bayrakları değiştirir: ZF = 1 ve PF = 1 olur. Bu yüzden FLAGS bu programda F246h'dir.

Programdan önceki tip 3 vektörü F000:1003h'dir: emu8086 her tip n için varsayılan vektörü F000:(1000h + n) adresine kurar (tip 0 için F000:1000h). Gerçek bir PC'de bu değer BIOS'a ve yüklü hata ayıklayıcıya göre değişir; programın doğruluğu bu değere bağlı değildir. Bağıl konumlar: `int 3` 011Ch'de ve tek bayttır, `isr3` 0130h'de, `eski` 0138h'dedir.

Satır	Komut	AX	BX	Tip 3 vektörü	CS:IP (sonraki)	SP	IF	Yığın / çıktı
2–3	`xor ax, ax` / `mov es, ax`	0000h	0000h	F000:1003	0700:0104	FFFEh	1	—
4	`mov ax, offset isr3`	**0130h**	0000h	F000:1003	0700:0107	FFFEh	1	—
5	`xchg ax, es:[3*4]`	**1003h**	0000h	**F000:0130**	0700:010C	FFFEh	1	—
6	`mov eski, ax`	1003h	0000h	F000:0130	0700:010F	FFFEh	1	eski = 1003h
7	`mov ax, cs`	**0700h**	0000h	F000:0130	0700:0111	FFFEh	1	—

Satır	Komut	AX	BX	Tip 3 vektörü	CS:IP (sonraki)	SP	IF	Yığın / çıktı
8	`xchg ax, es:[3*4+2]`	**F000h**	0000h	**0700:0130**	0700:0116	FFEEh	1	—
9	`mov eski+2, ax`	F000h	0000h	0700:0130	0700:0119	FFEEh	1	eski+2 = F000h
10	`mov ax, 1234h`	**1234h**	0000h	0700:0130	0700:011C	FFEEh	1	—
11	`int 3`	1234h	0000h	0700:0130	**0700:0130**	**FFF8h**	**0**	[FFF8h] = 011Dh, [FFFAh] = 0700h, [FFFCh] = F246h
17	`push ax`	1234h	0000h	0700:0130	0700:0131	**FFF6h**	0	[FFF6h] = 1234h
18	`mov ax, 0E21h`	**0E21h**	0000h	0700:0130	0700:0134	FFF6h	0	—
19	`int 10h`	0E21h	0000h	0700:0130	0700:0136	FFF6h	0	**çıktı: !**
20	`pop ax`	**1234h**	0000h	0700:0130	0700:0137	**FFF8h**	0	—
21	`iret`	1234h	0000h	0700:0130	**0700:011D**	**FFEEh**	**1**	yığın boşaldı
12–13	`mov bx, eski` / `mov es:[3*4], bx`	1234h	**1003h**	**0700:1003**	0700:0126	FFEEh	1	—
14–15	`mov bx, eski+2` / `mov es:[3*4+2], bx`	1234h	**F000h**	**F000:1003**	0700:012F	FFEEh	1	—
16	`ret`	1234h	F000h	F000:1003	**0700:0000**	**0000h**	1	DOS'a dönüş

Tablo 5.1.18: Vektörü sakla, değiştir, geri yükle programının izlemesi (SS = 0700h).

Tablo 5.1.18'in ilk evresi vektörün değiştirilmesidir. **Satır 5'te** XCHG yeni IP'yi (0130h) tabloya yazar ve eski IP'yi (1003h) AX'e getirir. Vektör o an F000:0130'dur: yarısı yeni, yarısı eski. **Satır 8'de** ikinci XCHG yeni CS'yi (0700h) yazar, eski CS'yi (F000h) AX'e getirir. Vektör artık 0700:0130'dur ve eski vektör F000:1003, `eski` değişkeninde saklıdır. Tip 3'ü yalnız program çağırdığı için bu örnekte CLI gerekmemiştir. Bir donanım tipinin vektörü değiştirilirken bu satırlar CLI ile STI arasına alınır.

İkinci evre kesmedir. **Satır 10'da** AX = 1234h olur; bu, ana programın korunması gereken değeridir. **Satır 11'de** `int 3` tek bayttır (CCh). Yığına FLAGS (F246h), CS (0700h) ve dönüş IP'si 011Dh itilir. Dönüş adresi 011Ch + 1 = 011Dh'dir; INT 3'ün tek baytlığı burada da görülür. CS:IP ← 0700:0130 olur.



Şekil 5.1.13: ISR'nin ilk komutu `push ax`'ten sonra: yığında dört sözcük vardır. Altta kesmenin ittiği FLAGS, CS ve IP, üstte ISR'nin sakladığı AX = 1234h; SP = FFF6h.

Üçüncü evre ISR'dir. **Satır 17'de** ISR kullanacağı AX'i önce yığına iter: SP = FFF6h ve [FFF6h] = 1234h (Şekil 5.1.13). **Satır 18–19** AX = 0E21h ile ekrana `!` yazar. **Satır 20'de** `pop ax` 1234h'yi geri alır; SP = FFF8h olur ve yığının tepesi yeniden kesmenin ittiği IP'dir. Bu noktada PUSH ile POP'un sayısı eşit olmalıdır. Eşit olmasaydı IRET yanlış sözcükleri dönüş adresi diye alırdı. **Satır 21'de** `iret` IP = 011Dh, CS = 0700h ve FLAGS = F246h'yi geri alır; SP = FFFEh ve IF = 1 olur.

Son evre geri yüklemedir. **Satır 12–15** `eski` değişkenindeki iki sözcüğü tabloya geri yazar. Burada da vektör bir an 0700:1003 gibi anlamsız bir ara değerden geçer. **Satır 16'da** program döndüğünde AX = 1234h korunmuştur ve vektör tablosu programdan önceki hâlindedir. Bir programın vektör tablosunda iz bırakmadan çıkması budur.

Sık yapılan hata

Programın her çıkış yolunda vektörü geri yüklemeyi unutmak. Program yalnız normal sonunda geri yükleme yapar, ama bir hata durumunda erkenden DOS'a dönerse, vektör artık yüklü olmayan bir kodu gösterir. Bir sonraki INT 3 ya da donanım kesmesi sistemi çökertir. Geri yükleme kodu, programın bütün çıkışlarının geçtiği tek bir yere konmalıdır.

5.1.6.4 ISR yazım kuralları

Önceki izlemeler ISR yazımının kurallarını tek tek ortaya koydu. Tablo 5.1.19 bu kuralları ve uyulmadığında ortaya çıkan belirtileri toplar.

Kural	Uyulmazsa
Kullanılan her yazmaç PUSH/POP ile korunur (FLAGS'i işlemci korur).	Ana program, değeri kendiliğinden değişmiş bir yazmaçla devam eder; hata rastgele anlarda görülür.

Kural	Uyulmazsa
Donanım ISR'si 8259A'ya EOI gönderir: `MOV AL, 20h` · `OUT 20h, AL`.	8259A ISR yazmacındaki biti silmez; aynı ve daha düşük öncelikli IRQ'lar bir daha gelmez.
ISR kısa tutulur; uzun iş bir bayrak değişkeniyle ana programa bırakılır.	Kesme gecikmesi büyür, hızlı aygıtların verisi kaybolur.
Donanım ISR'sinden INT 21h çağrılmaz.	DOS yeniden girişli değildir; kesilen DOS çağrısının verileri bozulur.
Vektör CLI ile ya da AH = 25h ile yazılır, çıkışta eskisi geri yüklenir.	Program bittikten sonra kesme, artık başka bir şeyin bulunduğu belleğe atlar.
ISR IRET ile biter.	Yığın 4 bayt kayar, IF kapalı kalır.

Tablo 5.1.19: ISR yazım kuralları ve uyulmadığında ortaya çıkan sorunlar.

Yazmaçları korumak. Bir donanım ISR'si programın herhangi bir komutundan sonra çalışabilir. Programın o anda hangi yazmacı kullandığı bilinemez. Bu yüzden ISR, kullandığı her yazmacı girişte iter ve çıkışta ters sırayla geri alır. Bayrakları saklamaya gerek yoktur; INT ve IRET bunu kendileri yapar. Kesim yazmaçları da unutulmamalıdır. ISR kendi verisine erişmek için DS'yi değiştirirse, DS'yi de saklamalıdır.

Kısa ISR ve bayrak değişkeni. Uzun süren bir iş ISR'nin içinde yapılmaz. ISR yalnız acil kısmı yapar: aygıttan veriyi okur, bir tampona koyar ve bir değişkeni 1 yapar. Ana program bu değişkene döngüsünde bakar ve uzun işi kesmeler açıkken yapar. Bu kalıp yoklamaya benzer görünür, ama yoklanan şey yavaş bir aygıt değil bellekteki bir değişkendir ve veri kaybolma tehlikesini ISR ortadan kaldırmıştır.

Tanım 5.1.9

Yeniden girişli (*reentrant*) yordam, bir çağrısı henüz bitmeden, örneğin bir kesme yüzünden, yeniden çağrıldığında da doğru çalışan yordamdır. Bunun için yordam kalıcı durumunu sabit bellek hücrelerinde değil, yığında ya da yazmaçlarda tutar.

DOS'u çağırmamak. DOS, çalışırken kendi iç değişkenlerini ve kendi yığınını sabit bellek alanlarında tutar. Program bir INT 21h çağrısının ortasındayken bir donanım kesmesi gelir ve ISR yeniden INT 21h çağırırsa, ikinci çağrı birincinin değişkenlerinin ve yığınının üzerine yazar. Birinci çağrıya dönüldüğünde veriler bozulmuştur. DOS yeniden girişli olmadığı için donanım ISR'leri DOS hizmetlerini kullanmaz. BIOS'un INT 10h gibi hizmetleri ise çoğu zaman bu açıdan daha güvenlidir; yukarıdaki örnek ISR'ler bu yüzden INT 10h'yi kullanır ve yazılım kesmesiyle çağrılır.

ISR içinde STI. ISR, IF = 0 ile başlar. Uzun bir ISR, işinin başında STI ile daha öncelikli donanım kesmelerine izin verebilir; 8259A aynı ve daha düşük öncelikli IRQ'ları EOI'ye kadar zaten bekletir. Bu durumda ISR'nin kendisi de yeniden girişli yazılmalıdır. Çünkü başka bir aygıtın kesmesi, ISR'nin ortasında başka bir yordam çalıştırabilir ve ISR'nin paylaştığı değişkenlere dokunabilir.

Bu kuralları uygulayan tipik bir donanım ISR'sinin iskeleti şöyledir. Kod, kuralların sırasını göstermek için yazılmıştır; aygıtın ne olduğuna göre ortadaki satırlar değişir.

```
isr_irq:
    push ax              ; kullanılan yazmaçları sakla
    push ds
    mov  ax, seg veri    ; ISR'nin kendi veri kesimi
    mov  ds, ax
    in   al, 60h          ; aygıttan veriyi al (örnek port)
    mov  son_kod, al      ; tampona / değişkene koy
    mov  hazir, 1         ; ana programa bayrak
    mov  al, 20h          ; EOI
    out  20h, al
    pop  ds               ; ters sırayla geri al
    pop  ax
    iret                  ; RET değil
```

Sık yapılan hata

Yazılım kesmesiyle çağrılan bir ISR'ye EOI koymak. EOI yalnız 8259A'dan gelen donanım kesmeleri içindir. INT 60h gibi bir yazılım kesmesinin ISR'si EOI gönderirse, 8259A o sırada hizmette olan başka bir IRQ'nun bitini siler ve o aygıtın kesme düzeni bozulur.

5.1.7 Alıştırmalar ve çözümleri

Sunumdaki altı alıştırma burada ara adımlarıyla çözülmüştür.

1. INT 13h vektörü hangi fiziksel adreslerde bulunur?

13h × 4 = 4Ch (onlu: 19 × 4 = 76). Vektör 0004Ch–0004Fh baytlarındadır. IP 0004Ch–0004Dh'de, CS 0004Eh–0004Fh'dedir.

2. INT 21h çalışmadan önce SP = FFFEh ise ISR içindeyken SP kaçtır? Yığında ne vardır?

INT üç sözcük, 6 bayt iter: FFFEh − 6 = FFF8h. Yığının tepesinde, [FFF8h]'de dönüş IP'si; [FFFAh]'da CS; [FFFCh]'de FLAGS vardır. FLAGS'in yığındaki kopyası IF = 1'i taşır; yazmaçtaki FLAGS ise IF = TF = 0'dır.

3. IMR = B8h iken yalnız IRQ0'ı (zamanlayıcı) da maskeleyen komutları yazın. IMR kaç olur?

Oku–değiştir–yaz kalıbı kullanılır: `IN AL, 21h` · `OR AL, 01h` · `OUT 21h, AL`. 1011 1000 OR 0000 0001 = 1011 1001 = B9h. IRQ3, 4, 5 ve 7 maskeli kalır, IRQ0 da maskelenir. Zamanlayıcı maskeli kaldıkça günün saati ilerlemez; bu yüzden böyle bir maske çok kısa tutulur.

4. Tip 61h'nin vektörü 0700:0120h yapılacak. Hangi fiziksel adreslere hangi baytlar yazılır?

61h × 4 = 184h. Önce IP, sonra CS yazılır; ikisi de little-endian yazılır. 00184h ← 20h, 00185h ← 01h, 00186h ← 00h, 00187h ← 07h. Bu ISR'nin fiziksel adresi 07000h + 0120h = 07120h'dir.

5. 5 MHz'lik 8086'da INTR geldiğinde MUL BX (en çok 133 tur) yeni başlamışsa kesme gecikmesi en çok ne kadardır?

Önce MUL bitmelidir: 133 tur. Ardından kabul: 61 tur. Toplam 194 tur. 194 × 200 ns = 38,8 µs ≈ 39 µs.

6. ISR'nin IRET yerine RET ile bitmesi ne gibi sorunlara yol açar?

Yakın RET yalnız IP'yi alır. CS ve FLAGS yığında kalır, SP 4 bayt kayar. FLAGS geri gelmediği için IF = 0 kalır ve maskelenebilir kesmeler bir daha kabul edilmez. ISR başka kesimdeyse CS de yanlış kalır ve program yanlış kesimde devam eder. İlk RET ya da POP, yığında kalan CS'yi veri ya da dönüş adresi diye alır. RET ile biten ISR izlemesinde bu durumların hepsi görülür.

5.1.8 Özet

- **Kesme**, işlemcinin o anki işini bir komut sınırında bırakıp bir kesme hizmet yordamını (ISR) çalıştırması ve IRET ile kaldığı yere dönmesidir. Kesilen program kesildiğini fark etmez.
- **Yoklamada** işlemci aygıta sürekli sorar ve zamanının çoğunu boşa harcar; **kesmede** aygıt hazır olunca işlemciyi uyarır. Kesmenin bedeli vektör tablosu, yığın ve ISR düzenidir.
- Kesmeler üç kaynaktan gelir: **yazılım** (INT n, INT 3, INTO), **donanım** (INTR, NMI) ve **iç** (bölme hatası INT 0, tek adım INT 1). Tip numarası belirlendikten sonra üçü de aynı adımları izler.
- **Kesme vektör tablosu** 00000h–003FFh'dedir: 256 tip × 4 bayt. Tip n'nin vektörü n × 4 adresindedir; düşük sözcük IP, yüksek sözcük CS'dir, ikisi de little-endian yazılır.
- **INT n**: FLAGS itilir, IF = TF = 0 yapılır, CS ve IP itilir (SP − 6), CS:IP ← vektör. İtilen IP, INT'ten sonraki komutun adresidir. **IRET** IP, CS ve FLAGS'i geri alır; IF ve TF eski değerlerine döner.
- ISR'yi RET ile bitirmek yığını 4 bayt kaydırır ve IF'yi kapalı bırakır; RETF de FLAGS'i geri almaz. ISR'ler her zaman IRET ile biter.
- **Bölme hatası** hem bölen sıfırken hem de bölüm hedefe sığmadığında oluşur. 8086 DIV'den sonraki komutun adresini iter. **INTO** yalnız OF = 1 iken INT 4'ü çağırır. **TF** = 1 iken her komuttan sonra INT 1 üretilir; TF yığın üzerinden değiştirilir. **INT 3** tek bayttır (CCh) ve kesme noktası için kullanılır.
- **INTR** düzey tetiklemelidir, yalnız IF = 1 iken ve komut sınırında kabul edilir; tip numarası ikinci INTA darbesinde 8259A'dan okunur. **NMI** yükselen kenarla tetiklenir, maskelenemez ve hep tip 2'dir.
- **Öncelik**: bölme hatası / INT n / INTO, sonra NMI, sonra INTR, en son tek adım. PC'de IRQ n'nin tipi 08h + n'dir; zamanlayıcı 08h, klavye 09h.
- **8259A**'da IRR bekleyen istekleri, IMR maskeleri (port 21h, bit 1 = engelli), ISR yazmacı hizmetteki IRQ'yu tutar. Donanım ISR'si sonunda EOI gönderir: `MOV AL, 20h` · `OUT 20h, AL`. IMR oku–değiştir–yaz kalıbıyla değiştirilir.
- SS yüklendikten ve STI'den sonra kesme bir komut ertelenir; REP'li dizgi komutları turlar arasında kesilebilir.

- **Kesme gecikmesi** = yürüyen komutun kalan süresi + kabul süresi (INTR için 61 tur). CLI ile kapalı geçen süre gecikmeye eklenir; CLI–STI arası ve ISR kısa tutulur.
- Kendi ISR'si kurulurken kullanılmayan bir tip seçilir ve vektör CLI/STI arasında (ya da INT 21h, AH = 25h ile) yazılır. Eski vektör saklanır ve çıkışta geri yüklenir. ISR kullandığı yazmaçları korur ve IRET ile biter.

5.1.9 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

kesme *interrupt* · kesme hizmet yordamı (ISR) *interrupt service routine* · yoklama *polling* · donanım / yazılım kesmesi *hardware / software interrupt* · istisna *exception* · kesme vektörü *interrupt vector* · kesme vektör tablosu (IVT) *interrupt vector table* · kesme bayrağı (IF) *interrupt flag* · tuzak bayrağı (TF) *trap flag* · tek adım *single-step* · kesme noktası (INT 3) *breakpoint* · maskelenebilir / maskelenemez kesme *maskable / non-maskable interrupt* · kesme onayı (INTA) *interrupt acknowledge* · programlanabilir kesme denetleyicisi (8259A) *programmable interrupt controller* · kesme maske yazmacı (IMR) *interrupt mask register* · kesme sonu (EOI) *end of interrupt* · kesme gecikmesi *interrupt latency* · yeniden girişli *reentrant* · yığın *stack* · bağıl konum *offset* · fiziksel adres *physical address*

5.1.10 Kendini Dene

1. Bir bellek dökümünde 0000:0020h adresinden başlayarak `A5 FE 00 F0` baytları görülüyor. Hangi tipin vektörüdür, PC'de hangi aygıta aittir? ISR'nin fiziksel adresi nedir?

   Cevap

   20h / 4 = 08h: tip 08h, yani IRQ0'a bağlı zamanlayıcı kesmesi. IP = FEA5h, CS = F000h. Fiziksel adres F0000h + FEA5h = FFEA5h'dir.

2. 0700:0200 adresindeki `INT 21h` çalışmadan önce SP = FFF0h ve FLAGS = F246h'dir. ISR'nin ilk komutunda SP, yığındaki üç sözcük ve FLAGS nedir?

   Cevap

   SP = FFF0h − 6 = FFEAh. [FFEAh] = 0202h (INT 21h iki bayt), [FFECh] = 0700h, [FFEEh] = F246h. Yazmaçtaki FLAGS'te IF (bit 9) silinir: F246h − 0200h = F046h. TF zaten 0'dır.

3. BL = 2 iken `DIV BL`, AX = 01FEh için ve AX = 0200h için ne sonuç verir?

   Cevap

   510 / 2 = 255 = FFh AL'ye sığar: AL = FFh, AH = 00h. 512 / 2 = 256 sığmaz; bölen sıfır olmadığı hâlde INT 0 üretilir. Kısa sınama: ikinci durumda AH = 02h ≥ BL = 02h'dir.

4. IMR = 00h iken zamanlayıcı ve klavye kesmelerini birlikte maskeleyen komut dizisi nedir? Yeni IMR kaçtır? Bu sırada bir tuşa basılırsa ne olur?

   Cevap

   `IN AL, 21h` · `OR AL, 03h` · `OUT 21h, AL`; IMR = 03h olur. Basılan tuşun isteği 8259A'nın IRR'sinde bekler; maske kaldırılınca hizmet görür.

5. 8 MHz'lik bir 8086'da INTR geldiğinde en çok 184 turluk bir IDIV yeni başlamışsa en kötü kesme gecikmesi kaç µs'dir?

   Cevap

   184 + 61 = 245 tur. 8 MHz'de bir tur 125 ns'dir: 245 × 125 ns ≈ 30,6 µs.

6. Bir program yığını taşımak için önce `MOV SP, 0FFFEh`, sonra `MOV SS, AX` yazıyor. Bu sıra neden güvenli değildir?

   Cevap

   Erteleme yalnız kesim yazmacına yazan komuttan sonraki komutu korur. İki komut arasında, yani MOV SP'den sonra kesme gelebilir. O anda SP yeni, SS eskidir ve FLAGS, CS, IP yanlış bir yere itilir. Doğru sıra önce `MOV SS`, hemen ardından `MOV SP`'dir.

7. Bir donanım ISR'si BX ve DX'i kullanıyor ama yalnız BX'i yığına itip geri alıyor. Ana programda ne gibi bir belirti görülür?

   Cevap

   Kesme programın herhangi bir komutundan sonra gelebildiği için DX, ana programın bilmediği anlarda değişir. Program çoğu zaman doğru çalışır, ama kesmenin tam DX kullanılırken geldiği seyrek anlarda yanlış sonuç verir. Hata tekrarlanması zor, rastgele görünen bir hatadır.

8. Hata ayıklayıcı 0150h adresindeki tek baytlık bir komutun yerine CCh yazmıştır. INT 3 yığına

hangi IP'yi iter? Hata ayıklayıcı bu değeri neden değiştirir?

Cevap

INT 3 tek bayt olduğu için itilen IP 0151h'dir. Hata ayıklayıcı saklanan asıl baytı 0150h'ye geri yazar ve yığındaki IP'yi 1 azaltarak 0150h yapar. IRET sonrasında asıl komut baştan, bozulmamış hâliyle yürütülür.

9. Bir program CLI ile kesmeleri kapatmışken bellek eşlik hatası oluşuyor. Kesme kabul edilir mi? Vektör hangi adresten okunur?

Cevap

Eşlik hatası PC'de NMI ile bildirilir. NMI maskelenemez; IF = 0 iken de kabul edilir. Tipi sabit 2'dir ve vektör 2 × 4 = 00008h–0000Bh baytlarından okunur. INTA döngüsü yapılmaz.

5.1.11 İleri okuma

- Intel, *The 8086 Family User's Manual* (1979), Interrupts kesimi: kesme türleri, öncelik sırası, INTR ve NMI zamanlaması ve komutların saat turu sayıları bu kaynaktadır. Tablo 5.1.6 ve 5.1.16'daki değerler buradan alınmıştır.
- B. B. Brey, *The Intel Microprocessors* (8. baskı), Bölüm 12: gerçek kipte kesmeler, 8259A'nın programlanması ve basamaklanması, PC'deki kesme düzeni.
- A. S. Tanenbaum, *Modern Operating Systems* (3. baskı): kesmelerin işletim sistemindeki yeri, aygıt sürücüleri ve kesme ile yoklamanın karşılaştırılması.
- R. Brown, *Ralf Brown's Interrupt List*: PC'deki bütün BIOS, DOS ve sürücü kesmelerinin numara numara başvuru listesi.

5.1.12 Kaynaklar

1. A. S. Tanenbaum, *Modern Operating Systems*, 3. baskı, Pearson, 2008.
2. Intel, *The 8086 Family User's Manual*, 1979 — Interrupts bölümü.
3. B. B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson, 2009 — Bölüm 6 ve 12.
4. emu8086 yardım belgeleri: “Interrupts” ve “Supported interrupts” sayfaları.
5. R. Brown, *Ralf Brown's Interrupt List* (BIOS/DOS kesme başvurusu).

KISIM II · BÖLÜM 5.2

Kesme Fonksiyonları

Bir 8086 programı ekrana yazı yazmak, klavyeden tuş okumak ya da saati öğrenmek için donanımı her seferinde baştan sürmek zorunda değildir. Bilgisayarın ROM'undaki BIOS ve onun üstünde çalışan DOS, bu işleri hazır yordamlar olarak sunar. Program bu yordamlara bir yazılım kesmesiyle ulaşır: AH yazmacına hizmet numarasını, öteki yazmaçlara parametreleri yazar ve `INT n` komutunu çalıştırır. Not, en sık kullanılan dört kesmenin hizmetlerini ele alır: INT 10h ekranı, INT 16h klavyeyi, INT 21h DOS'un karakter, dizgi, tarih ve saat hizmetlerini, INT 1Ah ise saat sayacını yönetir. Ekran belleğine BIOS'suz doğrudan yazma, BIOS veri alanı ve programı doğru sonlandırma yolları da işlenir. Her hizmet, kısa bir programın adım adım izlemesiyle yazmaç düzeyinde gösterilir.

ÖĞRENME ÇIKTILARI

Hizmet çağırmak
BIOS ve DOS hizmetlerini AH'deki hizmet numarası ve yazmaç parametreleriyle çağırmak; hangi yazmaçların değiştiğini önceden bilmek.

Ekrana yazmak
INT 10h ve INT 21h ile karakter, renkli karakter ve dizgi yazdırmak; imleci konumlamak; öznitelik baytını kurmak; B800h ve A000h belleğine doğrudan yazmak.

Klavyeden okumak
INT 16h ve INT 21h ile tuş okumak, tarama kodunu ASCII kodundan ayırmak, tuş beklemeden yoklamak ve tamponlu satır girişi almak.

Saati okumak
Tarih ve saati DOS'tan ikili, RTC'den paketli BCD olarak okuyup ekrana yazdırmak; BIOS veri alanındaki tık sayacıyla süre ölçmek.

Programı sonlandırmak
Programı RET, INT 20h ya da INT 21h/4Ch ile doğru biçimde bitirmek; COM ve EXE şablonlarının farkını açıklamak.

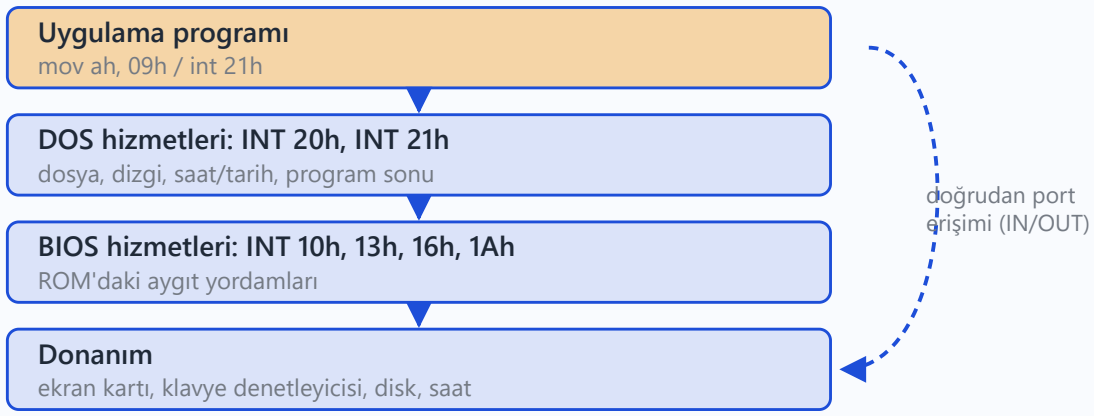
Program izlemek
Hizmet kullanan bir programı adım adım izleyip yazmaç, bayrak, bellek ve ekran durumunu her adımda belirlemek.

5.2.1 BIOS ve DOS hizmetleri

5.2.1.1 Hizmet katmanları

Ekrana tek bir karakter yazmak, donanım düzeyinde birkaç ayrı iş demektir. Karakterin kodu, ekran kartının belleğinde doğru hücreye yazılır. Hücrenin renk baytı ayarlanır. İmlecin yeni konumu ekran kartındaki denetleyiciye bir G/Ç portu üzerinden bildirilir. Satır sonuna gelinmişse alt satıra geçilir, ekranın sonunda ise bütün satırlar bir yukarı kaydırılır. Her programın bu işleri kendisi yapması hem zahmetli hem de tehlikelidir. Ekran kartı değişince programların hepsi yeniden yazılmak zorunda kalır.

IBM PC uyumlu bilgisayarlar bu sorunu katmanlarla çözer. Şekil 5.2.1 dört katmanı gösterir. En altta donanım vardır: ekran kartı, klavye denetleyicisi, disk ve saat. Donanıma doğrudan G/Ç portları (`IN`, `OUT`) ve bellek adresleri üzerinden ulaşılır. Bir üstte BIOS durur. BIOS'un yordamları donanımın ayrıntısını bilir ve bunu sabit bir arayüzün arkasına saklar. Onun üstündeki DOS, dosya, dizgi, tarih ve program yönetimi gibi daha soyut hizmetler sunar. En üstte uygulama programı vardır; o yalnız bu hizmetleri çağırır.



Şekil 5.2.1: Hizmet katmanları. Uygulama DOS'u `INT 21h` ile, BIOS'u `INT 10h`, `INT 16h`, `INT 1Ah` gibi kesmelerle çağırır. DOS kendi işinin çoğunu BIOS'a yaptırır; BIOS donanıma IN/OUT komutlarıyla ve bellek erişimiyle ulaşır.

Tanım 5.2.1

BIOS (*Basic Input/Output System*, temel giriş/çıkış sistemi), anakarttaki ROM'da bulunan ve donanımı doğrudan süren yordamlar topluluğudur. İşletim sistemi olmadan da çalışır. Bilgisayar açılınca donanımı sınar, kesme vektör tablosuna kendi yordamlarının adreslerini yazar ve işletim sistemini diskten yükler. **DOS** (*Disk Operating System*) ise BIOS'un üstünde çalışan, dosya, bellek ve program yönetimi sunan işletim sistemidir.

Bir programın BIOS ya da DOS hizmetine ulaşmak için kullandığı araç **yazılım kesmesidir** (*software interrupt*). `INT n` komutu, işlemcinin bellekteki kesme vektör tablosundan n numaralı girişi okuyup oradaki adrese dallanmasını sağlar. Dönüş adresi ve bayraklar yığına itilir; hizmet yordamı işini bitirince `IRET` ile programa döner. Buradaki konu bu mekanizmanın kendisi değil, onun sunduğu hizmetlerdir. Önemli olan şudur: program hizmetin bellekteki adresini bilmek zorunda değildir. Yalnız numarasını bilir.

Adres yerine numara kullanmanın yararı büyüktür. Farklı üreticilerin BIOS'ları yordamlarını farklı adreslere yerleştirir. DOS'un hangi sürümünün yüklü olduğu da adresleri değiştirir. Vektör tablosu açılışta o makinenin gerçek adresleriyle doldurulur. Böylece `INT 10h` yazan bir program, her PC uyumlu makinede kendi ekran yordamına ulaşır. 8086 üzerinde `INT n` komutu 51 saat turu sürer; doğrudan uzak `CALL` 28 turdur. Bedel neredeyse iki kat süredir, karşılığı taşınabilirliktir.

Aynı iş farklı katmanlarda yapılabilir. Ekrana bir karakter yazmanın üç yolu vardır: DOS'un `INT 21h/02h` hizmeti, BIOS'un `INT 10h/0Eh` hizmeti ve ekran belleğine (B800h kesimi) doğrudan yazma. Üst katman daha taşınabilirdir; çıktısı bir dosyaya da yönlendirilebilir. Alt katman daha hızlı ve esnektir; renk ve konum üzerinde tam denetim verir. Program, işin gereğine göre katmanı seçer ve aynı programda katmanları karıştırabilir.

5.2.1.2 Hizmet çağırma kalıbı

Bir kesme numarası tek bir hizmete değil, bir hizmet ailesine karşılık gelir. `INT 21h` karakter yazmak, dizgi yazmak, dosya açmak, saati okumak ve programı bitirmek gibi yüzlerce iş yapar. Hangi işin yapılacağını kesmeden önce AH yazmacına yazılan sayı belirler. Bu sayıya **hizmet numarası** ya da **fonksiyon numarası** denir. AH 8 bitlik olduğu için bir kesmenin en çok 256 hizmeti olabilir.

> **Tanım 5.2.2**
>
> **Hizmet çağırma kalıbı** dört adımdan oluşur: (1) AH ← hizmet numarası; (2) öteki parametreler AL, BX, CX, DX, DS:DX ya da ES:BP gibi yazmaçlara yazılır; (3) `INT n` çalıştırılır; (4) sonuç yazmaçlardan (AL, AX, CX, DX) ya da bayraklardan (ZF, CF) okunur. Hangi yazmacın ne taşıdığı her hizmet için ayrı tanımlıdır.

En kısa örnek, DOS'a ekrana bir harf yazdırmaktır:

```
mov ah, 02h    ; hizmet: karakter yaz
mov dl, 'A'    ; parametre
int 21h        ; DOS'u çağır
```

Birinci satır AH'ye 02h yükler: DOS'un “karakter yaz” hizmeti. İkinci satır yazılacak karakterin ASCII kodunu, 41h'yi DL'ye koyar. Bu hizmet karakteri DL'de bekler. Üçüncü satır kesmeyi çağırır. İşlemci 0000:0084h adresindeki vektörü okur (21h × 4 = 84h), DOS'un yordamına dallanır. Yordam karakteri

imlecin bulunduğu yere yazar, imleci bir ilerletir ve `IRET` ile döner. Program, `int 21h`'den sonraki komuttan devam eder.

Bu kalıpta işlemcinin donanım düzeyinde yaptığı iş bellidir. `MOV AH, 02h` iki baytlık bir komuttur (`B4 02`); BIU onu önden getirme kuyruğundan alır, EU yazmaca yazar ve bellek yol döngüsü gerekmez. `INT 21h` ise yığına üç sözcük iter (FLAGS, CS, IP) ve vektör tablosundan iki sözcük okur: beş bellek yol döngüsü. Hizmet yordamının kendisi yüzlerce komuttur. Dolayısıyla tek karakter yazdırmak bile binlerce saat turu sürer. Bu süre kullanıcının fark edemeyeceği kadar kısadır, ama ekranı karakter karakter dolduran bir döngüde belirginleşir.

Tablo 5.2.1 sık kullanılan kesmeleri ve sağlayıcılarını listeler.

Kesme	Sağlayan	Görev
`INT 10h`	BIOS	video: mod, imleç, karakter, dizgi, piksel
`INT 13h`	BIOS	disk
`INT 16h`	BIOS	klavye
`INT 1Ah`	BIOS	saat sayacı, gerçek zamanlı saat
`INT 20h`	DOS	programı sonlandır (COM)
`INT 21h`	DOS	yüzlerce hizmet: karakter ve dizgi G/Ç, dosya, tarih ve saat, çıkış

Tablo 5.2.1: Sık kullanılan BIOS ve DOS kesmeleri.

Hizmetler kendi kullandıkları yazmaçları değiştirebilir. Bu, kalıbın en çok hata doğuran yanıdır. Örneğin `INT 21h/02h` dönüşte AL'ye DL'deki karakteri koyar; `INT 21h/09h` ise AL'yi 24h yapar. AL'de saklanan bir değer bu çağrılardan sonra kaybolur. Sonra gerekecek değerler, hizmetin dokunmadığı bir yazmaçta ya da yığında tutulur. Bazı hizmetler AL'yi de bir alt seçim için kullanır. VESA ekran kartı hizmetleri buna örnektir: `AX = 4F00h` ile çağrılan hizmette AH = 4Fh aileyi, AL = 00h o ailedeki işi seçer.

> **Sık yapılan hata**
>
> Okunan bir karakteri AL'de bırakıp araya başka bir `INT 21h` çağrısı koymak. Örneğin önce `AH = 01h` ile tuş okunur, sonra satır başı için `AH = 02h, DL = 0Dh` çağrılır. Bu çağrı AL'yi 0Dh yapar ve okunan tuş kaybolur. Okunan değer çağrıdan hemen sonra BL gibi başka bir yazmaca kopyalanmalıdır.

5.2.1.3 ASCII ve denetim karakterleri

Ekrana yazılan ve klavyeden okunan her karakter bir sayıdır. PC'de bu sayılar **ASCII** (*American Standard Code for Information Interchange*) kodlamasından gelir. ASCII 0–127 arasındaki kodlara karakter atar. 20h–7Eh arası basılabilir karakterlerdir: boşluk, rakamlar, harfler ve noktalama. 00h–1Fh arası ise **denetim karakterleridir** (*control characters*). Bunlar ekranda bir şekil olarak değil, bir eylem olarak anlam taşır.

Kod	Ad	Etki
07h	BEL	zil sesi
08h	BS	imleç bir geri (karakter silinmez)
09h	TAB	sekme (DOS hizmetlerinde)
0Ah	LF	imleç bir alt satıra
0Dh	CR	imleç satır başına; Enter tuşunun kodu
1Bh	ESC	Esc tuşunun kodu

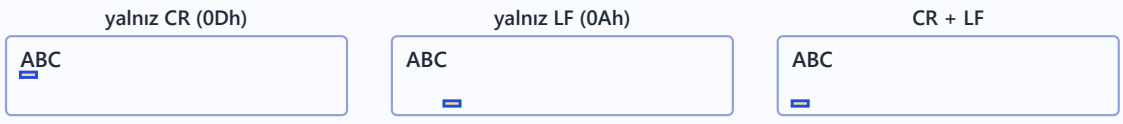
Tablo 5.2.2: Sık karşılaşılan denetim karakterleri ve INT 10h/0Eh ile INT 21h/02h'deki etkileri.

Basılabilir karakterlerin kodları rastgele dağılmamıştır; birkaç düzen, programları kısaltır. Tablo 5.2.3 bu düzenleri özetler. Rakamların kodu 30h'den başlar: '0' = 30h, '9' = 39h. Bu yüzden bir rakamın sayı değeri, kodundan 30h çıkarılarak bulunur. Tersine, 0–9 arasındaki bir sayı 30h eklenerek ekrana yazılabilir bir rakama dönüşür. Büyük harfler 41h–5Ah, küçük harfler 61h–7Ah arasındadır. İki grup arasındaki fark tam 20h'dir, yani tek bir bit: bit 5. Küçük harfi büyüğe çevirmek için bit 5 silinir (`SUB AL, 20h` ya da `AND AL, 0DFh`); büyüğü küçüğe çevirmek için kurulur (`OR AL, 20h`).

Karakter	ASCII	İlişki
boşluk	20h	basılabilir karakterlerin ilki
'0' … '9'	30h–39h	rakam = kod − 30h
'A' … 'Z'	41h–5Ah	bit 5 = 0
'a' … 'z'	61h–7Ah	bit 5 = 1 (+20h)
'$'	24h	INT 21h/09h hizmetinde dizgi sonu

Tablo 5.2.3: ASCII tablosundaki düzenler.

Yeni satıra geçmek iki ayrı karakter ister: CR (0Dh) ve LF (0Ah). Bu ayrım daktilodan kalmadır. CR, yazı kafasını satırın başına döndürür; LF, kâğıdı bir satır yukarı iter. Şekil 5.2.2 üç durumu karşılaştırır. Yalnız CR gönderilirse imleç satır başına döner ve sonraki yazı aynı satırın üzerine gelir. Yalnız LF gönderilirse imleç bir alt satıra iner ama sütunu değişmez; metin merdiven gibi sağa kayar. İkisi birlikte gönderilince imleç alt satırın başına gelir.



Şekil 5.2.2: CR ve LF'nin etkisi. Yalnız CR satırın başına döndürür; yalnız LF bir alt satıra indirir ama sütunu korur; CR LF birlikte yeni satır açar.

Türkçe harfler (ç, ğ, ı, ö, ş, ü) 7 bitlik ASCII'de yoktur. DOS bunları 128–255 arasındaki kodlarla, etkin **kod sayfasına** (*code page*) göre gösterir; Türkçe için bu sayfa CP857'dir. Aynı bayt başka bir kod sayfasında başka bir harf olarak görünebilir. Bu yüzden sunumdaki ve bu nottaki örnek programlar dizgilerde Türkçe harf kullanmaz: `'Cikis'`, `'Car'` gibi.

Örnek 5.2.1

AL = 37h ('7') ve BL = 32h ('2') iki rakam karakteridir. Toplamlarını tek basamak olarak ekrana yazmak için önce ikisinden de 30h çıkarılır: 07h ve 02h. Toplam 09h'dir. Ekrana yazmak için 30h eklenir: 39h = '9'. Karakterler doğrudan toplansaydı 37h + 32h = 69h = 'i' çıkardı. ASCII kodları sayı değil, karakter adıdır; aritmetikten önce sayıya çevrilir.

5.2.2 INT 10h: video hizmetleri

5.2.2.1 Hizmet tablosu ve video modları

Ekranla ilgili BIOS hizmetlerinin hepsi `INT 10h` altında toplanır. Ekran modunu seçmek, imleci taşımak, renkli karakter yazmak, bir dizgiyi tek çağrıyla basmak, ekranın bir bölgesini silmek ve grafik modda piksel çizmek bu hizmetlerle yapılır. Tablo 5.2.4 en sık kullanılanları, giriş ve çıkış yazmaçlarıyla verir.

AH	Hizmet	Giriş	Çıkış / açıklama
00h	video modu seç	AL = mod (03h: 80×25 renkli metin)	ekran temizlenir
01h	imleç şekli	CH = başlangıç, CL = bitiş satırı	CH = 20h → imleç gizli
02h	imleci konumla	BH = sayfa, DH = satır, DL = sütun	–
03h	imleç konumunu oku	BH = sayfa	DH, DL = konum; CH, CL = şekil
06h	pencereyi yukarı kaydır	AL = satır sayısı (0 = temizle), BH = öznitelik, CH,CL – DH,DL = köşeler	ekranı ya da bir pencereyi siler
09h	karakter ve öznitelik yaz	AL = karakter, BH = sayfa, BL = öznitelik, CX = tekrar	imleç ilerlemez
0Ah	yalnız karakter yaz	AL, BH, CX	hücrenin özniteliği korunur
0Ch	piksel yaz	AL = renk, BH = sayfa, CX = x, DX = y	grafik modlarda
0Eh	teletype çıkışı	AL = karakter, BH = sayfa	imleç ilerler; CR, LF, BS, BEL yorumlanır

AH	Hizmet	Giriş	Çıkış / açıklama
13h	dizgi yaz	AL = mod, BH, BL, CX = uzunluk, DH,DL = konum, ES:BP = dizgi	AT ve sonraki BIOS'larda

Tablo 5.2.4: INT 10h video hizmetleri.

Ekran kartı farklı **video modlarında** (*video mode*) çalışabilir. Metin modlarında ekran, her biri bir karakter tutan hücrelerden oluşur. Grafik modlarında ise ekran piksellerden oluşur ve her pikselin rengi ayrı ayrı yazılır. Mod, `AH = 00h` hizmetiyle AL'deki numara verilerek seçilir. Tablo 5.2.5 sık kullanılan modları ve görüntü belleğinin hangi kesimde başladığını gösterir.

AL	Mod	Görüntü belleği
01h	40×25 metin, 16 renk	B800h
03h	80×25 metin, 16 renk	B800h
07h	80×25 tek renk	B000h
04h	320×200 grafik, 4 renk	B800h
13h	320×200 grafik, 256 renk	A000h

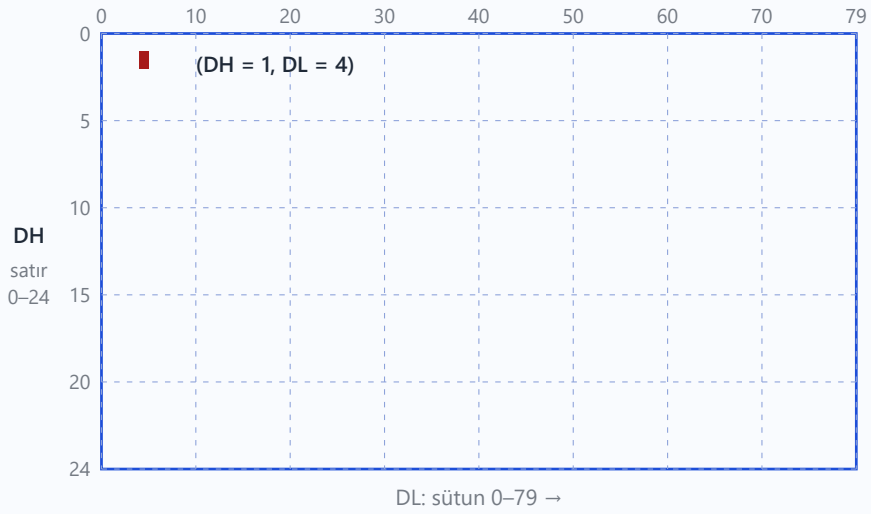
Tablo 5.2.5: Sık kullanılan video modları.

Mod seçmek ekranı da temizler. Bu yüzden programların başında sık görülen iki satır, temiz bir 80×25 ekran elde etmenin en kısa yoludur:

```
mov ah, 00h
mov al, 03h    ; ya da mov ax, 0003h
int 10h        ; temiz 80×25 ekran
```

AH ve AL tek bir `MOV AX, 0003h` komutuyla birlikte yüklenebilir. AH, AX'in yüksek baytı olduğundan 00h, AL düşük baytı olduğundan 03h olur. Üç baytlık bu komut, iki baytlık iki ayrı `MOV`'dan hem kısa hem hızlıdır.

Mod 03h'te ekran 25 satır ve 80 sütundur (Şekil 5.2.3). Satırlar 0'dan 24'e, sütunlar 0'dan 79'a numaralanır. Sol üst köşe (0, 0), sağ alt köşe (24, 79)'dur. BIOS hizmetlerinde satır numarası DH'de, sütun numarası DL'de verilir. emu8086'nın ekran penceresi varsayılan olarak bu moddadır; grafik modlarını sınırlı destekler.



Şekil 5.2.3: 80×25 metin ekranı. Satır DH'de (0–24), sütun DL'de (0–79) verilir; işaretli hücre (DH = 1, DL = 4) konumudur.

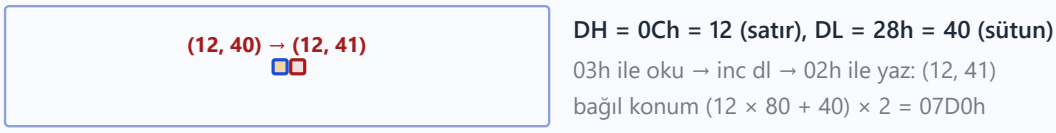
5.2.2.2 İmleç

İmleç (*cursor*), ekranda bir sonraki karakterin yazılacağı konumdur. BIOS her görüntü sayfası için bir imleç konumu tutar. `AH = 02h` imleci DH satırına ve DL sütununa taşır; `AH = 03h` imlecin konumunu DH ve DL'ye okur. İki hizmet de BH'de bir **görüntü sayfası** numarası bekler. 80×25 modunda ekran kartının belleği birden çok ekranı tutacak kadar büyüktür; VGA'da 8 sayfa (0–7) vardır. Sayfalardan yalnız biri görünür. Normalde sayfa 0 kullanılır.

Satır ve sütun, tek bir `MOV DX` komutuyla birlikte yüklenebilir. `mov dx, 0C28h` DH'ye 0Ch = 12, DL'ye 28h = 40 yazar: ekranın ortası. Aşağıdaki kod önce imleci ortaya alır, sonra konumu okuyup bir sağa kaydırır.

```
; imleci ortaya al
mov ah, 02h
mov bh, 0
mov dx, 0C28h  ; 12. satır, 40. sütun
int 10h
; imleci bir sağa kaydır
mov ah, 03h
mov bh, 0
int 10h        ; DH, DL = konum
inc dl
mov ah, 02h
int 10h
```

İlk dört satır imleci (12, 40)'a taşır. Sonraki üç satır konumu okur; dönüşte DH = 12, DL = 40 olur ve CH, CL'ye imlecin şekli yazılır. `INC DL` sütunu 41 yapar. Son iki satır yeni konumu yazar. BH, 03h çağrısından sonra değişmediği için yeniden yüklenmez. Şekil 5.2.4 bu adımları ve hücrenin ekran belleğindeki yerini gösterir.



Şekil 5.2.4: `mov dx, 0C28h` imleci (12, 40)'a koyar; 03h ile okunup `inc dl` ile artırılan konum 02h ile yazılınca imleç (12, 41)'e geçer.

İmleç konumu yalnız BIOS'un bir değişkeni değildir; ekran kartının donanımında da tutulur. Renkli metin modunda ekran denetleyicisinin (CRTC) iki yazmacı, imlecin hangi hücrede yanıp söneceğini belirler. BIOS bu yazmaçlara 3D4h (indis) ve 3D5h (veri) G/Ç portları üzerinden `OUT` komutlarıyla yazar. `AH = 02h` çağrısı, programı bu port ayrıntısından kurtarır. Aynı konum BIOS veri alanında da saklanır; `INT 21h` ile yazılan karakterler de bu imleç konumundan başlar.

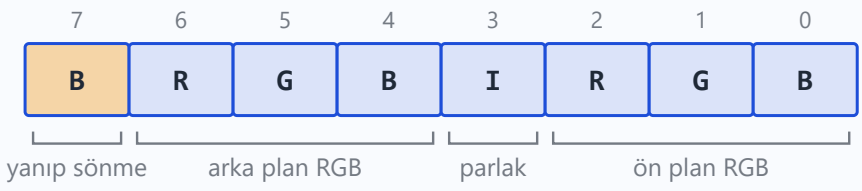
Her hücrenin ekran belleğinde de bir yeri vardır. Mod 03h'te hücre başına 2 bayt kullanılır ve satırlar art arda dizilir. (satır, sütun) hücresinin bağıl konumu şöyle hesaplanır:

$$\text{bağıl konum} = (\text{satır} \times 80 + \text{sütun}) \times 2$$

(12, 40) için (960 + 40) × 2 = 2000 = 07D0h bulunur. (1, 4) için (80 + 4) × 2 = 168 = 00A8h'dir. Çift adreste karakterin kodu, onu izleyen tek adreste rengi durur.

5.2.2.3 Öznitelik baytı

Metin modunda her hücrenin rengi bir baytla belirlenir. Bu bayta **öznitelik baytı** (*attribute byte*) denir. Şekil 5.2.5 baytın bitlerini gösterir. Düşük dört bit (bit 3–0) ön planın, yani yazının rengidir. Bit 2–0 kırmızı, yeşil ve mavi bileşenleri açar; bit 3 rengi parlaklaştırır. Bit 6–4 arka planın rengidir: yine kırmızı, yeşil, mavi. Bit 7 ise karakterin yanıp sönmesini açar.



Şekil 5.2.5: Öznitelik baytı: bit 7 yanıp sönme, bit 6–4 arka plan (RGB), bit 3 parlaklık, bit 2–0 ön plan (RGB).

1Eh = 0001 1110b mavi zemin üzerine sarı yazıdır.

Tanım 5.2.3

Öznitelik baytı, metin modunda bir karakter hücresinin ön ve arka plan rengini ve yanıp sönmesini belirleyen bayttır. Onaltılık yazıldığında yüksek basamak arka planı, düşük basamak ön planı verir: öznitelik = arka plan × 16 + ön plan. Bit 7 = 1 ise karakter yanıp söner (BIOS ayarına göre arka plan parlak da gösterilebilir).

On altı renk, dört bitin bütün birleşimlerinden çıkar. Tablo 5.2.6 numaraları ve adları verir. Parlak renkler (8–F), koyu karşılıklarına bit 3'ün eklenmesiyle oluşur: 1 mavi, 9 açık mavi; 4 kırmızı, C açık kırmızı. İki istisna dikkat çeker: 6 sarı değil kahverengidir, parlak hâli E ise sarıdır. 7 açık gri, F beyazdır.

No	Renk	No	Renk
0	siyah	8	koyu gri
1	mavi	9	açık mavi
2	yeşil	A	açık yeşil
3	camgöbeği	B	açık camgöbeği
4	kırmızı	C	açık kırmızı
5	eflatun	D	açık eflatun
6	kahverengi	E	sarı
7	açık gri	F	beyaz

Tablo 5.2.6: Metin modunun on altı rengi.

Örnek 5.2.2

Varsayılan öznitelik 07h'dir: arka plan 0 (siyah), ön plan 7 (açık gri). Mavi zemin üzerine beyaz yazı için arka plan 1, ön plan F seçilir: 1 × 16 + 15 = 31 = 1Fh. Kırmızı zemin üzerine sarı yazı 4Eh'dir. 8Fh = 1000 1111b ise bit 7 kurulu olduğu için siyah zemin üzerinde yanıp sönen beyaz yazıdır. Arka plana yalnız üç bit ayrıldığından açık yeşil (A) zemin doğrudan seçilemez: A0h yazılırsa en üstteki 1 biti bit 7'ye düşer ve yeşil zemin üzerinde yanıp sönme açılır.

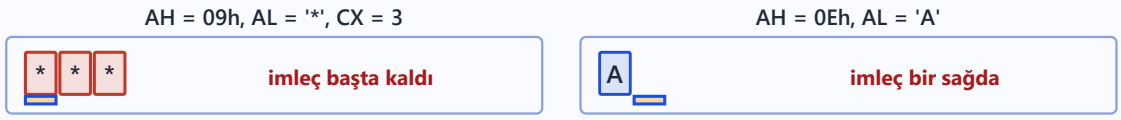
Sunumdaki öznitelik gezgini, ön plan ve arka plan seçildiğinde öznitelik baytını ve onu kullanan `INT 10h/09h` çağrısını üretir. Aynı hesap elle de kolaydır: iki renk numarası onaltılık iki basamak olarak yan yana yazılır.

5.2.2.4 Karakter yazma: 09h ve 0Eh

INT 10h karakter yazmak için iki ana hizmet sunar ve ikisi çok farklı davranır. `AH = 09h` karakteri AL'den, rengini BL'den alır ve aynı karakteri CX kez yan yana yazar. İmleci ise ilerletmez. `AH = 0Eh`, **teletype** (daktilo) çıkışıdır: karakteri imlecin olduğu yere yazar ve imleci bir ilerletir. Satır sonunda alt satıra geçer, son satırda ekranı yukarı kaydırır. CR, LF, BS ve BEL karakterlerini de komut olarak yorumlar. Tablo 5.2.7 farkları yan yana koyar.

Özellik	AH = 09h	AH = 0Eh (teletype)
Girişler	AL = karakter, BH = sayfa, BL = öznitelik, CX = tekrar	AL = karakter, BH = sayfa
Renk	var (BL)	metin modunda yok; hücrenin rengi korunur
İmleç	ilerlemez	ilerler; satır sonunda alt satıra geçer
CR, LF, BS, BEL	sembol olarak basılır (♪, ◘ ...)	komut olarak yorumlanır
Ekran kaydırma	yok	son satırda ekran yukarı kayar
Tipik kullanım	renkli kutular, çerçeveler, aynı karakteri CX kez	düz metin akışı

Tablo 5.2.7: AH = 09h ile AH = 0Eh karşılaştırması.



Şekil 5.2.6: 09h üç yıldızı yazar ama imleç ilk hücrede kalır; 0Eh 'A'yı yazar ve imleci bir sağa taşır.

09h'nin imleci ilerletmemesi ilk bakışta tuhaf görünür, ama hizmetin amacına uygundur. Bir çerçevenin üst kenarını çizmek için aynı karakter yan yana CX kez yazılır; imlecin nereye gideceğine program karar verir. Renkli düz metin yazmak için iki yaygın yol vardır. Birincisinde her karakter 09h ile CX = 1 olarak renkli yazılır, sonra imleç 02h ile bir sağa taşınır. İkincisinde bütün dizgi tek çağrıda 13h ile yazılır.

> **Sık yapılan hata**
>
> 09h ile art arda karakter yazıp imleci taşımamak. Her yeni karakter bir öncekinin üzerine yazılır ve ekranda yalnız sonuncusu görünür. 09h'den sonra imleç ya `AH = 02h` ile elle ilerletilir ya da renkli hücrenin üzerine 0Eh ile yazılır.

5.2.2.5 Kod izleme: mod, imleç, 09h ve 0Eh

Aşağıdaki program dört hizmeti art arda kullanır. Önce ekranı temizler, imleci (1, 4)'e taşır, oraya kırmızı zemin üzerine sarı üç yıldız yazar, sonra teletype ile 'A' ve 'B' yazar. Programın amacı, 09h ile 0Eh'nin imleç ve renk üzerindeki farkını bir arada göstermektir.

```
org 100h
mov ax, 0003h    ; AH=00h, AL=03h
int 10h          ; ekran silinir
mov ah, 02h      ; imleç konumla
mov bh, 0        ; sayfa 0
mov dx, 0104h    ; DH=1, DL=4
int 10h
mov ah, 09h      ; karakter + öznitelik
mov al, '*'
mov bl, 4Eh      ; kırmızı üstüne sarı
mov cx, 3        ; 3 kez
int 10h          ; imleç ilerlemez!
mov ah, 0Eh      ; teletype
mov al, 'A'
int 10h          ; imleç ilerler
mov al, 'B'
int 10h
ret
```

Program bir .COM programıdır: `org 100h` ile başlar, başlangıçta genel yazmaçlar sıfırdır. İlk iki satır mod 03h'ü kurar. Ekran silinir ve imleç (0, 0)'a gelir. Sonraki dört satır imleci taşır. `mov dx, 0104h` komutu DH'ye 01h, DL'ye 04h yazar. Little-endian yerleşim nedeniyle komutun baytları `BA 04 01`'dir. `int 10h`'den sonra imleç (1, 4)'tedir ama ekranda henüz bir şey yoktur.

Ardından 09h çağrısı hazırlanır. AL = 2Ah ('*'), BL = 4Eh, CX = 3 yüklenir. BH hâlâ 0'dır; yeniden yazmak gerekmez. Çağrıdan sonra (1, 4), (1, 5) ve (1, 6) hücrelerinde kırmızı zemin üzerine sarı yıldızlar vardır. İmleç ise yerinden kıpırdamamıştır: hâlâ (1, 4).

Son kısımda AH = 0Eh yüklenir ve AL'ye 'A' yazılır. `INT 10h`, 'A'yı imlecin bulunduğu (1, 4)'e yazar; yani ilk yıldızın üzerine. Sonra imleci (1, 5)'e taşır. 0Eh hizmeti AH'yi değiştirmediği için ikinci karakterde yalnız AL değiştirilir. 'B', (1, 5)'teki ikinci yıldızın üzerine yazılır ve imleç (1, 6)'ya geçer. Ekranda `AB*` görünür. 0Eh metin modunda öznitelik almadığı için 'A' ve 'B', yıldızlardan kalan 4Eh rengini korur: kırmızı zemin üzerinde sarı.

Adım	Komut	AX	BX	CX	DX	İmleç	Açıklama
1	`mov ax, 0003h`	0003h	0000h	0000h	0000h	(0, 0)	AH = 00h (mod seç), AL = 03h (80×25 metin)
2	`int 10h`	0003h	0000h	0000h	0000h	(0, 0)	mod kurulur, ekran silinir, imleç (0, 0)
3	`mov ah, 02h`	0203h	0000h	0000h	0000h	(0, 0)	AH = 02h: imleci konumla
4	`mov bh, 0`	0203h	0000h	0000h	0000h	(0, 0)	BH = sayfa 0
5	`mov dx, 0104h`	0203h	0000h	0000h	0104h	(0, 0)	DH = 1 (satır), DL = 4 (sütun)
6	`int 10h`	0203h	0000h	0000h	0104h	(1, 4)	imleç (1, 4)'e taşındı; ekran boş
7	`mov ah, 09h`	0903h	0000h	0000h	0104h	(1, 4)	AH = 09h: karakter + öznitelik
8	`mov al, '*'`	092Ah	0000h	0000h	0104h	(1, 4)	AL = '*' = 2Ah
9	`mov bl, 4Eh`	092Ah	004Eh	0000h	0104h	(1, 4)	BL = 4Eh: kırmızı zemin, sarı yazı
10	`mov cx, 3`	092Ah	004Eh	0003h	0104h	(1, 4)	CX = 3 tekrar
11	`int 10h`	092Ah	004Eh	0003h	0104h	(1, 4)	(1, 4)–(1, 6) hücrelerine `***`; imleç yerinde

Adım	Komut	AX	BX	CX	DX	İmleç	Açıklama
12	`mov ah, 0Eh`	0E2Ah	004Eh	0003h	0104h	(1, 4)	AH = 0Eh: teletype
13	`mov al, 'A'`	0E41h	004Eh	0003h	0104h	(1, 4)	AL = 'A'
14	`int 10h`	0E41h	004Eh	0003h	0104h	(1, 5)	'A' ilk yıldızın üzerine; imleç ilerledi
15	`mov al, 'B'`	0E42h	004Eh	0003h	0104h	(1, 5)	AL = 'B'
16	`int 10h`	0E42h	004Eh	0003h	0104h	(1, 6)	'B' ikinci yıldızın üzerine: ekranda `AB*`
17	`ret`	0E42h	004Eh	0003h	0104h	(1, 6)	yığından 0000h → IP
18	`INT 20h (PSP:0000h)`	0E42h	004Eh	0003h	0104h	(1, 6)	PSP'deki INT 20h programı bitirir

Tablo 5.2.8: Mod, imleç, 09h ve 0Eh programının izlemesi (her satır, komut yürütüldükten sonraki durum).

İzleme, BIOS hizmetlerinin yazmaçlara dokunma biçimini de gösterir. Hiçbir çağrı AX, BX, CX ya da DX'i değiştirmedi; yalnız ekranı ve imleci değiştirdi. INT 21h hizmetlerinin aksine bu INT 10h hizmetleri AL'yi korur. Yine de her hizmetin belgesine bakmadan bu güvene dayanmamak gerekir. `AH = 03h` gibi okuma hizmetleri dönüşte DX ve CX'i bilerek değiştirir.

5.2.2.6 Dizgi yazma: 13h

`AH = 13h`, bir dizgiyi tek çağrıda ve renkli yazar. AT ve sonraki bilgisayarların BIOS'unda bulunur. Parametreleri çoktur: AL yazma modunu, BH sayfayı, BL özniteliği, CX dizginin uzunluğunu, DH ve DL başlangıç konumunu verir. Dizginin adresi ise **ES:BP**'dedir. Bu, DOS'un dizgi hizmetinden (DS:DX) farklıdır ve sık karıştırılır. Dizginin sonunda '$' gibi bir işaret aranmaz; uzunluk CX'ten alınır.

AL	Dizginin biçimi	İmleç
00h	yalnız karakterler; hepsi BL özniteliğinde	yerinde kalır
01h	yalnız karakterler; hepsi BL özniteliğinde	dizginin sonuna ilerler
02h	karakter, öznitelik çiftleri	yerinde kalır
03h	karakter, öznitelik çiftleri	dizginin sonuna ilerler

Tablo 5.2.9: AH = 13h yazma modları (AL).

```
org 100h
mov ax, 1301h    ; AH=13h, AL=01h
mov bx, 001Eh    ; BH=0, BL=1Eh
mov cx, 7        ; dizgi uzunluğu
mov dx, 0205h    ; satır 2, sütun 5
lea bp, mesaj    ; ES:BP → dizgi
int 10h
ret
mesaj db 'Merhaba'
```

Program dört yazmaç çiftini birer komutla yükler. AX = 1301h, AH = 13h ve AL = 01h demektir: bütün karakterler BL renginde, imleç ilerler. BX = 001Eh, BH = 0 sayfası ve BL = 1Eh rengidir; mavi zemin üzerinde sarı yazı. CX = 7, `'Merhaba'` dizgisinin uzunluğudur. DX = 0205h başlangıcı satır 2, sütun

5'e koyar. `LEA BP, mesaj`, dizginin bağıl konumunu BP'ye yükler.

Dizginin bağıl konumu, önceki komutların bayt sayısından bulunur. Üç baytlık dört `MOV` 0100h–010Bh arasını kaplar. emu8086, doğrudan adresli `LEA BP, mesaj` komutunu `MOV BP, OFFSET mesaj` ile aynı üç baytlık koda (`BD 12 01`) derler; komut 010Ch–010Eh'dedir. İki baytlık `INT 10h` 010Fh–0110h'de, bir baytlık `RET` 0111h'dedir. `mesaj` bu yüzden 0112h'de başlar. Bir .COM programında bütün kesim yazmaçları aynı değeri gösterir, dolayısıyla ES = DS'dir ve ES:BP dizgiyi doğru gösterir.

Adım	Komut	AX	BX	CX	DX	BP	İmleç	Açıklama
1	`mov ax, 1301h`	1301h	0000h	0000h	0000h	0000h	(0, 0)	AH = 13h, AL = 01h (öznitelik BL'de, imleç ilerler)
2	`mov bx, 001Eh`	1301h	001Eh	0000h	0000h	0000h	(0, 0)	BH = 0 (sayfa), BL = 1Eh (mavi zemin, sarı yazı)
3	`mov cx, 7`	1301h	001Eh	0007h	0000h	0000h	(0, 0)	dizgi uzunluğu 7
4	`mov dx, 0205h`	1301h	001Eh	0007h	0205h	0000h	(0, 0)	DH = 2, DL = 5: başlangıç konumu
5	`lea bp, mesaj`	1301h	001Eh	0007h	0205h	0112h	(0, 0)	BP = `mesaj`'ın bağıl konumu; ES = DS
6	`int 10h`	1301h	001Eh	0007h	0205h	0112h	(2, 12)	(2, 5)'ten başlayarak `Merhaba`; imleç (2, 12)
7	`ret`	1301h	001Eh	0007h	0205h	0112h	(2, 12)	program biter
8	`INT 20h (PSP:0000h)`	1301h	001Eh	0007h	0205h	0112h	(2, 12)	

Tablo 5.2.10: AH = 13h ile renkli dizgi yazan programın izlemesi.

Çağrı, (2, 5)'ten başlayarak yedi hücreye `Merhaba` yazar ve AL = 01h olduğu için imleci (2, 12)'ye taşır: 5 + 7 = 12. AL = 00h olsaydı dizgi yine yazılır, ama imleç (2, 5)'te kalırdı. AL = 02h ya da 03h seçilseydi dizgi her karakterden sonra kendi öznitelik baytını taşımalıydı; o zaman BL kullanılmaz ve CX yine karakter sayısıdır.

Sık yapılan hata

13h'te dizgi adresini DS:DX'e koymak ve DX'i iki kez kullanmak. DX bu hizmette konumdur; `mov dx, offset mesaj` yazılırsa konum bozulur, BP ise rastgele bir yeri gösterir. Bir başka hata, CX'i vermeyip dizginin sonuna '$' koymaktır. 13h '$' tanımaz; CX'te ne yazıyorsa o kadar karakter basar.

5.2.2.7 Pencere kaydırma ve ekran temizleme: 06h

`AH = 06h`, ekranın dikdörtgen bir bölgesini, yani bir **pencereyi**, AL satır yukarı kaydırır. Pencerenin altında boşalan satırlar BH özniteliğiyle boşluk karakterine boyanır. AL = 0 verilirse kaydırma yerine pencerenin tamamı silinir. Bu özellik, ekranı mod değiştirmeden ve istenen renkte temizlemenin

standart yoludur. `AH = 07h` aynı işi aşağı doğru yapar.

Yazmaç	Anlamı
AL	kaydırılacak satır sayısı; 0 = pencereyi temizle
BH	boşalan satırların özniteliği
CH, CL	sol üst köşe (satır, sütun)
DH, DL	sağ alt köşe (satır, sütun)

Tablo 5.2.11: AH = 06h hizmetinin yazmaçları.

Aşağıdaki program önce bütün ekranı mavi zemine siler, sonra ortada kırmızı bir kutu açar.

```
org 100h
mov ax, 0600h    ; AH=06h, AL=0
mov bh, 1Fh      ; mavi/beyaz
mov cx, 0000h    ; (0,0)
mov dx, 184Fh    ; (24,79)
int 10h
mov ax, 0600h
mov bh, 4Fh      ; kırmızı/beyaz
mov cx, 0714h    ; (7,20)
mov dx, 103Bh    ; (16,59)
int 10h
ret
```

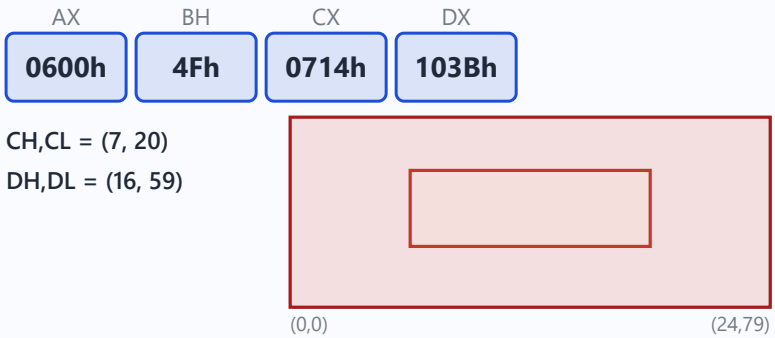
İlk çağrıda köşeler CX = 0000h ve DX = 184Fh'dir. 18h = 24 ve 4Fh = 79 olduğundan pencere bütün ekrandır. BH = 1Fh, silinen hücrelere mavi zemin ve beyaz yazı rengini verir. Çağrıdan sonra ekranın her hücresinde boşluk karakteri ve 1Fh özniteliği vardır. Sonra yazılacak karakterler, 0Eh ile yazılsalar bile, bu rengi alır. Mod 03h'ü yeniden seçmek de ekranı siler, ama renkleri varsayılan 07h'ye döndürür.

İkinci çağrıda AX yeniden 0600h yapılır. Bu gereklidir, çünkü INT 10h'nin bazı BIOS'larda AX'i koruyacağı garanti edilmez. CX = 0714h sol üst köşeyi (7, 20), DX = 103Bh sağ alt köşeyi (16, 59) yapar. Kutu 16 − 7 + 1 = 10 satır ve 59 − 20 + 1 = 40 sütundur. Yalnız bu bölge kırmızıya silinir; dışı mavi kalır. İki çağrı da imleci yerinden oynatmaz; temizlikten sonra imleç `AH = 02h` ile (0, 0)'a ya da kutunun içine taşınır.

Adım	Komut	AX	BX	CX	DX	Açıklama
1	`mov ax, 0600h`	0600h	0000h	0000h	0000h	AH = 06h (yukarı kaydır), AL = 0 (temizle)
2	`mov bh, 1Fh`	0600h	1F00h	0000h	0000h	BH = 1Fh: mavi zemin, beyaz yazı
3	`mov cx, 0000h`	0600h	1F00h	0000h	0000h	sol üst köşe (0, 0)
4	`mov dx, 184Fh`	0600h	1F00h	0000h	184Fh	sağ alt köşe (24, 79)
5	`int 10h`	0600h	1F00h	0000h	184Fh	tüm ekran mavi zemine silinir
6	`mov ax, 0600h`	0600h	1F00h	0000h	184Fh	ikinci çağrı için AH, AL yeniden
7	`mov bh, 4Fh`	0600h	4F00h	0000h	184Fh	BH = 4Fh: kırmızı zemin, beyaz yazı

Adım	Komut	AX	BX	CX	DX	Açıklama
8	`mov cx, 0714h`	0600h	4F00h	0714h	184Fh	sol üst (7, 20)
9	`mov dx, 103Bh`	0600h	4F00h	0714h	103Bh	sağ alt (16, 59)
10	`int 10h`	0600h	4F00h	0714h	103Bh	10 satır × 40 sütunluk kutu kırmızıya silinir
11	`ret`	0600h	4F00h	0714h	103Bh	program biter

Tablo 5.2.12: Ekranı ve bir pencereyi AH = 06h ile temizleyen programın izlemesi.



Şekil 5.2.7: İkinci çağrıdan sonra ekranın şeması: dış dikdörtgen 1Fh (mavi zemin) ile silinen 80 × 25 ekran, iç dikdörtgen 4Fh (kırmızı zemin) ile silinen (7, 20)–(16, 59) penceresidir. Renkler gerçek ekran renkleri değil, şemanın vurgu renkleridir.

Örnek 5.2.3

Ekranın alt yarısını (satır 13–24) siyah zemine temizlemek için AX = 0600h ve BH = 07h yüklenir. Köşeler CX = 0D00h (satır 13, sütun 0) ve DX = 184Fh (satır 24, sütun 79) olur. Bir bölgeyi tek satır yukarı kaydırmak için ise AL = 1 verilir. AX = 0601h, CX = 1400h, DX = 184Fh ile satır 20–24 penceresi bir satır yukarı kayar. Satır 24 BH renginde boşalır.

5.2.3 Görüntü belleğine doğrudan erişim

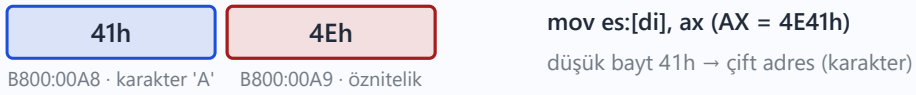
5.2.3.1 B800h metin belleği

INT 10h hizmetleri sonunda hep aynı işi yapar: ekran kartının belleğine bayt yazar. Bu bellek, işlemcinin adres uzayında sıradan RAM gibi görünür. 8086'nın 1 MB'lık adres uzayında A0000h–BFFFFh aralığı ekran kartlarına ayrılmıştır. Renkli metin modlarında görüntü belleği B8000h fiziksel adresinden başlar; program ona B800h kesimiyle ulaşır. Ekran kartının denetleyicisi bu belleği saniyede onlarca kez baştan sona okur ve her hücreyi ekrana çizer. Belleğe yazılan değer, bir sonraki taramada ekranda görünür.

Tanım 5.2.4

Görüntü belleği (*video memory*), ekran kartının ekranda gösterdiği içeriği tutan ve işlemcinin adres uzayına eşlenmiş bellektir. Mod 03h'te B800:0000h'dan başlayan 4000 bayttır (80 × 25 × 2). Her hücre iki bayt kaplar: çift adreste karakterin ASCII kodu, onu izleyen tek adreste öznitelik baytı.

Belleğe doğrudan yazmak, BIOS'u çağırmaktan çok daha hızlıdır. Bir `INT 10h` çağrısı yüzlerce komut yürütürken doğrudan yazma tek bir `MOV` komutudur. Bedeli de vardır. Doğrudan yazma imleci hareket ettirmez ve BIOS'un tuttuğu imleç bilgisini güncellemez. Çıktı DOS'un yönlendirmesine de girmez. Program, hücre adresini kendisi hesaplar ve ekran kartının hangi modda olduğunu bilmek zorundadır. emu8086 bu belleği destekler; yazılan hücreler ekran penceresinde görünür.



Şekil 5.2.8: Bir ekran hücresi iki bayttır. `mov es:[di], ax` ile AX = 4E41h yazılınca düşük bayt 41h ('A') çift adrese, yüksek bayt 4Eh (öznitelik) tek adrese gider.

Hücreye bir sözcük olarak yazmak, karakter ile rengini tek komutta yerleştirir. 8086 little-endian'dır: bir sözcüğün düşük baytı küçük adrese, yüksek baytı büyük adrese yazılır. Bu yüzden AX'in AL yarısına karakter, AH yarısına öznitelik konur. Şekil 5.2.8 bu yerleşimi gösterir. Sözcüğün adresi çift olduğundan 8086 bu yazmayı tek bir yol döngüsünde yapar: veri yolunun 16 bitinin hepsi kullanılır, BHE ve A0 hatları birlikte etkin olur. Tek adrese sözcük yazmak ise iki ayrı yol döngüsü gerektirirdi.

5.2.3.2 Kod izleme: B800h'e yazmak

Aşağıdaki program önce (1, 4) hücresine kırmızı zemin üzerine sarı bir 'A' koyar. Sonra `REP STOSW` ile ekranın ilk üç hücresini nokta karakteriyle doldurur. Sunumdaki biçimde ikinci kısım bütün ekranı doldurmak için CX = 2000 kullanır; izlenebilir kalsın diye burada CX = 3'tür.

```
org 100h
mov ax, 0B800h
mov es, ax
mov di, (1*80+4)*2  ; 00A8h
mov ax, 4E41h
mov es:[di], ax
xor di, di
mov ax, 072Eh
mov cx, 3           ; ilk 3 hücre
cld
rep stosw
ret
```

İlk iki satır ES'yi B800h yapar. Kesim yazmacına anlık değer doğrudan yazılamaz; `MOV ES, 0B800h` diye bir 8086 komutu yoktur. Bu yüzden değer önce AX'e alınır. Sayının başındaki 0, assembler'ın `B800h`'yi bir ad sanmaması içindir. Üçüncü satırdaki `(1*80+4)*2` ifadesi derleme anında hesaplanır; makine koduna yalnız sonucu, 00A8h girer. Program çalışırken çarpma yapılmaz.

`MOV ES:[DI], AX`, ES kesimini açıkça adlandırır. DI ile yapılan bellek erişiminde varsayılan kesim DS'dir; burada DS programın kesimini gösterdiği için kesim öneki (`26h` baytı) zorunludur. Komut çalışınca fiziksel adres B800h × 16 + 00A8h = B80A8h'dir. Bu adrese 41h, B80A9h adresine 4Eh yazılır. Karakter anında ekranda görünür, imleç ise yerinde kalır. 8086 kılavuzuna göre bu komut 9 + EA turudur; [DI] için EA 5 tur, kesim öneki 2 tur ekler: yaklaşık 16 saat turu. Aynı karakteri INT 10h ile yazmak bunun yüzlerce katı sürer.

İkinci kısımda `XOR DI, DI` DI'yi sıfırlar; bu, `MOV DI, 0`'dan bir bayt kısadır. AX = 072Eh, AL = 2Eh ('.') ve AH = 07h (açık gri) demektir. `CLD` yön bayrağını (DF) temizler, böylece dizgi komutu DI'yi artırarak ilerler. `REP STOSW` her turda ES:[DI] ← AX yazar, DI'yi 2 artırır ve CX'i 1 azaltır; CX sıfır olunca durur. Üç tur sonunda DI = 0006h ve CX = 0'dır.

Adım	Komut	AX	CX	DI	ES	DF	Açıklama
1	`mov ax, 0B800h`	B800h	0000h	0000h	0700h	0	kesim değeri önce AX'e
2	`mov es, ax`	B800h	0000h	0000h	B800h	0	ES = B800h: metin ekranının kesimi
3	`mov di, (1*80+4)*2`	B800h	0000h	00A8h	B800h	0	(1 × 80 + 4) × 2 = 168 = 00A8h
4	`mov ax, 4E41h`	4E41h	0000h	00A8h	B800h	0	AL = 41h ('A'), AH = 4Eh (öznitelik)
5	`mov es:[di], ax`	4E41h	0000h	00A8h	B800h	0	B80A8h ← 41h, B80A9h ← 4Eh; 'A' anında ekranda
6	`xor di, di`	4E41h	0000h	0000h	B800h	0	DI = 0: sol üst hücre
7	`mov ax, 072Eh`	072Eh	0000h	0000h	B800h	0	AL = 2Eh ('.'), AH = 07h
8	`mov cx, 3`	072Eh	0003h	0000h	B800h	0	üç hücre doldurulacak
9	`cld`	072Eh	0003h	0000h	B800h	0	DF = 0: DI artarak ilerler
10	`rep stosw`	072Eh	0000h	0006h	B800h	0	üç kez ES:[DI] ← AX, DI += 2
11	`ret`	072Eh	0000h	0006h	B800h	0	program biter

Tablo 5.2.13: Ekran belleğine doğrudan yazan programın izlemesi.



Şekil 5.2.9: Program bitince görüntü belleği: B800:0000h'dan itibaren üç kez `2E 07`, B800:00A8h'de `41 4E`. Ekranın ilk satırında üç nokta, ikinci satırın beşinci sütununda (sütun 4) 'A' görünür.

Bütün ekranı doldurmak için CX = 2000 verilir: 80 × 25 hücre. 8086'da `REP STOSW` yaklaşık 9 + 10 × CX saat turu sürer. 2000 hücre için bu 20.009 turdur; 5 MHz'lik bir işlemcide yaklaşık 4 ms eder. Aynı iş 2000 ayrı `INT 10h/0Eh` çağrısıyla yapılsaydı, kaydırma ve imleç güncellemesi yüzünden onlarca kat uzun sürerdi.

Sık yapılan hata

Karakter ve özniteliği ters yerleştirmek: `mov ax, 414Eh` yazılırsa düşük bayt 4Eh ('N') karakter, 41h ise kırmızı zemin üzerine mavi yazı rengi olur. Little-endian düzeninde karakter AL'de, öznitelik AH'dedir. İkinci yaygın hata, ES'yi ayarlamadan `MOV [DI], AX` yazmaktır. Kesim öneki olmadığında yazma DS kesimine, yani programın kendi verisinin üzerine gider.

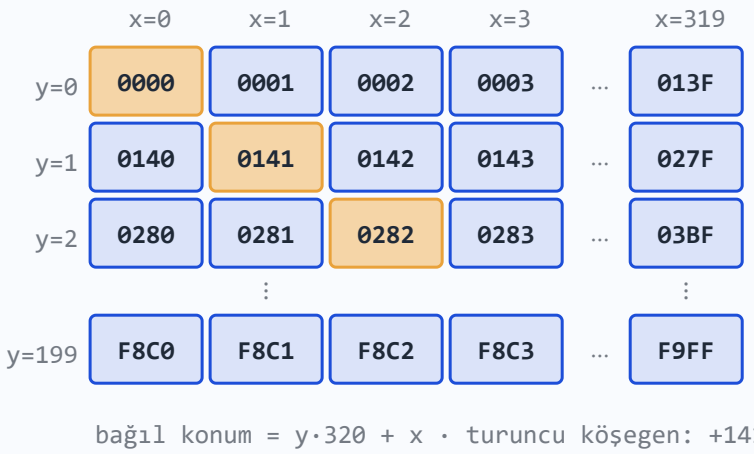
5.2.3.3 Mod 13h ve piksel adresi

Mod 13h, VGA'nın 320 × 200 çözünürlüklü, 256 renkli grafik modudur. `mov ax, 0013h` ve `int 10h` ile seçilir. Bu modun programcı için en büyük kolaylığı belleğin yapısıdır: her piksel tek bir bayttır ve pikseller satır satır art arda dizilir. Görüntü belleği A000:0000h'dan A000:F9FFh'ye kadar 64.000 bayttır (320 × 200). Baytın değeri pikselin renk numarasıdır; ekran kartı bu numarayı bir **palet** tablosunda gerçek renge çevirir. Varsayılan paletin ilk 16 rengi metin renkleriyle aynıdır: 0Eh sarı, 0Fh beyazdır.

Her satır 320 = 140h bayttır. (x, y) pikselinin bağıl konumu bu yüzden şöyle hesaplanır:

$$\text{bağıl konum} = y \times 320 + x$$

(100, 50) pikseli için 50 × 320 + 100 = 16.100 = 3EE4h bulunur. Son piksel (319, 199), 199 × 320 + 319 = 63.999 = F9FFh'dedir; belleğin son baytı. Şekil 5.2.10 ilk satırların ve son satırın bağıl konumlarını gösterir.



Şekil 5.2.10: Mod 13h'te piksel bağıl konumları. Her satır 140h bayttır; bir alt satırın aynı sütunu +140h, köşegende bir alt ve bir sağ +141h ileridedir.

BIOS bu modda da piksel hizmeti sunar. `AH = 0Ch`, AL'deki rengi CX sütunu (x, 0–319) ve DX satırındaki (y, 0–199) piksele yazar; BH sayfadır ve mod 13h'te 0'dır. `AH = 0Dh` aynı CX, DX ile pikselin rengini AL'ye okur. Koordinatla çalışmak kolaydır, ama her piksel için bir kesme çağrısı yavaştır.

Örnek 5.2.4

Ekranın ortasındaki (160, 100) pikselinin bağıl konumu 100 × 320 + 160 = 32.160 = 7DA0h'dir; fiziksel adresi A0000h + 7DA0h = A7DA0h olur. Sağ üst köşe (319, 0) ise 013Fh'de, sol alt köşe (0, 199) 199 × 320 = 63.680 = F8C0h'dedir.

5.2.3.4 Köşegen çizimi

Sol üst köşeden başlayan bir köşegen, (0, 0), (1, 1), (2, 2) ... piksellerinden oluşur. Önce BIOS ile koordinat kullanan sürüm:

```
mov ax, 0013h
int 10h          ; 320×200, 256 renk
xor cx, cx       ; x = 0
xor dx, dx       ; y = 0
mov bh, 0        ; sayfa
nokta: mov ax, 0C0Eh  ; AH=0Ch, AL=sarı
int 10h
inc cx           ; bir sağ
inc dx           ; bir alt
cmp dx, 200
jb nokta         ; 200 piksel
```

Döngü her turda AX'i yeniden yükler. Bunun nedeni, bazı BIOS'ların 0Ch hizmetinde AX'i değiştirmesidir; güvenli yol her çağrıdan önce AH ve AL'yi kurmaktır. CX ve DX birlikte artar. DX 200'e ulaşınca `JB` atlamaz ve döngü 200 piksel çizmiş olarak biter. Ekran 320 piksel geniş olduğu için köşegen sağ kenara değil, alt kenara ulaşır.

Aynı çizgi, A000h belleğine doğrudan yazarak da çizilir. Bu kez koordinat yerine bağıl konumla çalışılır:

```
mov ax, 0013h
int 10h
mov ax, 0A000h
mov es, ax
xor di, di       ; (0,0) → bağıl konum 0
mov al, 0Eh      ; sarı
mov cx, 200
ciz: mov es:[di], al
add di, 141h     ; 320 + 1
loop ciz
mov ah, 0
int 16h          ; tuş bekle
mov ax, 0003h
int 10h          ; metin moduna dön
```

Neden 141h eklenir? +140h (320) bir alt satırın aynı sütununa gider, +1 bir sağ sütuna. İkisi birlikte 141h = 321 eder ve köşegendeki bir sonraki pikseli verir. Son piksel (199, 199), 199 × 321 = 63.879 = F987h'dedir. Sağ üst köşeden inen ters köşegen için her adımda 13Fh (320 − 1) eklenir. Program sonunda bir tuş bekler, sonra metin moduna döner; böylece kullanıcı çizgiyi görebilir ve DOS istemi grafik ekranda kalmaz.

İki sürüm arasındaki hız farkı büyüktür. Doğrudan sürümde döngü gövdesi üç komuttur: bir bellek yazma, bir toplama ve LOOP. BIOS sürümünde her piksel için bir kesme çağrılır; kesme yordamı da koordinattan adresi kendisi hesaplar. emu8086'nın grafik desteği sınırlıdır: 0Ch hizmetiyle çizim yavaştır, A000h'a doğrudan yazma ise her sürümde görünmeyebilir. DOSBox gibi bir PC öykünücüsünde her iki sürüm de tam görünür.

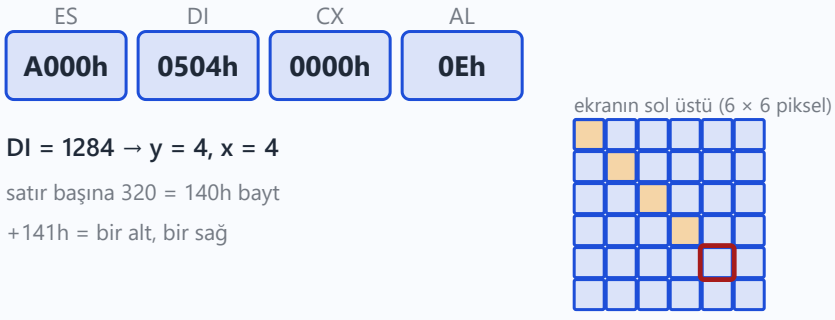
İzlemeyi kısa tutmak için aşağıdaki program yalnız dört piksel çizer ve mod değiştirmez. Döngünün her turunda DI'nin 141h arttığı açıkça görülür.

```
org 100h
mov ax, 0A000h
mov es, ax
xor di, di       ; (0,0) → bağıl konum 0
mov al, 0Eh      ; sarı
mov cx, 4        ; ilk 4 piksel
ciz: mov es:[di], al
add di, 141h     ; 320 + 1
loop ciz
ret
```

Adım	Komut	AL	CX	DI	ES	Açıklama
1	`mov ax, 0A000h`	00h	0000h	0000h	0700h	kesim değeri AX üzerinden
2	`mov es, ax`	00h	0000h	0000h	A000h	ES = A000h: mod 13h görüntü belleği
3	`xor di, di`	00h	0000h	0000h	A000h	DI = 0: piksel (0, 0)
4	`mov al, 0Eh`	0Eh	0000h	0000h	A000h	renk 0Eh (sarı)
5	`mov cx, 4`	0Eh	0004h	0000h	A000h	dört piksel
6	`mov es:[di], al`	0Eh	0004h	0000h	A000h	A0000h ← 0Eh: piksel (0, 0)
7	`add di, 141h`	0Eh	0004h	0141h	A000h	bir alt satır, bir sağ sütun
8	`loop ciz`	0Eh	0003h	0141h	A000h	CX = 3 ≠ 0: `ciz`'e dön
9	`mov es:[di], al`	0Eh	0003h	0141h	A000h	A0141h ← 0Eh: piksel (1, 1)
12	`mov es:[di], al`	0Eh	0002h	0282h	A000h	A0282h ← 0Eh: piksel (2, 2)
15	`mov es:[di], al`	0Eh	0001h	03C3h	A000h	A03C3h ← 0Eh: piksel (3, 3)
16	`add di, 141h`	0Eh	0001h	0504h	A000h	DI = 0504h = (4, 4)'ün yeri
17	`loop ciz`	0Eh	0000h	0504h	A000h	CX = 0: döngü biter
18	`ret`	0Eh	0000h	0504h	A000h	program biter

Tablo 5.2.14: A000h belleğine dört piksellik köşegen çizen programın izlemesi.

Tablodaki DI değerleri 0000h, 0141h, 0282h, 03C3h ve 0504h, 321'in katlarıdır: 0, 321, 642, 963 ve 1284. Her biri y × 320 + x biçiminde yazılınca x = y olduğu görülür; örneğin 963 = 3 × 320 + 3. Döngü bitince DI, çizilmemiş beşinci pikselin (4, 4) yerini gösterir. Fiziksel adresler A0000h, A0141h, A0282h ve A03C3h'tir.



Şekil 5.2.11: Döngü bitince ekranın sol üst 6 × 6 piksellik bölgesi: (0, 0)–(3, 3) sarı, DI = 0504h (1284) yani y

= 4, x = 4.

Sunumdaki piksel adresi gezgini, ekranda tıklanan noktanın (x, y) koordinatını, bağıl konumunu ve fiziksel adresini hesaplar. Köşegen ve ters köşegen düğmeleri, DI'ye 141h ya da 13Fh ekleyen döngüyü canlandırır.

5.2.4 INT 16h: klavye hizmetleri

5.2.4.1 Tarama kodu ve ASCII

Bir tuşa basıldığında klavyenin içindeki denetleyici, tuşun klavyedeki yerini gösteren bir sayı gönderir. PC'de bu sayı 60h G/Ç portundan okunur ve kesme denetleyicisi üzerinden IRQ 1 donanım kesmesini doğurur. BIOS'un klavye kesme yordamı (INT 09h) bu sayıyı alır, Shift ve Caps Lock durumuna bakarak ASCII karşılığını bulur. Sonra ikisini birlikte bir sözcük olarak bellekteki **klavye tamponuna** yazar. Program tuşları bu tampondan `INT 16h` ile okur. Böylece program meşgulken basılan tuşlar kaybolmaz; tamponda sıra bekler.

> **Tanım 5.2.5**
>
> **Tarama kodu** (*scan code*), klavyede basılan tuşun fiziksel konumunu gösteren koddur. Aynı tuş, Shift basılı olsa da olmasa da aynı tarama kodunu verir: 'a' ve 'A' için 1Eh. **ASCII kodu** ise tuşun o anki anlamını, yani üretilen karakteri gösterir: 'a' için 61h, 'A' için 41h.

AH	Hizmet	Çıkış
00h	tuş bekle ve oku	AH = tarama kodu, AL = ASCII; tuş tampondan silinir
01h	tuş var mı? (beklemez)	ZF = 1: tuş yok · ZF = 0: AX = tuş; tuş tampondan silinmez
02h	Shift durumları	AL bitleri: 0 sağ Shift, 1 sol Shift, 2 Ctrl, 3 Alt, 4 Scroll Lock, 5 Num Lock, 6 Caps Lock, 7 Insert

Tablo 5.2.15: INT 16h klavye hizmetleri.

`AH = 00h`, tamponda tuş yoksa bir tuşa basılana kadar bekler. Tuş gelince AH'ye tarama kodunu, AL'ye ASCII kodunu koyar ve tuşu tampondan siler. Tuş ekrana yansıtılmaz; ekranda bir şey görünmesi isteniyorsa program onu kendisi yazar. Tablo 5.2.16 bazı tuşların döndürdüğü değerleri listeler.

Tuş	AH (tarama)	AL (ASCII)	Tuş	AH (tarama)	AL
a / A	1Eh	61h / 41h	↑ / ↓	48h / 50h	00h
1	02h	31h	← / →	4Bh / 4Dh	00h
Enter	1Ch	0Dh	F1 … F10	3Bh … 44h	00h

Tuş	AH (tarama)	AL (ASCII)	Tuş	AH (tarama)	AL
Esc	01h	1Bh	Backspace	0Eh	08h
Boşluk	39h	20h	Ctrl+C	2Eh	03h

Tablo 5.2.16: Bazı tuşların INT 16h/00h dönüş değerleri (IBM PC/AT, ABD düzeni).

Ok tuşları, işlev tuşları, Home ve End gibi tuşların ASCII karşılığı yoktur. Bunlara **genişletilmiş tuşlar** denir ve INT 16h bunlar için AL = 00h döndürür. Program önce AL'ye bakar; AL sıfırsa tuşu AH'deki tarama kodundan tanır. F1 tuşu örneğin AH = 3Bh, AL = 00h verir. Ctrl ile basılan harfler ise ASCII'nin ilk 32 koduna düşer: Ctrl+A = 01h, Ctrl+C = 03h. Kodu, harfin kodunun düşük beş bitidir.

Program sonlarında sık görülen `MOV AH, 0` · `INT 16h` ikilisinin görevi basittir. Program bir tuş bekler, böylece çıktı ekranı kapanmadan okunabilir. Sunumdaki klavye gezgini, basılan bir tuşun INT 16h/00h ile hangi AH ve AL değerlerini döndüreceğini gösterir.

5.2.4.2 Kod izleme: tuş oku, büyük harfle yaz

Aşağıdaki program Enter'a basılana kadar tuş okur. Küçük harfleri büyüğe çevirip yazar, öteki karakterleri olduğu gibi yazar. Program INT 16h'nin yansıtma yapmadığını ve ASCII düzeninin bit 5 kuralını birlikte kullanır.

```
org 100h
oku:
mov ah, 0        ; tuş bekle
int 16h          ; AH=tarama, AL=ASCII
cmp al, 0Dh      ; Enter?
je bitti
cmp al, 'a'
jb yaz
cmp al, 'z'
ja yaz
sub al, 20h      ; küçük → büyük
yaz:
mov ah, 0Eh
int 10h          ; ekrana yaz
jmp oku
bitti:
ret
```

Döngü `oku` etiketinde başlar. `INT 16h` bir tuş bekler. `CMP AL, 0Dh` Enter'ı denetler; eşitse `JE` programı bitirir. Ardından iki karşılaştırma, AL'nin 'a'–'z' aralığında olup olmadığını sınar. `JB` ve `JA` işaretsiz karşılaştırma komutlarıdır: JB, CF = 1 iken; JA, CF = 0 ve ZF = 0 iken atlar. AL 'a'dan küçükse ya da 'z'den büyükse harf olmadığı için doğrudan `yaz`'a gidilir. Aralığın içindeyse `SUB AL, 20h` bit 5'i temizler. `yaz` kısmı karakteri teletype ile yazar ve döngü başına döner.

Kullanıcı sırayla Shift+o, k ve Enter tuşlarına basıyor. Tablo 5.2.17 önemli adımları verir; aynı turdaki benzer adımlar atlanmıştır.

Adım	Komut	AX	CF	ZF	Ekran (çıktı)	Açıklama
1	`mov ah, 0`	0000h	0	0	–	AH = 00h: tuş bekle ve oku
2	`int 16h`	184Fh	0	0	–	Shift+o: AH = 18h (tarama), AL = 4Fh ('O')
3	`cmp al, 0Dh`	184Fh	0	0	–	4Fh − 0Dh: sonuç sıfır değil
4	`je bitti`	184Fh	0	0	–	Enter değil, atlama yok
5	`cmp al, 'a'`	184Fh	1	0	–	4Fh < 61h: ödünç, CF = 1
6	`jb yaz`	184Fh	1	0	–	CF = 1: `yaz`'a atla
7	`mov ah, 0Eh`	0E4Fh	1	0	–	AH = 0Eh; AL'de 'O' duruyor
8	`int 10h`	0E4Fh	1	0	`O`	ekrana O
9	`jmp oku`	0E4Fh	1	0	`O`	döngü başına
11	`int 16h`	256Bh	1	0	`O`	k: AH = 25h, AL = 6Bh
14	`cmp al, 'a'`	256Bh	0	0	`O`	6Bh ≥ 61h: CF = 0
15	`jb yaz`	256Bh	0	0	`O`	atlama yok: küçük harf olabilir
16	`cmp al, 'z'`	256Bh	1	0	`O`	6Bh − 7Ah: ödünç, CF = 1, ZF = 0
17	`ja yaz`	256Bh	1	0	`O`	JA, CF = 0 ve ZF = 0 ister; atlamaz
18	`sub al, 20h`	254Bh	0	0	`O`	6Bh − 20h = 4Bh = 'K'
20	`int 10h`	0E4Bh	0	0	`OK`	ekrana K
23	`int 16h`	1C0Dh	0	0	`OK`	Enter: AH = 1Ch, AL = 0Dh
24	`cmp al, 0Dh`	1C0Dh	0	1	`OK`	0Dh − 0Dh = 0: ZF = 1
25	`je bitti`	1C0Dh	0	1	`OK`	`bitti`'ye atla
26	`ret`	1C0Dh	0	1	`OK`	program biter

Tablo 5.2.17: Tuş okuyup büyük harfle yazan programın izlemesi (basılan tuşlar: Shift+o, k, Enter).

İlk tuşta AL = 4Fh ('O') gelir. `CMP AL, 'a'` 4Fh − 61h hesaplar; sonuç negatiftir, ödünç gerekir ve CF = 1 olur. `JB` atlar: karakter küçük harf değildir ve olduğu gibi yazılır. İkinci tuşta AL = 6Bh ('k') gelir. 6Bh − 61h ödünç istemez, CF = 0 olur ve `JB` atlamaz. 6Bh − 7Ah ise ödünç ister: CF = 1, ZF = 0. JA'nın atlaması için hem CF hem ZF sıfır olmalıydı; atlamaz. Böylece AL'nin 61h ≤ AL ≤ 7Ah aralığında olduğu kesinleşir. `SUB AL, 20h` 6Bh'yi 4Bh'ye, yani 'K'ye çevirir. Enter ile gelen 0Dh, ilk karşılaştırmada ZF = 1 yapar ve program biter.

Tablonun AX sütununda bir ayrıntı daha görülür. Teletype çağrısından önce AH = 0Eh yapılır; döngü başında `MOV AH, 0` yeniden AH'yi sıfırlar ama AL, son yazılan karakteri korur. INT 16h dönüşünde ise AH ile AL'nin ikisi de yeni tuşla değişir. Ekranda sonuçta `OK` görünür. Enter'ın kendisi yansıtılmadığı için imleç satırda kalır.

5.2.4.3 Beklemeden yoklama

`AH = 00h` tuş gelene kadar programı durdurur. Bir oyun ya da animasyon ise tuş beklerken çalışmayı sürdürmek zorundadır. Bunun için `AH = 01h` kullanılır: hizmet hemen döner ve tuş olup olmadığını ZF ile bildirir. ZF = 1 ise tamponda tuş yoktur. ZF = 0 ise tuş vardır ve AX'te durur, ama tampondan si-

linmez. Program tuşu gerçekten almak için ardından `AH = 00h` çağırır. Bir aygıtın durumunu döngü içinde tekrar tekrar sormaya **yoklama** (*polling*) denir.

```
org 100h
xor dx, dx       ; iş sayacı
bekle:
inc dx           ; arka plan işi
mov ah, 01h      ; tuş var mı?
int 16h          ; ZF=1 → yok
jz bekle
mov ah, 00h      ; tuşu tampondan al
int 16h
ret
```

DX, bekleme sırasında yapılan işi temsil eder; her turda bir artar. Gerçek bir programda burada bir animasyon karesi çizilir ya da bir hesap ilerletilir. İzlemede ilk iki yoklamada tuş yoktur, üçüncüsünde kullanıcı x tuşuna basmıştır.

Adım	Komut	AX	DX	ZF	Açıklama
1	`xor dx, dx`	0000h	0000h	1	iş sayacı sıfırlanır
2	`inc dx`	0000h	0001h	0	arka plan işi: 1
3	`mov ah, 01h`	0100h	0001h	0	AH = 01h: tuş var mı?
4	`int 16h`	0100h	0001h	1	tampon boş: ZF = 1
5	`jz bekle`	0100h	0001h	1	ZF = 1: `bekle`'ye dön
6	`inc dx`	0100h	0002h	0	arka plan işi: 2
8	`int 16h`	0100h	0002h	1	yine tuş yok
10	`inc dx`	0100h	0003h	0	arka plan işi: 3
12	`int 16h`	2D78h	0003h	0	x var: ZF = 0, AX = 2D78h; tuş tamponda kalır
13	`jz bekle`	2D78h	0003h	0	ZF = 0: döngüden çık
14	`mov ah, 00h`	0078h	0003h	0	AH = 00h: tuşu al
15	`int 16h`	2D78h	0003h	0	aynı tuş okunur ve tampondan silinir
16	`ret`	2D78h	0003h	0	program biter

Tablo 5.2.18: AH = 01h ile tuş yoklayan döngünün izlemesi (üçüncü yoklamada x basılı).

İlk iki turda INT 16h ZF = 1 döndürür ve `JZ` döngüyü sürdürür. Program donmaz: her turda DX artar. Üçüncü turda tuş gelmiştir. INT 16h ZF = 0 yapar ve AX'e 2D78h koyar: tarama kodu 2Dh, ASCII 78h ('x'). Tuş tamponda kalmaya devam eder. `JZ` atlamaz, döngüden çıkılır. `AH = 00h` aynı tuşu okur ve bu kez tampondan siler; AX yine 2D78h'dir. Programın sonunda DX = 3, yani tuş beklenirken arka plan işi üç kez yapılmıştır.

Sık yapılan hata

01h ile tuşu görüp 00h ile almamak. 01h tuşu tampondan silmediği için döngünün sonraki turunda aynı tuş yine "var" görünür; program tek bir tuşa sonsuz kez tepki verir. Tuşa yanıt verildikten sonra `AH = 00h` ile tampon boşaltılır.

5.2.5 INT 21h: DOS karakter ve dizgi hizmetleri

5.2.5.1 Sık kullanılan hizmetler

`INT 21h`, DOS'un ana hizmet kesmesidir. Yüzlerce hizmetin içinden karakter ve dizgi G/Ç'si, tarih, saat ve program sonlandırma en sık kullanılanlardır. Tablo 5.2.19 bunları özetler.

AH	Hizmet	Giriş	Çıkış
01h	karakter oku ve yansıt	–	AL = karakter
02h	karakter yaz	DL = karakter	AL = DL
07h / 08h	karakter oku, yansıtmadan	–	AL (08h Ctrl+C'yi denetler)
09h	dizgi yaz	DS:DX → '$' ile biten dizgi	AL = 24h
0Ah	tamponlu satır oku	DS:DX → tampon	[tampon+1] = sayı, tampon+2 = metin
2Ah	tarihi al	–	CX = yıl, DH = ay, DL = gün, AL = haftanın günü (0 = Pazar)
2Ch	saati al	–	CH = saat, CL = dakika, DH = saniye, DL = salise
2Bh / 2Dh	tarihi / saati ayarla	2Ah / 2Ch ile aynı yazmaçlar	AL = 00h başarılı
4Ch	programı sonlandır	AL = çıkış kodu	(dönmez)

Tablo 5.2.19: Sık kullanılan INT 21h hizmetleri.

DOS hizmetlerinin BIOS hizmetlerinden önemli bir farkı, **G/Ç yönlendirmesine** uymalarıdır. Program `program > cikti.txt` biçiminde çalıştırılırsa 02h ve 09h ile yazılan her şey ekran yerine dosyaya gider. BIOS'un INT 10h çıktısı ise her zaman ekrana gider; DOS onu göremez. Benzer biçimde 01h ve 0Ah, girdiyi bir dosyadan da alabilir.

5.2.5.2 INT 10h/0Eh mi, INT 21h/02h mi?

Ekrana tek karakter yazan iki hizmet çok benzer görünür. İkisi de karakteri imlecin olduğu yere yazar ve imleci ilerletir; CR, LF gibi denetim karakterlerini yorumlar. Farkları, hangi katmanda çalıştıklarından gelir. Tablo 5.2.20 karşılaştırır.

Özellik	BIOS · INT 10h, AH = 0Eh	DOS · INT 21h, AH = 02h
Karakter	AL'de	DL'de
Gereken yazılım	yalnız video BIOS'u; açılış (boot) kodunda da çalışır	DOS yüklü olmalı
Yönlendirme	yok; hep ekrana	var; dosyaya, yazıcıya
Renk	grafik modda BL ile	yok
Dönüşte AL	korunur	AL = DL
Ctrl+Break denetimi	yok	var

Tablo 5.2.20: INT 10h/0Eh ile INT 21h/02h karşılaştırması.

Seçim, programın hangi ortamda çalışacağına bağlıdır. Bir önyükleme yükleyicisi DOS'tan önce çalıştığı için yalnız BIOS'u kullanabilir. Çıktısı bir dosyaya kaydedilecek bir araç ise DOS hizmetini seçmelidir. Aynı programda ikisi karıştırılabilir; ikisi de BIOS'un tuttuğu aynı imleç konumunu kullanır.

5.2.5.3 Kod izleme: karakter oku ve yaz

Aşağıdaki program klavyeden bir karakter okur, yeni bir satıra geçer ve okunan karakterden sonraki harfi yazar. Kısa olmasına karşın AL'nin hizmetler arasında nasıl bozulduğunu açıkça gösterir.

```
org 100h
mov ah, 01h      ; oku + yansıt
int 21h          ; AL = karakter
mov bl, al       ; sakla
mov ah, 02h      ; karakter yaz
mov dl, 0Dh      ; CR: satır başı
int 21h
mov dl, 0Ah      ; LF: alt satır
int 21h
inc bl           ; sonraki harf
mov dl, bl
int 21h
ret
```

Adım	Komut	AX	BX	DX	Ekran (çıktı)	Açıklama
1	`mov ah, 01h`	0100h	0000h	0000h	–	AH = 01h: oku ve yansıt
2	`int 21h`	016Bh	0000h	0000h	`k`	k: AL = 6Bh; ekranda k
3	`mov bl, al`	016Bh	006Bh	0000h	`k`	okunan karakter BL'de saklanır
4	`mov ah, 02h`	026Bh	006Bh	0000h	`k`	AH = 02h: DL'yi yaz
5	`mov dl, 0Dh`	026Bh	006Bh	000Dh	`k`	DL = 0Dh (CR)
6	`int 21h`	020Dh	006Bh	000Dh	`k⏎`	imleç satır başına; AL = 0Dh
7	`mov dl, 0Ah`	020Dh	006Bh	000Ah	`k⏎`	DL = 0Ah (LF)
8	`int 21h`	020Ah	006Bh	000Ah	`k↵`	imleç alt satıra; AL = 0Ah
9	`inc bl`	020Ah	006Ch	000Ah	`k↵`	BL = 6Ch ('l')
10	`mov dl, bl`	020Ah	006Ch	006Ch	`k↵`	DL = 'l'

Adım	Komut	AX	BX	DX	Ekran (çıktı)	Açıklama
11	`int 21h`	026Ch	006Ch	006Ch	`k↵l`	ekrana l; AL = 6Ch
12	`ret`	026Ch	006Ch	006Ch	`k↵l`	program biter

Tablo 5.2.21: Karakter okuyup bir sonrakini yeni satıra yazan programın izlemesi (basılan tuş: k). ⏎ CR, ↵ CR LF demektir.

`AH = 01h` bir tuş bekler ve tuşu ekrana yansıtır; kullanıcı k'ye bastığında ekranda k görünür ve AL = 6Bh olur. Okunan değer hemen BL'ye kopyalanır, çünkü sonraki `INT 21h` çağrıları AL'yi değiştirecektir. Gerçekten de CR yazıldıktan sonra AL = 0Dh, LF yazıldıktan sonra AL = 0Ah olur. 02h hizmeti dönüşte AL'ye DL'yi koyar. AH ise 02h olarak kalır; bu yüzden üç çağrı için bir kez yüklenmesi yeter. CR imleci satır başına, LF bir alt satıra götürür. `INC BL` 6Bh'yi 6Ch'ye, 'k'yi 'l'ye çevirir ve son çağrı bu harfi yeni satıra yazar. Ekranın son durumu iki satırdır: üstte yansıtılan k, altta l.

5.2.5.4 Kod izleme: dizgi yazdırma

`AH = 09h`, DS:DX'in gösterdiği dizgiyi '$' (24h) karakterini görene kadar yazar. '$' sonlandırıcıdır; kendisi yazılmaz. Dizginin içindeki 0Dh ve 0Ah satırı bitirir, yani denetim karakterleri burada da yorumlanır.

> **Tanım 5.2.6**
>
> **'$' ile biten dizgi**, DOS'un 09h hizmetinin beklediği biçimdir: karakterler bellekte art arda durur ve ilk 24h baytında dizgi biter. Uzunluk ayrıca verilmez. Hizmet dönüşte AL'yi 24h yapar.

```
org 100h
mov dx, offset msg1  ; DS:DX → dizgi
mov ah, 09h
int 21h
mov dx, offset msg2
int 21h              ; AH hâlâ 09h
ret
msg1 db 'Merhaba, 8086!', 0Dh, 0Ah, '$'
msg2 db 'Gule gule$'
```

Kod kısmı 13 bayttır (0100h–010Ch). `msg1` bu yüzden 010Dh'de başlar. `msg1` 14 karakter, CR, LF ve '$' olmak üzere 17 bayttır; `msg2` 010Dh + 17 = 011Eh'dedir. `OFFSET` işleci, değişkenin içeriğini değil bağıl konumunu verir.

Adım	Komut	AX	DX	Ekran (çıktı)	Açıklama
1	`mov dx, offset msg1`	0000h	010Dh	–	DS:DX → `msg1` (010Dh)
2	`mov ah, 09h`	0900h	010Dh	–	AH = 09h
3	`int 21h`	0924h	010Dh	`Merhaba, 8086!↵`	'$'a kadar yazılır; AL = 24h

Adım	Komut	AX	DX	Ekran (çıktı)	Açıklama
4	`mov dx, offset msg2`	0924h	011Eh	`Merhaba, 8086!↵`	DX → `msg2` (011Eh)
5	`int 21h`	0924h	011Eh	`Merhaba, 8086!↵Gule gule`	AH hâlâ 09h: ikinci dizgi
6	`ret`	0924h	011Eh	`Merhaba, 8086!↵Gule gule`	program biter

Tablo 5.2.22: AH = 09h ile iki dizgi yazan programın izlemesi.

İlk çağrı `Merhaba, 8086!` yazar ve CR LF ile yeni satıra geçer. AL = 24h olur. İkinci çağrıdan önce yalnız DX değiştirilir; AH hâlâ 09h'dir. Yine de her çağrıdan önce AH'yi yüklemek, programa sonradan eklenen bir satırın AH'yi bozma riskine karşı iyi bir alışkanlıktır. Çıktı iki satırdır. '$' karakterinin kendisi bu hizmetle yazdırılamaz; onu yazmak için 02h hizmeti ve DL = 24h kullanılır.

Sık yapılan hata

Dizginin sonuna '$' koymayı unutmak. 09h bellekte ilk 24h baytını bulana kadar ne varsa yazar: programın kodu, başka değişkenler, rastgele baytlar. Ekran çöple dolar, hatta denetim karakterleri yüzünden zil çalar. Tersi de olur: `'Fiyat: 5$ TL$'` dizgisi yalnız `Fiyat: 5` olarak yazılır, çünkü ilk '$' dizgiyi bitirir.

5.2.5.5 Tamponlu satır girişi

Kullanıcıdan bir ad ya da bir sayı almak, tuşları tek tek okumaktan fazlasını gerektirir. Kullanıcı yanlış yazınca Backspace ile siler, Enter'a basınca satırı bitirir. `AH = 0Ah` bu düzenlemeyi DOS'a yaptırır. Program yalnız bir tampon hazırlar ve adresini DS:DX'e koyar. DOS, Enter'a basılana kadar tuşları okur, yansıtır, Backspace ve Esc ile düzeltmeye izin verir. Bu sürede program çalışmaz; satır bitince çağrı döner.

Tanım 5.2.7

0Ah tamponu üç kısımdan oluşur. Bayt 0'a program kapasiteyi yazar: Enter dahil en çok kaç karakter alınacağı. Bayt 1'e DOS okunan karakter sayısını yazar (Enter sayılmaz). Bayt 2'den başlayarak metin gelir ve sonuna Enter'ın kodu 0Dh eklenir. Kapasite 0 olamaz.

Şekil 5.2.12, `tampon db 10, ?, 10 dup(?)` tanımlı bir tampona Ali yazılıp Enter'a basıldıktan sonraki durumu gösterir. Bayt 0 = 0Ah (10 karakter), bayt 1 = 03h, sonra 'A', 'l', 'i' ve 0Dh. Metin '$' ile bitmez; 09h ile yazdırmak için program 0Dh'nin yerine '$' koyar. Tampon dolduğunda DOS fazla tuşları kabul etmez ve zil çalar; yalnız Enter ve Backspace çalışmaya devam eder.



Şekil 5.2.12: Enter'dan sonra 0Ah tamponu: +0 kapasite (0Ah), +1 okunan sayı (03h), +2'den itibaren metin ve sonunda 0Dh.

Aşağıdaki program kullanıcının adını okur ve ona selam verir.

```
org 100h
mov dx, offset tampon
mov ah, 0Ah          ; tamponlu giriş
int 21h
mov bl, [tampon+1]   ; okunan sayı
mov bh, 0
mov byte ptr tampon[bx+2], '$'
mov dx, offset selam
mov ah, 09h
int 21h
lea dx, [tampon+2]
int 21h
ret
tampon db 10, ?, 10 dup(?)
selam  db 0Dh, 0Ah, 'Merhaba $'
```

Adım	Komut	AX	BX	DX	Ekran (çıktı)	Açıklama
1	`mov dx, offset tampon`	0000h	0000h	011Fh	–	DS:DX → tampon (011Fh)
2	`mov ah, 0Ah`	0A00h	0000h	011Fh	–	AH = 0Ah: satır oku
3	`int 21h`	0A00h	0000h	011Fh	`Ali⏎`	tampon = 0Ah, 03h, 'A', 'l', 'i', 0Dh
4	`mov bl, [tampon+1]`	0A00h	0003h	011Fh	`Ali⏎`	BL = [0120h] = 3
5	`mov bh, 0`	0A00h	0003h	011Fh	`Ali⏎`	BX = 0003h
6	`mov byte ptr tampon[bx+2], '$'`	0A00h	0003h	011Fh	`Ali⏎`	[011Fh + 3 + 2] = [0124h] ← '$'
7	`mov dx, offset selam`	0A00h	0003h	012Bh	`Ali⏎`	DX → `selam` (012Bh)
8	`mov ah, 09h`	0900h	0003h	012Bh	`Ali⏎`	AH = 09h
9	`int 21h`	0924h	0003h	012Bh	`Ali⏎↲Merhaba`	CR LF ve `Merhaba` yazılır
10	`lea dx, [tampon+2]`	0924h	0003h	0121h	`Ali⏎↲Merhaba`	DX = 0121h: metnin başı
11	`int 21h`	0924h	0003h	0121h	`Ali⏎↲Merhaba Ali`	`Ali` yazılır
12	`ret`	0924h	0003h	0121h	`Ali⏎↲Merhaba Ali`	program biter

Tablo 5.2.23: Ad okuyup selam veren programın izlemesi (yazılan: Ali, Enter).

Programın kodu 31 bayttır (0100h–011Eh). emu8086, doğrudan adresli `LEA DX, [tampon+2]` komutunu `MOV DX, OFFSET tampon+2` ile aynı üç baytlık koda (`BA 21 01`) derler; `tampon` bu yüzden 011Fh'de, `selam` 012Bh'de başlar. Çağrı dönünce tampon 011Fh'den itibaren `0A 03 41 6C 69 0D`

baytlarını tutar. `MOV BL, [tampon+1]` okunan sayıyı, 3'ü alır. BH sıfırlanır, çünkü sayı adres hesabında BX olarak kullanılacaktır. `tampon[bx+2]` adresi 011Fh + 3 + 2 = 0124h'tür; tam olarak 0Dh'nin bulunduğu bayt. Oraya '$' yazılır ve metin 09h ile yazılabilir hâle gelir. `BYTE PTR` belirteci burada açıklık içindir. `tampon` DB ile tanımlandığı için assembler yazmanın bir bayt olduğunu zaten bilir; değişken adı geçmeyen `[BX+2]` gibi bir adreste ise belirteç zorunludur.

Sonra `selam` yazılır: başta CR LF olduğu için yeni satıra geçilir ve `Merhaba` basılır. Enter'ın kendisi yalnız CR yansıttığından imleç satır başındaydı; LF onu alt satıra indirir. `LEA DX, [tampon+2]` metnin başını, 0121h'yi DX'e yükler. AH hâlâ 09h olduğundan son çağrı `Ali` yazar. Ekranda `Merhaba Ali` görünür.

Örnek 5.2.5

`tampon db 6, ?, 6 dup(?)` tanımlı bir tampona kullanıcı Mikro12 yazıp Enter'a basıyor. Kapasite Enter dahil 6 olduğundan en çok 5 karakter alınır. 'M', 'i', 'k', 'r', 'o' kabul edilir; '1' ve '2' tuşlarında zil çalar. Enter gelince tampon `06 05 4D 69 6B 72 6F 0D` olur. Okunan sayı 5'tir; '$' tampon + 5 + 2 = tampon + 7 adresine yazılır ve ekrana yalnız `Mikro` basılır.

Sık yapılan hata

Tamponun ilk baytını ayarlamamak ya da metni tampon+0'dan okumak. Kapasite 0 kalırsa hizmet beklenmedik biçimde davranır. Metin her zaman tampon+2'den başlar; tampon+0 ve tampon+1 sayılardır ve ekrana yazılırsa anlamsız karakterler çıkar.

5.2.6 Tarih ve saat

5.2.6.1 AH = 2Ah ve AH = 2Ch

DOS, sistemin tarihini ve saatini iki hizmetle verir. `AH = 2Ah` tarihi, `AH = 2Ch` saati döndürür. Tablo 5.2.24 dönüş yazmaçlarını gösterir. Değerlerin hepsi **ikili** sayıdır. Saat 14 ise CH = 0Eh, dakika 35 ise CL = 23h gelir. Yıl 16 bitlik CX'e sığar: 2026 = 07EAh.

Yazmaç	AH = 2Ah (tarih)	AH = 2Ch (saat)
CX / CH	CX = yıl (ör. 07EAh = 2026)	CH = saat (0–23)
CL	(CX'in parçası)	CL = dakika (0–59)
DH	ay (1–12)	saniye (0–59)
DL	gün (1–31)	salise (0–99)
AL	haftanın günü (0 = Pazar … 6 = Cumartesi)	–

Tablo 5.2.24: INT 21h/2Ah ve 2Ch dönüş değerleri.

İkili değerleri ekrana yazmak için her sayı onluk basamaklarına ayrılır. 0–99 arasındaki bir sayı için en kısa yol `AAM` komutudur. AAM, AL'yi 10'a böler; bölümü AH'ye, kalanı AL'ye yazar. 14 için AH = 1, AL = 4 olur. Ardından iki basamağa 30h eklenir ve ASCII rakamlar elde edilir. Yıl gibi 99'dan büyük sayılar için 10'a tekrarlı bölme kullanılır. BIOS'un `INT 1Ah/02h` hizmeti ise saati ikili değil, BCD olarak verir ve orada AAM kullanılmaz.

Örnek 5.2.6

AL = 23h (35) iken `AAM` çalışır: 35 ÷ 10 = 3, kalan 5. AH = 03h, AL = 05h, yani AX = 0305h. `ADD AX, 3030h` iki baytı birden ASCII'ye çevirir: AX = 3335h, AH = '3', AL = '5'. Önce AH, sonra AL yazılınca ekranda 35 görünür. AL = 07h için AX = 0007h → 3037h, ekranda 07: AAM tek basamaklı değerlere baştaki sıfırı kendiliğinden ekler.

5.2.6.2 Kod izleme: gün adını yazdırma

Haftanın günü AL'de 0–6 arasında bir sayı olarak gelir. Bunu bir ada çevirmenin doğal yolu bir **arama tablosudur**: günlerin adları bellekte eşit uzunlukta art arda dizilir ve AL, tablodaki konuma çevrilir. Aşağıdaki programda her ad '$' dahil 4 bayttır; bu yüzden konum AL × 4'tür.

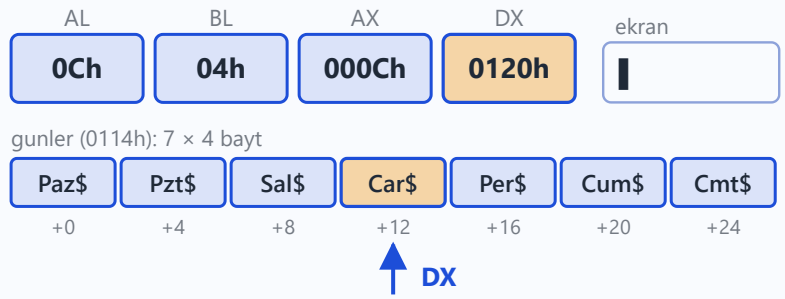
```
org 100h
mov ah, 2Ah
int 21h          ; AL = 0…6
mov ah, 0
mov bl, 4
mul bl           ; AX = AL × 4
mov dx, offset gunler
add dx, ax
mov ah, 09h
int 21h
ret
gunler db 'Paz$Pzt$Sal$Car$'
       db 'Per$Cum$Cmt$'
```

Tarih sistem saatinden gelir; izleme 7 Ocak 2026 Çarşamba içindir. `INT 21h` dönüşünde CX = 07EAh (2026), DH = 01h (Ocak), DL = 07h ve AL = 3 (Çarşamba) olur. DX bu anda 0107h'dir. `MOV AH, 0` AX'i 0003h yapar. `MUL BL` AL ile BL'yi çarpar ve 16 bitlik sonucu AX'e yazar: 3 × 4 = 12 = 000Ch. Bu, `Car$` adının tablonun başından uzaklığıdır.

`MOV DX, OFFSET gunler` DX'teki tarihi siler ve tablonun bağıl konumunu, 0114h'yi yükler. Tarih daha sonra gerekecek olsaydı bu satırdan önce saklanmalıydı. `ADD DX, AX` 0114h + 000Ch = 0120h verir: `Car$`'ın ilk baytı. `AH = 09h` ile ekrana `Car` yazılır.

Adım	Komut	AX	BX	CX	DX	Ekran (çıktı)	Açıklama
1	`mov ah, 2Ah`	2A00h	0000h	0000h	0000h	–	AH = 2Ah: tarihi al
2	`int 21h`	2A03h	0000h	07EAh	0107h	–	CX = 07EAh (2026), DH = 01h (ay), DL = 07h (gün), AL = 3 (Çarşamba)
3	`mov ah, 0`	0003h	0000h	07EAh	0107h	–	AX = 0003h
4	`mov bl, 4`	0003h	0004h	07EAh	0107h	–	her ad 4 bayt
5	`mul bl`	000Ch	0004h	07EAh	0107h	–	AX = 3 × 4 = 12 = 000Ch
6	`mov dx, offset gunler`	000Ch	0004h	07EAh	0114h	–	DX = `gunler` = 0114h; tarih silindi
7	`add dx, ax`	000Ch	0004h	07EAh	0120h	–	DX = 0120h: `Car$`
8	`mov ah, 09h`	090Ch	0004h	07EAh	0120h	–	AH = 09h
9	`int 21h`	0924h	0004h	07EAh	0120h	`Car`	ekrana Car
10	`ret`	0924h	0004h	07EAh	0120h	`Car`	program biter

Tablo 5.2.25: Gün adını tablodan yazan programın izlemesi (tarih 7 Ocak 2026, Çarşamba).



Şekil 5.2.13: `ADD DX, AX` sonrası: DX = 0120h, `gunler` tablosunun +12. baytını, yani `Car$` 'ı gösterir.

Kod kısmı 20 bayttır, bu yüzden `gunler` 0114h'de başlar. Komutların uzunlukları: `MOV AH, imm` 2, `INT` 2, `MOV BL, imm` 2, `MUL BL` 2, `MOV DX, imm` 3, `ADD DX, AX` 2, `RET` 1 bayttır. Çarpma yerine iki kez sola kaydırma da kullanılabilirdi (`SHL AX, 1` iki kez), çünkü 4 ikinin kuvvetidir. 8086'da MUL 70–77 saat turu sürer, iki SHL ise yalnız 4 tur.

Sık yapılan hata

`MOV AH, 0` satırını atlamak. Bu programda zarar vermez, çünkü `MUL BL` AX'in tamamını yeniden yazar. Ama çarpma iki `SHL AX, 1` ile yapılsaydı AH'de kalan 2Ah da kayardı: AX = 2A03h → A80Ch. DX = 0114h + A80Ch = A920h olur ve 09h bellekte alakasız bir yeri yazar. Tablo konumu hesaplanırken AX'in yüksek yarısı mutlaka sıfırlanmalıdır.

5.2.6.3 Kod izleme: saati yazdırma

Saati SS:DD biçiminde yazmak için iki sayı aynı yolla yazdırılır. Tekrarlanan kısım bir yordama alınır: `yaz2`, AL'deki 0–99 arasındaki sayıyı iki basamak olarak yazar.

```
org 100h
mov ah, 2Ch      ; sistem saatini al
int 21h          ; CH=saat CL=dakika
mov al, ch
call yaz2
mov dl, ':'
mov ah, 02h
int 21h
mov al, cl
call yaz2
ret
yaz2:            ; AL (0-99) → 2 hane
aam              ; AH=onlar AL=birler
add ax, 3030h    ; ASCII
mov bx, ax
mov dl, bh
mov ah, 02h
int 21h
mov dl, bl
int 21h
ret
```

İzleme saat 14:35:07,42 içindir. `INT 21h` dönüşünde CX = 0E23h (CH = 14, CL = 35) ve DX = 072Ah'dir (DH = 7 saniye, DL = 42 salise). `CALL yaz2` dönüş adresini, yani `call`'dan sonraki komutun adresi 0109h'yi yığına iter; SP FFFEh'den FFFCh'ye iner. Yordamın içinde `AAM` 14'ü 1 ve 4'e ayırır, `ADD AX, 3030h` ASCII'ye çevirir.

`MOV BX, AX` satırı yordamın kilit noktasıdır. İki basamak AH ve AL'dedir, ama ilk `INT 21h/02h` çağrısı AL'ye DL'yi koyacak ve birler basamağını silecektir. Bu yüzden ikisi de BX'e kopyalanır ve DL'ye sırayla BH'den ve BL'den alınır. CL'deki dakika ise hiçbir çağrıda bozulmaz: yordam CX'e dokunmaz, 02h hizmeti de CX'i değiştirmez. Dakika `MOV AL, CL` ile güvenle alınır.

Adım	Komut	AX	BX	CX	DX	SP	Ekran (çıktı)	Açıklama
1	`mov ah, 2Ch`	2C00h	0000h	0000h	0000h	FFFEh	–	AH = 2Ch: saati al
2	`int 21h`	2C00h	0000h	0E23h	072Ah	FFFEh	–	CH = 0Eh (14), CL = 23h (35), DH = 07h, DL = 2Ah (42)
3	`mov al, ch`	2C0Eh	0000h	0E23h	072Ah	FFFEh	–	AL = saat
4	`call yaz2`	2C0Eh	0000h	0E23h	072Ah	FFFCh	–	dönüş adresi 0109h yığında
5	`aam`	0104h	0000h	0E23h	072Ah	FFFCh	–	14 = 1 × 10 + 4: AH = 1, AL = 4

Adım	Komut	AX	BX	CX	DX	SP	Ekran (çıktı)	Açıklama
6	`add ax, 3030h`	3134h	0000h	0E23h	072Ah	FFFCh	–	iki basamak ASCII'ye: '1', '4'
7	`mov bx, ax`	3134h	3134h	0E23h	072Ah	FFFCh	–	BX'te saklanır; 02h AL'yi bozar
8	`mov dl, bh`	3134h	3134h	0E23h	0731h	FFFCh	–	DL = '1'
9	`mov ah, 02h`	0234h	3134h	0E23h	0731h	FFFCh	–	AH = 02h
10	`int 21h`	0231h	3134h	0E23h	0731h	FFFCh	`1`	ekrana 1; AL = 31h
11	`mov dl, bl`	0231h	3134h	0E23h	0734h	FFFCh	`1`	DL = '4'
12	`int 21h`	0234h	3134h	0E23h	0734h	FFFCh	`14`	ekrana 4
13	`ret`	0234h	3134h	0E23h	0734h	FFFEh	`14`	0109h'ye dönüş
14	`mov dl, ':'`	0234h	3134h	0E23h	073Ah	FFFEh	`14`	DL = ':'
16	`int 21h`	023Ah	3134h	0E23h	073Ah	FFFEh	`14:`	ekrana :
17	`mov al, cl`	0223h	3134h	0E23h	073Ah	FFFEh	`14:`	AL = dakika = 23h; CL bozulmadı
18	`call yaz2`	0223h	3134h	0E23h	073Ah	FFFCh	`14:`	dönüş adresi 0114h
19	`aam`	0305h	3134h	0E23h	073Ah	FFFCh	`14:`	35 = 3 × 10 + 5
20	`add ax, 3030h`	3335h	3134h	0E23h	073Ah	FFFCh	`14:`	'3', '5'
24	`int 21h`	0233h	3335h	0E23h	0733h	FFFCh	`14:3`	ekrana 3
26	`int 21h`	0235h	3335h	0E23h	0735h	FFFCh	`14:35`	ekrana 5
27	`ret`	0235h	3335h	0E23h	0735h	FFFEh	`14:35`	0114h'ye dönüş
28	`ret`	0235h	3335h	0E23h	0735h	0000h	`14:35`	yığından 0000h; SP = 0000h
29	`INT 20h` (`PSP:0000h`)	0235h	3335h	0E23h	0735h	0000h	`14:35`	INT 20h programı bitirir

Tablo 5.2.26: Saati SS:DD biçiminde yazan programın izlemesi (saat 14:35:07,42).

Son iki satır programın nasıl bittiğini gösterir. Yordamdan dönen `RET` (adım 27) yığından 0114h'yi alır ve SP'yi FFFEh'ye geri getirir. Ana programın `RET`'i (adım 28) ise yığının en üstündeki 0000h'yi IP'ye yükler ve SP = 0000h olur. 0000h, PSP'nin başıdır ve orada DOS'un koyduğu `INT 20h` komutu programı bitirir. Ekranda `14:35` kalır.

5.2.7 INT 1Ah ve BIOS veri alanı

5.2.7.1 Tık sayacı ve gerçek zamanlı saat

PC'de zamanı iki ayrı donanım tutar. Birincisi 8253/8254 programlanabilir zamanlayıcısıdır. Kanal 0, 1.193.182 Hz'lik bir saati 65.536'ya bölerek saniyede yaklaşık 18,2 kez IRQ 0 kesmesini üretir. BIOS'un bu kesmeye bağlı yordamı (INT 08h) her seferinde bellekteki bir sayacı bir artırır. Bu sayaca **tık sayacı** denir; gece yarısından beri geçen tık sayısını tutar. Sayaç bellekte olduğu için bilgisayar kapanınca kaybolur.

İkincisi, AT ve sonraki bilgisayarlarda bulunan **gerçek zamanlı saattir** (*real-time clock*, RTC). RTC, pille beslenen CMOS bir yongadır ve bilgisayar kapalıyken de çalışır. Saati, dakikayı, saniyeyi ve tarihi kendi yazmaçlarında BCD olarak tutar; 70h ve 71h G/Ç portları üzerinden okunur. Açılışta DOS saatini RTC'den alır, sonra tık sayacıyla ilerletir. `INT 1Ah` her iki saate de BIOS arayüzü sunar.

AH	İşlev	Dönüş
00h	tık sayacını oku	CX:DX = gece yarısından beri tık sayısı, AL = gün döndü bayrağı
02h	RTC saatini oku	CH = saat, CL = dakika, DH = saniye (BCD), DL = yaz saati; CF = 1: hata
04h	RTC tarihini oku	CH = yüzyıl, CL = yıl, DH = ay, DL = gün (BCD); CF = 1: hata

Tablo 5.2.27: INT 1Ah hizmetleri.

> **Tanım 5.2.8**
>
> **Paketli BCD** (*packed binary-coded decimal*), bir bayta iki onluk basamak koyan gösterimdir: yüksek dört bit (yüksek nibble) onlar basamağını, düşük dört bit birler basamağını tutar. Dakika 35, paketli BCD'de 35h = 0011 0101b olarak saklanır; ikili gösterimde ise 23h = 0010 0011b'dir.

BCD'nin yararı, onaltılık yazılışın doğrudan onluk değeri göstermesidir: 35h “otuz beş” okunur. Ekrana yazmak için bölme gerekmez; iki nibble ayrılır ve her birine 30h eklenir. Yüksek nibble dört bit sağa kaydırılarak, düşük nibble 0Fh ile maskelenerek ayrılır. 8086'da 1'den büyük sabit kaydırma yoktur; kaydırma sayısı CL'de verilir.

Adım	İşlem	Sonuç
1	35h = 0011 0101b	yüksek nibble 3, düşük nibble 5
2	`shr dl, cl` (CL = 4)	0000 0011b = 03h
3	`and dl, 0Fh`	0000 0101b = 05h
4	+30h	'3' = 33h, '5' = 35h

Tablo 5.2.28: Paketli BCD 35h'nin ASCII'ye çevrilmesi.

Aynı dakikayı `INT 21h/2Ch` ikili verir: CL = 23h. Orada AAM ya da 10'a bölme gerekir. İki yöntem karıştırılırsa ekrana yanlış sayı yazılır, ama hata ancak 10 ve üstündeki değerlerde görünür, çünkü 0–9 arasında BCD ile ikili aynıdır.

Örnek 5.2.7

Saat 14:23. RTC dakikayı CL = 23h (BCD) verir. Bu değer AAM kullanan `yaz2` yordamına gönderilirse AAM onu ikili 35 sayar ve ekrana 35 yazılır. DOS ise dakikayı CL = 17h (ikili 23) verir. Bu değer BCD yordamına gönderilirse nibble'lar 1 ve 7 olarak ayrılır, ekrana 17 yazılır. Doğrusu, RTC değerini SHR/AND ile, DOS değerini AAM ile yazdırmaktır.

emu8086'nın desteklediği kesmeler arasında INT 1Ah için yalnız 00h (tık sayacı) bulunur. 02h hizmeti gerçek DOS ortamında ya da DOSBox'ta çalışır; emu8086'da saat `INT 21h/2Ch` ile okunur.

5.2.7.2 Kod izleme: RTC saatini yazdırma

Aşağıdaki program RTC saatini SS:DD:ss biçiminde yazar. İki yordam girişi paylaşır: `ikinokta` önce ':' yazar, sonra hiçbir atlama olmadan `bcdyaz`'a düşer. Saat için doğrudan `bcdyaz`, dakika ve saniye için `ikinokta` çağrılır.

```
org 100h
mov ah, 02h
int 1Ah          ; CH=saat CL=dk DH=sn
jc son           ; CF=1: saat okunamadı
mov bl, cl       ; dakikayı sakla
mov cl, 4        ; SHR kaydırma sayısı
mov bh, ch
call bcdyaz      ; "14"
mov bh, bl
call ikinokta    ; ":35"
mov bh, dh
call ikinokta    ; ":07"
son: ret
ikinokta: mov dl, ':'
mov ah, 02h
int 21h
bcdyaz:          ; BH = paketli BCD
mov dl, bh
shr dl, cl       ; üst hane
add dl, 30h      ; ASCII rakam
mov ah, 02h
int 21h
mov dl, bh
and dl, 0Fh      ; alt hane
add dl, 30h
int 21h
ret
```

Program önce CF'ye bakar. RTC çalışmıyorsa ya da o an güncelleniyorsa hizmet CF = 1 döndürür; program yazdırmadan biter. İzlemede saat 14:35:07'dir ve CF = 0'dır. Dönüşte CH = 14h, CL = 35h, DH = 07h gelir; DL = 00h yaz saatinin kapalı olduğunu söyler.

Yazmaç kullanımı dikkatle planlanmıştır. CL'ye kaydırma sayısı 4 yazılacağından dakika önce BL'ye kopyalanır. Yordam değeri BH'de bekler; BH'ye sırayla saat (CH), dakika (BL) ve saniye (DH) konur. DL, karakter yazmak için gereklidir; ama saniye DH'de durduğu için DL'nin bozulması sorun olmaz. INT 21h/02h, BX'e, CX'e ve DH'ye dokunmaz.

Adım	Komut	AX	BX	CX	DX	SP	CF	Ekran (çıktı)	Açıklama
1	`mov ah, 02h`	0200h	0000h	0000h	0000h	FFFEh	0	–	AH = 02h: RTC saatini oku
2	`int 1Ah`	0200h	0000h	1435h	0700h	FFFEh	0	–	CH = 14h, CL = 35h, DH = 07h (paketli BCD); DL = 00h
3	`jc son`	0200h	0000h	1435h	0700h	FFFEh	0	–	CF = 0: atlama yok
4	`mov bl, cl`	0200h	0035h	1435h	0700h	FFFEh	0	–	dakika BL'de saklanır
5	`mov cl, 4`	0200h	0035h	1404h	0700h	FFFEh	0	–	CL = 4: kaydırma sayısı

Adım	Komut	AX	BX	CX	DX	SP	CF	Ekran (çıktı)	Açıklama
6	`mov bh, ch`	0200h	1435h	1404h	0700h	FFFEh	0	–	BH = saat = 14h
7	`call bcdyaz`	0200h	1435h	1404h	0700h	FFFCh	0	–	doğrudan `bcdyaz`
8	`mov dl, bh`	0200h	1435h	1404h	0714h	FFFCh	0	–	DL = 14h = 0001 0100b
9	`shr dl, cl`	0200h	1435h	1404h	0701h	FFFCh	0	–	4 bit sağa: 01h
10	`add dl, 30h`	0200h	1435h	1404h	0731h	FFFCh	0	–	DL = 31h = '1'
12	`int 21h`	0231h	1435h	1404h	0731h	FFFCh	0	`1`	ekrana 1
13	`mov dl, bh`	0231h	1435h	1404h	0714h	FFFCh	0	`1`	DL = 14h yeniden
14	`and dl, 0Fh`	0231h	1435h	1404h	0704h	FFFCh	0	`1`	üst 4 bit silinir: 04h
15	`add dl, 30h`	0231h	1435h	1404h	0734h	FFFCh	0	`1`	DL = 34h = '4'
16	`int 21h`	0234h	1435h	1404h	0734h	FFFCh	0	`14`	ekrana 4
17	`ret`	0234h	1435h	1404h	0734h	FFFEh	0	`14`	dönüş
18	`mov bh, bl`	0234h	3535h	1404h	0734h	FFFEh	0	`14`	BH = dakika = 35h
19	`call ikinokta`	0234h	3535h	1404h	0734h	FFFCh	0	`14`	`ikinokta` girişi
22	`int 21h`	023Ah	3535h	1404h	073Ah	FFFCh	0	`14:`	ekrana :
23	`mov dl, bh`	023Ah	3535h	1404h	0735h	FFFCh	0	`14:`	`bcdyaz`'a kendiliğinden geçilir
27	`int 21h`	0233h	3535h	1404h	0733h	FFFCh	0	`14:3`	ekrana 3
31	`int 21h`	0235h	3535h	1404h	0735h	FFFCh	0	`14:35`	ekrana 5
32	`ret`	0235h	3535h	1404h	0735h	FFFEh	0	`14:35`	dönüş
33	`mov bh, dh`	0235h	0735h	1404h	0735h	FFFEh	0	`14:35`	BH = saniye = 07h
37	`int 21h`	023Ah	0735h	1404h	073Ah	FFFCh	0	`14:35:`	ekrana :
42	`int 21h`	0230h	0735h	1404h	0730h	FFFCh	0	`14:35:0`	ekrana 0
46	`int 21h`	0237h	0735h	1404h	0737h	FFFCh	0	`14:35:07`	ekrana 7
47	`ret`	0237h	0735h	1404h	0737h	FFFEh	0	`14:35:07`	dönüş
48	`ret`	0237h	0735h	1404h	0737h	0000h	0	`14:35:07`	program biter: SP = 0000h

Tablo 5.2.29: RTC saatini SS:DD:ss biçiminde yazan programın izlemesi (saat 14:35:07).

Saat için `bcdyaz` iki kez yazar. DL = 14h = 0001 0100b; `SHR DL, CL` dört bit sağa kaydırır ve DL = 01h olur. 30h eklenince '1' yazılır. Sonra DL'ye yeniden 14h konur, `AND DL, 0Fh` üst dört biti siler ve DL = 04h olur; ekrana '4' gider. AH, ilk çağrıdan sonra hâlâ 02h olduğundan ikinci `INT 21h` için yeniden yüklenmez. Dakikada `ikinokta` girişi önce ':' yazar; ardından komutlar `bcdyaz` etiketine kendiliğinden geçer. Etiket bir komut değildir, yalnız bir adrestir. Son durumda ekranda `14:35:07` vardır.

> **Sık yapılan hata**
>
> `MOV CL, 4`'ü dakikayı saklamadan yazmak. CL'deki 35h kaybolur ve dakika yerine 04h yazdırılır. Bir başka hata, 8086 için `SHR DL, 4` yazmaktır. Bu biçim 80186 ve sonraki işlemcilerde vardır; 8086'da kaydırma sayısı ya 1 ya da CL'dir.

5.2.7.3 BIOS veri alanı

BIOS, donanımın durumunu bellekte sabit bir yerde tutar: 0040h kesiminde, fiziksel olarak 00400h'den başlayan alanda. Bu alana **BIOS veri alanı** (*BIOS data area*) denir. INT 16h klavye tamponunu, INT 1Ah tık sayacını, INT 10h video modunu ve imleç konumlarını buradan okur. Program da aynı baytları doğrudan okuyabilir. Tablo 5.2.30 sık kullanılan konumları listeler.

Adres	Boyut	İçerik
0040:0010	sözcük	donanım listesi (INT 11h'nin döndürdüğü değer)
0040:0013	sözcük	taban bellek, KB (INT 12h)
0040:0017	bayt	klavye bayrakları: bit 0 sağ Shift, 1 sol Shift, 2 Ctrl, 3 Alt, 4 Scroll Lock, 5 Num Lock, 6 Caps Lock, 7 Insert
0040:001A	sözcük	tampon başı: sıradaki okunacak tuşun bağıl konumu
0040:001C	sözcük	tampon sonu: sıradaki boş yerin bağıl konumu (baş = son → tampon boş)
0040:001E–003D	32 bayt	klavye tamponu: 16 sözcük (ASCII + tarama kodu), dairesel; en çok 15 tuş
0040:0049	bayt	geçerli video modu (03h, 13h …)
0040:004A	sözcük	ekrandaki sütun sayısı (mod 03h'te 80)
0040:0050	8 sözcük	8 sayfanın imleç konumu (düşük bayt sütun, yüksek bayt satır)
0040:006C	çift sözcük	tık sayacı: gece yarısından beri ≈ 18,2 Hz'lik kesme sayısı
0040:0070	bayt	gece yarısı geçti bayrağı (INT 1Ah/00h AL'de verir ve sıfırlar)

Tablo 5.2.30: BIOS veri alanından seçilmiş konumlar.

Fiziksel adres, kesim değeri 16 ile çarpılıp bağıl konum eklenerek bulunur: 0040:006C = 00400h + 006Ch = 0046Ch. Aynı bayta 0000:046Ch adresiyle de ulaşılır; iki kesim:bağıl konum çifti aynı fiziksel adresi gösterir.

Klavye tamponu **dairesel** bir kuyruktur. INT 09h yeni tuşu “son” konumuna yazar ve sonu iki bayt ilerletir; INT 16h “baş” konumundan okur ve başı ilerletir. Konumlardan biri alanın sonuna (003Eh) gelince yeniden 001Eh'ye döner. Baş ile son eşitse tampon boştur. Bu yüzden 16 sözcüklük alana en çok 15 tuş sığar: 16. tuş yazılsaydı son, başa eşit olur ve dolu tampon boş sanılırdı.

Örnek 5.2.8

0040:001Ah'de 0024h, 0040:001Ch'de 0028h okunuyor. Her tuş bir sözcük olduğundan bekleyen tuş sayısı (0028h − 0024h) ÷ 2 = 2'dir. Sıradaki tuşun ASCII kodu 0040:0024h'te, tarama kodu 0040:0025h'tedir. 0040:001Ah'nin fiziksel adresi 0041Ah'dir.

5.2.7.4 Tık sayacıyla bekleme

Tık sayacı, kısa beklemeler için hazır bir saattir. Program başlangıç değerini okur, sonra sayaç istenen kadar artana kadar döngüde bekler. Aşağıdaki kod ayrıca klavye bayraklarından Caps Lock'u sınar ve tamponda bekleyen tuş olup olmadığına bakar.

```
; yaklaşık 1 saniye bekle
mov ax, 0040h
mov es, ax
mov bx, es:[006Ch]   ; başlangıç tıkı
bekle:
mov ax, es:[006Ch]   ; şimdiki tık
sub ax, bx           ; geçen tık
cmp ax, 18           ; 18 tık ≈ 0,99 s
jb bekle
; Caps Lock açık mı? Bekleyen tuş var mı?
test byte ptr es:[0017h], 40h
jnz caps_acik        ; bit 6 = 1
mov ax, es:[001Ah]   ; tampon başı
cmp ax, es:[001Ch]   ; tampon sonu
je tampon_bos
```

Bir tıkın süresi zamanlayıcının bölme oranından hesaplanır. Tablo 5.2.31 değerleri verir.

Büyüklük	Hesap	Değer
Tık sıklığı	1.193.182 Hz ÷ 65.536	18,2065 Hz
Bir tık	65.536 ÷ 1.193.182 s	≈ 54,9 ms
18 tık	18 × 54,925 ms	≈ 0,989 s
5 saniye	5 × 18,2065	≈ 91 tık
Bir gün	sayacın sıfırlandığı değer	1800B0h = 1.573.040 tık

Tablo 5.2.31: Tık sayacının zaman değerleri.

Bekleme süresi tık sayısıyla ölçüldüğü için çözünürlük yaklaşık 55 ms'dir; daha kısa süreler bu yolla ölçülemez. Kod sayacın yalnız düşük sözcüğünü okur. Fark alındığı için düşük sözcük FFFFh'den 0000h'ye dönse bile sonuç doğrudur: 16 bitlik çıkarma, 2^{16}'ya göre (mod 65.536) doğru farkı verir. Yalnız gece yarısındaki sıfırlama hesabı bozar, çünkü sayaç 1800B0h'den 0'a atlar.

Caps Lock denetimi `TEST` ile yapılır. 40h = 0100 0000b maskesi bit 6'yı seçer; bit 1 ise sonuç sıfır değildir, ZF = 0 olur ve `JNZ` atlar. TEST bayrağı yalnız okur, BIOS veri alanındaki baytı değiştirmez. Tampon denetimi baş ile sonu karşılaştırır; eşitse bekleyen tuş yoktur. emu8086 BIOS veri alanının tamamını gerçek PC'deki gibi güncellemez; tık sayacı ve klavye tamponu örnekleri DOSBox'ta tam çalışır.

Sık yapılan hata

Bekleme süresini başlangıca ekleyip bulunan bitiş değeriyle sayacın eşitliğini aramak: `cmp ax, hedef` / `jne bekle`. Sayaç, döngünün tam o turda okumadığı bir anda hedefi geçebilir ve döngü bir sonraki taşmaya kadar sürer. Fark alıp `JB` ile “en az bu kadar geçti mi?” diye sormak güvenlidir.

5.2.8 Program sonlandırma

5.2.8.1 Üç yol ve çıkış kodu

Bir program işini bitirince denetimi DOS'a geri vermek zorundadır. İşlemci “programın sonu” diye bir şey bilmez; IP'nin gösterdiği baytı her zaman komut olarak yürütür. Sonlandırma komutu unutulursa işlemci programdan sonraki baytları, çoğu zaman programın verilerini, komut gibi çalıştırır. Sonuç kestirilemez: ekrana çöp basılabilir, sistem kilitlenebilir. DOS programı bitirmek için üç yol sunar (Tablo 5.2.32).

Yöntem	Nasıl çalışır?	Nerede?
`ret`	COM programında yığının üstünde 0000h vardır; RET, PSP:0000h'a döner, orada `int 20h` (CD 20) bulunur.	yalnız COM, yığın dengeliyse
`int 20h`	DOS programı sonlandırır; CS'nin PSP kesimini göstermesi gerekir.	COM
`mov ax, 4C00h` `int 21h`	AL'deki çıkış koduyla sonlandırır, açık dosyaları kapatır.	COM ve EXE · önerilen

Tablo 5.2.32: Programı sonlandırmanın üç yolu.

Tanım 5.2.9

PSP (*Program Segment Prefix*, program kesimi öneki), DOS'un her programın önüne koyduğu 256 baytlık alandır. Komut satırı, ortam değişkenlerinin adresi ve dönüş bilgileri burada tutulur. İlk iki baytı `CD 20`'dir, yani `INT 20h` komutu. COM programları PSP'den hemen sonra yüklendiği için kod `ORG 100h`'den başlar.

RET'in COM programını neden bitirdiği yığından anlaşılır. DOS bir COM programını başlatırken SP'yi FFFEh yapar ve oraya 0000h sözcüğünü koyar. Program yığını dengeli bırakmışsa, yani her PUSH'u bir POP ve her CALL'u bir RET izlemişse, son RET bu 0000h'yi IP'ye yükler. CS hâlâ programın kesimidir ve CS:0000h PSP'nin başıdır. Orada `INT 20h` çalışır ve program biter. Yığında fazladan bir sözcük kalmışsa RET o sözcüğü adres sanar ve rastgele bir yere atlar.

Önerilen yol `INT 21h/4Ch`'dir. Hem COM hem EXE programında çalışır, CS'nin değerine bağlı değildir ve DOS'a bir **çıkış kodu** verir. Çıkış kodu AL'dedir: 0 başarı, sıfırdan farklı değerler hata anlamına gelir. Bir toplu iş dosyası (`.BAT`) bu kodu `IF ERRORLEVEL 1` ile sınayıp farklı yollar izleyebilir.

5.2.8.2 COM ve EXE şablonları

emu8086 yeni bir program için iki şablon sunar. COM şablonunda kod, veri ve yığın tek bir 64 KB'lık kesimdedir. EXE şablonunda her biri ayrı kesimde durur.

```
org 100h
    mov dx, offset msg
    mov ah, 09h
    int 21h
    ret             ; ya da int 20h
msg db 'Merhaba$'
```

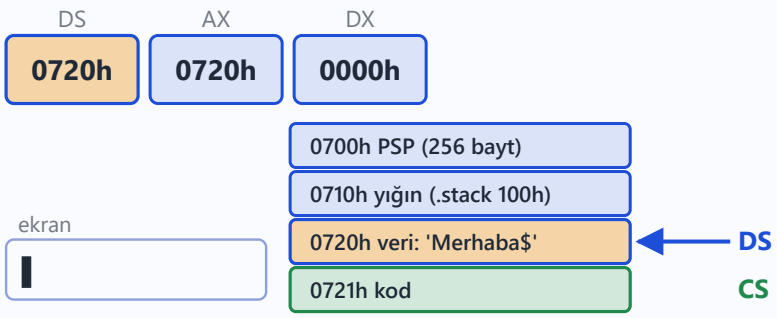
COM programında CS = DS = ES = SS'dir; program kesim yazmaçlarını ayarlamaz. Kod, veri ve yığın birlikte en fazla 64 KB olabilir. Dosya yalnız makine kodudur, başında bir başlık yoktur. DOS dosyayı PSP'nin hemen arkasına, 100h bağıl konumuna kopyalar ve yürütmeyi oradan başlatır.

```
.model small
.stack 100h
.data
msg db 'Merhaba$'
.code
basla:
mov ax, @data
mov ds, ax       ; DS'yi ayarla!
mov dx, offset msg
mov ah, 09h
int 21h
mov ax, 4C00h
int 21h
end basla
```

EXE dosyasının başında bir başlık vardır. Başlık, programın hangi kesimlerden oluştuğunu ve yükleme adresine göre düzeltilmesi gereken yerleri (*relocation*, yeniden yerleştirme bilgisi) tutar. Giriş noktası `end basla` ile belirtilir ve program 64 KB'tan büyük olabilir. DOS başlangıçta CS:IP'yi giriş noktasına, SS:SP'yi yığın kesimine kurar. DS ve ES ise PSP'yi gösterir. Program veri kesimine erişmeden önce DS'yi kendisi ayarlamalıdır: `mov ax, @data` ve `mov ds, ax`. `@data`, veri kesiminin kesim değeridir; bu değeri yükleme sırasında DOS yerine koyar.

Adım	Komut	AX	DS	DX	Ekran (çıktı)	Açıklama
1	`mov ax, @data`	0720h	0700h	0000h	–	@data = veri kesimi 0720h; DS hâlâ PSP'de
2	`mov ds, ax`	0720h	0720h	0000h	–	DS = 0720h: veri kesimi
3	`mov dx, offset msg`	0720h	0720h	0000h	–	`msg` veri kesiminin başında
4	`mov ah, 09h`	0920h	0720h	0000h	–	AH = 09h
5	`int 21h`	0924h	0720h	0000h	`Merhaba`	0720:0000'dan '$'a kadar
6	`mov ax, 4C00h`	4C00h	0720h	0000h	`Merhaba`	AH = 4Ch, AL = 00h
7	`int 21h`	4C00h	0720h	0000h	`Merhaba`	çıkış kodu 0 ile DOS'a dönüş

Tablo 5.2.33: EXE şablonunun izlemesi (kesim değerleri emu8086 yerleşimine göre).



Şekil 5.2.14: `MOV DS, AX` sonrası kesimler: PSP 0700h, yığın 0710h, veri 0720h (DS), kod 0721h (CS). Kesim değerleri emu8086'nın yerleşimidir; gerçek DOS'ta sayılar farklı olabilir, sıra ve DS ayarının gereği aynıdır.

Başlangıçta DS = ES = 0700h'dir, yani PSP. Yığın kesimi 0710h'dir: PSP'nin 256 baytı 16 paragraf olduğu için 0700h + 10h. `.stack 100h` 256 baytlık yığın ayırır, o da 10h paragraf eder; veri kesimi bu yüzden 0720h'de başlar. 8 baytlık veri bir paragrafa sığar ve kod 0721h'de başlar. `MOV AX, @data` AX'e 0720h yükler; `MOV DS, AX` DS'yi veri kesimine çevirir. Kesim yazmacına anlık değer yazılamadığı için iki adım gerekir. `msg` veri kesiminin başındadır: `OFFSET msg` = 0000h. 09h, DS:DX = 0720:0000h'dan itibaren `Merhaba` yazar. Son iki satır çıkış kodu 0 ile DOS'a döner.

EXE programında RET kullanılamaz. DOS EXE'nin yığınına 0000h koymaz; SS:SP programın kendi yığın kesimini gösterir. RET orada ne bulursa ona atlar. `MOV DS, AX` satırı unutulursa DX doğru bağıl konumu tutar ama kesim yanlıştır: DS:DX PSP'nin başını gösterir ve ekrana PSP'nin baytları, yani çöp basılır.

5.2.9 Uygulama: sonucu ikili yazdırmak

5.2.9.1 Program

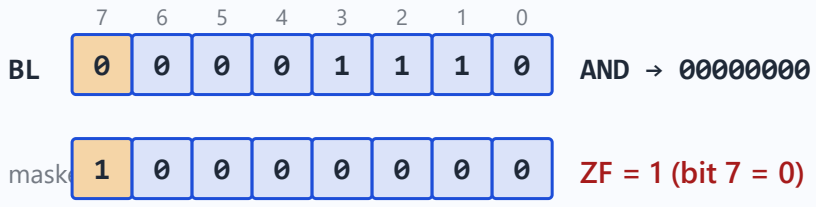
Bu kısımdaki program, birçok hizmeti küçük bir hesapla birleştirir. Önce 5 + 10 − 1 = 14 hesaplanır ve sonuç BL'ye yazılır. Sonra BL'nin sekiz biti, en anlamlı bitten (MSB) başlayarak tek tek '0' ya da '1' karakteri olarak ekrana basılır. En sonda ikili sayı eki 'b' yazılır ve bir tuşa basılması beklenir. Beklenen çıktı `00001110b`'dir.

```
org 100h
mov al, 5        ; 00000101b
mov bl, 10       ; 0Ah = 00001010b
; 5 + 10 = 15 = 0Fh = 00001111b
add bl, al
; 15 - 1 = 14 = 0Eh = 00001110b
sub bl, 1
mov cx, 8        ; sonucu ikilik tabanda yazdır
print:
mov ah, 2        ; yazdırma fonksiyonu
mov dl, '0'
test bl, 10000000b  ; ilk biti test et
jz zero
mov dl, '1'
zero:
int 21h
shl bl, 1
loop print
; ikili sayı eki b yazdır
mov dl, 'b'
int 21h
; klavye tuşu basılmasını bekle
mov ah, 0
int 16h
ret
```

Sayının bellekte ya da yazmaçta “ikili hâle getirilmesi” gerekmez; sayı zaten ikilidir. Yazdırmak, her biti bir ASCII karakterine çevirmektir: 0 biti için 30h, 1 biti için 31h. Döngü her turda bit 7'ye bakar, karakteri yazar ve sayıyı bir sola kaydırır. Böylece bir sonraki bit bit 7'nin yerine gelir. Sekiz tur sonunda bütün bitler soldan sağa yazılmış olur.

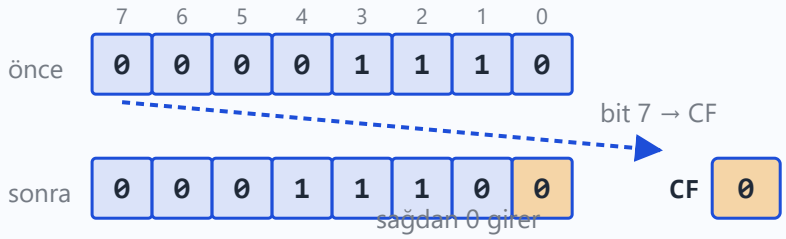
5.2.9.2 TEST ve SHL ile bit okuma

`TEST BL, 10000000b`, BL ile maskeyi bit bit VE'ler ama sonucu hiçbir yere yazmaz; yalnız bayrakları kurar. Maskenin tek 1 biti bit 7'dedir. BL'nin bit 7'si 0 ise VE sonucu 0 olur ve ZF = 1 kurulur. Bit 7 = 1 ise sonuç 80h olur ve ZF = 0'dır. `JZ` ZF = 1 iken atlar. TEST ayrıca CF ve OF'yi her zaman sıfırlar.



Şekil 5.2.15: `TEST BL, 10000000b` : BL = 00001110b maskeyle VE'lenir, sonuç 00000000b'dir; ZF = 1 (bit 7 = 0). BL değişmez.

`SHL BL, 1` bütün bitleri bir sola kaydırır. Bit 7'den çıkan bit CF'ye gider, bit 0'a 0 girer. Bir sonraki bit artık bit 7'nin yerindedir. TEST sonucu saklamadığı için BL'yi bozmaz; onu bilerek değiştiren yalnız SHL'dir.



Şekil 5.2.16: `SHL BL, 1` : 00001110b → 00011100b; çıkan bit 7 (0) CF'ye gider, sağdan 0 girer.

Aynı iş SHL ile CF'ye bakarak da yapılabilir: önce `SHL BL, 1`, sonra `JC` ile çıkan bitin 1 olup olmadığı sınanır. Bu sıralamada kaydırma yazdırmadan önce yapılmalıdır; çünkü bitin değeri artık BL'de değil CF'dedir. `MOV AH, 2` 'nin her turda yeniden yapılmasının bir nedeni de vardır: `INT 21h/02h` AH'yi bozmaz ama döngüye sonradan bir başka hizmet eklenirse AH korunmuş olur.

5.2.9.3 Kod izleme

Hesaplama kısmında bayrakların nasıl değiştiği Tablo 5.2.34'te görülür. `MOV` komutları bayraklara dokunmaz. `ADD BL, AL` 0Ah + 05h = 0Fh üretir ve elde yoktur (CF = 0). Sonuçta dört 1 biti vardır; parite çift olduğu için PF = 1'dir. `SUB BL, 1` 0Eh = 00001110b üretir; üç 1 biti PF'yi 0 yapar, ödünç yoktur (CF = 0).

Adım	Komut	AH	DL	BL	CX	ZF	CF	PF	Ekran (çıktı)	Açıklama
1	`mov al, 5`	00h	00h	00h	0000h	0	0	0	–	AL = 05h; MOV bayrak değiştirmez
2	`mov bl, 10`	00h	00h	0Ah	0000h	0	0	0	–	BL = 0Ah
3	`add bl, al`	00h	00h	0Fh	0000h	0	0	1	–	0Ah + 05h = 0Fh; dört 1 biti: PF = 1
4	`sub bl, 1`	00h	00h	0Eh	0000h	0	0	0	–	0Fh − 1 = 0Eh; üç 1 biti: PF = 0
5	`mov cx, 8`	00h	00h	0Eh	0008h	0	0	0	–	sekiz bit için sekiz tur

Tablo 5.2.34: Hesaplama kısmının izlemesi (bayraklar her komuttan sonra).

Döngünün sekiz turu aynı kalıbı izler. Tablo 5.2.35 iki turu adım adım verir: bit 7'nin 0 olduğu ilk tur ve 1 olduğu beşinci tur. İlk turda TEST ZF = 1 yapar, `JZ` atlar ve DL'deki '0' yazılır. Beşinci turda BL = E0h'dir; TEST ZF = 0 yapar, JZ atlamaz, `MOV DL, '1'` çalışır ve '1' yazılır. Her iki turda da SHL çıkan biti CF'ye atar, LOOP ise CX'i bir azaltır.

Adım	Komut	AH	DL	BL	CX	ZF	CF	PF	Ekran (çıktı)	Açıklama
6	`mov ah, 2`	02h	00h	0Eh	0008h	0	0	0	–	tur 1: AH = 2
7	`mov dl, '0'`	02h	30h	0Eh	0008h	0	0	0	–	varsayılan '0'
8	`test bl, 10000000b`	02h	30h	0Eh	0008h	1	0	1	–	0Eh AND 80h = 0: ZF = 1
9	`jz zero`	02h	30h	0Eh	0008h	1	0	1	–	ZF = 1: `zero` 'ya atla

Adım	Komut	AH	DL	BL	CX	ZF	CF	PF	Ekran (çıktı)	Açıklama
10	`int 21h`	02h	30h	0Eh	0008h	1	0	1	`0`	ekrana 0
11	`shl bl, 1`	02h	30h	1Ch	0008h	0	0	0	`0`	BL = 1Ch; çıkan bit 0
12	`loop print`	02h	30h	1Ch	0007h	0	0	0	`0`	CX = 7 ≠ 0: `print`'e
34	`mov ah, 2`	02h	30h	E0h	0004h	0	0	0	`0000`	tur 5: AH = 2
35	`mov dl, '0'`	02h	30h	E0h	0004h	0	0	0	`0000`	varsayılan '0'
36	`test bl, 10000000b`	02h	30h	E0h	0004h	0	0	0	`0000`	E0h AND 80h = 80h: ZF = 0
37	`jz zero`	02h	30h	E0h	0004h	0	0	0	`0000`	atlama yok
38	`mov dl, '1'`	02h	31h	E0h	0004h	0	0	0	`0000`	DL = '1'
39	`int 21h`	02h	31h	E0h	0004h	0	0	0	`00001`	ekrana 1
40	`shl bl, 1`	02h	31h	C0h	0004h	0	1	1	`00001`	BL = C0h; çıkan bit CF = 1
41	`loop print`	02h	31h	C0h	0003h	0	1	1	`00001`	CX = 3 ≠ 0: `print`'e

Tablo 5.2.35: Döngünün 1. ve 5. turları adım adım (5. turda bit 7 = 1).

Tablo 5.2.36 sekiz turun özetidir. BL her turda bir sola kayar: 0Eh, 1Ch, 38h, 70h, E0h, C0h, 80h, 00h. Bit 7 sütunu, yazılan karakterlerin sırasını verir: 0, 0, 0, 0, 1, 1, 1, 0. Bu, 0Eh'nin ikili yazılışıdır. Sekizinci turdan sonra CX = 0 olur ve LOOP atlamaz.

Tur	BL (TEST öncesi)	Bit 7	Yazılan	BL (SHL sonrası)	CF	CX	Ekran
1	0Eh = 00001110	0	`0`	1Ch	0	0007h	`0`
2	1Ch = 00011100	0	`0`	38h	0	0006h	`00`
3	38h = 00111000	0	`0`	70h	0	0005h	`000`
4	70h = 01110000	0	`0`	E0h	0	0004h	`0000`
5	E0h = 11100000	1	`1`	C0h	1	0003h	`00001`
6	C0h = 11000000	1	`1`	80h	1	0002h	`000011`
7	80h = 10000000	1	`1`	00h	1	0001h	`0000111`
8	00h = 00000000	0	`0`	00h	0	0000h	`00001110`

Tablo 5.2.36: Sekiz turun özeti (her satır, o turun LOOP komutundan sonraki durum).

Döngüden sonra DL = 'b' yapılır ve AH hâlâ 2 olduğu için `INT 21h` doğrudan 'b'yi yazar. `MOV AH, 0` ve `INT 16h` bir tuş bekler; izlemede Enter'a basıldığında AH = 1Ch (tarama kodu), AL = 0Dh döner. Son `RET` programı bitirir. Ekranın son hâli şöyledir:

```
00001110b_
```

Örnek 5.2.9

Aynı döngü BL = B5h ile çalışsın. B5h = 1011 0101b'dir. Turlarda bit 7 sırasıyla 1, 0, 1, 1, 0, 1, 0, 1 olur ve ekrana `10110101b` yazılır. Turlarda SHL'nin CF'ye attığı bitler de aynı sırayı izler, çünkü çıkan bit her turda TEST'in baktığı bittir.

Programdan çıkan dersler kısadır. LOOP CX'i azalttığı için döngü içinde CX başka bir işe kullanılmaz. INT 21h/02h AL'yi değiştirir (AL = DL); AL'de saklanan bir değer varsa önce korunur. TEST bayrakları kurar ama işleneni değiştirmez; bu yüzden aynı yazmaç birçok maskeyle art arda sınanabilir.

5.2.9.4 Mini menü

Son program, bir kullanıcı arayüzünün iskeletidir: bir menü gösterir, bir tuş okur ve tuşa göre iş yapar. '1' tuşu selam verir, 'q' programı bitirir, başka bir tuş menüyü yeniden gösterir.

```
org 100h
goster:
mov dx, offset menu
mov ah, 09h
int 21h
mov ah, 01h      ; seçimi oku
int 21h
cmp al, 'q'
je cikis
cmp al, '1'
jne goster
mov dx, offset selam
mov ah, 09h
int 21h
jmp goster
cikis:
mov ax, 4C00h    ; çıkış kodu 0
int 21h
menu  db 0Dh, 0Ah, '1:Selam q:Cikis $'
selam db 0Dh, 0Ah, 'Selam!$'
```

Kullanıcı önce 1'e, sonra q'ya basıyor. Tablo 5.2.37 izlemeyi verir.

Adım	Komut	AX	DX	ZF	Ekran (çıktı)	Açıklama
1	`mov dx, offset menu`	0000h	0121h	0	–	DX → menü metni
2	`mov ah, 09h`	0900h	0121h	0	–	AH = 09h
3	`int 21h`	0924h	0121h	0	`↵1:Selam q:Cikis`	CR LF ve menü yazılır
4	`mov ah, 01h`	0124h	0121h	0	`↵1:Selam q:Cikis`	AH = 01h: oku ve yansıt
5	`int 21h`	0131h	0121h	0	`↵1:Selam q:Cikis 1`	1: AL = 31h
6	`cmp al, 'q'`	0131h	0121h	0	`↵1:Selam q:Cikis 1`	31h − 71h ≠ 0: ZF = 0
7	`je cikis`	0131h	0121h	0	`↵1:Selam q:Cikis 1`	atlama yok

Adım	Komut	AX	DX	ZF	Ekran (çıktı)	Açıklama
8	`cmp al, '1'`	0131h	0121h	1	`↵1:Selam q:Cikis 1`	31h − 31h = 0: ZF = 1
9	`jne goster`	0131h	0121h	1	`↵1:Selam q:Cikis 1`	eşit: atlama yok
10	`mov dx, offset selam`	0131h	0134h	1	`↵1:Selam q:Cikis 1`	DX → `selam`
12	`int 21h`	0924h	0134h	1	`↵1:Selam q:Cikis 1↵Selam!`	yeni satıra Selam!
13	`jmp goster`	0924h	0134h	1	`↵1:Selam q:Cikis 1↵Selam!`	menüye dön
16	`int 21h`	0924h	0121h	1	`↵1:Selam q:Cikis 1↵Selam!↵1:Selam q:Cikis`	menü yeniden yazılır
18	`int 21h`	0171h	0121h	1	`↵1:Selam q:Cikis 1↵Selam!↵1:Selam q:Cikis q`	q: AL = 71h
19	`cmp al, 'q'`	0171h	0121h	1	`↵1:Selam q:Cikis 1↵Selam!↵1:Selam q:Cikis q`	eşit: ZF = 1
20	`je cikis`	0171h	0121h	1	`↵1:Selam q:Cikis 1↵Selam!↵1:Selam q:Cikis q`	`cikis`'a atla
21	`mov ax, 4C00h`	4C00h	0121h	1	`↵1:Selam q:Cikis 1↵Selam!↵1:Selam q:Cikis q`	AH = 4Ch, AL = 00h
22	`int 21h`	4C00h	0121h	1	`↵1:Selam q:Cikis 1↵Selam!↵1:Selam q:Cikis q`	çıkış kodu 0 ile DOS'a dönüş

Tablo 5.2.37: Mini menü programının izlemesi (basılan tuşlar: 1, q).

Menü metni CR LF ile başladığı için her gösterimde yeni bir satıra yazılır. `AH = 01h` tuşu okur ve yansıtır; kullanıcının bastığı 1 menünün hemen yanında görünür. İlk karşılaştırma 31h ile 71h ('q') arasındadır; eşit değildir, ZF = 0 olur ve `JE` atlamaz. İkinci karşılaştırma '1' ile yapılır; eşittir, ZF = 1 olur ve `JNE` atlamaz. `selam` yazılır ve `JMP` menüye döner. İkinci turda q okunur, `CMP AL, 'q'` ZF = 1 yapar ve `JE cikis`'a atlar. `MOV AX, 4C00h` AH = 4Ch ve AL = 00h'yi tek komutta kurar; program çıkış kodu 0 ile biter. Ekranın son durumu şudur:

```
1:Selam q:Cikis 1
Selam!
1:Selam q:Cikis q
```

Veri, son komuttan sonra durur. `INT 21h/4Ch` geri dönmediği için işlemci `menu`'nün baytlarına hiç ulaşmaz. Sonlandırma satırları silinseydi işlemci 0Dh, 0Ah, '1' … baytlarını komut olarak çözmeye çalışırdı.

5.2.9.5 Sık yapılan hatalar

Hizmet çağıran programlardaki hataların çoğu birkaç kalıba girer. Tablo 5.2.38 bunları nedenleri ve düzeltmeleriyle toplar.

Hata	Sonuç	Düzeltme
'$' unutuldu	INT 21h/09h belleği '$' bulana kadar yazar; ekrana çöp dolar	dizginin sonuna '$'
DS ayarlanmadı (EXE)	DX doğru ama kesim yanlış; PSP yazılır	`mov ax, @data` · `mov ds, ax`
AL'nin bozulması	02h ve 09h AL'yi değiştirir; okunan karakter kaybolur	değeri başka yazmaçta saklamak
09h sonrası imleç	INT 10h/09h imleci ilerletmez; sonraki yazı üstüne gelir	imleci 02h ile taşımak
13h'te DS:DX	BIOS dizgi hizmeti adresi ES:BP'de bekler	`lea bp, dizgi` ; DX konum
0Ah tamponu	kapasite yazılmamış ya da metin tampon+0'dan okunmuş	bayt 0 = kapasite; metin tampon+2'de
Yalnız LF	imleç alt satıra iner, satır başına gelmez	CR + LF (0Dh, 0Ah)
Sonlandırma yok	veriler komut olarak yürütülür	RET (COM) ya da INT 21h/4Ch

Tablo 5.2.38: Kesme hizmetleriyle yapılan sık hatalar.

Sık yapılan hata

Okunan tuşu yeni satırda yeniden yazmak için şu sıra kullanılıyor: `AH = 01h` ile oku, `AH = 02h` ile CR ve LF yaz, sonra `MOV DL, AL` ile tuşu yaz. Kullanıcı k'ye bastığında ekranda yalnız yansıtılan k görünür ve imleç iki satır aşağı iner. Son 02h çağrısından sonra AL = 0Ah'dir; `MOV DL, AL` tuşu değil LF'yi alır. Okunan tuş hemen `MOV BL, AL` ile saklanır, sonra `MOV DL, BL` yazılır.

5.2.10 Alıştırmalar ve çözümleri

Sunumdaki alıştırmalar aşağıda çözümleriyle verilir. Program yazılan alıştırmalarda kod, ardından çözümün kısa açıklaması gelir. Çözümler tek yol değildir; aynı işi başka hizmetlerle yapan programlar da doğrudur.

Alıştırma 1. Ekranın tam ortasına (12, 38) mavi zemin üzerine beyaz “8086” yazan program INT 10h/13h ile nasıl yazılır?

```
org 100h
mov ax, 1301h      ; AH=13h, AL=01h
mov bx, 001Fh      ; BH=0, BL=1Fh
mov cx, 4          ; 4 karakter
mov dx, 0C26h      ; satır 12, sütun 38
lea bp, yazi       ; ES:BP → dizgi
int 10h
ret
yazi db '8086'
```

Dört karakter 38–41. sütunlara yazılır; 80 sütunluk ekranın ortası 40 olduğundan yazı ortalanmıştır. DX = 0C26h, DH = 0Ch = 12 ve DL = 26h = 38 demektir. BL = 1Fh mavi zemin, beyaz yazıdır. COM programında ES = DS olduğu için ES ayrıca kurulmaz. İlk hücrenin görüntü belleğindeki yeri (12 × 80 + 38) × 2 = 07CCh'dir.

Alıştırma 2. Kullanıcıdan iki rakam okuyup (01h) toplamlarını tek basamak olarak yazdıran program (toplam ≤ 9) nasıl yazılır?

```
org 100h
mov ah, 01h
int 21h            ; birinci rakam
sub al, 30h
mov bl, al
mov ah, 01h
int 21h            ; ikinci rakam
sub al, 30h
add bl, al         ; BL = toplam
mov ah, 02h
mov dl, 0Dh
int 21h
mov dl, 0Ah
int 21h
mov dl, bl
add dl, 30h        ; ASCII rakam
int 21h
ret
```

Her rakamdan 30h çıkarılarak sayı değeri bulunur. Birinci sayı BL'de saklanır, çünkü ikinci okuma AL'yi değiştirir. Toplam BL'de oluşur ve CR LF'den sonra 30h eklenerek yazılır. Kullanıcı 3'e ve 5'e bastığında ekranda `35`, alt satırda `8` görünür.

Alıştırma 3. Esc'ye basılana kadar basılan tuşların tarama kodunu onaltılık yazdıran program nasıl yazılır?

```
org 100h
oku:
mov ah, 0
int 16h            ; AH = tarama, AL = ASCII
cmp al, 1Bh        ; Esc?
je son
lea bx, hex        ; XLAT tablosu
mov dh, ah         ; tarama kodunu sakla
mov al, dh
mov cl, 4
shr al, cl         ; üst basamak
xlat
mov dl, al
mov ah, 02h
int 21h
mov al, dh
and al, 0Fh        ; alt basamak
xlat
mov dl, al
int 21h
mov dl, ' '
int 21h
jmp oku
son: ret
hex db '0123456789ABCDEF'
```

Tarama kodu AH'de gelir ve DH'de saklanır; çünkü AL ve AH, XLAT ve 02h çağrılarıyla değişecektir. `XLAT`, AL ← [BX + AL] işlemini yapar: 0–15 arasındaki bir sayıyı tablodaki karaktere çevirir. Üst basamak dört bit sağa kaydırılarak, alt basamak 0Fh ile maskelenerek ayrılır. a ve 1 tuşlarına basılınca ekranda `1E 02` görünür; Esc programı bitirir.

Alıştırma 4. 0Ah ile okunan metni tersten yazdıran program nasıl yazılır?

```
org 100h
mov dx, offset tampon
mov ah, 0Ah
int 21h            ; satır oku
mov ah, 02h
mov dl, 0Ah        ; alt satır (Enter CR basmıştı)
int 21h
mov cl, [tampon+1] ; okunan sayı
mov ch, 0
jcxz bitti
mov si, offset tampon+1
add si, cx         ; son karakter
std                ; geriye doğru
ters:
lodsb
mov dl, al
int 21h
loop ters
cld
bitti: ret
tampon db 20, ?, 20 dup(?)
```

Metin tampon+2'den başlar ve uzunluğu [tampon+1]'dedir. Son karakter tampon + 2 + sayı − 1 = tampon + 1 + sayı adresindedir; SI oraya kurulur. `STD` yön bayrağını kurar ve `LODSB` her turda AL ← [SI] yapıp SI'yi bir azaltır. CX = sayı olduğundan LOOP her karakteri bir kez yazar. Boş satırda (sayı 0) `JCXZ` döngüye hiç girmez; yoksa LOOP CX'i 0'dan FFFFh'ye indirip 65.536 tur dönerdi. Program sonunda `CLD` yön bayrağını varsayılana döndürür. Mikro yazılınca alt satırda `orkiM` görünür.

Alıştırma 5. Tarih GG.AA.YYYY biçiminde nasıl yazdırılır? Yıl için hangi yöntem kullanılır?

```
org 100h
mov ah, 2Ah
int 21h            ; CX=yıl DH=ay DL=gün
mov bx, cx         ; yılı sakla
mov al, dl
call iki           ; GG
mov dl, '.'
int 21h
mov al, dh
call iki           ; AA
mov dl, '.'
int 21h
mov ax, bx         ; yıl
mov cx, 0          ; basamak sayacı
mov si, 10
bol:
xor dx, dx
div si             ; DX:AX / 10
push dx            ; kalan = basamak
inc cx
test ax, ax
jnz bol
mov ah, 02h
yaz:
pop dx
add dl, 30h
int 21h
loop yaz
ret
iki:               ; AL (0-99) → 2 hane, DH korunur
push dx
aam
add ax, 3030h
push ax
mov dl, ah
mov ah, 02h
int 21h
pop ax
mov dl, al
mov ah, 02h
int 21h
pop dx
ret
```

Gün ve ay 99'dan küçük olduğu için AAM yeterlidir; `iki` yordamı DH'deki ayı yığında korur. Yıl 16 bitlik bir sayıdır ve AAM ile ayrılamaz. Yıl 10'a tekrar tekrar bölünür: `DIV SI` DX:AX'i 10'a böler, bölümü AX'e, kalanı DX'e yazar. Kalanlar birler, onlar, yüzler, binler sırasıyla çıkar; yazılış sırası ise tersidir. Bu yüzden kalanlar yığına itilir ve sonra çekilerek yazılır. Her bölmeden önce DX sıfırlanır, çünkü DIV bölüneni DX:AX olarak alır. 7 Ocak 2026 için ekrana `07.01.2026` yazılır.

Alıştırma 6. `mov ah, 09h` / `int 21h` ile '$' karakteri nasıl yazdırılır?

Yazdırılamaz; 09h ilk '$'ı dizginin sonu sayar ve yazmaz. '$' için 02h hizmeti kullanılır: `mov ah, 02h`, `mov dl, '$'` (24h), `int 21h`. INT 10h/0Eh ile AL = 24h da aynı işi görür.

Alıştırma 7. Mod 13h'te (160, 100) pikselinin A000h kesimindeki bağıl konumu nedir?

100 × 320 + 160 = 32.160 = 7DA0h. Fiziksel adres A7DA0h'dir.

Alıştırma 8. INT 1Ah/02h DH = 59h döndürdü. Ekrana ne yazılır? Aynı saniye INT 21h/2Ch ile hangi değerle gelir?

59h paketli BCD'dir: üst nibble 5, alt nibble 9; ekrana `59` yazılır. INT 21h/2Ch saniyeyi ikili verir: 59 = 3Bh, yani DH = 3Bh.

Alıştırma 9. 0040:006C tık sayacıyla yaklaşık 5 saniye beklemek için kaç tık beklenmelidir?

5 × 18,2065 ≈ 91 tık. Tık başına 54,9 ms ile 91 tık 4,998 saniyedir.

5.2.11 Özet

- BIOS donanımı doğrudan süren ROM yordamlarıdır; DOS onun üstünde dosya, dizgi, saat ve program hizmetleri sunar. Program ikisine de `INT n` ile, hizmetin adresini bilmeden ulaşır.
- Hizmet çağırma kalıbı: AH ← hizmet numarası, öteki yazmaçlara parametreler, `INT n`, sonuç yazmaçlarda ya da bayraklarda. Hizmetler bazı yazmaçları değiştirir: INT 21h/02h AL = DL, 09h AL = 24h yapar.
- Yeni satır CR LF (0Dh, 0Ah) ister. Rakam = kod − 30h; büyük ve küçük harf bit 5 ile ayrılır.
- INT 10h: 00h mod seçer ve ekranı siler; 02h/03h imleci yazar ve okur. 09h renkli yazar ama imleci ilerletmez; 0Eh teletype yazar ve imleci ilerletir. 13h ES:BP'deki CX uzunluklu dizgiyi yazar; 06h bir pencereyi kaydırır ya da siler.
- Öznitelik = arka plan × 16 + ön plan; bit 7 yanıp sönme. 07h varsayılan, 1Fh mavi zemin beyaz yazı, 4Eh kırmızı zemin sarı yazıdır.
- Mod 03h'te görüntü belleği B800:0000h'dadır; hücre (satır × 80 + sütun) × 2 adresindedir, çift baytta karakter, tek baytta öznitelik. Mod 13h'te A000:0000h'dan itibaren piksel başına bir bayt, adres y × 320 + x'tir.
- INT 16h/00h tuş bekler ve AH'ye tarama kodunu, AL'ye ASCII'yi verir; genişletilmiş tuşlarda AL = 00h'dir. 01h beklemeden yoklar: ZF = 1 tuş yok.
- INT 21h: 01h oku ve yansıt, 02h DL'yi yaz, 09h '$' ile biten dizgiyi yaz, 0Ah tamponlu satır oku. DOS çıktısı yönlendirilebilir, BIOS çıktısı yönlendirilemez.
- 0Ah tamponu: bayt 0 kapasite, bayt 1 okunan sayı, metin bayt 2'den başlar ve 0Dh ile biter.
- INT 21h/2Ah ve 2Ch tarih ve saati ikili verir (AAM ile yazdırılır); INT 1Ah/02h RTC saatini paketli BCD verir (SHR ve AND ile yazdırılır).

- BIOS veri alanı 0040h kesimindedir: klavye bayrakları 0017h, tampon başı ve sonu 001Ah ve 001Ch, tık sayacı 006Ch. Tık sıklığı 18,2065 Hz, bir tık ≈ 54,9 ms'dir.
- Program RET (yalnız dengeli yığınlı COM), INT 20h (COM) ya da önerilen INT 21h/4Ch ile biter; AL çıkış kodudur. EXE programı DS'yi `@data` ile kendisi ayarlar.

5.2.12 Kavram listesi

BIOS *Basic Input/Output System* · yazılım kesmesi *software interrupt* · kesme vektörü *interrupt vector* · ASCII *ASCII* · video modu *video mode* · imleç *cursor* · öznitelik baytı *attribute byte* · piksel *pixel* · little-endian *little-endian* · tarama kodu *scan code* · klavye tamponu *keyboard buffer* · yoklama *polling* · paketli / paketsiz BCD *packed / unpacked BCD* · nibble *nibble* · gerçek zamanlı saat (RTC) *real-time clock* · PSP (program kesimi öneki) *program segment prefix* · program sonlandırma *program termination* · öykünücü *emulator*

5.2.13 Kendini Dene

1. Mod 03h'te imleç (24, 0)'dayken INT 10h, AH = 09h ile AL = '-', BL = 1Eh, CX = 80 yazılıyor. Ekranda ne görünür, imleç nerede kalır?

   Cevap

   Son satırın 80 hücresinin hepsine mavi zemin üzerine sarı '-' yazılır. 09h imleci ilerletmez ve ekranı kaydırmaz; imleç (24, 0)'da kalır. Aynı satır 80 kez 0Eh ile yazılsaydı son karakterden sonra imleç alt satıra geçmeye çalışır ve ekran bir satır yukarı kayardı.

2. Kırmızı zemin üzerinde yanıp sönen beyaz yazının öznitelik baytı nedir?

   Cevap

   Bit 7 = 1 (yanıp sönme), arka plan 100b (kırmızı), ön plan 1111b (beyaz): 1100 1111b = **CFh**.

3. Mod 03h'te son satırın ilk hücresinin (24, 0) karakter baytı hangi bağıl konumdadır? Fiziksel adresi nedir?

   Cevap

   (24 × 80 + 0) × 2 = 3840 = 0F00h. Karakter B800:0F00h'da, öznitelik B800:0F01h'dedir. Fiziksel adres B8000h + 0F00h = **B8F00h**.

4. `tampon db 4, ?, 4 dup(?)` tanımlı tampona INT 21h/0Ah ile okunurken kullanıcı abcdef yazıp Enter'a basıyor. Tamponun ilk altı baytı ne olur?

   Cevap

   Kapasite Enter dahil 4 olduğundan en çok 3 karakter alınır; d, e ve f'de zil çalar. Tampon `04 03 61 62 63 0D` olur: kapasite, okunan sayı 3, 'a', 'b', 'c' ve Enter.

5. `mov ah, 02h`, `mov dl, 'Z'`, `int 21h` komutlarından sonra AL kaçtır?

   Cevap

   INT 21h/02h dönüşte AL'ye DL'yi koyar: AL = 5Ah ('Z'). AL'de daha önce saklanan bir değer varsa kaybolmuştur.

6. Saat 23:00. Saati DOS'tan (INT 21h/2Ch) okuyan program CH'yi BCD yazan yordama (SHR/AND) gönderiyor. Ekranda ne görünür? RTC'den okunan CH, AAM kullanan yordama gönderilseydi ne görünürdü?

   Cevap

   DOS ikili verir: CH = 23 = 17h. BCD yordamı nibble'ları 1 ve 7 olarak ayırır ve **17** yazar. RTC ise CH = 23h verir; AAM bunu ikili 35 sayar (23h = 35) ve **35** yazılır. Her iki durumda da doğru değer 23'tür.

7. Tık sayacıyla yaklaşık 3 saniye beklemek için kaç tık beklenmelidir? Bu yolla 10 ms'lik bir bekleme ölçülebilir mi?

   Cevap

   3 × 18,2065 ≈ 54,6; yaklaşık **55 tık**. 10 ms ölçülemez: bir tık yaklaşık 54,9 ms'dir ve sayaç bu süreden kısa aralıkları ayırt edemez.

8. Mod 13h'te (200, 150) pikselinin bağıl konumu nedir? Bir sağdaki ve bir alttaki piksellerin konumları nedir?

   Cevap

   150 × 320 + 200 = 48.200 = **BC48h**. Bir sağdaki (201, 150) BC49h, bir alttaki (200, 151) BC48h + 140h = BD88h'dedir.

9. BIOS veri alanında tampon başı 003Ch, tampon sonu 0020h okunuyor. Kaç tuş bekliyor?

> **Cevap**
>
> Tampon dairesel olduğu için son, başın gerisinde kalmış olabilir. Baştan alanın sonuna kadar (003Ch–003Dh) bir sözcük, alanın başından sona kadar (001Eh–001Fh) bir sözcük vardır: **2 tuş**. Genel formül (son − baş + 32) mod 32, bölü 2'dir: (20h − 3Ch + 20h) = 4 bayt.

10. Bir EXE programı `MOV AX, 4C00h` · `INT 21h` yerine `RET` ile bitiyor. Ne olur?

> **Cevap**
>
> EXE'nin yığınında DOS'un koyduğu 0000h sözcüğü yoktur ve CS PSP'yi göstermez. RET, yığının üstünde ne bulursa onu IP'ye yükler ve kod kesiminde rastgele bir adrese atlar. Program kilitlenebilir. EXE programı INT 21h/4Ch ile bitirilir.

5.2.14 İleri okuma

- Ralf Brown, *Ralf Brown's Interrupt List*: INT 10h, 16h, 1Ah ve 21h'nin bütün hizmetlerini, yazmaç düzeyinde ve BIOS sürüm farklarıyla anlatan başvuru listesi.
- IBM, *Personal Computer AT Technical Reference*, 1984: BIOS kaynak listesi; BIOS veri alanının, klavye ve zamanlayıcı kesmelerinin özgün tanımı.
- Peter Abel, *IBM PC Assembly Language and Programming*, 5. baskı: ekran, klavye ve DOS hizmetlerini çok sayıda örnek programla işler.
- Barry B. Brey, *The Intel Microprocessors*, 8. baskı: BIOS ve DOS çağrılarını 8086 donanımı ve kesme yapısıyla birlikte ele alır.

5.2.15 Kaynaklar

1. R. Brown, *Ralf Brown's Interrupt List* (INT 10h, 16h, 21h ayrıntılı başvuru).
2. IBM, *Personal Computer AT Technical Reference*, BIOS bölümü, 1984.
3. P. Abel, *IBM PC Assembly Language and Programming*, 5. baskı, Prentice Hall, 2001.
4. B. B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson, 2009.
5. emu8086 yardım belgeleri: *supported interrupt functions*.
6. Ş. Ç. Öztürk, O. Ergin, *Intel 8086 ile Mikroişlemci Programlamaya Giriş*, 2009.
7. Intel, *8086 16-Bit HMOS Microprocessor* veri sayfası (komut süreleri).

KISIM III

Komutlar ve Programlama

Bölüm 6–9

KISIM III · BÖLÜM 6

Aritmetik, Mantık ve Kaydırma Komutları

Bir işlemcinin asıl işi hesap yapmaktır. 8086'da bu işi yürütme biriminin içindeki aritmetik-mantık birimi (ALU) görür: iki sayıyı toplar, birini ötekinden çıkarır, bitlerini VE, VEYA, dışlayan VEYA kapılarından geçirir, sola ya da sağa kaydırır. Her işlemin sonunda ALU, sonucu yazmaca ya da belleğe yazmakla kalmaz; sonucun nasıl çıktığını bayrak yazmacındaki altı durum bitine de işler. Sonraki komutlar, özellikle koşullu atlamalar, kararlarını bu bitlere bakarak verir. Bu not 8086'nın aritmetik ve mantık komutlarını tek tek ele alır ve her komutun hangi bayrağı nasıl değiştirdiğini gösterir. Önce aynı bit dizisinin işaretsiz ve işaretli iki yorumu ile elde ve taşma bayrakları arasındaki fark açıklanır. Ardından toplama, çıkarma, karşılaştırma ve mantık komutları, çarpma ile bölmenin örtük yazmaçları, işaret genişletme, tek işlenenli komutlar, kaydırma ve döndürme gelir. 16 bitlik bir işlemcide 32 bitlik toplama, çıkarma ve çarpmanın nasıl yapıldığı adım adım izlenir. Son kısımlar dizi toplama, ikili kodlu onluk (BCD) aritmetik ve sıcaklık dönüşümü programlarını yazmaç ve bayrak düzeyinde inceler.

Kod (8086): `github.com/sercankulcu/microprocessors-8086/Bolum06`

ÖĞRENME ÇIKTILARI

Bayrakları hesaplamak
Bir toplama ya da çıkarmanın sonucundan CF, ZF, SF, OF, PF ve AF'nin değerini bulmak; aynı sonucun işaretsiz ve işaretli yorumunu ayırıp CF ile OF farkını açıklamak.

Komutları doğru işlenenle kullanmak
İki işlenenli, çarpma-bölme ve tek işlenenli komut gruplarının geçerli işlenen biçimlerini bilmek; her komutun hangi bayrağı değiştirdiğini, hangisini koruduğunu ve hangisini tanımsız bıraktığını söylemek.

Çarpma ve bölme
MUL, IMUL, DIV ve IDIV'in örtük yazmaçlarını (AL, AX, DX:AX) kullanmak; bölmeden önce bölüneni CBW, CWD ya da üst yarıyı sıfırlayarak hazırlamak; bölme hatasının nedenlerini tanımak.

Çok sözcüklü aritmetik
ADC ve SBB ile 32 bitlik toplama ve çıkarma, dört MUL ile 32 × 32 bitlik çarpma yapmak; eldenin sözcükten sözcüğe nasıl taşındığını izlemek.

Bit işlemleri
AND, OR, XOR ve TEST ile bit temizlemek, kurmak, terslemek ve sınamak; kaydırma ve döndürme komutlarının CF'ye etkisini izlemek; kaydırma ve toplamayla çarpıp bölmek.

Uygulama programlarını izlemek
Dizi toplama, BCD düzeltme ve sıcaklık dönüşümü programlarını yazmaç, bellek ve bayrak düzeyinde adım adım izlemek; programın geçerli olduğu sayı aralığını bulmak.

6.1 Bayraklar ve işlemin sonucu

6.1.1 ALU sonucu nereye yazar?

8086'nın içi iki birime ayrılır. **Yol arayüz birimi** (BIU) bellekten komut baytlarını getirir, önden getirme kuyruğuna koyar ve bellek işlenenlerini okuyup yazar. **Yürütme birimi** (EU) kuyruktan aldığı komutu çözer ve yürütür. Toplama, çıkarma, VE, VEYA, kaydırma gibi bütün hesaplar EU'nun içindeki 16 bitlik **aritmetik-mantık biriminde** (*arithmetic logic unit*, ALU) yapılır. ALU'nun iki girişi ve bir çıkışı vardır. Girişler yazmaçlardan, komutun içindeki anlık değerden ya da BIU'nun bellekten getirdiği sayıdan gelir. Çıkış hedef yazmaca ya da BIU aracılığıyla belleğe gider.

ALU'nun ikinci bir çıkışı daha vardır: sonucun özelliklerini anlatan birkaç bit. Sonuç sıfır mı, en soldaki biti 1 mi, en soldaki bitten dışarı elde çıktı mı? Bu sorulara verilen evet/hayır cevapları, 16 bitlik **bayrak yazmacındaki** (FLAGS) durum bitlerine yazılır. Şekil 6.1 bayrak yazmacının bit düzenini gösterir. On altı bitin dokuzu kullanılır. Altısı **durum bayrağıdır**: CF, PF, AF, ZF, SF ve OF son aritmetik ya da mantık işleminin sonucunu anlatır. Üçü **kontrol bayrağıdır**: TF, IF ve DF işlemcinin davranışını yönetir ve aritmetik komutlar onlara dokunmaz.



Şekil 6.1: 16 bitlik bayrak yazmacı. Kesik çizgili bitler 8086'da kullanılmaz; altı durum bayrağı (CF, PF, AF, ZF, SF, OF) aritmetik ve mantık komutlarının sonucunu tutar.

Bayraklar, komutlar arasında bilgi taşıyan bir köprüdür. `CMP AL, BL` komutu iki sayıyı karşılaştırır ama sonucu hiçbir yazmaca yazmaz; yalnız bayrakları değiştirir. Hemen ardından gelen `JZ`, `JC` ya da `JO` gibi bir koşullu atlama, kararını bu bayraklara bakarak verir. Çok sözcüklü toplamada ise ilk toplamanın eldesi CF'de bekler ve ikinci toplama onu ADC ile alır. Bu yüzden her komut için iki soru sorulur: sonucu nereye yazar ve hangi bayrakları nasıl değiştirir?

Tablo 6.1 durum ve kontrol bayraklarını, hangi koşulda 1 olduklarını ve kısa birer örneği toplar. Bayrakların her birinin ayrıntılı anlatımı ve donanımdaki üretimi 8086 yazmaçlarını anlatan notta verilmiştir. Burada yalnız bir toplama ya da çıkarmadan sonra bayrakları hesaplamaya yetecek kurallar özetlenir.

Bit	Bayrak	Tür	1 olur, eğer...	Örnek
0	CF (elde)	durum	işaretsiz taşma: toplamada en anlamlı bitten dışarı elde, çıkarmada ödünç	FFh + 01h → AL = 00h, CF = 1; 7 − 9 → CF = 1
2	PF (eşlik)	durum	sonucun düşük baytında çift sayıda 1 biti var	96h = 1001 0110b → dört 1 → PF = 1
4	AF (yardımcı elde)	durum	bit 3'ten bit 4'e elde ya da ödünç	08h + 08h = 10h → AF = 1
6	ZF (sıfır)	durum	sonucun bütün bitleri 0	`SUB AL, AL` → ZF = 1
7	SF (işaret)	durum	sonucun en anlamlı biti (MSB) 1	96h → SF = 1
8	TF (tuzak)	kontrol	her komuttan sonra INT 1 (tek adım)	hata ayıklayıcılar kullanır
9	IF (kesme)	kontrol	INTR ucundan gelen kesmeler kabul edilir	`STI` → 1, `CLI` → 0
10	DF (yön)	kontrol	dizgi komutları SI ve DI'yi azaltır	`CLD` → 0, `STD` → 1
11	OF (taşma)	durum	işaretli taşma: sonuç işaretli aralığa sığmadı	100 + 50 → AL = 96h (−106), OF = 1

Tablo 6.1: 8086'nın bayrakları. Durum bayrakları sonucu anlatır; kontrol bayrakları işlemcinin davranışını yönetir.

Durum bayraklarının donanımdaki karşılığı basittir. ALU'nun toplayıcısı bitleri sağdan sola, tam toplayıcı zinciriyle toplar. Bayt işleminde bit 7'nin, sözcük işleminde bit 15'in dışarı verdiği elde doğrudan CF'ye gider. Bit 3'ün verdiği elde AF'ye gider. SF sonucun en anlamlı bitinin kopyasıdır. ZF, sonucun bütün bitlerini giriş alan bir VEYA-DEĞİL (NOR) kapısının çıkışıdır. PF, düşük baytın sekiz bitini dışlayan VEYA ağacından geçirip tersler; sözcük sonucunda bile yalnız düşük bayta bakar. OF ise en anlamlı bite *giren* elde ile o bitten *çıkan* eldenin dışlayan VEYA'sıdır. Bu son kural aşağıdaki örneklerde sayılarla görülür.

Çıkarmada ALU ayrı bir çıkarıcı kullanmaz. A − B işlemini A + (B'nin birler tümleyeni) + 1 diye toplayarak yapar; bu, ikiye tümleyenle eklemektir. Bu toplamada en soldan elde çıkması, aslında ödünç gerekmediği anlamına gelir. 8086 bu yüzden çıkarmada eldeyi tersleyip CF'ye yazar. Sonuçta CF, çıkarmada **ödünç** (*borrow*) bayrağı gibi davranır: işaretsiz yorumda küçük sayıdan büyük sayı çıkarılmışsa CF = 1 olur.

6.1.2 CF mi, OF mi?

Bir yazmaçtaki sekiz bit, kendi başına ne işaretsiz ne işaretli bir sayıdır. 1001 0110b dizisi işaretsiz okunursa 150, ikiye tümleyenle işaretli okunursa −106 demektir. İşlemci toplamayı yalnız bitler üzerinde yapar ve iki yorum için aynı sonucu üretir. Hangi yorumun geçerli olduğunu bilen programcıdır. İşlemci her iki yorumu da hesaba katar ve iki ayrı bayrak tutar: CF işaretsiz yorumda, OF işaretli yorumda sonucun doğru olup olmadığını söyler.

Tanım 6.1

Elde bayrağı (*carry flag*, CF), işlem işaretsiz yorumla yapıldığında sonucun hedefin boyutuna sığmadığını gösterir. Bayt toplamasında gerçek toplam 255'i aşarsa, çıkarmada sonuç 0'ın altına inerse CF = 1 olur.

Taşma bayrağı (*overflow flag*, OF), işlem ikiye tümleyenli işaretli yorumla yapıldığında sonucun aralığa sığmadığını gösterir. Bayt için aralık −128…127, sözcük için −32.768…32.767'dir. Aynı işaretli iki sayının toplamı ters işaretli çıkarsa OF = 1 olur.

Tablo 6.2 beş bayt toplamasını iki yorumla yan yana koyar. Her satırda AL'ye yazılan sonuç tektir; değişen yalnız onun nasıl okunduğudur.

AL + işlenen	İşaretsiz yorum	İşaretli yorum	AL	CF	OF
`FFh + 01h`	255 + 1 = 256	−1 + 1 = 0	00h	1	0
`64h + 32h`	100 + 50 = 150	100 + 50 = 150	96h	0	1
`C8h + 64h`	200 + 100 = 300	−56 + 100 = 44	2Ch	1	0
`80h + 80h`	128 + 128 = 256	−128 + (−128) = −256	00h	1	1
`05h + 0Ah`	5 + 10 = 15	5 + 10 = 15	0Fh	0	0

Tablo 6.2: Aynı bayt toplamasının işaretsiz ve işaretli yorumu. CF işaretsiz, OF işaretli sonucun yanlış çıktığını söyler.

İlk satırda işaretsiz toplam 256'dır ve bir bayta sığmaz. AL'de 256 − 256 = 0 kalır, dışarı çıkan elde CF'ye düşer. İşaretli yorumda ise −1 + 1 = 0 tamamen doğrudur; OF = 0. Bu satır, CF = 1'in işaretli hesapta bir hata olmadığını gösterir. İkinci satır tam tersidir. 150, bayta işaretsiz olarak sığar, CF = 0'dır. Ama işaretli bayt en çok 127 olabilir; iki pozitif sayının toplamı 96h çıkmıştır ve 96h'nin işaret biti 1'dir. İşaretli yorumda bu −106 demektir ve yanlıştır: OF = 1.

Üçüncü satırda iki yorum yine ayrışır. İşaretsiz 300 sığmaz, CF = 1 olur. İşaretli yorumda ise sayılardan biri negatif (−56), öteki pozitiftir (100). Farklı işaretli iki sayının toplamı, mutlak değerce büyük olanından daha küçüktür; bu yüzden asla aralığın dışına çıkamaz. Sonuç 44 doğrudur ve OF = 0'dır. Dördüncü satırda iki yorum da yanlıştır: 256 bayta sığmaz, −256 de sığmaz. İki bayrak birlikte 1 olur. Son satırda iki yorum da doğrudur ve iki bayrak da 0'dır.

OF'nin kuralı, en anlamlı bitteki eldeler cinsinden de söylenebilir. İkinci satırı bit bit toplayalım: 0110 0100b + 0011 0010b. Bit 5'te 1 + 1 = 10b olur; 0 yazılır, elde 1 bit 6'ya geçer. Bit 6'da 1 + 0 + 1 = 10b; 0 yazılır, elde bit 7'ye girer. Bit 7'de 0 + 0 + 1 = 1 yazılır ve dışarı elde çıkmaz. Bit 7'ye giren elde 1, çıkan elde 0'dır. İkisi farklı olduğu için OF = 1'dir. Dördüncü satırda ise bit 7'ye hiç elde girmez ama 1 + 1 dışarı elde çıkarır: giren 0, çıkan 1, yine OF = 1. İlk satırda bit 7'ye elde girer ve dışarı da elde çıkar: ikisi aynı olduğundan OF = 0.

Kural (taşma bayrağı)

Toplamada OF = (en anlamlı bite giren elde) XOR (en anlamlı bitten çıkan elde). Bunun işaret diliyle söylenişi şudur: aynı işaretli iki sayının toplamı ters işaretli çıkarsa OF = 1 olur; farklı işaretli iki sayı toplanınca OF hiçbir zaman 1 olmaz. Çıkarmada (A − B) kural şu biçimi alır: işaretleri farklı iki sayının farkının işareti A'nınkinden farklıysa OF = 1 olur.

Sık yapılan hata

CF'yi "taşma" diye okumak ve işaretli sayılarla çalışırken CF'ye bakmak. −1 + 1 toplamasında CF = 1 olur, oysa işaretli sonuç doğrudur. İşaretli sayılarla çalışan bir program OF'ye (ve karşılaştırmada SF ile OF'ye), işaretsiz sayılarla çalışan bir program CF'ye bakar. İşlemci hangisinin geçerli olduğunu bilmez; seçim komutu yazanındır.

6.1.3 Kod izleme: CF, OF ve diğer bayraklar

Aşağıdaki kısa program bayrakların bir komuttan ötekine nasıl taşındığını gösterir. Program 9Ch ile başlar; 9Ch işaretsiz yorumda 156, işaretli yorumda −100'dür. Sonra iki ayrı toplama, ardından kendisiyle toplama, INC ve CMC gelir.

```
MOV  AL, 9Ch     ; 156 ya da -100
ADD  AL, 70h     ; + 112
MOV  AL, 50h     ; 80
ADD  AL, 50h     ; + 80
ADD  AL, AL      ; A0h + A0h
MOV  AL, 0FFh
INC  AL          ; CF'ye dokunmaz
CMC              ; CF tersine
RET
```

Komut	Makine kodu	AL	CF	OF	SF	ZF	AF	PF
`MOV AL, 9Ch`	`B0 9C`	9Ch	0	0	0	0	0	0
`ADD AL, 70h`	`04 70`	0Ch	**1**	0	0	0	0	1
`MOV AL, 50h`	`B0 50`	50h	1	0	0	0	0	1

Komut	Makine kodu	AL	CF	OF	SF	ZF	AF	PF
`ADD AL, 50h`	`04 50`	A0h	0	**1**	1	0	0	1
`ADD AL, AL`	`02 C0`	40h	**1**	**1**	0	0	0	0
`MOV AL, 0FFh`	`B0 FF`	FFh	1	1	0	0	0	0
`INC AL`	`FE C0`	00h	1	0	0	**1**	1	1
`CMC`	`F5`	00h	**0**	0	0	1	1	1

Tablo 6.3: CF, OF ve diğer bayrakların izlemesi. Her satır komut yürütüldükten sonraki durumdur; başlangıçta bütün durum bayrakları 0'dır.

İlk toplamada 156 + 112 = 268 bulunur. 268 bir bayta sığmaz; AL'de 268 − 256 = 12 = 0Ch kalır ve CF = 1 olur. İşaretli yorumda −100 + 112 = 12 doğrudur; işaretleri farklı iki sayı toplandığı için OF zaten 1 olamaz. 0Ch = 0000 1100b'de iki tane 1 vardır, bu yüzden PF = 1'dir. Düşük yarımlar C + 0 = C olduğundan bit 3'ten elde çıkmaz: AF = 0.

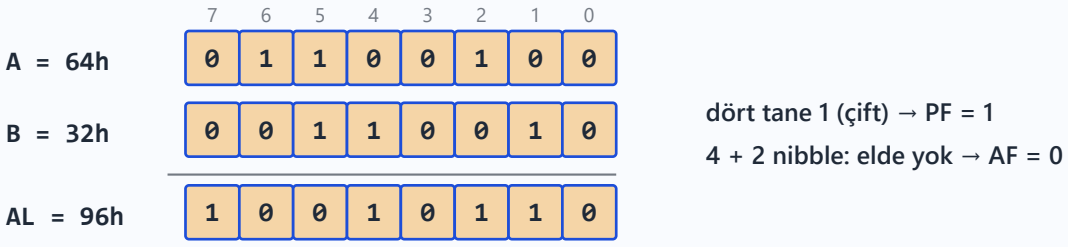
Üçüncü satırdaki MOV hiçbir bayrağı değiştirmez. Bu, sık unutulan bir kuraldır: MOV, LEA, XCHG gibi veri taşıma komutları ALU'dan geçmez ve bayraklar önceki aritmetik komuttan ne kaldıysa öyle kalır. CF = 1 ve PF = 1 yerinde durur.

Dördüncü satırda 80 + 80 = 160 işaretsiz yorumda doğrudur; CF = 0. Ama iki pozitif sayının toplamı A0h = 1010 0000b çıkmıştır ve işaret biti 1'dir. İşaretli yorumda bu −96'dır ve yanlıştır: OF = 1, SF = 1. Beşinci satırda A0h kendisiyle toplanır. İşaretsiz 160 + 160 = 320 sığmaz, CF = 1 olur. İşaretli −96 + (−96) = −192 de sığmaz, OF = 1 olur. AL'de kalan 40h iki yorumda da yanlış bir sonuçtur.

Yedinci satırdaki `INC AL`, FFh'ye 1 ekler ve AL = 00h olur. Sonuç sıfır olduğundan ZF = 1, düşük yarımdaki F + 1 elde verdiğinden AF = 1'dir. Aynı işlem `ADD AL, 1` ile yapılsaydı dışarı elde çıkacak ve CF = 1 olacaktı. INC ise CF'ye hiç dokunmaz; tablodaki CF = 1 beşinci satırdaki toplamadan kalmıştır. Son satırdaki `CMC` (*complement carry*) yalnız CF'yi tersler. Aynı ailede `STC` CF'yi 1, `CLC` CF'yi 0 yapar.

6.1.4 ADD ile bayraklar adım adım

Bayrakları elle hesaplamanın en güvenilir yolu, toplamayı kâğıt üzerinde ikili tabanda yapmaktır. `MOV AL, 64h` ve `ADD AL, 32h` komutları 100 ile 50'yi toplar. Şekil 6.2 iki işleneni, sonucu ve her bayrağın hangi bitten okunduğunu gösterir.



Şekil 6.2: 64h + 32h = 96h. Bitler sağdan sola toplanır; bayraklar sonucun bitlerinden ve eldelerden okunur.

1. **Sonuç.** 0110 0100b + 0011 0010b = 1001 0110b = 96h. İşaretsiz yorumla 150 doğrudur, işaretli yorumla −106 yanlıştır.
2. **CF.** En soldaki bitten dışarı elde çıkmadı; 150 ≤ 255. CF = 0.
3. **OF.** Bit 7'ye giren elde 1, çıkan elde 0. İki pozitif sayının toplamı negatif göründü. OF = 1.
4. **SF ve ZF.** Sonucun bit 7'si 1'dir, SF = 1. Sonuç sıfır değildir, ZF = 0.
5. **PF.** 1001 0110b'de dört tane 1 vardır; dört çift sayıdır, PF = 1.
6. **AF.** Düşük yarımlar 4 + 2 = 6'dır; bit 3'ten bit 4'e elde geçmez, AF = 0.

Sonuç: AL = 96h; CF = 0, ZF = 0, SF = 1, OF = 1, PF = 1, AF = 0. Bu hesap emu8086'da çalıştırıldığında bayrak penceresinde aynı değerler görülür. Sunumdaki bayrak hesaplayıcısı ve Bayrak Laboratuvarı sayfası, başka sayılar ve başka komutlar için aynı tabloyu kurar.

Örnek 6.1

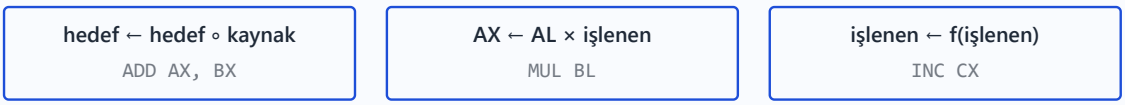
`MOV AL, 0C0h` ve `ADD AL, 0C0h` sonrası bayraklar nedir?

C0h = 1100 0000b; işaretsiz 192, işaretli −64. Toplam 1 1000 0000b = 180h'dir. AL'ye 80h yazılır ve dışarı çıkan elde CF'ye düşer: CF = 1. Bit 7'ye giren elde 1 (bit 6'daki 1 + 1'den), çıkan elde de 1'dir; OF = 0. Gerçekten −64 + (−64) = −128 işaretli bayta tam sığar. SF = 1, ZF = 0. 80h'de tek bir 1 biti vardır: PF = 0. Düşük yarımlar 0 + 0 olduğundan AF = 0'dır.

6.2 Komut grupları ve işlenen kuralları

6.2.1 Üç grup

8086'nın aritmetik ve mantık komutları, işlenenlerini nasıl aldıklarına göre üç gruba ayrılır. Bu ayrım ezberi kolaylaştırır: aynı gruptaki komutlar aynı işlenen biçimlerini kabul eder ve makine kodunda da benzer kodlanır. Şekil 6.3 üç grubun işlenen düzenini yan yana gösterir.



Şekil 6.3: Üç grubun işlenen biçimleri. 1. grupta sonuç hedefe yazılır; 2. grupta örtük AL, AX ve DX kullanılır; 3. grupta tek işlenen hem kaynak hem hedeftir.

- **1. grup, iki işlenenli komutlar:** ADD, SUB, CMP, AND, TEST, OR, XOR. Bu listeye eldeli toplama ADC ile ödünçlü çıkarma SBB de eklenir. Biçim `komut hedef, kaynak`'tır; işlem hedef ile kaynak arasında yapılır ve sonuç hedefe yazılır.
- **2. grup, çarpma ve bölme:** MUL, IMUL, DIV, IDIV. Komutta tek bir işlenen yazılır. İşlemin öteki yarısı ve sonucun gideceği yer sabittir: AL, AX ya da DX:AX. Bunlara **örtük işlenen** denir.
- **3. grup, tek işlenenli komutlar:** INC, DEC, NOT, NEG. İşlenen hem okunur hem yazılır: `INC CX` CX'i okur, 1 ekler ve sonucu yine CX'e yazar.

Tablo 6.4 her komutun ne yaptığını tek satırda özetler. Kaydırma ve döndürme komutları da tek işlenen alır, ama kaydırma sayısını ikinci bir işlenen olarak verdikleri için ayrı bir aile olarak ele alınır.

Grup	Komut	Ne yapar?
1	`ADD`	iki değeri toplar, sonucu hedefe yazar
	`SUB`	kaynağı hedeften çıkarır, sonucu hedefe yazar
	`CMP`	kaynağı hedeften çıkarır, sonucu saklamaz; yalnız bayrakları ayarlar
	`AND`	bit düzeyinde VE
	`TEST`	bit düzeyinde VE; sonucu saklamaz
	`OR`	bit düzeyinde VEYA
	`XOR`	bit düzeyinde dışlayan VEYA
2	`MUL`	işaretsiz çarpma
	`IMUL`	işaretli çarpma
	`DIV`	işaretsiz bölme (bölüm ve kalan)
	`IDIV`	işaretli bölme (bölüm ve kalan)
3	`INC`	1 artırır
	`DEC`	1 azaltır
	`NOT`	bütün bitleri tersler (birler tümleyeni)
	`NEG`	işaretini değiştirir (ikiye tümleyen)

Tablo 6.4: Üç gruptaki komutların anlamı.

6.2.2 İki işlenenli komutların işlenenleri

1. gruptaki komutların hepsi aynı beş işlenen birleşimini kabul eder. Bunlar Tablo 6.5'te örnekleriyle verilmiştir. REG, sekiz ya da on altı bitlik genel amaçlı yazmaçlardan biridir: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, SI, DI, BP, SP. Memory bir bellek işlenenidir (`[BX]`, `[BX+SI+7]`, bir değişken adı). Immediate ise komutun içine yazılmış anlık değerdir (5, −24, 3Fh, 10001101b).

Biçim	Örnek	Makine kodu
REG, memory	`ADD AX, [BX]`	`03 07`
memory, REG	`SUB var1, CL`	`28 0E` + adres
REG, REG	`XOR AX, AX`	`33 C0`
memory, immediate	`AND BYTE PTR [SI], 0Fh`	`80 24 0F`
REG, immediate	`CMP AL, 'A'`	`3C 41`

Tablo 6.5: ADD, SUB, CMP, AND, TEST, OR ve XOR'un geçerli işlenen birleşimleri.

Kurallar MOV'un kurallarına benzer ve aynı donanım kısıtından gelir. İki işlenen birden bellekte olamaz: `ADD [BX], [SI]` geçersizdir, çünkü komutun ModR/M baytı yalnız tek bir bellek adresi tarif edebilir. İki işlenen aynı boyutta olmalıdır: `ADD AL, BX` 8 bitlik bir yazmaca 16 bitlik bir sayıyı eklemeye çalışır ve assembler reddeder. Hedef sabit olamaz: `ADD 5, AL` sonucu nereye yazacağını bilemez. Kesim yazmaçları aritmetik işleneni olamaz: `ADD DS, AX` geçersizdir; DS yalnız MOV, PUSH ve POP ile değiştirilir.

Bellek işleneni ile sabit birlikte kullanıldığında boyut ikisinden de anlaşılmaz. `AND [SI], 0Fh` yazımında 0Fh bir bayt da bir sözcük de olabilir; bu yüzden `BYTE PTR` ya da `WORD PTR` belirtilmelidir. İşlenenlerden biri yazmaç ya da türü bilinen bir değişkense boyut ondan alınır.

> **Sık yapılan hata**
>
> İki değişkeni doğrudan toplamaya çalışmak: `ADD toplam, sayi`. İki işlenen de bellektedir ve bu komut yoktur. Biri önce bir yazmaca alınır: `MOV AX, sayi` ve `ADD toplam, AX`.

6.2.3 Sonuç nereye yazılır, bayraklar nasıl etkilenir?

İki işlenenli bir işlemden sonra sonuç her zaman ilk işlenende, yani hedefte saklanır. `SUB AX, BX` komutu AX − BX'i hesaplar ve AX'e yazar; BX değişmez. CMP ile TEST bu kuralın iki istisnasıdır. CMP, SUB'ın yaptığı çıkarmayı yapar ama sonucu atar. TEST, AND'in yaptığı bit bit VE'yi yapar ama sonucu atar. İkisi de yalnız bayrakları değiştirir ve böylece bir yazmacı bozmadan sınamayı mümkün kılar.

Bayrak etkisi, işlemin aritmetik mi mantık mı olduğuna bağlıdır. Tablo 6.6 iki aileyi karşılaştırır.

Komut	CF	ZF	SF	OF	PF	AF
`ADD, ADC, SUB, SBB, CMP`	✓	✓	✓	✓	✓	✓

Komut	CF	ZF	SF	OF	PF	AF
`AND, OR, XOR, TEST`	0	✓	✓	0	✓	?

Tablo 6.6: İki işlenenli komutların bayrak etkisi. ✓ sonuca göre ayarlanır, 0 her zaman temizlenir, ? tanımsızdır.

Aritmetik komutlar altı durum bayrağının hepsini sonuca göre ayarlar. Mantık komutlarında ise elde kavramı yoktur: her bit kendi başına işlenir ve bir bitten ötekine hiçbir şey geçmez. Bu yüzden CF ve OF her zaman 0 yapılır. ZF, SF ve PF sonuca göre ayarlanır. AF'nin değeri Intel belgelerinde tanımsız bırakılmıştır; bir program mantık komutundan sonra AF'ye güvenmemelidir. Tanımsız, bayrağın rastgele olduğu anlamına gelmez; gerçek işlemcide belirli bir değer alır, ama bu değer hiçbir belgede söz verilmiş değildir ve başka bir x86 işlemcide farklı çıkabilir.

6.2.4 Süre: işlemci içinde ve yolda geçen zaman

Bir komutun süresi, işlenenlerinin nerede durduğuna sıkı sıkıya bağlıdır. Yazmaçtaki iki sayı ALU'ya doğrudan girer; `ADD AX, BX` 8086'da 3 saat turu sürer. Bellek işleneni varsa EU önce etkin adresi hesaplar, sonra BIU'dan bir okuma yol döngüsü ister. Bir yol döngüsü en az 4 saat turudur: adres yola konur, bellek okunur, veri yongaya alınır. Hedef bellekteyse sonucu geri yazmak için bir yol döngüsü daha gerekir. Bu okuma-değiştirme-yazma sırasında 8086'nın RD ucu önce, WR ucu sonra etkin olur. Tablo 6.7 Intel'in 8086 kılavuzundaki süreleri verir; EA, etkin adresin hesap süresidir (5–12 tur).

Biçim	ADD, ADC, SUB, SBB, AND, OR, XOR	CMP	TEST
reg, reg	3	3	3
reg, mem	9 + EA	9 + EA	9 + EA
mem, reg	16 + EA	9 + EA	9 + EA
reg, imm	4	4	5
mem, imm	17 + EA	10 + EA	11 + EA
AL/AX, imm	4	4	4

Tablo 6.7: İki işlenenli komutların 8086'daki saat turu süreleri (Intel kılavuzu). Tek adresteki sözcük erişimine her aktarım için 4 tur eklenir.

Tablodaki farkların kaynağı yol döngüleridir. `ADD [BX], AX` belleği hem okur hem yazar ve 16 + EA tur sürer. `CMP [BX], AX` aynı çıkarmayı yapar ama sonucu yazmadığı için yalnız okur: 9 + EA tur. 5 MHz'lik bir 8086'da bir saat turu 200 ns'dir; `ADD [BX], AX` (EA = 5) yaklaşık 21 × 200 ns = 4,2 µs sürer. Aynı toplama iki yazmaç arasında 0,6 µs'dir. Sık kullanılan değerleri döngü boyunca yazmaçta tutmanın nedeni budur.

6.3 Toplama, çıkarma ve karşılaştırma

6.3.1 Kod izleme: ADD, SUB, CMP

Aşağıdaki program bir toplama, bir çıkarma ve iki karşılaştırma yapar; son satır negatif bir sayıyı sıfıra tamamlar. Her komutun bayrak etkisi, işlemin bit düzeyindeki ayrıntısıyla birlikte izlenir.

```
mov al, 0F0h     ; 240
add al, 20h      ; 240 + 32 = 272
mov bl, 7
sub bl, 9        ; 7 - 9
cmp al, 10h      ; AL - 10h
cmp bl, 0
add bl, 2        ; FEh + 2
```

Komut	Makine kodu	AL	BL	CF	ZF	SF	OF	PF	AF
`mov al, 0F0h`	`B0 F0`	F0h	00h	0	0	0	0	0	0
`add al, 20h`	`04 20`	10h	00h	**1**	0	0	0	0	0
`mov bl, 7`	`B3 07`	10h	07h	1	0	0	0	0	0
`sub bl, 9`	`80 EB 09`	10h	FEh	**1**	0	1	0	0	1
`cmp al, 10h`	`3C 10`	10h	FEh	0	**1**	0	0	1	0
`cmp bl, 0`	`80 FB 00`	10h	FEh	0	0	1	0	0	0
`add bl, 2`	`80 C3 02`	10h	00h	**1**	**1**	0	0	1	1

Tablo 6.8: ADD, SUB ve CMP izlemesi (başlangıçta bütün durum bayrakları 0).

Satır 2. F0h + 20h = 110h'dir. Dokuz bitlik bu sonucun düşük sekiz biti, 10h, AL'ye yazılır; dışarı taşan 1 CF'ye düşer. İşaretli yorumda −16 + 32 = 16 doğrudur ve OF = 0'dır. 10h = 0001 0000b'de tek bir 1 vardır, PF = 0. Düşük yarımlar 0 + 0 olduğundan AF = 0. Makine kodu `04 20`'dir: 04, "AL'ye anlık bayt ekle" anlamına gelen kısa akümülatör biçimidir.

Satır 4. 7 − 9 = −2 bulunur ve 8 bitte ikiye tümleyenle FEh olarak yazılır. Küçük sayıdan büyük sayı çıkarıldığı için ödünç alınmıştır: CF = 1. Sonucun en anlamlı biti 1'dir, SF = 1. Düşük yarımlarda da 7 − 9 ödünç ister, AF = 1. İşaretli yorumda 7 − 9 = −2 doğrudur ve OF = 0'dır. Bu komut AL'yi değil BL'yi kullandığı için kısa biçim seçilemez; genel `80 /5` biçimi üç bayt tutar.

Satır 5. `CMP AL, 10h`, 10h − 10h = 0 çıkarmasını yapar. Sonuç sıfırdır, ZF = 1. Sonuç saklanmaz: AL hâlâ 10h'dir. Eşitlik sınamasının özü budur: iki sayı eşitse farkları sıfırdır ve ZF = 1 olur. Hemen ardından `JE` (ya da aynı komutun öteki adı `JZ`) gelirse atlama yapılır. Önceki satırdan kalan CF = 1 ve AF = 1 de silinmiştir; CMP bütün durum bayraklarını yeniden yazar.

Satır 6. `CMP BL, 0`, FEh − 0 işlemidir. Sonuç FEh'dir: sıfır değil (ZF = 0), işaret biti 1 (SF = 1), ödünç yok (CF = 0). Bu üç bayrak iki farklı yorumu taşır. İşaretsiz okunursa FEh = 254 > 0'dır; bunu CF = 0 söyler. İşaretli okunursa FEh = −2 < 0'dır; bunu SF ≠ OF söyler.

Satır 7. FEh + 2 = 100h'dir. BL'de 00h kalır; CF = 1 ve ZF = 1 olur. İşaretli yorumda −2 + 2 = 0 doğrudur, OF = 0. Bu satır CF ile OF arasındaki farkı bir kez daha gösterir: CF = 1, işaretli hesabın yanlış olduğu anlamına gelmez.

6.3.2 CMP ile büyüklük karşılaştırması

CMP'nin asıl işi, iki sayının eşit olup olmadığını değil hangisinin büyük olduğunu bulmaktır. `CMP A, B` A − B'yi hesaplar. İşaretsiz sayılarda A < B ise çıkarma ödünç ister ve CF = 1 olur. İşaretli sayılarda durum daha inceliklidir. Fark taşmadıysa (OF = 0) sonucun işareti doğrudur ve A < B ise SF = 1'dir. Fark taşmışsa (OF = 1) işaret biti ters dönmüştür; bu kez A < B, SF = 0 ile gösterilir. İki durum tek bir kuralda birleşir: işaretli A < B ⇔ SF ≠ OF.

Komutlar	CF	SF	OF	İşaretsiz	İşaretli
`AL = 05h, BL = FEh` ; `CMP AL, BL`	1	0	0	5 < 254	5 > −2 (SF = OF)
`AL = 7Fh, BL = 80h` ; `CMP AL, BL`	1	1	1	127 < 128	127 > −128 (SF = OF)
`AL = 80h, BL = 01h` ; `CMP AL, BL`	0	0	1	128 > 1	−128 < 1 (SF ≠ OF)

Tablo 6.9: Üç CMP örneği. İşaretsiz sonuç CF'den, işaretli sonuç SF ile OF'nin karşılaştırmasından okunur.

İkinci satırda 7Fh − 80h = FFh çıkar: SF = 1. Ama işaretli 127 − (−128) = 255 bayta sığmaz, OF = 1'dir. İşaret biti burada yanıltır ve SF = OF olduğu için doğru karar "büyük"tür. Koşullu atlamalar bu kuralları hazır hâlde sunar. İşaretsiz karşılaştırmada `JB` / `JA` (below/above, CF ve ZF'ye bakar), işaretli karşılaştırmada `JL` / `JG` (less/greater, SF, OF ve ZF'ye bakar) kullanılır. Aynı CMP'den sonra hangi atlamanın seçildiği, sayıların nasıl yorumlanacağını belirler.

6.3.3 ADC ile 32 bitlik toplama

8086'nın ALU'su en çok 16 bitlik sayıları toplar. 131.071 gibi 16 bite sığmayan bir sayı iki sözcüğe bölünerek saklanır: DX yüksek sözcüğü, AX düşük sözcüğü tutar ve çift DX:AX diye yazılır. İki 32 bitlik sayıyı toplamak için kâğıt üzerindeki toplamanın yöntemi izlenir: önce sağdaki basamaklar toplanır, eldesi soldaki basamaklara eklenir. Burada "basamak" 16 bitlik bir sözcüktür ve elde CF'de taşınır.

Tanım 6.2

ADC (*add with carry*, eldeli toplama) hedef ← hedef + kaynak + CF işlemini yapar. **SBB** (*subtract with borrow*, ödünçlü çıkarma) hedef ← hedef − kaynak − CF işlemini yapar. İkisinin işlenen kuralları ve bayrak etkisi ADD ile SUB'ınkiyle aynıdır.

```
; DX:AX = 0001FFFFh, CX:BX = 00020001h
mov ax, 0FFFFh
mov dx, 0001h
mov bx, 0001h
mov cx, 0002h
add ax, bx      ; düşük sözcükler
adc dx, cx      ; yüksek + CF
```

Komut	Makine kodu	DX	AX	CX	BX	CF	ZF
`mov ax, 0FFFFh`	`B8 FF FF`	0000h	FFFFh	0000h	0000h	0	0
`mov dx, 0001h`	`BA 01 00`	0001h	FFFFh	0000h	0000h	0	0
`mov bx, 0001h` · `mov cx, 0002h`	`BB 01 00` · `B9 02 00`	0001h	FFFFh	0002h	0001h	0	0
`add ax, bx`	`03 C3`	0001h	**0000h**	0002h	0001h	**1**	1
`adc dx, cx`	`13 D1`	**0004h**	0000h	0002h	0001h	0	0

Tablo 6.10: 0001FFFFh + 00020001h toplamasının izlemesi.

İlk sayı DX:AX = 0001FFFFh = 131.071, ikinci sayı CX:BX = 00020001h = 131.073'tür. `ADD AX, BX` düşük sözcükleri toplar: FFFFh + 0001h = 10000h. AX'e 0000h yazılır, dışarı çıkan 1 CF'ye düşer. Bu CF, yüksek sözcüğe taşınacak eldedir. ZF = 1 olması burada yanıltıcıdır: yalnız düşük sözcük sıfırdır, 32 bitlik toplam sıfır değildir. `ADC DX, CX` bu eldeyi kullanır: 0001h + 0002h + 1 = 0004h. Sonuç DX:AX = 00040000h = 262.144'tür ve 131.071 + 131.073 ile tutar.

Yöntem sözcük sayısından bağımsızdır. 48 bitlik bir sayı üç sözcükte tutulursa ilk sözcükler ADD ile, sonraki iki sözcük ADC ile toplanır. Elde zincir boyunca CF'de taşınır. ADD ile ADC arasına CF'yi değiştiren bir komut girmemelidir. MOV, INC, DEC ve LOOP CF'ye dokunmaz; bu yüzden bir döngü içinde sayaç artırılırken bile elde korunur. Buna karşılık araya giren bir ADD ya da CMP eldeyi siler.

6.3.4 SBB ile 32 bitlik çıkarma

Çıkarmada da aynı düzen geçerlidir. En düşük sözcük SUB ile çıkarılır; düşük sözcük yetmezse bir üst sözcükten ödünç alınır ve bu ödünç CF'de bekler. SBB, üst sözcükten kaynakla birlikte bu ödüncü de çıkarır.

```
; DX:AX = 00040000h, CX:BX = 00010001h
mov ax, 0000h
mov dx, 0004h
mov bx, 0001h
mov cx, 0001h
sub ax, bx      ; düşük sözcükler
sbb dx, cx      ; yüksek - CF
```

Komut	Makine kodu	DX	AX	CX:BX	CF	SF
`mov ax, 0000h` · `mov dx, 0004h`	`B8 00 00` · `BA 04 00`	0004h	0000h	0000:0000h	0	0
`mov bx, 0001h` · `mov cx, 0001h`	`BB 01 00` · `B9 01 00`	0004h	0000h	0001:0001h	0	0
`sub ax, bx`	`2B C3`	0004h	**FFFFh**	0001:0001h	**1**	1
`sbb dx, cx`	`1B D1`	**0002h**	FFFFh	0001:0001h	0	0

Tablo 6.11: 00040000h – 00010001h çıkarmasının izlemesi.

Birinci sayı 262.144, çıkarılan 65.537'dir. `SUB AX, BX`, 0000h − 0001h işlemini yapar. Düşük sözcük yetmez; ödünç alınır, AX = FFFFh ve CF = 1 olur. `SBB DX, CX` ödüncü öder: 0004h − 0001h − 1 = 0002h. Sonuç DX:AX = 0002FFFFh = 196.607'dir ve 262.144 − 65.537 ile tutar.

Örnek 6.2

00012345h − 0000ABCDh işlemi SUB ve SBB ile nasıl yapılır? SUB sonrası CF kaçtır?

Birinci sayı DX:AX = 0001:2345h, ikinci sayı CX:BX = 0000:ABCDh olarak yüklenir. `SUB AX, BX` : 2345h − ABCDh negatiftir; 16 bitte 12345h − ABCDh = 7778h kalır ve ödünç alındığı için CF = 1 olur. `SBB DX, CX` : 0001h − 0000h − 1 = 0000h. Sonuç 00007778h = 30.584'tür (74.565 − 43.981).

Sık yapılan hata

İki sözcükte de SUB (ya da ADD) kullanmak. Böyle yazılırsa düşük sözcükten gelen ödünç ya da elde kaybolur; sonuç tam 10000h kadar yanlış çıkar. Kural: en düşük sözcükte ADD/SUB, sonraki her sözcükte ADC/SBB.

6.4 Mantıksal işlemler

6.4.1 Bit bit çalışan kapılar

Mantık komutları, sayısal tasarımdan bilinen kapıların yazılım karşılığıdır. Tablo 6.12 dört işlemin doğruluk tablosunu verir. AND ancak iki giriş de 1 ise 1 verir. OR en az bir giriş 1 ise 1 verir. XOR girişler farklıysa 1 verir. NOT tek girişlidir ve girişi tersler.

A	B	A AND B	A OR B	A XOR B	NOT A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0

A	B	A AND B	A OR B	A XOR B	NOT A
1	1	1	1	0	0

Tablo 6.12: AND, OR, XOR ve NOT doğruluk tabloları.

8086'nın mantık komutları bu kuralları bir sayının bütününe değil, bitlerine tek tek uygular. `AND AL, BL` komutu aslında sekiz bağımsız VE kapısıdır: AL'nin bit 0'ı BL'nin bit 0'ıyla, bit 1'i bit 1'iyle VE'lenir ve sonuç AL'nin aynı bitine yazılır. Komşu bitler arasında elde ya da ödünç geçmez. ALU'da bu işlem, toplayıcının elde zinciri devre dışı bırakılarak yapılır ve her bitte doğru kapının çıkışı seçilir. Bu yüzden mantık komutları toplama kadar hızlıdır: yazmaçlar arasında 3 saat turu.

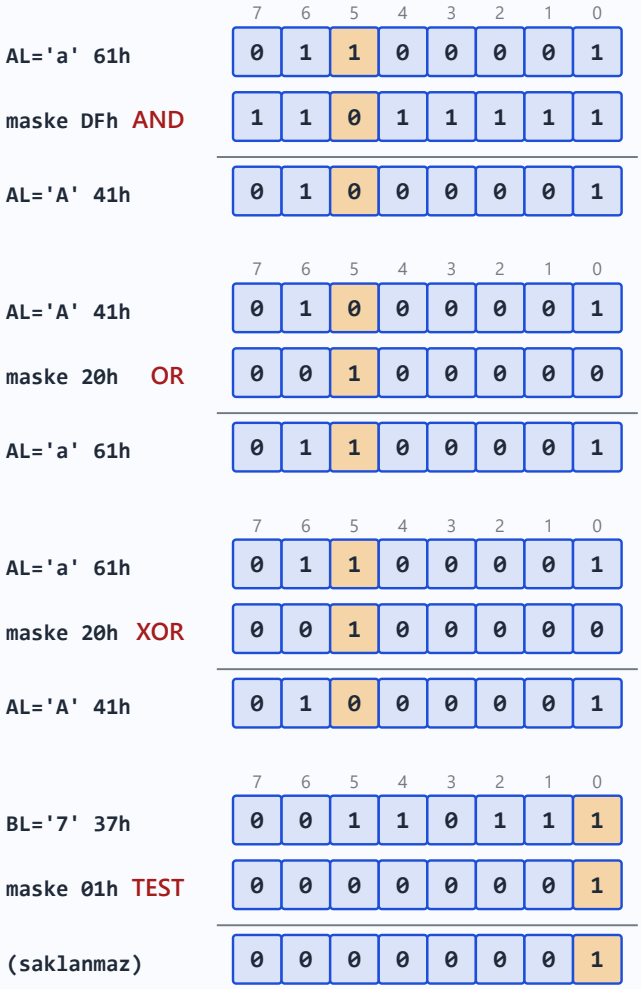
Bit bit işlemenin bir sonucu, elde ve taşmanın anlamsız olmasıdır. 8086 bunu açıkça belirtir: AND, OR, XOR ve TEST'ten sonra CF = 0 ve OF = 0'dır. ZF, SF ve PF sonuca göre ayarlanır, AF tanımsızdır. NOT ise bir istisnadır: hiçbir bayrağı değiştirmez.

6.4.2 Maskeleme

Mantık komutlarının en yaygın kullanımı bir sayının bazı bitlerini değiştirip ötekileri korumaktır. Bunun için ikinci işlenen olarak bir **maske** verilir. Maskenin her biti, karşı gelen bitin başına ne geleceğini seçer. Bu yönteme **maskeleme** (*masking*) denir.

- **AND ile bit temizleme.** x AND 0 = 0, x AND 1 = x'tir. Maskede 0 olan bitler sıfırlanır, 1 olanlar korunur.
- **OR ile bit kurma.** x OR 1 = 1, x OR 0 = x'tir. Maskede 1 olan bitler kesin 1 olur, 0 olanlar korunur.
- **XOR ile bit tersleme.** x XOR 1 = NOT x, x XOR 0 = x'tir. Maskede 1 olan bitler terslenir, 0 olanlar korunur.
- **TEST ile bit sınama.** TEST, AND yapar ama sonucu yazmaz. Maskedeki bit (ya da bitlerden biri) 1 ise ve sayıda da 1 ise sonuç sıfırdan farklıdır: ZF = 0. Sayı değişmez.

ASCII kodlamasında büyük ve küçük harfler yalnız bit 5'te ayrılır: 'A' = 41h = 0100 0001b, 'a' = 61h = 0110 0001b. Bu yüzden harf dönüşümü tek bir mantık komutuyla yapılır. Şekil 6.4 dört işlemi aynı harflerle gösterir.



Şekil 6.4: Maskeleme. AND DFh bit 5'i temizler ('a' → 'A'); OR 20h bit 5'i kurar ('A' → 'a'); XOR 20h bit 5'i tersler; TEST 01h bit 0'ı sınar ve BL'yi değiştirmez.

Maske değerleri ikili yazıldığında anlamı açıkça görülür. DFh = 1101 1111b, yalnız bit 5'i 0 olan bir maskedir; AND ile bit 5'i siler. 20h = 0010 0000b, yalnız bit 5'i 1 olan maskedir; OR ile bit 5'i kurar, XOR ile tersler. Son şekilde BL = '7' = 37h = 0011 0111b, maske 01h ile TEST edilir. Sonuç 01h ≠ 0 olduğundan ZF = 0'dır: bit 0 birdir, yani sayı tektir.

6.4.3 Sık kullanılan kalıplar

Bazı mantık komutu kullanımları o kadar yaygındır ki assembly programlarında neredeyse kalıp hâline gelmiştir. Tablo 6.13 en sık rastlananları toplar.

Kod	Etki	Neden?
`XOR AX, AX`	AX = 0, ZF = 1, CF = 0	2 bayt (`33 C0`); `MOV AX, 0` 3 bayttır (`B8 00 00`) ve bayraklara dokunmaz
`AND AL, 0Fh`	'7' (37h) → 7	ASCII rakamı sayıya çevirir; yüksek yarım silinir
`OR AL, 30h`	7 → '7' (37h)	0–9 arası sayıyı ASCII rakama çevirir
`AND AL, 0DFh`	'a' → 'A'	bit 5'i temizler: büyük harf

Kod	Etki	Neden?
`OR AL, 20h`	'A' → 'a'	bit 5'i kurar: küçük harf
`TEST AL, 1`	ZF = 0 ise tek	AL değişmeden bit 0 sınanır
`TEST AL, 80h`	ZF = 0 ise negatif	işaret biti sınanır
`OR AX, AX`	AX değişmez	ZF ve SF ayarlanır: AX sıfır mı, negatif mi?

Tablo 6.13: Mantık komutlarıyla sık kullanılan kalıplar.

ASCII rakamları '0' = 30h'den '9' = 39h'ye kadar uzanır. Yüksek yarım her rakamda 3'tür, düşük yarım rakamın değeridir. `AND AL, 0Fh` yüksek yarımı sildiği için klavyeden okunan bir rakamı sayıya çevirir. `OR AL, 30h` bunun tersini yapar ve ekrana yazılacak karakteri hazırlar. Aynı iş `SUB AL, 30h` ve `ADD AL, 30h` ile de yapılabilir; mantık komutlarının farkı, girdi zaten dönüştürülmüşse yine doğru sonuç vermesidir.

`OR AX, AX` ilk bakışta anlamsızdır: bir sayıyı kendisiyle VEYA'lamak onu değiştirmez. Komutun amacı bayraklardır. AX sıfırsa ZF = 1, en anlamlı biti 1 ise SF = 1 olur. Aynı iş `TEST AX, AX` ve `CMP AX, 0` ile de yapılır. OR ile TEST iki bayt tutar; CMP ise sabiti de taşıdığı için üç bayttır.

6.4.4 Kod izleme: AND, OR, XOR, TEST

```
mov al, 'a'      ; 61h
and al, 0DFh     ; büyük harf
or  al, 20h      ; küçük harf
xor al, 20h      ; tersle
mov bl, '7'      ; 37h
and bl, 0Fh      ; sayı 7
test bl, 1       ; tek mi?
xor bl, bl       ; BL = 0
```

Komut	Makine kodu	AL	BL	ZF	SF	PF	CF	OF
`mov al, 'a'`	`B0 61`	61h	00h	0	0	0	0	0
`and al, 0DFh`	`24 DF`	41h 'A'	00h	0	0	1	0	0
`or al, 20h`	`0C 20`	61h 'a'	00h	0	0	0	0	0
`xor al, 20h`	`34 20`	41h 'A'	00h	0	0	1	0	0
`mov bl, '7'`	`B3 37`	41h	37h	0	0	1	0	0
`and bl, 0Fh`	`80 E3 0F`	41h	07h	0	0	0	0	0
`test bl, 1`	`F6 C3 01`	41h	07h	**0**	0	0	0	0
`xor bl, bl`	`32 DB`	41h	00h	**1**	0	1	0	0

Tablo 6.14: Mantık komutlarının izlemesi. Mantık komutlarından sonra CF = OF = 0'dır; AF tanımsızdır (tabloda gösterilmedi).

61h AND DFh = 41h = 'A' bulunur. 41h = 0100 0001b'de iki tane 1 vardır, PF = 1. 41h OR 20h = 61h = 'a'; 0110 0001b'de üç tane 1 vardır, PF = 0. 61h XOR 20h yine 41h verir: bit 5 terslenmiştir. XOR komutu iki kez uygulandığında sayıyı eski hâline getirir; bu özellik basit şifreleme ve bit değiştirme işlerinde kullanılır.

BL'ye ASCII '7' = 37h yüklenir ve `AND BL, 0Fh` onu sayısal 7'ye çevirir. `TEST BL, 1` 07h AND 01h = 01h hesaplar; sonuç sıfır değildir, ZF = 0'dır ve sayı tektir. BL değişmeden 07h kalır. Son satırda BL kendisiyle XOR'lanır. Her bit kendisinden farklı olmadığı için sonuç 00h'dir; ZF = 1 olur. Sıfır tane 1 biti çift sayıdır, PF = 1.

Tablonun makine kodu sütunu, AL ile çalışan komutların kısa biçimlerini gösterir. `AND AL, imm8` = 24, `OR AL, imm8` = 0C, `XOR AL, imm8` = 34, iki baytlık komutlardır. BL ile aynı işlemler genel `80` ya da `F6` biçimine düşer ve üç bayt tutar. Bu, AL ve AX'in akümülatör olarak taşıdığı küçük ama sürekli bir üstünlüktür.

> **Sık yapılan hata**
>
> Bir biti sınamak için AND kullanmak ve sayıyı bozmak. `AND AL, 80h` işaret bitine bakar ama AL'nin öteki yedi bitini siler. Sayı sonra yeniden kullanılacaksa TEST yazılmalıdır. Aynı ilişki CMP ile SUB arasında da vardır.

6.5 Çarpma ve bölme

6.5.1 MUL ve IMUL

İki 8 bitlik sayının çarpımı 16 bite, iki 16 bitlik sayının çarpımı 32 bite kadar çıkabilir: 255 × 255 = 65.025 bir bayta değil ancak bir sözcüğe sığar. 8086 bu yüzden çarpmanın sonucunu her zaman işlenenin iki katı genişlikte bir yere yazar. Çarpmanın öteki işleneni ve sonucun yeri sabittir; komutta yalnız bir işlenen yazılır. Şekil 6.5 iki boyutu gösterir.



Şekil 6.5: Bayt çarpmada AL × işlenen AX'e, sözcük çarpmada AX × işlenen DX:AX'e yazılır.

Tanım 6.3

MUL işaretsiz, **IMUL** işaretli çarpma yapar. İşlenen bayt ise AX ← AL × işlenen; işlenen sözcük ise DX:AX ← AX × işlenen olur. İşlenen bir yazmaç ya da bellek işleneni olabilir (`MUL BL` , `IMUL WORD PTR [SI]`). 8086'da sabit işlenen yoktur: `MUL 5` geçersizdir.

Çarpımın işaretsiz ya da işaretli olması sonucun bitlerini değiştirir; toplamadaki gibi tek bir işlem iki yorumu karşılamaz. AL = C8h ve BL = 03h için `MUL BL` 200 × 3 = 600 = 0258h verir. Aynı bitlerle `IMUL BL` ise C8h'yi −56 diye okur ve −56 × 3 = −168 = FF58h verir. Düşük baytlar (58h) aynıdır, yüksek baytlar farklıdır. Bu yüzden 8086'nın iki ayrı çarpma komutu vardır.

Çarpmadan sonra yalnız CF ve OF anlamlıdır ve ikisi her zaman eşittir. Sonucun üst yarısı (AH ya da DX) gerçekten gerekiyorsa CF = OF = 1 olur. MUL için "gerekli" üst yarının sıfırdan farklı olması demektir. IMUL için üst yarının alt yarının işaret uzantısından farklı olması demektir: sonuç alt yarıya işaretli olarak sığıyorsa üst yarı yalnız işaret bitinin kopyalarıdır ve bilgi taşımaz. SF, ZF, AF ve PF çarpmadan sonra tanımsızdır.

AL, BL	Komut	Hesap	AX	CF = OF	Neden
C8h, 03h	`MUL BL`	200 × 3 = 600	0258h	1	AH = 02h ≠ 0
C8h, 03h	`IMUL BL`	−56 × 3 = −168	FF58h	1	−168 işaretli bayta sığmaz
FEh, 05h	`IMUL BL`	−2 × 5 = −10	FFF6h	0	AH = FFh, AL'nin işaret uzantısı
FEh, 05h	`MUL BL`	254 × 5 = 1270	04F6h	1	AH = 04h ≠ 0
06h, 09h	`MUL BL`	6 × 9 = 54	0036h	0	AH = 0

Tablo 6.15: Bayt çarpma örnekleri ve CF/OF.

Çarpma 8086'nın en yavaş komutlarındandır. ALU'da donanım çarpıcı yoktur; EU, mikrokodla kaydır-topla yöntemini bit bit uygular. Intel kılavuzuna göre `MUL` yazmaç işleneniyle baytta 70–77, sözcükte 118–133 saat turu sürer; `IMUL` baytta 80–98, sözcükte 128–154 tur. Bellek işleneni varsa EA süresiyle birkaç tur daha eklenir. Süre aralık olarak verilir, çünkü mikrokod döngüsü çarpanın bitlerine göre farklı yollardan geçer. Bir toplama yalnız 3 tur sürdüğüne göre, sabit bir sayıyla çarpmayı kaydırma ve toplamaya çevirmek çoğu zaman kazançlıdır.

6.5.2 DIV ve IDIV

Bölme, çarpmanın tersi olarak, işlenenin iki katı genişlikte bir **bölünen** ister. Bölen bir bayt ise bölünen AX'tir; bölen bir sözcük ise bölünen DX:AX'tir. Sonuç iki parçadır: **bölüm** ve **kalan**. Şekil 6.6 sözcük bölmeyi gösterir.



Şekil 6.6: Sözcük bölmede DX:AX bölünen, işlenen bölendir; bölüm AX'e, kalan DX'e yazılır.

Komut	İşlenen bayt ise	İşlenen sözcük ise
`DIV` (işaretsiz)	AL ← AX / işlenen, AH ← kalan	AX ← DX:AX / işlenen, DX ← kalan
`IDIV` (işaretli)	AL ← AX / işlenen, AH ← kalan	AX ← DX:AX / işlenen, DX ← kalan

Tablo 6.16: DIV ve IDIV'in örtük işlenenleri.

Bölmeden sonra bütün durum bayrakları tanımsızdır; bölme sonucu hakkında bayraklardan bilgi alınamaz. Bölme çarpmadan da yavaştır: `DIV` yazmaçla baytta 80–90, sözcükte 144–162 tur; `IDIV` baytta 101–112, sözcükte 165–184 tur sürer.

IDIV bölümü sıfıra doğru keser ve kalanı bölünenle aynı işaretli bırakır. −23 / 5 için bölüm −4,6'dır; sıfıra doğru kesilince −4 olur. Kalan, −23 = (−4) × 5 + kalan eşitliğinden −3 bulunur. Bu kurala göre her zaman bölünen = bölüm × bölen + kalan eşitliği sağlanır ve kalanın mutlak değeri bölenden küçüktür.

Tanım 6.4

Bölme hatası (*divide error*), bölen 0 olduğunda ya da bölüm hedefine (AL ya da AX) sığmadığında oluşur. İşlemci bölmeyi bitirmez; 0 numaralı kesmeyi (INT 0) üretir. Kesme vektör tablosunun ilk girişi (0000:0000) bu kesmenin hizmet yordamını gösterir. DOS bu yordamda “Divide overflow” iletisini yazar ve programı sonlandırır.

Bölümün sığıp sığmayacağı bölmeden önce kolayca sınanır. Bayt bölmede bölüm 255'i aşmamalıdır; bu ancak AH < bölen ise sağlanır. Sözcük bölmede aynı kural DX < bölen biçimini alır. AX = 1000h (4096) ve BL = 2 için AH = 10h ≥ 2'dir: bölüm 2048 AL'ye sığmaz ve bölme hatası oluşur. DX:AX = FFFFFFFFh ve BX = 1 için de bölen sıfır olmadığı hâlde DX = FFFFh ≥ 1'dir ve bölüm sığmaz. IDIV'de sınır biraz daha dardır: 8086'da bayt bölümü −127…127, sözcük bölümü −32.767…32.767 aralığında olmalıdır; −128 ve −32.768 de bölme hatası verir.

8086'nın bölme hatasında yığına ittiği dönüş adresi bir ayrıntı daha taşır. INT 0 hizmeti çağrılırken itilen IP, bölme komutundan *sonraki* komutu gösterir. 80286 ve sonraki işlemcilerde ise bölme komutunun kendisini gösterir; böylece hata giderildikten sonra komut yeniden denenebilir. Bu fark yalnız kendi INT 0 yordamını yazan programları ilgilendirir.

6.5.3 Çarpma ve bölme adım adım

Tablo 6.17 sunumdaki adım adım çarpma ve bölme örneklerini tek tabloda toplar. Her satır bağımsız bir denemedir; yazmaçlar satırın başında verilen değerlerle başlar.

Başlangıç	Komut	Sonuç	Açıklama
AL = C8h, BL = 03h	`MUL BL`	AX = 0258h, CF = OF = 1	200 × 3 = 600; AL'ye sığmadı
AL = C8h, BL = 03h	`IMUL BL`	AX = FF58h, CF = OF = 1	−56 × 3 = −168; işaretli bayta sığmaz
AL = FEh, BL = 05h	`IMUL BL`	AX = FFF6h, CF = OF = 0	−2 × 5 = −10; AH yalnız işaret uzantısı
AX = 0017h, BL = 05h	`DIV BL`	AL = 04h, AH = 03h (AX = 0304h)	23 = 4 × 5 + 3
AX = FFE9h, BL = 05h	`IDIV BL`	AL = FCh, AH = FDh (AX = FDFCh)	−23 = (−4) × 5 + (−3)
AX = 1000h, BL = 02h	`DIV BL`	INT 0	bölüm 2048 AL'ye sığmaz
DX:AX = FFFFFFFFh, BX = 0001h	`DIV BX`	INT 0	bölüm FFFFFFFFh AX'e sığmaz

Tablo 6.17: Bayt ve sözcük çarpma-bölme örnekleri.

Dördüncü satırda AX = 0017h = 23, BL = 5'tir. Bölüm 4 AL'ye, kalan 3 AH'ye yazılır ve yeni AX 0304h olur. Bu değer bir sayı olarak okunmamalıdır; AX'in iki yarısı artık iki ayrı sonuç taşır. Beşinci satırda aynı sayının negatifi bölünür. −23 = FFE9h'dir. Bölüm −4 = FCh, kalan −3 = FDh'dir; doğrulama: −4 × 5 + (−3) = −23.

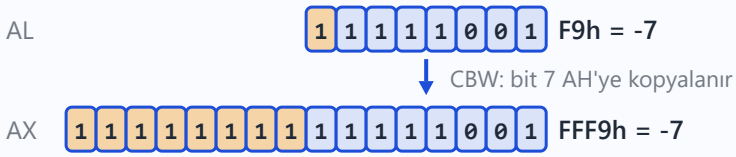
6.5.4 Bölmeye hazırlık: CBW ve CWD

Bölünen her zaman bölenin iki katı genişlikte olduğu için, bölmeden önce bölüneni genişletmek gerekir. Bölünecek sayı çoğu zaman yalnız AL'de ya da AX'te durur; üst yarının (AH ya da DX) ne olacağına programcı karar verir.

- **İşaretsiz bölmeden önce** üst yarı sıfırlanır: `MOV AH, 0` ya da `XOR DX, DX`. Sayının değeri değişmez, yalnız önüne sıfırlar eklenir.
- **İşaretli bölmeden önce** işaret genişletilir: işaret biti üst yarının bütün bitlerine kopyalanır. 8086'nın bunun için iki komutu vardır.

Tanım 6.5

İşaret genişletme (*sign extension*), işaretli bir sayıyı değerini koruyarak daha geniş bir yere taşımaktır: en anlamlı bit yeni eklenen bütün bitlere kopyalanır. **CBW** (*convert byte to word*) AL'yi AX'e genişletir: AL'nin bit 7'si 0 ise AH = 00h, 1 ise AH = FFh olur. **CWD** (*convert word to doubleword*) AX'i DX:AX'e genişletir: AX'in bit 15'i 0 ise DX = 0000h, 1 ise DX = FFFFh olur. İki komut da işlenen almaz ve hiçbir bayrağı değiştirmez.



Şekil 6.7: CBW: AL = F9h (−7) işaret biti 1 olduğu için AH = FFh olur; AX = FFF9h yine −7'dir.

F9h, işaretli bayt olarak −7'dir. Aynı −7 sözcük olarak FFF9h yazılır: 65.536 − 7 = 65.529 = FFF9h. CBW bu dönüşümü tek komutta yapar ve 2 saat turu sürer. CWD ise 5 tur sürer. İkisi de örtük işlenen kullanır ve tek baytlık komutlardır (`98` ve `99`).

```
; işaretsiz: 1000 / 7
mov ax, 1000
xor dx, dx      ; DX:AX = 1000
mov bx, 7
div bx          ; AX = 142, DX = 6
; işaretli: -1000 / 7
mov ax, -1000
cwd             ; DX = FFFFh
idiv bx         ; AX = -142, DX = -6
```

İlk parçada 1000 = 03E8h AX'e yüklenir ve DX sıfırlanır; bölünen DX:AX = 000003E8h olur. 1000 / 7 = 142, kalan 6'dır: AX = 008Eh, DX = 0006h. İkinci parçada −1000 = FC18h yüklenir. CWD, bit 15 bir olduğu için DX'e FFFFh koyar ve DX:AX = FFFFFC18h, 32 bitte −1000 olur. IDIV bölümü −142 = FF72h, kalanı −6 = FFFAh verir.

Sık yapılan hata

Sözcük bölmeden önce DX'i unutmak. DX'te önceki bir işlemden kalan sayı bölüneni büyütür. İşaretli −1000 için CWD yerine `XOR DX, DX` yazılırsa bölünen 0000FC18h = 64.536 olur. IDIV bu durumda 64.536 / 7 = 9219 (2403h) gibi anlamsız bir sonuç verir. DX'te büyük bir değer kalmışsa bölüm sığmaz ve program bölme hatasıyla durur.

6.5.5 Kod izleme: DIV, CWD ve IDIV

Aşağıdaki program aynı bölme işini üç kez yapar: önce işaretsiz, sonra işaretli, en sonda DX'i temizlemeyi unutarak.

```
MOV  AX, 2000
XOR  DX, DX      ; DX:AX = 2000
MOV  BX, 9
DIV  BX          ; işaretsiz bölme
MOV  AX, -2000
CWD              ; işaret genişlet
IDIV BX          ; işaretli bölme
MOV  AX, 2000    ; DX temizlenmedi!
DIV  BX
RET
```

Komut	Makine kodu	DX	AX	BX	DX:AX (onluk)
`MOV AX, 2000`	`B8 D0 07`	0000h	07D0h	0000h	2000
`XOR DX, DX`	`33 D2`	0000h	07D0h	0000h	2000
`MOV BX, 9`	`BB 09 00`	0000h	07D0h	0009h	2000
`DIV BX`	`F7 F3`	0002h	00DEh	0009h	bölüm 222, kalan 2
`MOV AX, -2000`	`B8 30 F8`	0002h	F830h	0009h	0002F830h = 194.608 (yanlış bölünen)
`CWD`	`99`	FFFFh	F830h	0009h	FFFFF830h = −2000
`IDIV BX`	`F7 FB`	FFFEh	FF22h	0009h	bölüm −222, kalan −2
`MOV AX, 2000`	`B8 D0 07`	FFFEh	07D0h	0009h	FFFE07D0h ≈ 4,29 milyar
`DIV BX`	`F7 F3`	bölme hatası: DX = FFFEh ≥ 9, bölüm AX'e sığmaz → INT 0			

Tablo 6.18: DIV, CWD ve IDIV izlemesi. Bölmeden sonra bayraklar tanımsızdır.

İlk bölmede 2000 / 9 = 222, kalan 2'dir: AX = 00DEh, DX = 0002h. Beşinci satır, CWD'nin neden gerektiğini gösterir. AX'e −2000 = F830h yüklenmiştir ama DX'te önceki bölmenin kalanı (2) durur. DX:AX'i 32 bitlik bir sayı olarak okuyan IDIV, bu hâliyle 194.608'i bölerdi. CWD, AX'in işaret biti 1 olduğu için DX'e FFFFh yazar ve bölüneni doğru −2000 yapar. IDIV −2000 / 9 için bölüm −222 = FF22h, kalan −2 = FFFEh verir. Kalan, bölünen gibi negatiftir.

Son iki satır klasik hatayı canlandırır. AX'e 2000 yüklenmiştir, ama DX'te IDIV'in kalanı (FFFEh) durur. İşaretsiz bölünen FFFE07D0h, yaklaşık 4,29 milyardır. Bunu 9'a bölmek 16 bite sığmayan bir bölüm verir; kural da bunu önceden söyler: DX = FFFEh ≥ 9. İşlemci INT 0 üretir ve program durur. Çözüm, bölmeden önce `XOR DX, DX` yazmaktır.

6.5.6 İşaretli taşma: JO ve CBW

İki işaretli baytın toplamı bayta sığmayabilir. Bu durumda doğru sonuç, iki sayıyı önce sözcüğe genişletip 16 bitte toplamakla bulunur. Aşağıdaki program önce bayt toplamasını dener; taşma olursa sözcük toplamasına geçer. Programda iki atlama komutu vardır. `JO etiket` (*jump if overflow*) OF = 1 ise belirtilen etikete atlar, değilse bir sonraki komuta geçer. `JMP etiket` koşulsuz atlar. Tersi olan `JNO`, OF = 0 ise atlar.

```
mov al, -92      ; A4h
mov bl, -92
add al, bl       ; 8 bit toplama
jo  genislet     ; OF = 1 ise
jmp bitti
genislet:
mov al, -92      ; birinci sayı
cbw              ; AL -> AX
mov dx, ax
mov al, bl       ; ikinci sayı
cbw
add ax, dx       ; 16 bit toplama
bitti:
ret
```

Komut	AX	DX	CF	OF	SF	Açıklama
`mov al, -92` · `mov bl, -92`	00A4h	0000h	0	0	0	−92 = 256 − 92 = 164 = A4h
`add al, bl`	0048h	0000h	1	**1**	0	A4h + A4h = 148h; AL = 48h = +72
`jo genislet`	0048h	0000h	1	1	0	OF = 1: atlanır
`mov al, -92` · `cbw`	FFA4h	0000h	1	1	0	AL'nin MSB'si 1 → AH = FFh
`mov dx, ax` · `mov al, bl` · `cbw`	FFA4h	FFA4h	1	1	0	iki sayı da sözcükte −92
`add ax, dx`	**FF48h**	FFA4h	1	**0**	1	FFA4h + FFA4h = 1FF48h; AX = −184

Tablo 6.19: JO ve CBW ile işaretli taşmanın giderilmesi.

Bayt toplamasında iki negatif sayının toplamı pozitif çıkmıştır: AL = 48h = +72. Doğru sonuç −184'tür ve −128...127 aralığında değildir. OF = 1 olduğu için JO genislet etiketine atlar. CF = 1 de olmuştur, ama işaretli yorumda CF'nin bir anlamı yoktur. Genişletme kısmında birinci sayı yeniden AL'ye yüklenir ve CBW ile FFA4h'ye genişletilir. DX'e saklanır. İkinci sayı BL'den alınır ve o da genişletilir. 16 bitlik toplam FFA4h + FFA4h = 1FF48h'dir; dışarı çıkan 1 CF'ye düşer, AX'te FF48h = −184 kalır. Sonuç sözcüğe sığmıştır, OF = 0'dır.

6.5.7 CBW ile işaretli bayt dizisinin toplamı

Aynı fikir bir dizinin bütün elemanlarına uygulanabilir. Aşağıdaki program dört işaretli baytı 16 bitlik bir toplamda biriktirir. Program bir döngü kullanır ve döngüyü `LOOP` komutu kurar.

Tanım 6.6

LOOP etiket komutu iki iş yapar: CX'i 1 azaltır, ardından CX sıfırdan farklıysa etikete atlar; CX sıfır olduysa bir sonraki komuta geçer. Döngüden önce CX'e tur sayısı yüklenir. CX = 4 ile başlayan bir döngü gövdesi dört kez çalışır. LOOP hiçbir bayrağı değiştirmez; CX'in azaltılması da bayraklara yansımaz.

```
org 100h
jmp basla
dizi db 100, -30, 90, -7
toplam dw 0
basla:
lea si, dizi
mov cx, 4
xor bx, bx      ; BX = 16 bit toplam
dongu:
mov al, [si]
cbw             ; AL -> AX (işaretle)
add bx, ax
inc si
loop dongu
mov toplam, bx  ; 153 = 0099h
ret
```

Programın başındaki `JMP basla`, veri tanımlarının üzerinden atlar. .COM programında kod ve veri aynı kesimdedir ve işlemci 0100h'den başlar; atlama olmasaydı dizinin baytları komut sanılırdı. `LEA SI, dizi` dizinin bağıl konumunu (0102h) SI'ya yükler. emu8086 bu satırı `MOV SI, 0102h` ile aynı üç baytlık koda (`BE 02 01`) çevirir. BX toplamı tutar ve sıfırlanır.

Tur	AL	CBW sonrası AX	BX (toplam)	CX (tur başında)	Açıklama
1	64h	0064h	0064h = 100	4	+100, MSB = 0 → AH = 00h
2	E2h	FFE2h	0046h = 70	3	−30, MSB = 1 → AH = FFh
3	5Ah	005Ah	00A0h = 160	2	+90
4	F9h	FFF9h	0099h = 153	1	−7; LOOP sonrası CX = 0

Tablo 6.20: CBW ile işaretli bayt toplamı. Her tur için `cbw` ve `add bx, ax` sonrası durum.

İkinci turda BX = 0064h'ye FFE2h eklenir: 0064h + FFE2h = 1 0046h. Dışarı çıkan elde atılır ve BX = 0046h = 70 kalır; 100 + (−30) = 70 doğrudur. Bu, ikiye tümleyenin bilinen özelliğidir: negatif bir sayıyı eklemek, elde atıldığında çıkarmakla aynı sonucu verir. Üçüncü turda toplam 160 olur. Toplama bayt olarak yapılsaydı 70 + 90 = 160 > 127 olacak, AL = A0h (−96) ve OF = 1 çıkacaktı. Sözcükte ise 160 rahatça sığar. Döngüden sonra toplam = 0099h = 153 belleğe yazılır.

Sık yapılan hata

İşaretli baytları genişletirken `MOV AH, 0` kullanmak. Bu, sayıyı işaretsiz genişletir: −30 = E2h, 00E2h = 226 olarak eklenir; −7 de 249 sayılır ve toplam 153 yerine 665 çıkar. İşaretsiz veride MOV AH, 0, işaretli veride CBW kullanılır.

6.5.8 CWD ve ADC ile 32 bitlik toplam

Elemanlar sözcük olduğunda toplam 16 bite de sığmayabilir. Çözüm, her elemanı CWD ile 32 bite genişletmek ve 32 bitlik bir toplamda ADD-ADC çiftiyle biriktirmektir.

```
org 100h
jmp basla
dizi dw 30000, 25000, -4000
toplam dw 0, 0       ; 32 bit toplam
basla:
lea si, dizi
mov cx, 3
dongu:
mov ax, [si]
cwd                  ; AX -> DX:AX
add [toplam], ax     ; düşük sözcük
adc [toplam+2], dx   ; yüksek + CF
add si, 2
loop dongu
ret
```

Toplam iki sözcüktür: düşük sözcük `toplam`'da, yüksek sözcük `toplam+2`'de durur. Sözcük dizisinde her eleman iki bayt tuttuğu için SI her turda 2 artar.

Tur	AX	CWD: DX	ADD sonrası toplam, CF	ADC sonrası toplam+2	32 bitlik toplam
1	7530h (30.000)	0000h	7530h, CF = 0	0000h	00007530h = 30.000
2	61A8h (25.000)	0000h	D6D8h, CF = 0	0000h	0000D6D8h = 55.000
3	F060h (−4000)	FFFFh	C738h, CF = 1	0000h	0000C738h = 51.000

Tablo 6.21: CWD ve ADC ile 32 bitlik toplamın izlemesi (her turda ADD ve ADC sonrası).

İkinci turda 7530h + 61A8h = D6D8h bulunur. 16 bit işaretli yorumda bu sayı negatiftir ve OF = 1 olur; ama toplam 32 bit tutulduğu için sorun yoktur: yüksek sözcük 0000h'dir ve 55.000 doğrudur. Üçüncü turda −4000 = F060h'nin işaret biti 1'dir; CWD DX'e FFFFh koyar ve DX:AX = FFFFF060h, 32 bitte −4000 olur. Düşük sözcükte D6D8h + F060h = 1C738h; toplam C738h olur ve CF = 1'dir. Yüksek sözcükte 0000h + FFFFh + 1 = 10000h; toplam+2 = 0000h kalır. Sonuç 0000C738h = 51.000 = 30.000 + 25.000 − 4000'dir.

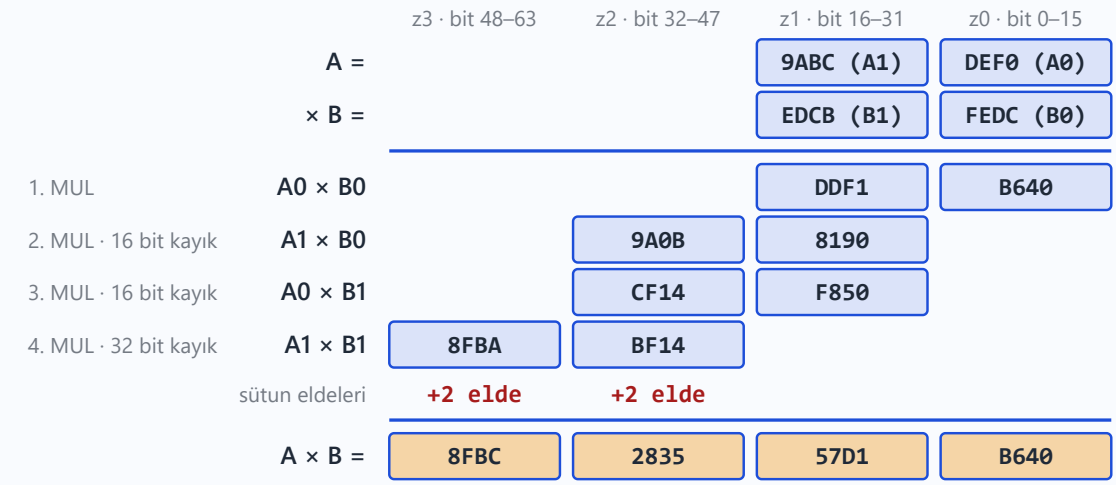
Programdaki `add si, 2` CF'yi değiştirir; ama bu bir sorun değildir, çünkü sonraki turun eldesi yeniden ADD ile başlar. Elde yalnız ADD ile hemen ardındaki ADC arasında korunmalıdır. Bu iki komut bellek işleneni kullanır: her biri sözcüğü okur, toplar ve geri yazar. 8086'da her biri 16 + EA saat turu sürer ve iki yol döngüsü (bir okuma, bir yazma) gerektirir.

6.5.9 32 × 32 bitlik çarpma

8086 en çok 16 × 16 bit çarpar. 32 bitlik iki sayının çarpımı ise kâğıt üzerindeki çarpmanın yöntemiyle, 16 bitlik "basamaklarla" yapılır. A = A1:A0 ve B = B1:B0 olarak yarılara ayrılırsa

> $A \times B = A1 \cdot B1 \cdot 2^{32} + (A1 \cdot B0 + A0 \cdot B1) \cdot 2^{16} + A0 \cdot B0$

bulunur. Dört kısmi çarpımın her biri bir MUL ile hesaplanır ve 32 bitlik sonucu DX:AX'e düşer. Kısmi çarpımlar, sonucun dört sözcüğüne (z3:z2:z1:z0, toplam 64 bit) konumlarına göre eklenir. 2^{16} ile çarpılmış bir çarpım bir sözcük sola kaymıştır ve z2:z1'e eklenir; 2^{32} ile çarpılmış olan iki sözcük sola kaymıştır ve z3:z2'ye eklenir.



Şekil 6.8: A = 9ABCDEF0h ile B = EDCBFEDCh'nin çarpımı. Dört kısmi çarpım sütunlarına göre toplanır; sütun eldeleri bir üst sözcüğe geçer.

Örnekte A = 9ABCDEF0h, B = EDCBFEDCh'dir. Kısmi çarpımlar A0 × B0 = DDF1B640h, A1 × B0 = 9A0B8190h, A0 × B1 = CF14F850h ve A1 × B1 = 8FBABF14h'dir. z0 sütununda yalnız B640h vardır. z1 sütununun toplamı DDF1h + 8190h + F850h = 257D1h'dir: 57D1h yazılır, 2 elde z2'ye geçer. z2 sütunu 9A0Bh + CF14h + BF14h + 2 = 22835h'dir: 2835h yazılır, 2 elde z3'e geçer. z3 = 8FBAh + 2 = 8FBCh. Sonuç 8FBC 2835 57D1 B640h'dir. Onluk denetim: 2.596.069.104 × 3.989.569.244 = 10.357.197.452.617.037.376.

Sütun eldesi 2 olabilir, ama bir ADD en çok 1 elde verir. Kod bu yüzden toplamaları tek tek yapar ve her ADD'nin eldesini hemen bir ADC ile bir üst sözcüğe taşır.

```
org 100h
mov ax, [x]       ; A0
mul [y]           ; A0*B0
mov [z], ax
mov [z+2], dx
mov ax, [x+2]     ; A1
mul [y]           ; A1*B0
add [z+2], ax
adc [z+4], dx
mov ax, [x]       ; A0
mul [y+2]         ; A0*B1
add [z+2], ax
adc [z+4], dx
adc [z+6], 0
mov ax, [x+2]     ; A1
mul [y+2]         ; A1*B1
add [z+4], ax
adc [z+6], dx
ret
x dw 0DEF0h, 9ABCh  ; A = 9ABCDEF0h
y dw 0FEDCh, 0EDCBh ; B = EDCBFEDCh
z dw 4 dup(0)       ; 64 bit sonuç
```

Komut(lar)	DX:AX	CF	z3	z2	z1	z0
`mov ax,[x]` · `mul [y]`	DDF1:B640h	1	0000h	0000h	0000h	0000h
`mov [z],ax` · `mov [z+2],dx`	DDF1:B640h	1	0000h	0000h	DDF1h	B640h
`mov ax,[x+2]` · `mul [y]`	9A0B:8190h	1	0000h	0000h	DDF1h	B640h
`add [z+2],ax`	9A0B:8190h	1	0000h	0000h	5F81h	B640h
`adc [z+4],dx`	9A0B:8190h	0	0000h	9A0Ch	5F81h	B640h
`mov ax,[x]` · `mul [y+2]`	CF14:F850h	1	0000h	9A0Ch	5F81h	B640h
`add [z+2],ax`	CF14:F850h	1	0000h	9A0Ch	57D1h	B640h
`adc [z+4],dx`	CF14:F850h	1	0000h	6921h	57D1h	B640h
`adc [z+6],0`	CF14:F850h	0	0001h	6921h	57D1h	B640h
`mov ax,[x+2]` · `mul [y+2]`	8FBA:BF14h	1	0001h	6921h	57D1h	B640h
`add [z+4],ax`	8FBA:BF14h	1	0001h	2835h	57D1h	B640h
`adc [z+6],dx`	8FBA:BF14h	0	**8FBCh**	**2835h**	**57D1h**	**B640h**

Tablo 6.22: 32 × 32 bitlik çarpmanın izlemesi. z3…z0 sonucun dört sözcüğüdür (başlangıçta 0).

İlk MUL'den sonra DX ≠ 0 olduğu için CF = 1'dir; bu CF yalnız “sonuç 16 bite sığmadı” der ve sonraki toplamalarla ilgisi yoktur. İkinci kısmi çarpımda z1 + AX = DDF1h + 8190h = 1 5F81h; z1 = 5F81h olur ve CF = 1'dir. ADC, z2 = 0 + 9A0Bh + 1 = 9A0Ch yapar. z2 önceden sıfır olduğu için burada yeni elde oluşmaz.

Üçüncü kısmi çarpım daha ilginçtir. z1 = 5F81h + F850h = 1 57D1h; elde z2'ye gider. z2 = 9A0Ch + CF14h + 1 = 1 6921h; bu kez elde z3'e taşmalıdır. `adc [z+6], 0` komutu bu işi yapar: z3'e hiçbir sayı eklemez, yalnız eldeyi ekler ve z3 = 0001h olur. Bu satır unutulursa sonuç tam 2^{48} eksik çıkar. Dördüncü kısmi çarpım z3:z2'ye eklenir: z2 = 6921h + BF14h = 1 2835h, z3 = 0001h + 8FBAh + 1 = 8FBCh. Bellekte z sırasıyla `40 B6 D1 57 35 28 BC 8F` baytlarını taşır; little-endian düzende bu 8FBC283557D1B640h'dir.

Dört MUL'ün her biri 124–139 tur artı EA sürer; programın tamamı 8086'da 600 turu aşar. Aynı iş 80386'da tek bir 32 bitlik MUL ile yapılır. 16 bitlik bir işlemcide çok sözcüklü aritmetiğin bedeli budur.

6.6 Tek işlenenli komutlar

6.6.1 INC, DEC, NOT ve NEG

Üçüncü gruptaki dört komut tek bir işlenen alır. İşlenen bir yazmaç (AX, BX, …, SP ya da 8 bitlik yarılar) veya bir bellek işleneni olabilir; sabit olamaz. Bellek işleneninde boyut belirtilmelidir: `INC BYTE PTR [BX]`. Komut işleneni okur, değiştirir ve aynı yere geri yazar.

- **INC** işlenene 1 ekler, **DEC** 1 çıkarır. ZF, SF, OF, PF ve AF'yi sonuca göre ayarlar, ama **CF'ye dokunmaz**.
- **NOT** her biti tersler. Sonuç işlenenin **birler tümleyenidir**. Hiçbir bayrağı değiştirmez.
- **NEG** işlenenin işaretini değiştirir: 0 − işlenen hesaplar. Sonuç **ikiye tümleyendir**: bitler terslenir ve 1 eklenir. Altı durum bayrağının hepsini ayarlar; CF, işlenen sıfırdan farklıysa 1, sıfırsa 0 olur.

Komut	CF	ZF	SF	OF	PF	AF
`INC, DEC`	—	✓	✓	✓	✓	✓
`NOT`	—	—	—	—	—	—
`NEG`	✓ (işlenen ≠ 0 ise 1)	✓	✓	✓	✓	✓

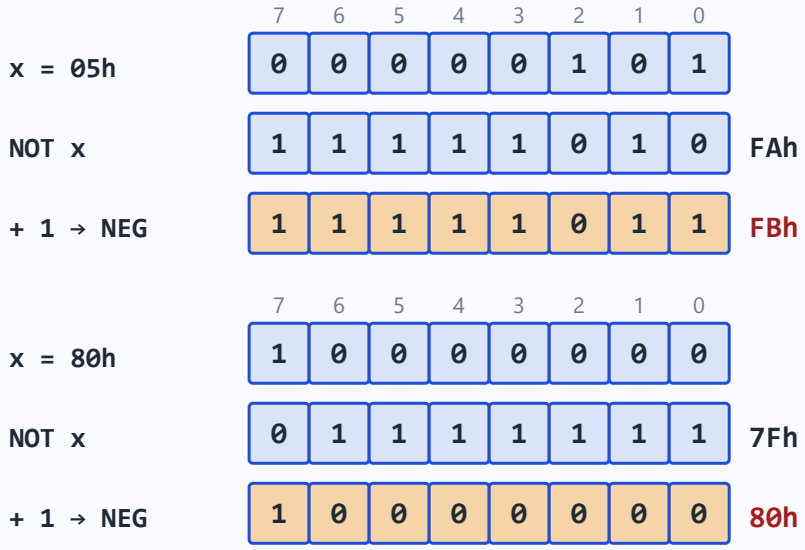
Tablo 6.23: Tek işlenenli komutların bayrak etkisi. ✓ sonuca göre ayarlanır, — değişmez.

INC ve DEC'in CF'yi koruması bir tasarım kararıdır. Çok baytlı ya da çok sözcüklü bir toplamada döngü her turda bir ADC yapar ve eldeyi bir sonraki tura taşır. Aynı döngü işaretçiyi INC ile ilerletir ve sayacı DEC ile azaltır. Bu iki komut CF'yi silseydi elde kaybolurdu. 8086 tasarımcıları bu yüzden INC ve DEC'i CF'den bağımsız bıraktı.

INC ve DEC aynı zamanda çok kısa komutlardır. 16 bitlik bir yazmacı artıran `INC SI` tek bayttır (`46`) ve 2 saat turu sürer; `ADD SI, 1` üç bayt ve 4 turdur. 8 bitlik yazmaçta INC iki bayt (`FE C0` gibi) ve 3 tur, bellekte 15 + EA turdur. NEG ve NOT yazmaçta 3, bellekte 16 + EA tur sürer.

6.6.2 NEG: ikiye tümleyen adım adım

İkiye tümleyen, ikili sayılarda negatif sayıyı göstermenin standart yoludur. Bir sayının negatifi, bitleri terslenip 1 eklenerek bulunur. NEG bu iki adımı tek komutta yapar. Şekil 6.9 5'in ve −128'in negatifini adım adım gösterir.



Şekil 6.9: NEG'in iki adımı: önce NOT (birler tümleyeni), sonra +1. Solda 05h → FAh → FBh (−5); sağda 80h → 7Fh → 80h: −128'in negatifi bayta sığmaz.

x	NOT x	NEG x	CF	OF	Açıklama
05h (5)	FAh (−6)	FBh (−5)	1	0	pozitifin negatifi
FEh (−2)	01h	02h (2)	1	0	negatifin negatifi pozitiftir
80h (−128)	7Fh	80h	1	**1**	+128 bayta sığmaz
00h	FFh	00h	**0**	0	ZF = 1; CF = 0 olan tek durum

Tablo 6.24: Dört bayt değerinde NEG.

Birler tümleyeninin değeri, ikiye tümleyenden her zaman 1 eksiktir: NOT 5 = −6, NEG 5 = −5. −128'in durumu özeldir. İşaretli bayt aralığı −128…127'dir ve +128 bu aralıkta yoktur. NEG 80h sonucu yine 80h verir ve OF = 1 ile taşmayı bildirir. NEG'in CF kuralı da 0 − x çıkarmasından gelir: x sıfır değilse 0'dan pozitif bir sayı çıkarmak ödünç ister ve CF = 1 olur; x = 0 ise ödünç gerekmez.

6.6.3 Kod izleme: INC, DEC, NOT, NEG

```
mov al, 0FFh
inc al          ; FFh + 1
dec al          ; 00h - 1
mov bl, 7Fh     ; +127
inc bl          ; +128 ?
not bl
neg bl
```

Komut	Makine kodu	AL	BL	CF	ZF	SF	OF	PF	AF
`mov al, 0FFh`	`B0 FF`	FFh	00h	0	0	0	0	0	0
`inc al`	`FE C0`	00h	00h	**0**	1	0	0	1	1
`dec al`	`FE C8`	FFh	00h	**0**	0	1	0	1	1
`mov bl, 7Fh`	`B3 7F`	FFh	7Fh	0	0	1	0	1	1
`inc bl`	`FE C3`	FFh	80h	0	0	1	**1**	0	1
`not bl`	`F6 D3`	FFh	7Fh	0	0	1	1	0	1
`neg bl`	`F6 DB`	FFh	81h	**1**	0	1	0	1	1

Tablo 6.25: INC, DEC, NOT ve NEG izlemesi (başlangıçta bütün durum bayrakları 0).

`INC AL` FFh'yi 00h yapar ve ZF = 1 olur. ADD AL, 1 olsaydı CF = 1 olacaktı; INC CF'yi değiştirmez ve CF = 0 kalır. `DEC AL` 00h − 1 = FFh hesaplar. Ödünç gerekmiştir, ama CF yine değişmez; yalnız SF = 1 olur. Bu iki satır, INC ve DEC'in sonucu sarmalayıp sarmalamadığını bayraklardan anlamanın tek yolunun ZF ve SF olduğunu gösterir.

BL'ye en büyük pozitif bayt olan 7Fh = +127 yüklenir. `INC BL` ile BL 80h olur: işaretsiz yorumda 128 doğrudur, işaretli yorumda −128'dir. İşaretli taşma olmuştur, OF = 1. `NOT BL` 80h'yi 7Fh'ye çevirir ama bayraklara dokunmaz: SF hâlâ 1 görünür, oysa BL'nin işaret biti artık 0'dır. Bu, NOT'tan sonra bayraklara bakmanın yanıltıcı olabileceğini gösterir. `NEG BL` +127'nin negatifini alır: −127 = 81h. İşlenen sıfır olmadığı için CF = 1'dir; sonuç aralıktadır, OF = 0.

6.7 Kaydırma ve döndürme

6.7.1 Kaydırma: SHL/SAL, SHR, SAR

Kaydırma komutları bir sayının bütün bitlerini bir yöne kaydırır. Bir uçtan çıkan bit CF'ye düşer, öteki uçtan yeni bir bit girer. Giren bitin ne olacağını komut belirler. Sayısal tasarımdaki kaydırmalı yazmacın (*shift register*) yazılım karşılığıdır; tek fark, 8086'nın kaydırmayı bir saat darbesiyle değil bir komutla yapmasıdır. Şekil 6.10 üç kaydırmanın bit akışını gösterir.



Şekil 6.10: SHL/SAL (sağdan 0 girer, b7 CF'ye), SHR (soldan 0 girer, b0 CF'ye) ve SAR (işaret biti yerinde kalıp kendini kopyalar, b0 CF'ye).

Tanım 6.7

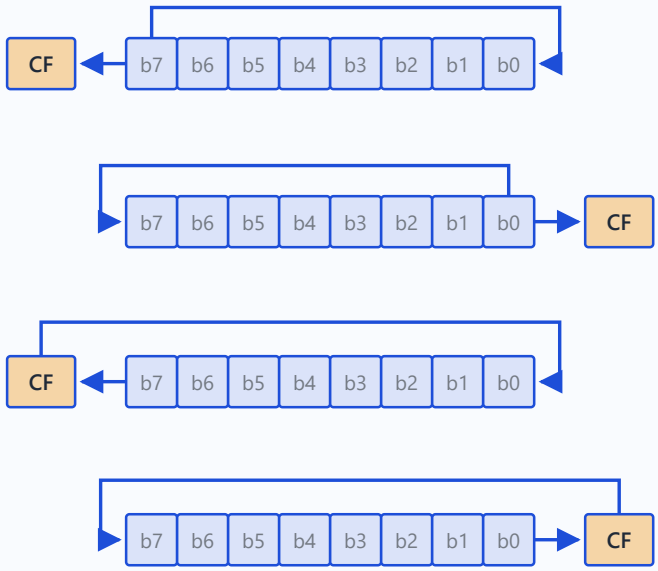
SHL (*shift logical left*) ve aynı komutun öteki adı **SAL** bitleri sola kaydırır; sağdan 0 girer, en anlamlı bit CF'ye çıkar. Her kaydırma işaretsiz sayıyı 2 ile çarpar. **SHR** (*shift logical right*) **mantıksal kaydırmadır**: soldan 0 girer, en az anlamlı bit CF'ye çıkar; işaretsiz sayıyı 2'ye böler. **SAR** (*shift arithmetic right*) **aritmetik kaydırmadır**: en anlamlı bit yerinde kalır ve sağa da kopyalanır; işaretli sayıyı 2'ye böler.

96h = 1001 0110b örneğiyle üç kaydırma şöyle çalışır. `SHL AL, 1`: bitler sola kayar, b7 (1) CF'ye çıkar, sağdan 0 girer: 0010 1100b = 2Ch, CF = 1. İşaretsiz 150 × 2 = 300 bayta sığmamıştır; CF bunu gösterir. `SHR AL, 1`: b0 (0) CF'ye çıkar, soldan 0 girer: 0100 1011b = 4Bh = 75 = 150 / 2. `SAR AL, 1`: işaret biti 1 kopyalanır: 1100 1011b = CBh = −53. İşaretli yorumda 96h = −106'dır ve −106 / 2 = −53 doğrudur.

SAR'ın bölmesi IDIV'inkinden farklı yuvarlar. SAR her zaman eksi sonsuza doğru yuvarlar; IDIV sıfıra doğru keser. Pozitif sayılarda ikisi aynı sonucu verir, negatif tek sayılarda ayrılır. −7 = F9h için `SAR AL, 1` FCh = −4 verir (−3,5 aşağı yuvarlanmıştır). −7 / 2 için IDIV ise −3 verir. Bu fark, bölmeyi kaydırmaya çevirirken akılda tutulmalıdır.

6.7.2 Döndürme: ROL, ROR, RCL, RCR

Döndürme komutları kaydırmaya benzer, ama bir uçtan çıkan bit kaybolmaz; öteki uçtan geri girer. Bitler bir halka üzerinde döner. Şekil 6.11 dört döndürmenin bit akışını gösterir.



Şekil 6.11: ROL ve ROR 8 bitlik halkada döndürür ve çıkan bitin bir kopyasını CF'ye yazar. RCL ve RCR, CF'yi de halkaya katar: bayt için 9, sözcük için 17 bitlik halka.

- **ROL** (sola döndür): b7 hem b0'a hem CF'ye gider.
- **ROR** (sağa döndür): b0 hem b7'ye hem CF'ye gider.
- **RCL** (CF üzerinden sola döndür): b7 CF'ye, eski CF b0'a gider.
- **RCR** (CF üzerinden sağa döndür): b0 CF'ye, eski CF b7'ye gider.

Komut	CF = 0 ile	CF = 1 ile	Sonuç CF	OF
`SHL AL, 1`	2Ch (0010 1100)	2Ch	1	1
`SHR AL, 1`	4Bh (0100 1011)	4Bh	0	1
`SAR AL, 1`	CBh (1100 1011)	CBh	0	0
`ROL AL, 1`	2Dh (0010 1101)	2Dh	1	1
`ROR AL, 1`	4Bh (0100 1011)	4Bh	0	1
`RCL AL, 1`	2Ch (0010 1100)	2Dh (0010 1101)	1	1
`RCR AL, 1`	4Bh (0100 1011)	CBh (1100 1011)	0	1 / 0

Tablo 6.26: AL = 96h (1001 0110b) için yedi komutun tek adımlık sonucu. RCL ve RCR'de başlangıç CF'si sonucu değiştirir.

Döndürmede hiçbir bit kaybolmaz. `ROL AX, CL` ile CL = 8 kez döndürülen 1234h, 3412h olur: yüksek ve düşük baytlar yer değiştirir. Döndürme komutları yalnız CF'yi ve (sayı 1 iken) OF'yi değiştirir; SF, ZF, PF ve AF'ye dokunmaz. Kaydırma komutları ise SF, ZF ve PF'yi sonuca göre ayarlar; AF tanımsızdır.

RCL ve RCR çok sözcüklü kaydırmanın anahtarıdır. 32 bitlik DX:AX'i 2 ile çarpmak için önce düşük sözcük kaydırılır, sonra yüksek sözcük CF üzerinden döndürülür: `SHL AX, 1` AX'in b15'ini CF'ye çıkarır, `RCL DX, 1` bu biti DX'in b0'ına sokar. Örneğin DX:AX = 0000FFFFh iken bu iki komuttan sonra

DX:AX = 0001FFFEh olur. Bölme için ters sıra izlenir: `SHR DX, 1` ve ardından `RCR AX, 1`.

6.7.3 Kaydırma sayısı, OF ve süre

8086'da kaydırma sayısı yalnız iki biçimde verilir: sabit 1 ya da CL yazmacı. `SHL AX, 1` ve `SHR AL, CL` geçerlidir. `SHL AL, 4` gibi 1'den büyük bir sabit, 80186 ve sonraki işlemcilerde eklenmiştir; 8086 kodunda dört kez `SHL AL, 1` ya da `MOV CL, 4` ve `SHL AL, CL` yazılır. emu8086 bu biçimi kabul eder, ama ürettiği kod gerçek bir 8086'da çalışmaz. 8086, CL'deki sayıyı maskelemez; CL = 255 ise kaydırma 255 kez yapılır. 80286 ve sonrası sayıyı 5 bite (0–31) indirir.

OF kaydırma ve döndürmede yalnız sayı 1 iken tanımlıdır. Sola bir adımlık kaydırmada OF, işaret bitinin değişip değişmediğini söyler: OF = (yeni b7) XOR CF. 40h = 0100 0000b sola kaydırılınca 80h olur; işaret biti 0'dan 1'e döner ve OF = 1'dir. C0h sola kaydırılınca da 80h olur; ama işaret biti 1'den 1'e kalır ve OF = 0'dır: −64 × 2 = −128 doğrudur. Sayı CL ile verildiğinde ve 1'den büyükse OF tanımsızdır.

Süre bakımından kaydırma çarpmadan çok daha hızlıdır. Intel kılavuzuna göre yazmaçta `SHL reg, 1` 2 saat turu, `SHL reg, CL` 8 + 4n tur sürer (n kaydırma sayısı). Bellekte bir adımlık kaydırma 15 + EA, CL ile 20 + EA + 4n turdur. Bir bayt MUL'ün 70 turu aştığı düşünülürse, 2'nin kuvvetleriyle çarpıp bölmenin neden kaydırmaya bırakıldığı anlaşılır.

6.7.4 Kod izleme: kaydırarak çarpma

Donanım çarpıcısı olmayan işlemciler çarpmayı kâğıt üzerindeki yöntemle yapar. Çarpanın bitleri sağdan sola tek tek incelenir. Bit 1 ise o anki çarpılan sonuca eklenir; her adımda çarpılan bir sola kaydırılır, yani 2 ile çarpılır. Aşağıdaki program 13 × 5'i bu yöntemle hesaplar.

```
org 100h
mov bl, carpilan  ; 13
mov dl, carpan    ; 5 = 0101b
mov cx, 4         ; 4 bit
kontrol:
shr dl, 1         ; en sağ bit → CF
jnc atla
add sonuc, bl     ; bit 1 ise ekle
atla:
shl bl, 1         ; çarpılan × 2
loop kontrol
ret
carpilan db 13
carpan   db 5
sonuc    db 0
```

Programda iki denetim komutu vardır. `JNC atla` (*jump if no carry*) CF = 0 ise atla etiketine geçer ve toplamayı atlar. `LOOP kontrol` CX'i 1 azaltır ve CX sıfır değilse kontrol etiketine döner. CX = 4 ile döngü dört kez çalışır; çarpan 4 bitlik olduğu için bu yeterlidir.

Tur	DL (SHR sonrası)	CF	Toplama	sonuc	BL (SHL sonrası)	CX (LOOP sonrası)
1	02h	1	0 + 0Dh	0Dh (13)	1Ah (26)	3
2	01h	0	atlanır	0Dh	34h (52)	2
3	00h	1	0Dh + 34h	41h (65)	68h (104)	1
4	00h	0	atlanır	41h	D0h	0

Tablo 6.27: 13 × 5'in kaydırarak hesaplanması. Her satır bir tur: SHR'nin çıkardığı bit, toplama ve SHL sonrası durum.

Çarpan 5 = 0101b'dir. Birinci turda SHR en sağdaki biti (1) CF'ye çıkarır ve DL = 02h olur. CF = 1 olduğundan JNC atlamaz; sonuca çarpılanın kendisi, 13 × 1 eklenir. İkinci turda çıkan bit 0'dır ve toplama atlanır. Üçüncü turda çıkan bit yine 1'dir. Bu sırada BL iki kez kaydırılmış ve 13 × 4 = 52 olmuştur; sonuç 13 + 52 = 65 = 41h olur. Dördüncü turda bit 0'dır. Döngü biter ve sonuc = 41h = 65 = 13 × 5 kalır. Son turda BL = D0h olmuştur; bu değer (208) artık kullanılmaz.

Program çarpanın 4 bitlik olduğunu ve sonucun bayta sığacağını varsayar. Daha büyük çarpanlar için döngü 8 tur döner, çarpılan ve sonuç da sözcükte tutulur. Mikroişlemcilerin ilk kuşağında (Intel 8080, Zilog Z80) çarpma komutu yoktu ve her çarpma bu tür bir altyordamla yapılırdı. 8086'nın MUL komutu da içeride aynı yöntemi mikrokodla uygular.

6.7.5 Kaydırmayla hızlı çarpma ve bölme

Sabit bir sayıyla çarpma, o sayının ikili açılımına göre kaydırma ve toplamaya dönüştürülebilir. 3 = 2 + 1 olduğundan AX × 3 = AX × 2 + AX'tir:

```
; AX ← AX × 3 = AX×2 + AX
mov bx, ax      ; sakla
shl ax, 1       ; AX × 2
add ax, bx      ; ×2 + ×1
; işaretsiz ÷ 8 ve kalan
mov dx, ax
and dx, 0007h   ; kalan = alt 3 bit
mov cl, 3
shr ax, cl      ; bölüm
; işaretli ÷ 4
mov cl, 2
sar ax, cl      ; -7 → -2
```

AX = 25 için ilk parça BX = 25, AX = 50 ve son olarak AX = 75 verir. Toplam süre 2 + 2 + 3 = 7 tur kadardır; MUL ise 118 turu aşar. İkinci parça 2'nin kuvvetine bölmeyi gösterir. Sağa n bit kaydırmak 2^n'e böler; kaydırma sırasında dışarı çıkan alt n bit kalandır. Kalan, kaydırmadan önce AND ile alınır: alt üç biti 1 olan maske 0007h'dir. AX = 100 için kalan DX = 4, bölüm AX = 12 olur; 12 × 8 + 4 = 100.

Üçüncü parça işaretli sayıyı SAR ile böler. AX = −7 için SAR iki adımda −2 verir; IDIV ile −7 / 4 ise −1 olurdu. SAR eksi sonsuza, IDIV sıfıra doğru yuvarlar. Derleyiciler işaretli bölmeyi kaydırmaya çevirirken bu farkı gidermek için negatif sayılara önce $(2^n - 1)$ ekler.

6.7.6 Kod izleme: 37 × 10 ve 370 ÷ 16

10 = 8 + 2 olduğu için AX × 10 = AX × 8 + AX × 2'dir. Aşağıdaki program önce 37'yi bu yolla 10 ile çarpar, ardından sonucu 16'ya böler ve kalanı ayrıca bulur.

```
org 100h
mov ax, 37
shl ax, 1      ; AX × 2
mov bx, ax     ; sakla
shl ax, 1      ; AX × 4
shl ax, 1      ; AX × 8
add ax, bx     ; ×8 + ×2
mov bx, ax
and bx, 000Fh  ; kalan: alt 4 bit
mov cl, 4
shr ax, cl     ; bölüm: ÷ 16
ret
```

Komut	Makine kodu	AX	AX (ikili)	BX	CL
`mov ax, 37`	`B8 25 00`	0025h = 37	`0000 0000 0010 0101`	0000h	00h
`shl ax, 1`	`D1 E0`	004Ah = 74	`0000 0000 0100 1010`	0000h	00h
`mov bx, ax`	`8B D8`	004Ah	`0000 0000 0100 1010`	004Ah = 74	00h
`shl ax, 1`	`D1 E0`	0094h = 148	`0000 0000 1001 0100`	004Ah	00h
`shl ax, 1`	`D1 E0`	0128h = 296	`0000 0001 0010 1000`	004Ah	00h
`add ax, bx`	`03 C3`	0172h = 370	`0000 0001 0111 0010`	004Ah	00h
`mov bx, ax`	`8B D8`	0172h	`0000 0001 0111 0010`	0172h	00h
`and bx, 000Fh`	`83 E3 0F`	0172h	`0000 0001 0111 0010`	0002h	00h
`mov cl, 4`	`B1 04`	0172h	`0000 0001 0111 0010`	0002h	04h
`shr ax, cl`	`D3 E8`	0017h = 23	`0000 0000 0001 0111`	0002h	04h

Tablo 6.28: 37 × 10 ve 370 ÷ 16 izlemesi.

Her SHL, bitleri bir sola kaydırır ve sayıyı ikiye katlar: 37, 74, 148, 296. Arada saklanan ×2 değeri (74) ×8 değerine eklenir ve 370 = 37 × 10 bulunur. Bölme için önce kalan ayrılır: 370 = 0172h'nin alt dört biti 0010b = 2'dir ve `AND BX, 000Fh` yalnız onları bırakır. Sonra `SHR AX, CL` dört bit sağa kaydırır: 0172h → 0017h = 23. Denetim: 23 × 16 + 2 = 370.

Makine kodlarında iki ayrıntı göze çarpar. `shl ax, 1` (D1 E0) ile `shr ax, cl` (D3 E8) farklı işlem kodları kullanır: D1 "bir kez", D3 "CL kez" anlamına gelir. `and bx, 000Fh` ise 83 biçimiyle üç bayt tutar: sabit 0Fh tek bayta sığdığı için işlemci onu çalışırken 16 bite işaret genişletir. Intel kılavuzundaki sürelerle program 4 + 2 + 2 + 2 + 2 + 3 + 2 + 4 + 4 + (8 + 4 × 4) = 49 saat turu tutar; sözcük işlenenli bir MUL ile bir DIV ise birlikte 260 turu aşar.

6.8 Örnek: dizi toplama

6.8.1 Program

İki diziyi eleman eleman toplayıp sonucu üçüncü bir diziye yazmak, aritmetik komutların bellekle birlikte kullanıldığı en yalın örnektir. Aşağıdaki program vec3[i] = vec1[i] + vec2[i] işlemini i = 0…3 için yapar.

```
org 100h
jmp start
vec1 db 1, 2, 5, 6
vec2 db 3, 5, 6, 1
vec3 db ?, ?, ?, ?
start:
lea si, vec1
lea bx, vec2
lea di, vec3
mov cx, 4
sum:
mov al, [si]
add al, [bx]
mov [di], al
inc si
inc bx
inc di
loop sum
ret
```

Programın yapısı satır satır şöyledir.

- `JMP start` veri tanımlarının üzerinden atlar. .COM programında kod ve veri aynı kesimdedir ve yürütme 0100h'den başlar. Atlama olmasaydı işlemci vec1'in baytlarını (01 02 05 06) komut diye çözmeye çalışırdı.
- `vec3 db ?, ?, ?, ?` dört baytlık yer ayırır. Soru işareti "ilk değer önemsiz" demektir; emu8086 bu baytlara 0 yazar.
- `LEA` (*load effective address*) bir değişkenin değerini değil adresini (bağıl konumunu) yükler. SI, BX ve DI üç dizinin işaretçileri olur.

- `MOV AL, [SI]` SI'nın gösterdiği bellek baytını okur; bu, yazmaç dolaylı adreslemedir. `ADD AL, [BX]` ikinci dizinin elemanını ekler, `MOV [DI], AL` sonucu üçüncü diziye yazar.
- Üç INC, işaretçileri bir sonraki elemana ilerletir. Diziler bayt dizisi olduğu için her işaretçi 1 artar; sözcük dizisi olsaydı 2 artması gerekirdi.
- `LOOP sum` döngüyü kurar.

LOOP komutu, döngü sayacı olarak her zaman CX'i kullanır. Her yürütülüşünde önce CX'i 1 azaltır, sonra CX'e bakar. CX sıfırdan farklıysa etikete (sum) geri atlar; sıfırsa bir sonraki komuta (RET) geçer. Döngüden önce CX'e 4 yüklendiği için gövde dört kez çalışır: LOOP'tan sonra CX sırasıyla 3, 2, 1 ve 0 olur ve son seferde atlama yapılmaz. LOOP bayrakları değiştirmez. 8086'da atlayan bir LOOP 17, atlamayan 5 saat turu sürer.

6.8.2 Kod izleme

Adım	SI	BX	DI	CX	AL	vec3	Açıklama
`jmp start`	0000h	0000h	0000h	0000h	00h	0 0 0 0	veriler (0102h–010Dh) atlanır
`lea si / bx / di`	0102h	0106h	010Ah	0000h	00h	0 0 0 0	üç dizinin adresi
`mov cx, 4`	0102h	0106h	010Ah	0004h	00h	0 0 0 0	4 tur
tur 1: `mov al,[si]` · `add al, [bx]`	0102h	0106h	010Ah	0004h	04h	0 0 0 0	1 + 3 = 4
tur 1: `mov [di],al`	0102h	0106h	010Ah	0004h	04h	**4** 0 0 0	vec3[0] = 4
tur 1: üç `inc` · `loop`	0103h	0107h	010Bh	0003h	04h	4 0 0 0	CX = 3 ≠ 0 → sum'a dön
tur 2	0104h	0108h	010Ch	0002h	07h	4 **7** 0 0	2 + 5 = 7
tur 3	0105h	0109h	010Dh	0001h	0Bh	4 7 **11** 0	5 + 6 = 11
tur 4	0106h	010Ah	010Eh	0000h	07h	4 7 11 **7**	6 + 1 = 7; CX = 0, döngü biter

Tablo 6.29: Dizi toplama programının izlemesi.



Şekil 6.12: Program bittiğinde bellek: vec3 = 4, 7, 11, 7 (04h 07h 0Bh 07h). Son turdan sonra SI vec2'nin, BX vec3'ün başını gösterir.

Son turdan sonra işaretçiler dizilerin bir eleman ötesini gösterir. SI = 0106h artık vec2'nin ilk elemanıdır; BX = 010Ah vec3'ün ilk elemanıdır. Bu, dizilerin bellekte art arda durmasının doğal sonucudur ve döngü bittiği için zararsızdır. Toplamlar 255'i aşmadığı için CF hiçbir turda 1 olmaz; büyük elemanlarla toplam bir bayta sığmayabilir ve o zaman ADD'den sonra CF'ye bakmak gerekir.

6.8.3 Bellekte program ve veri

Tablo 6.30 programın emu8086 bellek penceresindeki görünümünü verir (CS = DS = 0700h). Kod ve veri aynı kesimde, art arda durur.

Adres	Baytlar	Anlamı
0100h	`EB 0C`	`JMP 010Eh` ; hedef = 0102h + 0Ch
0102h	`01 02 05 06`	vec1 (veri)
0106h	`03 05 06 01`	vec2 (veri)
010Ah	`04 07 0B 07`	vec3 (program çalıştıktan sonra)
010Eh	`BE 02 01`	`MOV SI, 0102h` (LEA'nın kodu)
0111h	`BB 06 01`	`MOV BX, 0106h`
0114h	`BF 0A 01`	`MOV DI, 010Ah`
0117h	`B9 04 00`	`MOV CX, 0004h` ; sabit little-endian
011Ah	`8A 04`	`MOV AL, [SI]` (sum:)
011Ch	`02 07`	`ADD AL, [BX]`
011Eh	`88 05`	`MOV [DI], AL`
0120h	`46 43 47`	`INC SI` , `INC BX` , `INC DI` (birer bayt)
0123h	`E2 F5`	`LOOP 011Ah` ; F5h = −11
0125h	`C3`	`RET`

Tablo 6.30: Dizi toplama programının bellekteki baytları (emu8086).

Tablodan üç şey okunur. Birincisi, emu8086 doğrudan adresli `LEA SI, vec1` satırını `MOV SI, 0102h` ile aynı üç baytlık koda çevirir. İki komut aynı işi yapar: değişkenin bağıl konumu derleme anında bilindiği için onu bir sabit olarak yüklemek yeter. MASM ise LEA'yı kendi işlem koduyla (`8D 36 02 01`) dört bayt olarak üretir; o zaman sonraki adresler birer bayt kayar. İkincisi, atlama komutları hedef adresi değil, bir sonraki komuta göre **uzaklığı** taşır. `JMP` 'nin uzaklığı 0Ch'dir: 0102h + 0Ch = 010Eh. LOOP'un uzaklığı F5h'dir; işaretli olarak −11'dir ve 0125h − 0Bh = 011Ah'ye, yani sum etiketine geri götürür. Üçüncüsü, MOV CX, 4 komutundaki sabit, bellekte düşük bayt önce, `04 00` olarak durur.

6.9 BCD aritmetiği

6.9.1 İkili kodlu onluk sayılar

İnsanlar onluk tabanda düşünür, işlemci ikili tabanda hesaplar. Hesap makineleri, kasa yazılımları, saat yongaları ve yedi parçalı göstergeler gibi onluk basamakları doğrudan gösteren aygıtlarda sayıyı her seferinde ikiliden onluğa çevirmek pahalıdır. Bunun yerine sayı basamak basamak saklanabilir.

> **Tanım 6.8**
>
> **İkili kodlu onluk** (*binary coded decimal*, BCD) gösterimde her onluk basamak ayrı bir 4 bitlik grupta, 0000–1001 değerleriyle tutulur; 1010–1111 kullanılmaz. **Paketsiz BCD** (*unpacked*) her bayta bir basamak koyar; üst yarım 0'dır. **Paketli BCD** (*packed*) bir bayta iki basamak koyar: yüksek yarım onlar, düşük yarım birler basamağıdır.

Gösterim	Bitler	Bayt(lar)
İkili	`0000 1110`	`0Eh`
Paketsiz BCD	`0000 0001 · 0000 0100`	`01h 04h` (AH, AL)
Paketli BCD	`0001 0100`	`14h`
ASCII	`0011 0001 · 0011 0100`	`31h 34h` ('1', '4')

Tablo 6.31: 14 sayısının dört gösterimi.

Paketli BCD'de 14 sayısı 14h baytıyla yazılır; onaltılık gösterimi onluk basamakları aynen okur. ASCII rakamlar ise paketsiz BCD'nin yüksek yarımına 3 eklenmiş hâlidir. Bu yakınlık sayesinde klavyeden okunan rakamlar `AND 0Fh` ile paketsiz BCD'ye, paketsiz BCD de `OR 30h` ile ekrana yazılacak karaktere dönüşür.

İşlemcinin toplayıcısı BCD bilmez; iki BCD sayıyı ikili sayılar gibi toplar. Sonuç çoğu zaman yanlış bir BCD'dir. 9 + 5 ikili olarak 0Eh verir; doğru BCD sonuç “14”tür. 8086 bu yüzden ikili işlemden sonra sonucu BCD'ye çeviren **BCD düzeltme** komutları sunar. Hepsi yalnız AL (ve gerektiğinde AH) üzerinde çalışır; BCD aritmetiği bu yüzden bayt bayt yapılır.

6.9.2 Düzeltme komutları

Komut	Ne zaman?	Etki	Süre (tur)
`AAA`	paketsiz toplamadan sonra	AL'yi BCD'ye düzeltir, elde AH'ye	4
`AAS`	paketsiz çıkarmadan sonra	AL'yi düzeltir, ödünç AH'den	4
`AAM`	paketsiz çarpmadan sonra	AH ← AL / 10, AL ← AL mod 10	83
`AAD`	paketsiz bölmeden **önce**	AL ← AH × 10 + AL, AH ← 0	60

Komut	Ne zaman?	Etki	Süre (tur)
`DAA`	paketli toplamadan sonra	AL'deki iki basamağı düzeltir	4
`DAS`	paketli çıkarmadan sonra	AL'deki iki basamağı düzeltir	4

Tablo 6.32: BCD düzeltme komutları. Hepsi tek baytlık (AAM ve AAD iki baytlık) örtük işlenenli komutlardır.

Düzeltmenin fikri 6 sayısına dayanır. Bir yarım bayt 16 değer alabilir, onluk basamak ise 10. Bir basamağın toplamı 9'u aşınca ikili toplam BCD'den 6 eksik kalır: 9 + 5 = 0Eh, oysa "14" = 14h = 0Eh + 6'dır. Basamak toplamı 15'i aşarsa (örneğin 9 + 8 = 17) yarım bayttan elde çıkar ve AF = 1 olur; bu durumda da 6 eklemek gerekir: 9 + 8 = 11h, 11h + 6 = 17h. Düzeltme komutları bu iki koşulu (yarım değeri > 9 ya da AF = 1) denetler ve gerekiyorsa 6 ekler ya da çıkarır.

Kural (AAA ve DAA)

AAA: AL'nin düşük yarımı 9'dan büyükse ya da AF = 1 ise AL'ye 6, AH'ye 1 eklenir ve AF = CF = 1 olur; değilse AF = CF = 0 olur. Her iki durumda AL'nin yüksek yarımı silinir. **AAS** aynı denetimi yapar, ama AL'den 6, AH'den 1 çıkarır.

DAA: Önce düşük yarım denetlenir: 9'dan büyükse ya da AF = 1 ise AL'ye 06h eklenir ve AF = 1 olur. Sonra yüksek basamak denetlenir: işlemden önceki AL 99h'den büyükse ya da CF = 1 ise AL'ye 60h eklenir ve CF = 1 olur. **DAS** aynı denetimlerle 06h ve 60h çıkarır.

Kuralların uygulanışı birkaç örnekte görülür. 38h + 45h = 7Dh; düşük yarım D > 9 olduğundan DAA 06h ekler ve AL = 83h olur: 38 + 45 = 83. 99h + 01h = 9Ah; düşük yarım A > 9, AL + 06h = A0h; ilk AL 99h'den büyük olduğundan 60h da eklenir ve AL = 00h, CF = 1 olur: 99 + 1 = 100, "00" ve yüzler basamağı CF'de. Çıkarmada 83h − 38h = 4Bh bulunur; düşük yarım B > 9 olduğundan DAS 06h çıkarır ve AL = 45h olur: 83 − 38 = 45. 25h − 48h = DDh; iki düzeltme de gerekir ve DAS AL = 77h, CF = 1 verir. Bu, 125 − 48 = 77 gibi okunur: CF = 1, yüksek basamaktan ödünç alındığını söyler.

6.9.3 Kod izleme: AAA ile 9 + 5

Aşağıdaki program iki paketsiz BCD basamağını toplar, sonucu AAA ile düzeltir ve iki rakamı ekrana yazar. Yazdırma kısmı BIOS'un 10h kesmesinin 0Eh hizmetini (*teletype*) kullanır: AH = 0Eh iken `INT 10h`, AL'deki karakteri imlecin bulunduğu yere yazar ve imleci bir ilerletir.

```
; first number '9':
mov     ah, 09h
; second number '5':
mov     al, 05h
; al = al + ah = 09h + 05h = 0eh
add     al, ah
; clear tens byte of bcd
xor     ah, ah
; adjust result to bcd form, ah = 1, al = 4 -> '14'
aaa
; --- sonucu yazdır (ek) ---
or      ax, 3030h   ; ASCII'ye çevir
mov     bl, al      ; '4' sakla
mov     al, ah      ; '1'
mov     ah, 0Eh
int     10h
mov     al, bl
int     10h
ret
```

Komut	Makine kodu	AX	BL	CF	AF	Açıklama
`mov ah, 09h` · `mov al, 05h`	`B4 09` · `B0 05`	0905h	00h	0	0	iki basamak
`add al, ah`	`02 C4`	090Eh	00h	0	0	9 + 5 = 0Eh; ikili doğru, BCD değil
`xor ah, ah`	`32 E4`	000Eh	00h	0	?	onlar basamağı temizlendi
`aaa`	`37`	**0104h**	00h	**1**	**1**	E > 9: AL + 6 = 14h, AH + 1, yüksek yarım silinir
`or ax, 3030h`	`0D 30 30`	3134h	00h	0	?	'1', '4'
`mov bl, al` · `mov al, ah`	`8A D8` · `8A C4`	3131h	34h	0	?	'4' saklandı, AL = '1'
`mov ah, 0Eh` · `int 10h`	`B4 0E` · `CD 10`	0E31h	34h	0	?	ekrana '1'
`mov al, bl` · `int 10h`	`8A C3` · `CD 10`	0E34h	34h	0	?	ekrana '4'

Tablo 6.33: AAA ile 9 + 5 izlemesi. Ekranda “14” görünür.

09h + 05h = 0Eh, ikili olarak 14'tür ve doğrudur; ama BCD'de E diye bir basamak yoktur. Düşük yarımlar 9 + 5 toplanırken bit 3'ten elde çıkmamıştır, AF = 0'dır. AAA düşük yarımın E > 9 olduğunu görür. AL'ye 6 ekler: 0Eh + 6 = 14h. AH'yi 1 artırır, AF ve CF'yi 1 yapar, AL'nin yüksek yarımını siler: AL = 04h. Sonuç AX = 0104h, yani paketsiz “14”tür. `OR AX, 3030h` iki baytın yüksek yarımlarına 3 koyar ve AX = 3134h, ASCII '1' ve '4' olur. Programın geri kalanı bu iki karakteri sırayla ekrana yazar.

`XOR AH, AH` satırının amacı, AAA'nın AH'ye ekleyeceği eldeyi temiz bir onlar basamağına yazmaktır. Bu satır olmasaydı AH'de birinci basamak (09h) kalır ve AAA onu 0Ah yapardı. AAA AL'nin yalnız düşük yarımına bakar; bu yüzden toplamadan önce ASCII rakamlarından 30h çıkarmak gerekmez. '7' + '8' = 37h + 38h = 6Fh'de düşük yarım F > 9'dur; AAA AH = 01h, AL = 05h verir ve `OR AX, 3030h` ile '1', '5' elde edilir.

6.9.4 Kod izleme: AAS ile 5 − 8

```
; make 5 - 8, al = 0fdh (not in bcd form)
mov     al, 05h
mov     bl, 08h
sub     al, bl
; convert to bcd, al = 7
; and 1 is borrowed from ah,
; like calculating 15 - 8:
aas
; --- yazdır ---
or      al, 30h
mov     ah, 0Eh
int     10h
mov     ah, 0
int     16h
ret
```

Komut	AX	BL	CF	AF	SF	Açıklama
`mov al, 05h` · `mov bl, 08h`	0005h	08h	0	0	0	
`sub al, bl`	00FDh	08h	1	1	1	5 − 8 = −3 = FDh
`aas`	**FF07h**	08h	**1**	**1**	?	AL − 6 = F7h, AH − 1 = FFh; yüksek yarım silinir
`or al, 30h`	FF37h	08h	0	?	0	AL = '7'
`mov ah, 0Eh` · `int 10h`	0E37h	08h	0	?	0	ekrana '7'
`mov ah, 0` · `int 16h`	—	08h	0	?	0	tuş beklenir; AL = basılan tuşun kodu

Tablo 6.34: AAS ile 5 − 8 izlemesi. Program AX = 0000h ile başlar.

5 − 8 = −3, ikiye tümleyenle FDh'dir. Ödünç alınmıştır (CF = 1); düşük yarımda da 5 − 8 ödünç istemiştir (AF = 1). AAS, AF = 1 olduğu için AL'den 6 çıkarır: FDh − 6 = F7h. AH'den 1 ödünç alır: 00h − 1 = FFh. AL'nin yüksek yarımını siler: AL = 07h. Sonuç AX = FF07h'dir. Ekranda “7” görünür.

Bu sonuç ilk bakışta tuhaftır: 5 − 8 = −3 olmalıydı. AAS'ın yaptığı, kâğıt üzerindeki çıkarmadaki gibi bir üst basamaktan 10 ödünç almaktır: 15 − 8 = 7. CF = 1, ödünç alındığını ve sonucun negatif olduğunu söyler. 7, −3'ün 10'a tümleyenidir (10 − 3 = 7). Çok basamaklı bir BCD çıkarmada bu ödünç, bir sonraki basamağın çıkarmasında SBB ile ödenir; AH'nin FFh olması da bunun bir yansımasıdır. AH = 01h ile başlansaydı (paketsiz “15”) AAS AX = 0007h, yani doğru “07” sonucunu verirdi.

6.9.5 AAM ve AAD

Çarpma ve bölmenin düzeltmesi toplamadan farklı çalışır, çünkü burada 6 eklemek yetmez; sayı gerçekten 10'a bölünmeli ya da 10 ile çarpılmalıdır. 8086 bu iki işi mikrokodla yapan iki komut sunar.

Tanım 6.9

AAM (*ASCII adjust after multiply*) AL'yi 10'a böler: bölüm AH'ye, kalan AL'ye yazılır. İki paketsiz basamağın çarpımını (en çok 9 × 9 = 81) iki paketsiz basamağa ayırır. **AAD** (*ASCII adjust before division*) AH'deki onlar ve AL'deki birler basamağını ikili sayıya çevirir: AL ← AH × 10 + AL, AH ← 0. Bölme komutundan *önce* yazılır. İkisi de SF, ZF ve PF'yi AL'ye göre ayarlar; OF, AF ve CF tanımsızdır.

```
mov al, 7
mov bl, 8
mul bl      ; AX = 0038h (56)
aam         ; AH = 05h, AL = 06h
mov ax, 0504h  ; '54' paketsiz
aad            ; AX = 0036h (54)
mov bl, 6
div bl         ; AL = 09h, AH = 00h
```

İlk parçada 7 × 8 = 56 = 0038h bulunur. AAM, 56'yı 10'a böler: AH = 5, AL = 6. AX = 0506h artık paketsiz “56”dır. İkinci parçada AX = 0504h paketsiz “54”tür. AAD bunu 5 × 10 + 4 = 54 = 36h'ye çevirir ve AH'yi sıfırlar. Normal DIV ile 54 / 6 = 9 bulunur: AL = 09h, kalan AH = 00h. AAD unutulsaydı DIV, 0504h = 1284'ü bölerdi ve 1284 / 6 = 214 olurdu.

AAM ve AAD'nin makine kodu ilginçtir: `D4 0A` ve `D5 0A`. İkinci bayt 0Ah, yani 10'dur. 8086'da bu bayt gerçekten taban olarak kullanılır; `D4 10` AL'yi 16'ya böler. Intel bunu belgelememiştir ve assembler'lar her zaman 0Ah üretir. `D4 00` sıfıra bölme olduğu için INT 0 üretir. AAM 83, AAD 60 saat turu sürer; mikrokodla yapılan bir bölme ve çarpmadır.

6.9.6 Kod izleme: AAM, AAD ve DAA

```
MOV  AL, 6
MOV  BL, 9
MUL  BL          ; AX = 6 × 9
AAM              ; basamaklara ayır
MOV  AX, 0702h   ; '72' paketsiz
AAD              ; ikiliye çevir
MOV  BL, 8
DIV  BL
MOV  AL, 19h     ; paketli 19
ADD  AL, 28h     ; + 28
DAA              ; BCD düzelt
RET
```

Komut	Makine kodu	AX	BL	AF	CF	Açıklama
`MOV AL, 6` · `MOV BL, 9`	`B0 06` · `B3 09`	0006h	09h	0	0	
`MUL BL`	`F6 E3`	0036h	09h	?	0	6 × 9 = 54; AL'ye sığdı
`AAM`	`D4 0A`	**0504h**	09h	?	?	paketsiz “54”
`MOV AX, 0702h`	`B8 02 07`	0702h	09h	?	?	paketsiz “72”
`AAD`	`D5 0A`	**0048h**	09h	?	?	7 × 10 + 2 = 72 = 48h
`MOV BL, 8` · `DIV BL`	`B3 08` · `F6 F3`	0009h	08h	?	?	72 / 8 = 9, kalan 0
`MOV AL, 19h`	`B0 19`	0019h	08h	?	?	paketli “19”
`ADD AL, 28h`	`04 28`	0041h	08h	**1**	0	9 + 8 = 17: yarımdan elde
`DAA`	`27`	**0047h**	08h	1	0	AF = 1 → AL + 06h = 47h

Tablo 6.35: AAM, AAD ve DAA izlemesi. ? tanımsız bayrak.

Programın son üç satırı DAA'nın AF'ye neden ihtiyaç duyduğunu gösterir. 19h + 28h = 41h'dir. Sonucun düşük yarımı 1'dir ve 9'dan büyük değildir; yalnız ona bakılsa düzeltme gerekmez gibi görünür. Ama birler basamağında 9 + 8 = 17 olmuş ve yarımdan dışarı elde çıkmıştır: AF = 1. DAA bu bayrağı görür ve 06h ekler: 41h + 06h = 47h. Sonuç “47” = 19 + 28'dir. Yüksek yarım 4 ≤ 9 ve CF = 0 olduğu için ikinci düzeltme yapılmaz.

Sık yapılan hata

Düzeltme komutunu toplamadan hemen sonra yazmamak. DAA ve AAA, AF ile CF'ye bakar; araya bir MOV girse sorun çıkmaz, ama bayrakları değiştiren bir komut (ADD, CMP, AND, hatta AF'yi değiştiren INC) düzeltmeyi bozar. Bir başka hata, paketli BCD'de AAA ya da paketsiz BCD'de DAA kullanmaktır: AAA yüksek yarımı siler ve paketli sayının onlar basamağını yok eder.

6.10 Örnek: sıcaklık dönüşümü

6.10.1 Santigrattan Fahrenhayta

Santigrat derecesi (°C) ile Fahrenhayt derecesi (°F) arasındaki bağıntı f = c × 9 / 5 + 32'dir. Aşağıdaki program bu formülü tam sayı işlemleriyle hesaplar. Sıcaklıklar negatif olabileceği için işaretli komutlar (IMUL, IDIV) kullanılır.

```
org 100h
start:
;celsius to fahrenheit according to f = c * 9 / 5 + 32
mov cl, 15
mov al, 9
imul cl
mov cl, 5
idiv cl
add al, 32
ret
```

Komut	Makine kodu	AX	CL	CF	OF	Açıklama
`mov cl, 15`	`B1 0F`	0000h	0Fh	0	0	c = 15
`mov al, 9`	`B0 09`	0009h	0Fh	0	0	
`imul cl`	`F6 E9`	0087h	0Fh	1	1	9 × 15 = 135; işaretli bayta sığmaz, AX'te eksiksiz
`mov cl, 5`	`B1 05`	0087h	05h	1	1	bölen
`idiv cl`	`F6 F9`	001Bh	05h	?	?	135 / 5 = 27, kalan 0
`add al, 32`	`04 20`	003Bh	05h	0	0	27 + 32 = 59

Tablo 6.36: 15 °C'nin Fahrenhayta çevrilmesi.

9 × 15 = 135 bulunur. 135, işaretli bir bayta (en çok 127) sığmaz; IMUL bu yüzden CF = OF = 1 yapar. Ama bu bir hata değildir: sonuç AX'te 0087h olarak eksiksiz durur. IDIV, AX'i 5'e böler; bölüm 27 = 1Bh AL'ye, kalan 0 AH'ye yazılır. 27 + 32 = 59 = 3Bh bulunur: 15 °C = 59 °F. Program bittiğinde AX = 003Bh, CX = 0005h'dir.

Formüldeki işlem sırası önemlidir. Önce 9 ile çarpılır, sonra 5'e bölünür. Tersi yapılsaydı, yani önce 9 / 5 hesaplansaydı, tam sayı bölmesi 1 verirdi ve f = c + 32 gibi yanlış bir formül ortaya çıkardı. Tam sayı aritmetiğinde bölme mümkün olduğunca sona bırakılır.

6.10.2 Fahrenhayttan Santigrata

```
org 100h
start:
;fahrenheit to celsius according to: c = (f - 32) * 5 / 9
mov cl, 59
sub cl, 32
mov al, 5
imul cl
mov cl, 9
idiv cl
ret
```

Komut	Makine kodu	AX	CL	CF	OF	Açıklama
`mov cl, 59`	`B1 3B`	0000h	3Bh	0	0	f = 59
`sub cl, 32`	`80 E9 20`	0000h	1Bh	0	0	59 − 32 = 27
`mov al, 5`	`B0 05`	0005h	1Bh	0	0	
`imul cl`	`F6 E9`	0087h	1Bh	1	1	5 × 27 = 135
`mov cl, 9` · `idiv cl`	`B1 09` · `F6 F9`	000Fh	09h	?	?	135 / 9 = 15, kalan 0

Tablo 6.37: 59 °F'nin Santigrata çevrilmesi.

59 − 32 = 27, 5 × 27 = 135 ve 135 / 9 = 15 bulunur: 59 °F = 15 °C. Sonuç AL'dedir; AH kalanı (0) taşır. İki program birbirinin tersidir ve 15 ile 59 arasında tam olarak gidip gelir, çünkü iki bölmede de kalan sıfırdır.

6.10.3 Neden IMUL ve IDIV? Sınırlar

Negatif sıcaklıklar işaretli komutların gerekçesidir. c = −10 için IMUL, −10 = F6h'yi işaretli okur ve AX = FFA6h = −90 verir. IDIV −90 / 5 = −18 = EEh hesaplar; 32 eklenince 0Eh = 14 bulunur: −10 °C = 14 °F doğrudur. Aynı hesap MUL ve DIV ile yapılsaydı F6h, 246 sanılırdı. MUL 9 × 246 = 2214 = 08A6h verir ve DIV bu sayıyı 5'e bölünce bölüm 442 AL'ye sığmaz: program bölme hatasıyla durur.

Tam sayı bölmesi kesirli kısmı atar. IDIV sıfıra doğru kestiği için c = 1 °C, 9 / 5 = 1 ile 33 °F olur; doğrusu 33,8'dir. Yuvarlama gerekiyorsa AH'deki kalan incelenir: kalan bölenin yarısından büyükse bölüme 1 eklenir.

Programın en önemli sınırı, sonucun işaretli bir bayt olan AL'de tutulmasıdır. Tablo 6.38 santigrattan fahrenhayta dönüşümün sınırlarını sayılarla gösterir.

c	IMUL: AX	IDIV: AL	ADD sonrası AL	OF	Sonuç
−72	FD78h (−648)	bölüm −129 sığmaz → INT 0			bölme hatası
−71	FD81h (−639)	81h (−127)	A1h (−95)	0	doğru
−10	FFA6h (−90)	EEh (−18)	0Eh (14)	0	doğru

c	IMUL: AX	IDIV: AL	ADD sonrası AL	OF	Sonuç
53	01DDh (477)	5Fh (95)	7Fh (127)	0	doğru (en büyük)
54	01E6h (486)	61h (97)	81h	1	yanlış: −127 okunur
72	0288h (648)	bölüm 129 sığmaz → INT 0			bölme hatası

Tablo 6.38: °C → °F programının sınır değerleri.

Geçerli aralık c ∈ [−71, 53]'tür. c = 54'te bölüm 97 sığar, ama 97 + 32 = 129 işaretli bayta sığmaz; ADD OF = 1 yapar ve AL = 81h, −127 diye okunur. c = 72'de bölümün kendisi (129) AL'ye sığmaz ve bölme hatası oluşur. Fahrenhayttan santigrata dönüşümde sınır f − 32'nin bir bayta sığmasıdır: f ∈ [−96, 127]. Daha geniş aralıklar için sözcük işlemleri gerekir: c AX'e alınır ve `IMUL BX` ile 16 bitte çarpılır. Çarpım zaten DX:AX'te durduğu için CWD gerekmez; doğrudan `IDIV` ile bölünür.

6.11 Hangi komut hangi bayrağı etkiler?

Tablo 6.39 bu notta anlatılan bütün komutların bayrak etkisini tek tabloda toplar. Bir programı izlerken ya da bir koşullu atlamanın neden beklenmedik davrandığını ararken ilk bakılacak yer burasıdır.

Komut	CF	ZF	SF	OF	PF	AF
`ADD SUB CMP ADC SBB`	✓	✓	✓	✓	✓	✓
`NEG`	✓	✓	✓	✓	✓	✓
`INC DEC`	—	✓	✓	✓	✓	✓
`AND OR XOR TEST`	0	✓	✓	0	✓	?
`NOT`	—	—	—	—	—	—
`MUL IMUL`	✓	?	?	✓	?	?
`DIV IDIV`	?	?	?	?	?	?
`SHL SHR SAR`	✓	✓	✓	✓[1]	✓	?
`ROL ROR RCL RCR`	✓	—	—	✓[1]	—	—
`AAA AAS`	✓	?	?	?	?	✓
`AAM AAD`	?	✓	✓	?	✓	?
`DAA DAS`	✓	✓	✓	?	✓	✓
`CBW CWD MOV LEA LOOP`	—	—	—	—	—	—

Tablo 6.39: Komutların durum bayraklarına etkisi. ✓ sonuca göre, 0 temizlenir, — değişmez, ? tanımsız, [1] yalnız sayı 1 iken tanımlı.

Tablo birkaç genel kurala indirgenebilir. Veri taşıyan komutlar (MOV, LEA, XCHG, CBW, CWD) ve LOOP bayraklara dokunmaz. Toplama ve çıkarma ailesi bütün bayrakları yazar; INC ve DEC bunlardan yalnız CF'yi korur. Mantık komutları CF ile OF'yi temizler. Çarpma yalnız CF ile OF'ye, bölme hiçbir bayra-

ğa anlamlı bir değer yazmaz. Döndürme yalnız CF ve OF'yi değiştirir. Tanımsız bir bayrağa güvenen program bir işlemcide çalışıp ötekinde bozulabilir.

6.12 Sık yapılan hatalar

Hatalı	Ne olur?	Doğrusu
İşaretli sayılarda CF'ye bakmak	−1 + 1 gibi doğru sonuçlar "taşmış" sanılır	işaretli: OF; karşılaştırmada SF ≠ OF
`DIV BX` öncesi DX'i unutmak	eski DX bölüneni büyütür; yanlış sonuç ya da INT 0	`XOR DX, DX` ya da `CWD`
İşaretli bayt için `MOV AH, 0`	−30 = E2h, 226 sayılır	`CBW`
`MUL 10`	8086'da sabit işlenen yok; assembler red-deder	`MOV BL, 10` · `MUL BL`
`SHL AX, 4`	80186 komutu; 8086'da çalışmaz	`MOV CL, 4` · `SHL AX, CL`
İki sözcükte de ADD	düşük sözcüğün eldesi kaybolur	ADD, ardından ADC
ADD ile ADC arasına CMP koymak	CF silinir	arada yalnız MOV, INC, DEC, LOOP
Bit sınamak için AND	sayının öteki bitleri silinir	`TEST`
Negatif sayıyı SAR ile bölüp IDIV sonucu beklemek	−7 / 2: SAR −4, IDIV −3	yuvarlama yönünü bilerek seçmek
AAD'yi DIV'den sonra yazmak	paketsiz sayı ikili sanılır: 0702h = 1794 bölünür	AAD bölmeden önce

Tablo 6.40: Aritmetik ve mantık komutlarında sık yapılan hatalar.

> **Sık yapılan hata**
>
> MUL ve IMUL'ün işlenen boyutunu yanlış seçmek. `MUL BX` sözcük çarpmasıdır ve sonucu DX:AX'e yazar; DX'te saklanan bir değer varsa sessizce silinir. Bayt çarpma isteniyorsa işlenen 8 bitlik olmalıdır (`MUL BL`); o zaman yalnız AX değişir.

6.13 Alıştırmalar ve çözümleri

Sunumdaki on bir alıştırma burada ara adımlarıyla çözülmüştür.

1. `MOV AL, 0C0h` ve `ADD AL, 0C0h` sonrası AL, CF, OF, SF, ZF, PF nedir?

C0h + C0h = 180h; AL = 80h, CF = 1. −64 + (−64) = −128 işaretli bayta sığar: OF = 0. SF = 1, ZF = 0. 80h'de tek bir 1 biti vardır: PF = 0.

2. `CMP AL, BL` sonrası işaretsiz "AL < BL" hangi bayrakla, işaretli "AL < BL" hangi bayraklarla anlaşılır?

İşaretsiz: CF = 1 (çıkarma ödünç istemiştir); bunu `JB` sınar. İşaretli: SF ≠ OF; bunu `JL` sınar. SF tek başına yetmez, çünkü fark taşarsa işaret biti ters döner.

3. AX = 0FFFFh, BX = 0FFFFh iken `MUL BX` sonrası DX, AX, CF nedir?

65.535^2 = 4.294.836.225 = FFFE0001h. DX = FFFEh, AX = 0001h. Üst yarı sıfır olmadığından CF = OF = 1.

4. AX = 0100h, BL = 0 iken `DIV BL` ne olur?

Sıfıra bölme: işlemci INT 0 (bölme hatası) üretir. AX ve bayraklar değişmez; denetim kesme hizmetine geçer.

5. Dizi toplama programını sözcük (DW) dizileri için değiştirin.

Veriler `dw` ile tanımlanır, AL yerine AX kullanılır ve işaretçiler 2'şer ilerler:

```
vec1 dw 1000, 2000, 3000, 4000
vec2 dw 5, 6, 7, 8
vec3 dw 4 dup(?)
; ...
sum:
mov ax, [si]
add ax, [bx]
mov [di], ax
add si, 2
add bx, 2
add di, 2
loop sum
```

Bu verilerle vec3 = 1005, 2006, 3007, 4008 olur. `INC SI` iki kez yazılabilir; ama `ADD SI, 2` CF'yi değiştirir, INC değiştirmez. Döngü ADC kullanıyorsa INC'ler tercih edilir.

6. Bir bayttaki 1 bitlerini sayan programı SHL/TEST ve döngüyle yazın. PF ile karşılaştırın.

```
mov al, 0B5h     ; sayılacak bayt
xor bl, bl       ; sayaç
mov cx, 8
tekrar:
test al, 80h     ; en soldaki bit 1 mi?
jz sifir
inc bl
sifir:
shl al, 1
loop tekrar
```

B5h = 1011 0101b için BL = 5 bulunur. Her turda TEST en soldaki biti sınar, SHL bir sonraki biti en sola getirir. `JZ` (ZF = 1 ise atla) bit 0 ise sayacı artırmayı atlar. PF yalnız 1'lerin sayısının tek mi çift mi olduğunu söyler: B5h için PF = 0'dır (beş tek sayıdır), ama kaç tane 1 olduğunu vermez.

7. DX = 0005h, AX = 0000h, BX = 0005h iken `DIV BX` ne olur?

Bölünen DX:AX = 50000h = 327.680'dir. Bölüm 65.536 olurdu ve AX'e sığmaz. Kural da bunu söyler: DX = 5 ≥ bölen 5. Bölen sıfır olmasa da bölme hatası (INT 0) oluşur.

8. `MOV AL, 70h` ve `ADD AL, 30h` sonrası AL ve OF nedir? Doğru toplam 16 bitte nasıl elde edilir?

AL = A0h, OF = 1: 112 + 48 = 160 > 127. İki sayı ayrı ayrı CBW ile genişletilip 16 bitte toplanır: `MOV AL, 70h` · `CBW` · `MOV DX, AX` · `MOV AL, 30h` · `CBW` · `ADD AX, DX` → AX = 00A0h = 160.

9. 00012345h − 0000ABCDh işlemini SUB/SBB ile yapın. SUB sonrası CF kaçtır?

2345h − ABCDh = 7778h, CF = 1 (ödünç). 0001h − 0000h − 1 = 0000h. Sonuç 00007778h = 30.584.

10. 32 × 32 bit çarpmada dört kısmi çarpım sonucun hangi sözcüklerine eklenir?

A0·B0 → z1:z0; A1·B0 ve A0·B1 → z2:z1; A1·B1 → z3:z2. Her ADD'nin eldesi ADC ile bir üst sözcüğe taşınır; üçüncü kısmi çarpımdan sonra z3'e yalnız elde ekleyen bir `ADC [z+6], 0` gerekebilir.

11. DX = 0000h, AX = 0FFFh, CF = 0 iken `SHL AX, 1` ve `RCL DX, 1` sonrası DX:AX nedir?

SHL: AX = 1FFEh, çıkan bit (b15 = 0) CF'ye: CF = 0. RCL: DX'e 0 girer, DX = 0000h. DX:AX = 00001FFEh = 0FFFh × 2.

6.14 Özet

- **Bayraklar**: aritmetik ve mantık komutları sonucu hedefe, sonucun özelliklerini bayrak yazmacına yazar. CF işaretsiz taşmayı (toplamada elde, çıkarmada ödünç), OF işaretli taşmayı gösterir; OF = en anlamlı bite giren ve çıkan eldelerin XOR'u. ZF sonuç sıfır, SF en anlamlı bit, PF düşük baytta çift sayıda 1, AF bit 3'ten elde.
- **İki işlenenli komutlar** (ADD, ADC, SUB, SBB, CMP, AND, OR, XOR, TEST) sonucu hedefe yazar; CMP ve TEST yalnız bayrakları değiştirir. İki bellek işleneni, farklı boyut, sabit hedef ve kesim yazmacı kullanılamaz. Yazmaç-yazmaç 3, bellek hedefli 16 + EA saat turu.
- **Karşılaştırma**: CMP sonrası işaretsiz “küçük” CF = 1, işaretli “küçük” SF ≠ OF.
- **Çok sözcüklü aritmetik**: en düşük sözcükte ADD/SUB, üsttekilerde ADC/SBB; elde CF'de taşınır. 32 × 32 çarpma dört MUL ve bir ADD/ADC zinciriyle yapılır.
- **Mantık komutları** bit bit çalışır; CF = OF = 0 yapar. AND bit temizler, OR kurar, XOR tersler, TEST sınar. NOT hiçbir bayrağı değiştirmez.

- **Çarpma ve bölme**: bayt işlemde AX = AL × işlenen, AL = AX / işlenen (AH kalan); sözcükte DX:AX. MUL/IMUL yalnız CF = OF'yi anlamlı ayarlar; DIV/IDIV bütün bayrakları tanımsız bırakır. Bölen 0 ya da bölüm sığmazsa INT 0. IDIV sıfıra doğru keser.
- **İşaret genişletme**: CBW (AL → AX) ve CWD (AX → DX:AX) işaretli bölmeden ve işaretli toplamadan önce; işaretsiz veride üst yarı sıfırlanır.
- **INC ve DEC** CF'yi korur; **NEG** ikiye tümleyen alır ve işlenen sıfır değilse CF = 1 yapar; NEG 80h taşar.
- **Kaydırma ve döndürme**: sayı 1 ya da CL. SHL ×2, SHR işaretsiz ÷2, SAR işaretli ÷2 (eksi sonsuza yuvarlar). Döndürmede bit kaybolmaz; RCL/RCR CF ile 9 ya da 17 bitlik halka kurar ve çok sözcüklü kaydırmayı sağlar. Kaydırmayla çarpma MUL'den çok daha hızlıdır.
- **BCD**: AAA/AAS paketsiz toplama/çıkarma sonrası, DAA/DAS paketli toplama/çıkarma sonrası, AAM çarpma sonrası, AAD bölme öncesi. Düzeltme 6 (ve paketlide 60h) eklemeye ya da çıkarmaya dayanır; AF bu yüzden vardır.
- **Uygulama**: dizi toplama LOOP ile, sıcaklık dönüşümü IMUL/IDIV ile yapılır; işaretli bayt sonucu yalnız belirli bir aralıkta doğrudur.

6.15 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

aritmetik-mantık birimi (ALU) *arithmetic logic unit* · yürütme birimi (EU) *execution unit* · bayrak yazmacı (FLAGS) *flags register* · durum bayrağı / kontrol bayrağı *status / control flag* · elde bayrağı (CF) *carry flag* · taşma bayrağı (OF) *overflow flag* · sıfır, işaret, eşlik bayrağı *zero, sign, parity flag* · yardımcı elde bayrağı (AF) *auxiliary carry flag* · elde / ödünç *carry / borrow* · işaretli / işaretsiz sayı *signed / unsigned number* · ikiye tümleyen / birler tümleyeni *two's / one's complement* · aritmetik komut / mantık komutu *arithmetic / logical instruction* · örtük işlenen *implicit operand* · maskeleme *masking* · çarpma / bölme *multiplication / division* · bölünen / bölen / bölüm / kalan *dividend / divisor / quotient / remainder* · işaret genişletme *sign extension* · kaydırma / döndürme *shift / rotate* · mantıksal / aritmetik kaydırma *logical / arithmetic shift* · ikili kodlu onluk (BCD) *binary-coded decimal* · paketli / paketsiz BCD *packed / unpacked BCD* · BCD düzeltmesi *BCD adjustment* · nibble · döngü *loop* · koşullu atlama *conditional jump* · kesme vektörü *interrupt vector* · mikrokod *microcode* · saat turu *clock cycle* · yol döngüsü *bus cycle*

6.16 Kendini Dene

1. `MOV AL, 9Ch` ve `ADD AL, 9Ch` sonrası AL ve altı durum bayrağı nedir?

   Cevap

   9Ch + 9Ch = 138h; AL = 38h, CF = 1. İşaretli −100 + (−100) = −200 bayta sığmaz ve iki negatifin toplamı pozitif (38h) çıkmıştır: OF = 1. SF = 0, ZF = 0. 38h = 0011 1000b'de üç tane 1: PF = 0. Düşük yarımlar C + C = 18h; bit 3'ten elde çıkar: AF = 1.

2. `SUB AL, BL` komutu AL = 30h, BL = 50h için ve AL = 80h, BL = 01h için hangi CF ve OF değerlerini verir?

   Cevap

   30h − 50h = E0h: işaretsiz 48 − 80 ödünç ister, CF = 1; işaretli 48 − 80 = −32 doğrudur, OF = 0. 80h − 01h = 7Fh: işaretsiz 128 − 1 = 127 doğrudur, CF = 0; işaretli −128 − 1 = −129 sığmaz ve negatiften pozitif çıkmıştır, OF = 1.

3. AL = F0h, BL = 10h iken `IMUL BL` ve `MUL BL` sonrası AX ve CF nedir? AL = F8h olsaydı IMUL ne verirdi?

   Cevap

   IMUL: −16 × 16 = −256 = FF00h; −256 bayta sığmaz, CF = OF = 1. MUL: 240 × 16 = 3840 = 0F00h; AH ≠ 0, CF = 1. AL = F8h (−8) için IMUL −128 = FF80h verir; AH yalnız AL'nin işaret uzantısıdır ve CF = OF = 0 olur.

4. −50 sayısı 7'ye ve 8'e IDIV ile bölünüyor (AX = −50). Bölüm ve kalanlar nedir? −50, SAR ile 3 bit kaydırılırsa ne bulunur?

   Cevap

   IDIV 7: −50 / 7 = −7,14; sıfıra doğru −7 = F9h, kalan −50 − (−49) = −1 = FFh. IDIV 8: bölüm −6 = FAh, kalan −2 = FEh. SAR 3 (÷ 8): CEh = 1100 1110b → 1111 1001b = F9h = −7; −6,25 eksi sonsuza doğru −7'ye yuvarlanmıştır. IDIV ile SAR farklı sonuç verir.

5. CF = 1 ve AL = 81h iken art arda üç `RCL AL, 1` komutundan sonra AL ve CF nedir?

   Cevap

   1000 0001b, CF = 1 → 0000 0011b, CF = 1 → 0000 0111b, CF = 0 → 0000 1110b, CF = 0. AL = 0Eh, CF = 0. Dokuz bitlik halkada bitler dönmüş, hiçbiri kaybolmamıştır.

6. Paketli BCD ile 58 + 67 yapılıyor: `MOV AL, 58h`, `ADD AL, 67h`, `DAA`. ADD ve DAA sonrası AL, AF ve CF nedir?

   Cevap

   ADD: 58h + 67h = BFh; düşük yarımda 8 + 7 = 15 = Fh, elde yok (AF = 0), CF = 0. DAA: düşük yarım F > 9 → +06h = C5h, AF = 1; ilk AL BFh > 99h → +60h = 125h, AL = 25h, CF = 1. Sonuç "25" ve yüzler basamağında 1: 58 + 67 = 125.

7. AL = B6h iken `AND AL, 0F0h`, `OR AL, 05h`, `XOR AL, 0FFh` sonrası AL ve PF nedir?

   Cevap

   AND: B0h (düşük yarım silinir). OR: B5h (bit 0 ve 2 kurulur). XOR: 4Ah = 0100 1010b (bütün bitler terslenir). Üç tane 1 vardır: PF = 0. Mantık komutlarından sonra CF = OF = 0'dır.

8. 48 bitlik DX:BX:AX = 0001 FFFF FFFFh sayısına 1 eklemek için hangi üç komut yazılır? Sonuç nedir?

   Cevap

   `ADD AX, 1` · `ADC BX, 0` · `ADC DX, 0`. AX = 0000h ve CF = 1; BX = FFFFh + 0 + 1 = 0000h ve CF = 1; DX = 0001h + 0 + 1 = 0002h. Sonuç 0002 0000 0000h'dir. `INC AX` kullanılsaydı CF değişmeyecek ve elde kaybolacaktı.

9. `MOV AX, 8000h` ve `NEG AX` sonrası AX, CF ve OF nedir? Neden?

   Cevap

   AX = 8000h kalır; CF = 1 (işlenen sıfır değil), OF = 1. 8000h işaretli sözcükte −32.768'dir; +32.768 sözcük aralığına (en çok 32.767) sığmaz. NOT 8000h = 7FFFh, +1 = 8000h.

10. `INC AX` ile `ADD AX, 1` arasında makine kodu, süre ve bayrak bakımından ne fark vardır?

    Cevap

    `INC AX` tek bayttır (`40`) ve 2 saat turu sürer; CF'ye dokunmaz. `ADD AX, 1` üç bayttır (`05 01 00` , ya da 83 biçimiyle de üç bayt) ve 4 tur sürer; CF dahil altı bayrağı yazar. AX = FFFFh iken ADD, CF = 1 yapar; INC CF'yi olduğu gibi bırakır.

6.17 İleri okuma

- B. B. Brey, *The Intel Microprocessors* (8. baskı), Bölüm 5: aritmetik ve mantık komutları, BCD ve ASCII düzeltmeleri, kaydırma ve döndürme; bu notun konularını 8086'dan sonraki işlemcilere de uzatır.
- Intel, *The 8086 Family User's Manual* (1979), komut kümesi bölümü: her komutun bayrak etkisi, kodlaması ve saat turu süreleri birinci elden buradadır.
- K. R. Irvine, *Assembly Language for x86 Processors* (7. baskı), Bölüm 7: tam sayı aritmetiği, çok sözcüklü kaydırma ve toplama, ASCII ve paketli onluk aritmetik.
- emu8086 belgeleri, "8086 instructions": her komutun kısa tanımı, bayrak tablosu ve örnek kodu; sunumdaki BCD örneklerinin kaynağıdır.

6.18 Kaynaklar

1. Intel, *The 8086 Family User's Manual*, 1979 — Instruction Set bölümü.
2. B. B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson, 2009 — Bölüm 5.
3. emu8086 yardım belgeleri: "8086 instructions" ve "Arithmetic and logic instructions".
4. K. R. Irvine, *Assembly Language for x86 Processors*, 7. baskı, Pearson, 2014 — Bölüm 7.
5. A. S. Tanenbaum, *Modern Operating Systems*, 3. baskı, Pearson, 2008.
6. M. Bodur, *Mikroişlemcilere Giriş: Assembler ile Yazılım ve Arayüz*, EMO, 2014.
7. Ş. Ç. Öztürk, O. Ergin, *Intel 8086 ile Mikroişlemci Programlamaya Giriş*, 2009.

KISIM III · BÖLÜM 7

Akış Kontrolü

İşlemci, kendi hâline bırakıldığında komutları bellekteki sırasıyla yürütür. Her komuttan sonra komut işaretçisi IP, komutun uzunluğu kadar artar ve bir sonraki komutu gösterir. Karar veren, bir işi tekrarlayan ya da bir hata durumunda başka yola sapan her program bu düz çizgiyi kırmak zorundadır. Akış kontrol komutları bunu IP'ye yeni bir değer yükleyerek yapar. Bu not önce koşulsuz atlamayı (JMP) ve atlama uzaklığının makine koduna nasıl yazıldığını anlatır. Ardından bayraklara bakan koşullu atlamalar, CMP komutu ve işaretli ile işaretsiz sayıların neden ayrı atlama ailelerine ihtiyaç duyduğu gelir. Yüksek düzeyli dillerdeki if/else ve while yapılarının assembly karşılıkları kurulur. Son olarak CX yazmacını sayaç olarak kullanan LOOP ailesi ve JCXZ incelenir. Her örnek program yazmaç, bayrak ve bellek düzeyinde adım adım izlenir; atlamanın önden getirme kuyruğu ve saat turları üzerindeki etkisi de ayrıca açıklanır.

Kod (8086): `github.com/sercankulcu/microprocessors-8086/Bolum07`

ÖĞRENME ÇIKTILARI

Atlama mekanizması
JMP ve etiketlerle programın akışını yönlendirmek; IP'nin nasıl değiştiğini ve atlama uzaklığının (bağıl konumun) nasıl hesaplandığını açıklamak; kısa, yakın, uzak ve dolaylı atlamayı ayırt etmek.

Koşullu atlamalar
Her koşullu atlamanın hangi bayrağa (ZF, CF, SF, OF, PF) baktığını söylemek, zıt komutunu ve eş anlamlılarını bulmak, makine kodunu hesaplamak.

Doğru karşılaştırma
CMP sonrasında işaretsiz sayılar için JA/JB, işaretli sayılar için JG/JL ailesini doğru seçmek; SF = OF kuralının nedenini taşma üzerinden açıklamak.

Yüksek düzey yapılar
if/else, while ve for yapılarını koşulun tersiyle atlayan kalıplarla yazmak; menzil dışı koşullu atlamayı ters koşul ve JMP ile çözmek.

Döngüler
LOOP, LOOPE/LOOPZ, LOOPNE/LOOPNZ ve JCXZ ile sayaçlı döngüler yazmak; CX = 0 tuzağını ve iç içe döngüde sayacı saklama gereğini açıklamak; örnek programları adım adım izlemek.

7.1 Sıralı yürütme ve akış kontrolü

7.1.1 IP ve önden getirme kuyruğu

8086'nın yürüteceği komutlar bellekte, kod kesiminde art arda dizilmiş baytlardır. İşlemci bu baytların hangisinin sıradaki komut olduğunu iki yazmaçla bilir: **CS** (kod kesimi) ve **IP** (komut işaretçisi, *instruction pointer*). CS kesimin başlangıcını, IP ise sıradaki komutun bu kesim içindeki bağıl konumunu tutar. Komutun fiziksel adresi, öteki bellek erişimlerinde olduğu gibi CS × 10h + IP ile bulunur. Bir .COM programında CS = 0700h ve ilk komut 0100h'de ise işlemci ilk komutu 07000h + 0100h = 07100h adresinden okur.

Programcı IP'ye MOV ile değer yazamaz; `MOV IP, AX` diye bir komut yoktur. IP'yi yalnız işlemcinin kendisi değiştirir. Olağan durumda değişiklik basittir: bir komut çözülünce IP o komutun bayt sayısı kadar artar. Üç baytlık `MOV AX, 5` (`B8 05 00`) 0100h'deyse komut bitince IP = 0103h olur. İki baytlık `MOV AL, 25` (`B0 19`) IP'yi iki artırır. Bu düzene **sıralı yürütme** denir.

Sıralı yürütmenin 8086'da bir donanım karşılığı vardır. İşlemci iki birimden oluşur. **Yol arayüz birimi** (BIU, *bus interface unit*) adres ve veri yollarını yönetir; komut baytlarını bellekten okur. **Yürütme birimi** (EU, *execution unit*) bu baytları çözer ve işlemi yapar. BIU, EU'nun işini bitirmesini beklemez. Yol boş kaldığında CS:IP'nin ilerisindeki baytları önceden okur ve altı baytlık **önden getirme kuyruğuna** (*prefetch queue*) koyar. EU sıradaki komuta geçtiğinde baytlar çoğu zaman kuyrukta hazırdır. Böylece bellekten okuma ile komut yürütme aynı anda yapılır. Bu düzen basit bir iş hattıdır (*pipeline*).

Kuyruğun bir varsayımı vardır: sıradaki komut bellekte hemen arkadan gelir. Akış kontrol komutları bu varsayımı bozar. Bir atlama yürütüldüğünde IP başka bir adrese gider. Kuyruktaki baytlar artık yanlış komutlara aittir. EU kuyruğu boşaltır ve BIU yeni CS:IP'den okumaya baştan başlar. Atlamanın saat turu bedeli büyük ölçüde bu yeniden doldurma işinden gelir.

> **Tanım 7.1**
>
> **Komut işaretçisi** (IP), yürütülecek sıradaki komutun kod kesimindeki bağıl konumunu tutan 16 bitlik yazmaçtır. Komutun fiziksel adresi CS × 10h + IP'dir. **Akış kontrol komutu** (*flow control instruction*), IP'ye (uzak atlamada CS'ye de) komut uzunluğundan farklı bir değer yükleyen komuttur.

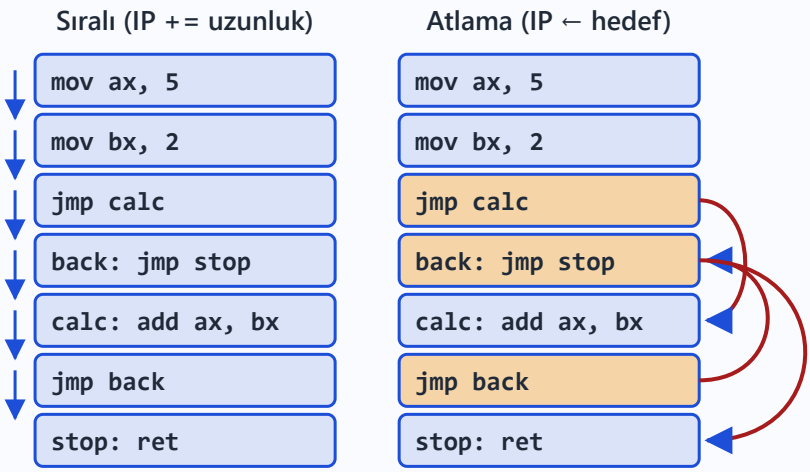
7.1.2 Akış kontrol komutlarının türleri

8086'nın akış kontrol komutları üç gruba ayrılır. Birinci grup **koşulsuz atlamadır**: `JMP etiket` her durumda atlar. İkinci grup **koşullu atlamalardır** (`Jcc`). Bunlar bayrak yazmacındaki bir ya da birkaç bayrağa bakar; koşul doğruysa atlar, değilse sıradaki komuta geçer. Koşullu atlamalar da kendi

içinde üçe bölünür. Bunlar tek bayrağı sınayanlar (JZ, JC, JS, JO, JP ve tersleri), işaretli sayılar için olanlar (JG, JL ailesi) ve işaretsiz sayılar için olanlardır (JA, JB ailesi). Üçüncü grup **döngü komutlarıdır**: `LOOP`, `LOOPE/LOOPZ`, `LOOPNE/LOOPNZ` ve `JCXZ`. Bunlar sayaç olarak CX'i kullanır.

Akışı değiştiren başka komutlar da vardır. `CALL` ile `RET` bir alt programa gidip geri dönmeyi sağlar; dönüş adresini yığında saklar. `INT` ile `IRET` kesme hizmetine gidip dönmeyi sağlar. Bu notta her programın sonunda görülen `RET` de bir akış kontrol komutudur. Bir .COM programı başladığında yığının tepesinde 0000h vardır. `RET` bu değeri IP'ye yükler. 0000h bağıl konumunda DOS'un programın başına koyduğu PSP (program kesimi öneki) bulunur. PSP'nin ilk iki baytı `INT 20h`'dir ve program böylece sona erer. İzleme tablolarındaki son satırda IP = 0000h ve SP = 0000h görülmesinin nedeni budur.

Akış kontrolünün yüksek düzeyli dillerdeki karşılığı karar yapıları (if/else, switch) ve döngülerdir (for, while, do-while). Derleyici bunların hepsini karşılaştırma, koşullu atlama ve koşulsuz atlama komutlarına çevirir. Assembly ile çalışan programcı bu çeviriyi elle yapar. Şekil 7.1, aynı program parçasının sıralı yürütmede ve atlamalı yürütmede nasıl gezildiğini karşılaştırır.



Şekil 7.1: Solda komutlar bellekteki sırasıyla yürütülür ve IP her adımda komut uzunluğu kadar artar. Sağda `jmp` komutları IP'ye yeni değer yükler; akış calc, back ve stop etiketlerine sıçrar.

7.2 Koşulsuz atlama: JMP

7.2.1 JMP ve biçimleri

`JMP etiket` komutu denetimi programın başka bir noktasına her koşulda aktarır. İleri de atlayabilir, geri de. Komut hiçbir bayrağı değiştirmez; atlamadan önceki bayrak durumu atlamadan sonra da aynen geçerlidir. Bu özellik önemlidir: bir CMP'nin ayarladığı bayraklar araya giren bir JMP'den sonra hâlâ okunabilir.

Tanım 7.2

Koşulsuz atlama (*unconditional jump*), yürütmenin bayraklara bakılmaksızın belirtilen hedef adresten sürmesidir. 8086'da `JMP` komutuyla yapılır. Hedef komutun içinde ya bir **uzaklık** (*jump displacement*) olarak, ya mutlak bir CS:IP çifti olarak ya da bir yazmaç veya bellek sözcüğünde dolaylı olarak verilir.

8086 JMP komutunun dört biçimini tanır. Hangisinin kullanılacağı hedefin ne kadar uzakta olduğuna ve nerede yazılı olduğuna bağlıdır. Tablo 7.1 bu biçimleri özetler.

Biçim	Kod	Bayt	Menzil	Değişen	Saat turu
kısa (`JMP SHORT`)	`EB rel8`	2	−128…+127	IP	15
yakın (`JMP NEAR`)	`E9 rel16`	3	kesim içi her yer	IP	15
uzak (`JMP FAR`)	`EA ofs seg`	5	1 MB'ın her yeri	CS ve IP	15
dolaylı, yazmaç (`JMP BX`)	`FF E3`	2	kesim içi	IP	11
dolaylı, bellek (`JMP [BX]`)	`FF 27`	2–4	kesim içi	IP	18 + EA

Tablo 7.1: JMP komutunun biçimleri, makine kodları ve 8086'daki süreleri.

Kısa atlama (*short jump*) iki bayttır: işlem kodu EBh ve 8 bitlik işaretli uzaklık (rel8). Uzaklık −128 ile +127 arasında olabilir. Bu aralık, atlama komutundan hemen sonraki komutun adresine göre sayılır. Döngülerin ve kısa if bloklarının çoğu bu aralığa sığar, bu yüzden kısa atlama en sık görülen biçimdir.

Yakın atlama (*near jump*) üç bayttır: E9h ve 16 bitlik uzaklık (rel16). 16 bitlik toplama sarmalı olduğundan rel16 ile kod kesiminin 64 KB'lık alanındaki her bağıl konuma (0000h–FFFFh) ulaşılır. Uzaklık işaretli yorumlandığında bu ±32 KB demektir; ama sarma sayesinde kesimin öbür ucuna da gidilebilir. Kısa ve yakın atlamada yalnız IP değişir, CS aynı kalır. Bu yüzden ikisi de **kesim içi** (*intrasegment*) atlamadır.

Uzak atlama (*far jump*) beş bayttır: EAh, ardından hedefin 16 bitlik bağıl konumu ve 16 bitlik kesim değeri. Bayt sırası little-endian'dır; önce bağıl konumun düşük baytı gelir. Örneğin 0800h:0000h adresine uzak atlama `EA 00 00 00 08` olarak kodlanır. Komut IP'ye 0000h, CS'ye 0800h yükler ve yürütme 08000h fiziksel adresinden sürer. Bu, başka bir kesime geçmenin tek yoludur ve **kesimler arası** (*intersegment*) atlama diye anılır.

Dolaylı atlamada hedef komutun içinde yazılı değildir. `JMP BX` IP'ye BX'in değerini yükler. `JMP [BX]` ise DS:BX adresindeki sözcüğü okur ve onu IP'ye yükler. Dolaylı atlama, hedefin çalışma anında hesaplandığı durumlar için uygundur. Bir menüde seçilen numaraya göre farklı bir işleve gitmek bunun tipik örneğidir. Programcı bir **atlama tablosu** kurar: `tablo dw is0, is1, is2`. Seçim numarası 2 ile çarpılıp SI'ya konur ve `JMP tablo[SI]` doğru adrese gider. Bu komutun makine kodu `FF A4` ile tablonun 16 bitlik adresinden oluşur ve dört bayttır.

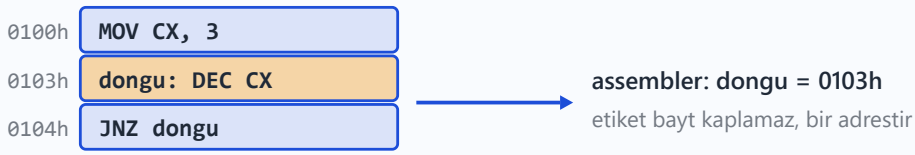
Programcının çoğu zaman biçim seçmesi gerekmez. Assembler hedefi biliyorsa ve hedef kısa menzildeyse 2 baytlık kısa biçimi kendisi seçer. Hedef kodda daha aşağıda ve henüz tanımlanmamışsa bazı assembler'lar önce 3 bayt yer ayırır. Çok geçişli assembler'lar (emu8086 bunlardan biridir) adresleri kesinleştirdikten sonra kısa biçime iner. `JMP SHORT etiket` yazımı kısa biçimi zorlar; hedef menzil dışındaysa assembler hata verir. `JMP NEAR PTR etiket` yazımı ise hedef yakında olsa bile 3 baytlık biçimi seçtirir.

> **Sık yapılan hata**
>
> "JMP en çok 127 bayt ileri gidebilir" yargısı yanlıştır. Bu sınır yalnız kısa biçim içindir. Yakın JMP kod kesiminin her yerine, uzak JMP ise 1 MB'lık adres alanının her yerine ulaşır. 127 baytlık sınır asıl olarak 8086'nın *koşullu* atlamaları için önemlidir, çünkü onların 16 bitlik biçimi yoktur.

7.2.2 Etiketler

Assembly programcısı atlama hedefini adresle değil **etiketle** (*label*) yazar. Etiket, bir adrese verilmiş addır. Tanımlamak için ad yazılır ve sonuna iki nokta konur: `dongu:`. Etiket ayrı bir satırda durabilir ya da bir komutun önüne yazılabilir: `back: jmp stop`. İki yazımda da etiket, kendisinden sonra gelen ilk komutun adresini alır.

Etiket bellekte yer kaplamaz. Assembler kaynak kodu okurken bir **konum sayacı** tutar. Her komutun bayt sayısını bu sayaca ekler ve bir etiketle karşılaşınca sayacın o anki değerini etiketin adresi olarak kaydeder. Sonra `jmp dongu` gibi bir satırı kodlarken bu adresi kullanır. Şekil 7.2'de `dongu` etiketi, kendisini izleyen `DEC CX` komutunun adresine, 0103h'ye dönüşür. Makine kodunda `dongu` diye bir şey yoktur; yalnız `JNZ` komutunun uzaklık baytında bu adrese giden bir sayı vardır.



Şekil 7.2: `dongu` etiketi, ardından gelen `DEC CX` komutunun adresine (0103h) dönüşür; `JNZ dongu` bu adrese atlar. Etiket bellekte bayt kaplamaz.

Etiket adları için birkaç kural vardır. Ad harf, rakam ve alt çizgi gibi karakterlerden oluşur, ama rakamla başlayamaz. Ad boşluk içeremez. Komut adları (`mov` , `loop`) ve yazmaç adları (`ax` , `cx`) etiket olamaz, çünkü assembler bir satırın başındaki sözcüğü önce komut ya da yazmaç olarak tanımaya çalışır. Aynı etiket bir programda bir kez tanımlanır; ikinci tanım hangi adresin kastedildiğini belirsizleştirir ve hataya yol açar.

Etiket	Durum	Neden
`dongu:`	geçerli	harflerden oluşur
`next_char:`	geçerli	alt çizgi kullanılabilir
`k1:`, `calc:`	geçerli	rakam ilk karakter değil
`1basla:`	geçersiz	rakamla başlıyor
`mov:`	geçersiz	komut adı
`iki kelime:`	geçersiz	boşluk içeriyor

Tablo 7.2: Geçerli ve geçersiz etiket örnekleri.

Etiketin bir türü de vardır. Kod içindeki etiketler yakın türdedir: yalnız bağıl konumu taşırlar ve bu yüzden kesim içi atlamalarda kullanılır. Veri tanımlarının adları (`msg db 'Hello', 0`) da birer adrestir, ama bunlara değişken denir ve veri tipleri (bayt, sözcük) vardır. Bu fark JMP'de önem kazanır. `jmp calc` yazıldığında assembler calc'ın adresine giden bir uzaklık üretir. `tablo dw 0103h` gibi bir değişkenin adı yazıldığında ise `jmp tablo`, dolaylı atlama olarak kodlanır (`FF 26` ve tablonun adresi). İşlemci tablonun içindeki sözcüğü okur ve onu IP'ye yükler. Yazım birbirine benzese de biri etikete, öbürü değişkenin içeriğine gider.

7.2.3 Kod izleme: JMP ile ileri ve geri

Aşağıdaki program JMP'nin hem ileri hem geri atlayabildiğini gösterir. Program iki sayı yükler, toplamayı yapan `calc` bloğuna ileri atlar, oradan geriye `back` satırına döner ve son olarak yeniden ileri, `stop` satırına gider. Kaynakta satırların sırası ile yürütülme sırası aynı değildir.

```
org   100h
mov   ax, 5          ; ax = 5
mov   bx, 2          ; bx = 2
jmp   calc           ; 'calc' etiketine git
back: jmp stop       ; 'stop' etiketine git
calc:
add   ax, bx         ; ax = ax + bx
jmp   back           ; 'back' etiketine dön
stop:
ret                  ; işletim sistemine dön
```

`org 100h` direktifi bir komut değildir. Assembler'a konum sayacını 0100h'den başlatmasını söyler, çünkü DOS bir .COM programını kesimin 0100h bağıl konumuna yükler. İlk komut `mov ax, 5` bu yüzden 0100h'dedir. İşlemci programa başlarken IP = 0100h'dir.

Tablo 7.3 programın her komutunu, adresini, makine kodunu ve komut bittikten sonraki durumu verir. Son sütundaki IP, sıradaki komutun adresidir.

Adım	Adres	Komut	Kod	AX	BX	IP (sonra)
0	—	başlangıç	—	0000h	0000h	0100h

Adım	Adres	Komut	Kod	AX	BX	IP (sonra)
1	0100h	`mov ax, 5`	`B8 05 00`	0005h	0000h	0103h
2	0103h	`mov bx, 2`	`BB 02 00`	0005h	0002h	0106h
3	0106h	`jmp calc`	`EB 02`	0005h	0002h	010Ah
4	010Ah	`add ax, bx`	`01 D8`	0007h	0002h	010Ch
5	010Ch	`jmp back`	`EB FA`	0007h	0002h	0108h
6	0108h	`jmp stop`	`EB 04`	0007h	0002h	010Eh
7	010Eh	`ret`	`C3`	0007h	0002h	0000h

Tablo 7.3: JMP örneğinin izlemesi (CS = 0700h; değerler komut yürütüldükten sonraki durumdur).

İlk iki adım sıralı yürütmedir: IP önce 3, sonra yine 3 artar. Üçüncü adımda `jmp calc` yürütülür. Sıralı yürütmede IP 0108h olurdu. Atlama IP'ye calc'ın adresini, 010Ah'yi yükler. Arada kalan 0108h'deki `back: jmp stop` komutu bu turda yürütülmez. Dördüncü adımda `add ax, bx` AX'i 5 + 2 = 7 yapar. ADD bayrakları da etkiler. Sonuç sıfır değildir (ZF = 0), işaret biti 0'dır (SF = 0), elde ve taşma yoktur (CF = OF = 0). 07h = 0000 0111b üç tane 1 içerdiği için eşlik tek (PF = 0).

Beşinci adım geriye atlamadır. `jmp back` 010Ch'dedir ve IP'yi 0108h'ye, yani kendinden önceki bir adrese çeker. Altıncı adımda 0108h'deki `jmp stop` nihayet yürütülür ve IP 010Eh olur. Yedinci adımda `ret` yığından 0000h değerini alır; IP = 0000h, SP = FFFEh + 2 = 0000h olur. PSP'deki `INT 20h` programı bitirir. Sonuç AX = 0007h'dir. Yürütme sırası kaynak satırlarına göre 2 → 3 → 4 → 7 → 8 → 5 → 10'dur. Üç JMP de bayraklara dokunmadığından ADD'in ayarladığı bayraklar programın sonuna kadar aynı kalır.

Tablodaki `add ax, bx` için verilen `01 D8`, ADD'in "yazmaç/bellek ← yazmaç" biçimidir. Aynı komut "yazmaç ← yazmaç/bellek" biçimiyle `03 C3` olarak da kodlanabilir ve bazı assembler'lar bunu üretir. İki kodlama da iki bayttır ve aynı işi yapar; adresler ve atlama uzaklıkları değişmez.

7.2.4 Atlama uzaklığının hesabı

Kısa ve yakın atlamanın makine kodunda hedefin adresi yazılı değildir. Onun yerine hedefin, *atlama komutundan sonraki komutun* adresine göre işaretli uzaklığı yazılıdır. Bu yaklaşıma **göreceli adresleme** (*relative addressing*) denir. Hesap şöyledir:

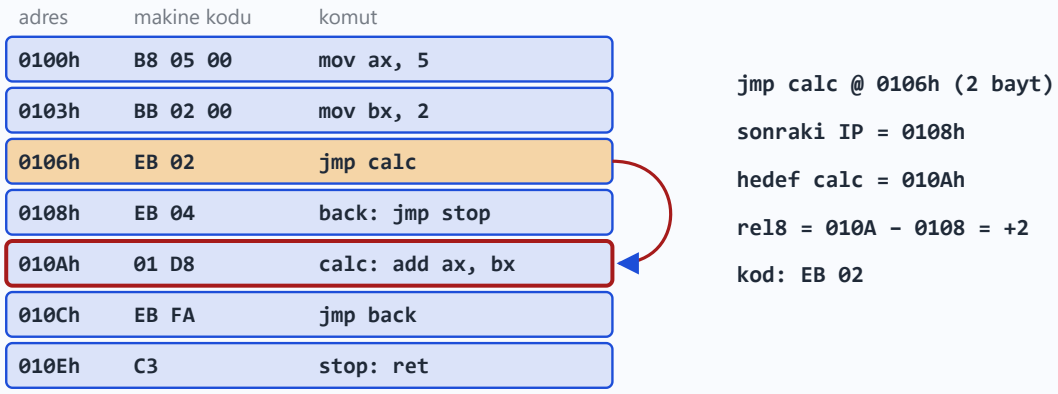
> uzaklık = hedef − (atlama komutunun adresi + atlama komutunun uzunluğu)

Parantez içindeki toplam, işlemcinin atlama komutunu okuyup bitirdiği andaki IP'dir. İşlemci atlamayı yürütürken bu IP'ye uzaklığı ekler: IP ← IP + uzaklık. Uzaklık negatifse ikiye tümleyen gösterimle saklanır ve toplama yine aynı devreyle yapılır.

Tanım 7.3

Atlama uzaklığı (*jump displacement*), atlama hedefinin, atlama komutundan sonraki komutun adresine göre işaretli farkıdır. Kısa atlamada ve 8086'nın bütün koşullu atlamalarında 8 bitlik (rel8, −128…+127), yakın atlamada 16 bitliktir (rel16). İşlemci yürütürken IP ← IP + işaret genişletilmiş uzaklık hesabını yapar.

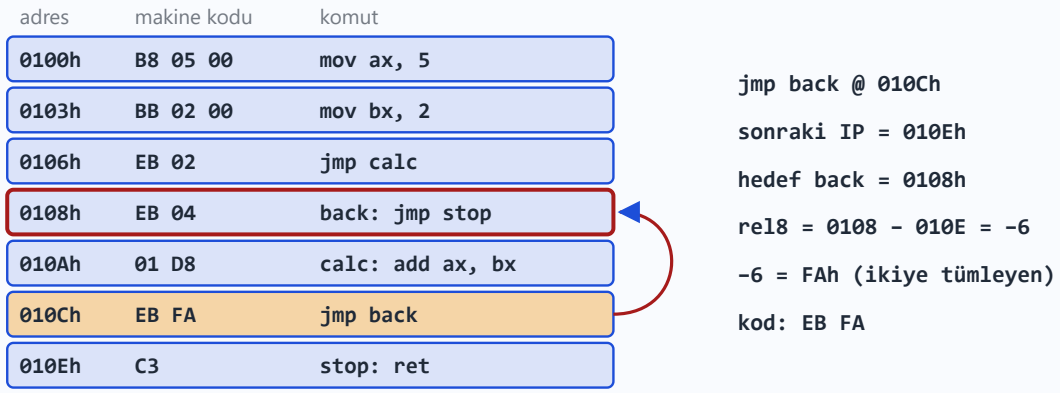
Şekil 7.3, JMP örneğindeki ilk atlamanın, `jmp calc` komutunun kodunu çıkarır. Komut 0106h'dedir ve iki bayttır. İşlemci komutu bitirdiğinde IP = 0108h'dir. Hedef calc = 010Ah'dir. Uzaklık 010Ah − 0108h = +2'dir ve makine kodu `EB 02` olur. Bu iki bayt, 0108h'deki iki baytlık `jmp stop` komutunun üzerinden atlamak demektir.



Şekil 7.3: İleri atlama: `jmp calc` 0106h'de, sonraki komut 0108h'de, hedef 010Ah'de. Uzaklık 010Ah − 0108h = +2, kod `EB 02`.

0108h'deki `jmp stop` için hesap aynıdır. Sonraki komut 010Ah'dedir, hedef stop = 010Eh'dir. Uzaklık 010Eh − 010Ah = +4, kod `EB 04`'tür. İşlemci yürütürken IP ← 010Ah + 4 = 010Eh hesabını yapar.

Geri atlamada uzaklık negatiftir. 010Ch'deki `jmp back` komutundan sonra IP = 010Eh'dir; hedef back = 0108h'dir. Uzaklık 0108h − 010Eh = −6'dır. −6'nın 8 bitlik ikiye tümleyeni şöyle bulunur: 6 = 0000 0110b; bitler ters çevrilince 1111 1001b; 1 eklenince 1111 1010b = FAh. Makine kodu `EB FA`'dır (Şekil 7.4).



Şekil 7.4: Geri atlama: `jmp back` 010Ch'de, sonraki IP 010Eh, hedef 0108h. Uzaklık −6, ikiye tümleyenle FAh; kod `EB FA`.

İşlemci bu sekiz bitlik sayıyı IP'ye eklemeden önce **işaret genişletme** yapar: en anlamlı bit 1 olduğundan üst sekiz bit 1'lerle doldurulur ve FAh, FFFAh olur. Toplama 16 bitliktir: 010Eh + FFFAh = 1 0108h. On yedinci bit yazmaca sığmaz ve atılır; IP = 0108h olur. Pozitif uzaklıkta da aynı işlem yapılır, yalnız üst bitler 0'larla dolar: 02h → 0002h.

Örnek 7.1

0150h adresindeki kısa JMP 0140h'ye atlamak istiyor. Makine kodu nedir?

Komut iki bayttır, sonraki IP = 0152h. Uzaklık = 0140h − 0152h = −12h = −18. 18 = 0001 0010b; tersi 1110 1101b; 1 eklenince 1110 1110b = EEh. Kod `EB EE`'dir. Sağlama: 0152h + FFEEh = 1 0140h → IP = 0140h.

Aynı hedefe 0100h'den gidilseydi uzaklık 0140h − 0102h = +3Eh olurdu; kod `EB 3E`. 0300h'den gidilseydi uzaklık 0140h − 0302h = −1C2h = −450 olurdu. Bu sayı −128'den küçüktür; kısa biçim kullanılamaz. Assembler yakın biçimi seçer: komut 3 bayt, sonraki IP = 0303h, uzaklık 0140h − 0303h = −1C3h = FE3Dh; kod `E9 3D FE`.

Göreceli adreslemenin önemli bir yararı vardır. Kod bellekte başka bir yere taşındığında uzaklıklar değişmez, çünkü hem atlama komutu hem hedefi aynı miktarda kayar. Böyle koda **konumdan bağımsız kod** denir. Bir .COM programını DOS hangi kesime yüklerse yüklesin, kesim içindeki kısa ve yakın atlamalar doğru çalışır. Uzak atlamada ise hedef mutlak bir kesim değeri içerir; program taşınırsa bu değerin düzeltilmesi gerekir.

7.2.5 Atlamanın donanımdaki bedeli

Tablo 7.1'de JMP'nin 15 saat turu sürdüğü görülür. Bu, iki baytlık bir komut için uzun bir süredir; iki yazmacı toplayan `ADD AX, BX` yalnız 3 tur sürer. Farkın nedeni önden getirme kuyruğudur. Atlama yürütüldüğünde EU, BIU'ya kuyruğu boşaltmasını ve yeni adresten okumaya başlamasını söyler. BIU

yeni CS:IP'yi adres yoluna çıkarır ve bir yol döngüsüyle (dört saat turu) iki bayt okur. EU, yeni komutun ilk baytı kuyruğa gelene kadar bekler. 15 turun önemli bir kısmı bu bekleyişten oluşur.

Koşullu atlamalarda bu bedel daha açık görülür. Koşul sağlanır ve atlama yapılırsa komut 16 tur sürer. Koşul sağlanmazsa yalnız 4 tur sürer, çünkü IP sıradaki komutu gösterir ve kuyruktaki baytlar geçerliliğini korur. Aynı komutun iki durumu arasında dört kat fark vardır. Bu yüzden sık yürütülen döngülerde atlamanın "çoğunlukla atlanmayan" tarafa düşecek biçimde yazılması 8086'da bile hız kazandırır.

Komut	Atlarsa	Atlamazsa
`JMP` kısa / yakın / uzak	15	—
`JMP` yazmaç / bellek	11 / 18 + EA	—
`Jcc` (bütün koşullu atlamalar)	16	4
`LOOP`	17	5
`LOOPE` / `LOOPZ`	18	6
`LOOPNE` / `LOOPNZ`	19	5
`JCXZ`	18	6

Tablo 7.4: Akış kontrol komutlarının 8086'daki saat turu sayıları (Intel 8086 kılavuzu).

Bu sayılar 5 MHz'lik bir 8086'da somut bir anlam kazanır. Bir saat turu 1 / 5 MHz = 200 ns'dir. 15 turluk bir JMP 3 µs sürer. Atlanmayan bir koşullu atlama 0,8 µs, atlanan bir koşullu atlama 3,2 µs sürer. Bin turluk bir döngünün yalnız döngü komutu bu yüzden milisaniyeler mertebesinde zaman tutabilir.

Atlamanın bir yan etkisi daha vardır. Hedef adres tek sayıysa BIU ilk yol döngüsünde yalnız bir bayt getirebilir. 8086'nın veri yolu 16 bittir ve bir sözcüğü tek yol döngüsünde ancak çift adresten okuyabilir. Sık kullanılan döngü başlarını çift adrese hizalamak (`EVEN` direktifi) bu yüzden küçük bir kazanç sağlar. 8 bitlik veri yolu olan 8088'de bu fark yoktur, ama her bayt için ayrı yol döngüsü gerektiğinden kuyruğu doldurmak daha da uzun sürer.

7.3 Koşullu atlamalar

7.3.1 Koşullu atlama fikri

Koşulsuz atlama tek başına karar veremez; her seferinde aynı yere gider. Karar için atlamanın bir koşula bağlanması gerekir. 8086'da bu koşul her zaman **bayrak yazmacındaki** bayraklardan okunur. Koşullu atlama komutlarının genel adı `Jcc`'dir; cc yerine koşulun kısaltması gelir (Z, NZ, C, A, G …). `Jcc etiket` komutunun anlamı şudur: koşul doğruysa IP ← etiket, değilse IP sıradaki komutu göstermeye devam eder.

Tanım 7.4

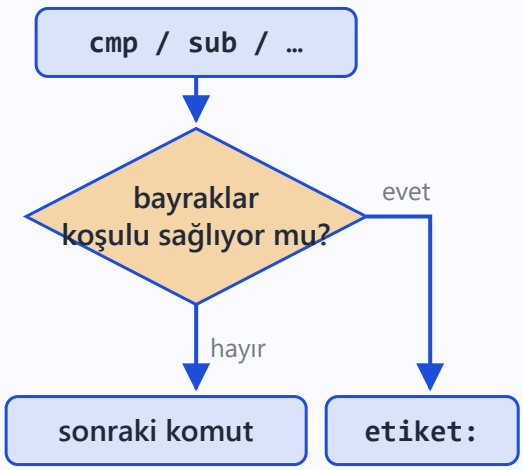
Koşullu atlama (*conditional jump*), bayrak yazmacındaki bir ya da birkaç bayrağın durumuna göre ya hedefe atlayan ya da sıradaki komuta geçen komuttur. Koşullu atlama bayrakları yalnız okur; hiçbirini değiştirmez.

Bayrakları koşullu atlamadan önce gelen bir komut ayarlar. Bu komut çoğunlukla `CMP`'dir, ama `SUB`, `ADD`, `TEST`, `AND`, `INC`, `DEC` gibi aritmetik ve mantık komutları da bayrakları ayarlar. Bayrakları ayarlayan komut ile onları okuyan atlama arasına bayrağa dokunmayan komutlar (`MOV`, `JMP`, `LOOP`, `PUSH`, `POP`) girebilir; karar yine ilk komutun sonucuna göre verilir. Araya bayrağı değiştiren bir komut girerse karar o komuta göre verilir. Bu, akış kontrolündeki hataların başlıca kaynaklarından biridir.

Atlamalarda kullanılan durum bayrakları şunlardır:

- **ZF** (sıfır bayrağı): sonuç sıfırsa 1.
- **CF** (elde bayrağı): toplamada en anlamlı bitten elde çıkarsa, çıkarmada ödünç alınırsa 1.
- **SF** (işaret bayrağı): sonucun en anlamlı biti; işaretli yorumda sonuç negatifse 1.
- **OF** (taşma bayrağı): işaretli yorumda sonuç hedefin bit sayısına sığmıyorsa 1.
- **PF** (eşlik bayrağı): sonucun düşük baytında çift sayıda 1 varsa 1.
- **AF** (yardımcı elde bayrağı): bit 3'ten elde ya da ödünç varsa 1. Bu bayrağı sınayan bir atlama yoktur; AF yalnız BCD düzeltme komutlarında (DAA, DAS, AAA, AAS) kullanılır.

Şekil 7.5 koşullu atlamanın akışını özetler. Bayrakları ayarlayan komuttan sonra işlemci bir yol ayrımına gelir. Koşul sağlanıyorsa etikete, sağlanmıyorsa sıradaki komuta gider. Yüksek düzeyli dillerdeki her if bu yol ayrımının bir örneğidir.



Şekil 7.5: Koşullu atlamanın akış şeması: bayrakları ayarlayan komuttan sonra koşul doğruysa etikete atlanır, değilse sıradaki komutla devam edilir.

7.3.2 Tek bayrağı sınayan atlamalar

En basit koşullu atlamalar tek bir bayrağa bakar. Her bayrak için iki komut vardır: biri bayrak 1 iken, öbürü bayrak 0 iken atlar. Komut adındaki N harfi "not" (değil) anlamına gelir: `JZ` sıfırsa, `JNZ` sıfır değilse atlar. Tablo 7.5 bu komutları, koşullarını ve zıtlarını verir.

Komut	Anlamı	Koşul	Zıt komut
`JZ`, `JE`	sıfırsa / eşitse (*Jump if Zero / Equal*)	ZF = 1	`JNZ`, `JNE`
`JC`, `JB`, `JNAE`	elde varsa / aşağıdaysa	CF = 1	`JNC`, `JNB`, `JAE`
`JS`	sonuç negatifse	SF = 1	`JNS`
`JO`	işaretli taşma varsa	OF = 1	`JNO`
`JPE`, `JP`	eşlik çiftse	PF = 1	`JPO`, `JNP`
`JNZ`, `JNE`	sıfır değilse / eşit değilse	ZF = 0	`JZ`, `JE`
`JNC`, `JNB`, `JAE`	elde yoksa / aşağıda değilse	CF = 0	`JC`, `JB`, `JNAE`
`JNS`	sonuç negatif değilse	SF = 0	`JS`
`JNO`	taşma yoksa	OF = 0	`JO`
`JPO`, `JNP`	eşlik tekse	PF = 0	`JPE`, `JP`

Tablo 7.5: Tek bayrağı sınayan koşullu atlamalar.

Bu komutların kullanım alanları farklıdır. `JZ/JNZ` en sık kullanılanlardır: bir sayacın sıfıra inip inmediği, iki değerin eşit olup olmadığı, bir dizginin sonundaki 0 baytına gelinip gelinmediği bunlarla sınanır. `JC/JNC` işaretsiz toplamada taşmayı, çıkarmada ödünç almayı yakalar. DOS ve BIOS hizmetlerinin bazıları da hata durumunu CF ile bildirir; hizmet çağrısından sonra `JC hata` yazılır. `JS/JNS` bir sonucun işaretine bakar. `JO/JNO` işaretli aritmetikte taşmayı yakalar. `JP/JNP` seyrek kullanılır; seri iletişimde eşlik denetimi gibi işlerde işe yarar.

> **Sık yapılan hata**
>
> JC adındaki C, elde (*carry*) bayrağıdır; taşma (*overflow*) ise OF'dir. İkisi farklı kavramlardır. CF işaretsiz yorumdaki sığmama durumunu, OF işaretli yorumdaki sığmama durumunu bildirir. 7Fh + 01h = 80h toplamasında CF = 0'dır (127 + 1 = 128 bir bayta sığar), ama OF = 1'dir (işaretli +127 + 1 = +128, −128…127 aralığına sığmaz).

7.3.3 Kodlama ve eş anlamlılar

8086'da bütün koşullu atlamalar sabit uzunluktadır: **2 bayt**. İlk bayt işlem kodu, ikinci bayt 8 bitlik işaretli uzaklıktır (rel8). Bu yüzden bir koşullu atlama, kendisinden sonraki komuta göre en çok 128 bayt geriye ya da 127 bayt ileriye gidebilir. 16 bitlik uzaklıklı koşullu atlamalar (`0F 8x` işlem kodları) ancak 80386 ile gelmiştir; 8086'da yoktur.

Aynı koşulun birden çok adı olabilir. Örneğin CF = 1 koşulu "elde var" (JC), "aşağıda" (JB, işaretsiz küçük) ve "yukarıda ya da eşit değil" (JNAE) diye üç biçimde okunabilir. Bu adlara **eş anlamlı** komutlar denir. Assembler hepsini aynı makine koduna çevirir; işlemci hangi adın yazıldığını bilmez. Programcı, koşulun o programdaki anlamına en uygun adı seçer. İki sayının eşitliği sınanıyorsa `JE`, bir sayacın sıfırlanması sınanıyorsa `JZ` yazmak kodu okunur kılar.

70 JO	**71** JNO	**72** JB / JC / JNAE	**73** JAE / JNB / JNC
74 JE / JZ	**75** JNE / JNZ	**76** JBE / JNA	**77** JA / JNBE
78 JS	**79** JNS	**7A** JP / JPE	**7B** JNP / JPO
7C JL / JNGE	**7D** JGE / JNL	**7E** JLE / JNG	**7F** JG / JNLE
E0 LOOPNE / LOOPNZ	**E1** LOOPE / LOOPZ	**E2** LOOP	**E3** JCXZ

Yukarıdaki işlem kodu dizisinde bir düzen vardır. Koşullu atlamaların hepsi 70h–7Fh aralığındadır ve ikişer ikişer eşlenir. Her çiftte işlem kodunun en düşük biti koşulu tersine çevirir: 74h JE iken 75h JNE, 72h JB iken 73h JAE, 7Ch JL iken 7Dh JGE. Üst bitler hangi bayrağın ya da bayrak birleşiminin sınanacağını seçer. İşlemcinin koşul devresi önce bu birleşimi hesaplar, sonra en düşük bit 1 ise sonucu tersler. Bu düzen, bir koşulun tersini bulmayı kolaylaştırır: işlem kodunun son bitini değiştirmek yeterlidir. Döngü komutları da E0h–E3h aralığında yan yana durur. Kısa JMP'nin kodu ise EBh'dir.

7.3.4 Kod izleme: JNC = JNB = JAE

Eş anlamlı komutların gerçekten aynı makine kodunu ürettiği küçük bir programla görülebilir. Program art arda üç eş anlamlı atlama içerir; üçü de aynı `a` etiketine gider.

```
org 100h
jnc a          ; 0100h: 73 07
jnb a          ; 0102h: 73 05
jae a          ; 0104h: 73 03
mov ax, 4      ; 0106h: B8 04 00
a: mov ax, 5   ; 0109h: B8 05 00
ret            ; 010Ch: C3
```

Üç atlamanın işlem kodu da 73h'dir; üçü de CF = 0 koşulunu sınar. Yalnız ikinci baytları farklıdır, çünkü her biri farklı adreste durur ve hedefe olan uzaklıkları farklıdır. `jnc a` 0100h'dedir: sonraki adres 0102h, uzaklık 0109h − 0102h = 7. `jnb a` 0102h'dedir: uzaklık 0109h − 0104h = 5. `jae a` 0104h'dedir: uzaklık 0109h − 0106h = 3.

Adım	Adres	Komut	CF	AX	IP (sonra)	Açıklama
0	—	başlangıç	0	0000h	0100h	.COM başlangıcı

Adım	Adres	Komut	CF	AX	IP (sonra)	Açıklama
1	0100h	`jnc a`	0	0000h	0109h	CF = 0 → atlar: 0102h + 07h
2	0109h	`mov ax, 5`	0	0005h	010Ch	aradaki üç komut yürütülmedi
3	010Ch	`ret`	0	0005h	0000h	PSP'ye dönüş

Tablo 7.6: Eş anlamlı atlamalar programının izlemesi (başlangıçta CF = 0).

Program başladığında bütün durum bayrakları 0'dır, CF de 0'dır. İlk komut `jnc a` bu yüzden atlar ve IP = 0109h olur. `jnb`, `jae` ve `mov ax, 4` hiç yürütülmez. AX = 0005h olur ve program biter. Başlangıçta CF = 1 olsaydı (örneğin programın başına `STC` eklenseydi) üç atlama da atlamazdı. İşlemci üç komutu sırayla geçer, AX önce 4, sonra 5 olurdu. İki durumda da son değer aynıdır, ama yürütülen komut sayısı farklıdır: CF = 0 iken 3, CF = 1 iken 6 komut.

7.3.5 Atlama kodunu elle hesaplamak

Sunumdaki atlama kodu hesaplayıcısı, bir atlama komutunun adresinden ve hedefinden makine kodunu bulur. Aynı hesap elle üç adımda yapılır. Önce komutun uzunluğu belirlenir: JMP SHORT, bütün Jcc'ler ve LOOP ailesi 2 bayt, JMP NEAR 3 bayttır. Sonra uzaklık hesaplanır: uzaklık = hedef − (adres + uzunluk). Son olarak uzaklık menzile sığıyor mu diye bakılır ve ikiye tümleyenle yazılır.

Örnek 7.2

(a) 0120h'deki `JNE` 0100h'ye atlayacak. Sonraki IP = 0122h; uzaklık = 0100h − 0122h = −22h. 22h = 0010 0010b; tersi 1101 1101b; artı 1: 1101 1110b = DEh. Kod `75 DE`. İşlemci 0122h + FFDEh = 1 0100h → IP = 0100h hesaplar.

(b) 0105h'deki `LOOP` 0100h'ye dönecek. Uzaklık = 0100h − 0107h = −7 = F9h. Kod `E2 F9`.

(c) 0100h'deki `JE` 0190h'ye atlayacak. Uzaklık = 0190h − 0102h = 8Eh = 142. Bu sayı +127'den büyüktür; 8 bitlik işaretli sayı olarak 8Eh, −114 diye okunurdu. Kısa biçim kullanılamaz ve 8086'da koşullu atlamanın uzun biçimi yoktur. Çözüm, koşulu ters çevirip yakın JMP kullanmaktır.

(d) 0100h'deki `JMP NEAR` 0190h'ye atlayacak. Komut 3 bayt, sonraki IP = 0103h, uzaklık = 008Dh. Kod `E9 8D 00` (little-endian: önce düşük bayt).

Kısa biçimde sonuç −128…+127 aralığına sığmıyorsa klasik assembler'lar "*relative jump out of range*" hatası verir. Bu hata, uzaklığın bir bayta sığmadığını söyler; çözümü komutu değiştirmek ya da kodu yeniden düzenlemektir.

7.4 Karşılaştırma: CMP

7.4.1 CMP komutu

Bir program çoğunlukla iki değeri karşılaştırıp sonuca göre karar verir: sayı 100'den büyük mü, karakter 'A' mı, sayaç sona geldi mi. Bunun için `CMP hedef, kaynak` komutu kullanılır. CMP, SUB komutunun yaptığı çıkarmayı yapar: hedef − kaynak. Ancak sonucu hiçbir yere yazmaz. Yalnız bayrakları (CF, ZF, SF, OF, PF, AF) çıkarmanın sonucuna göre ayarlar. İki işlenen de değişmeden kalır.

> **Tanım 7.5**
>
> **Karşılaştırma** (CMP, *compare*), hedef − kaynak çıkarmasını yapıp sonucu atan, yalnız bayrakları ayarlayan komuttur. Bayraklar SUB'daki kurallarla ayarlanır. CMP'den sonra gelen koşullu atlama bu bayraklara bakarak karar verir.

CMP'nin işlenen kuralları SUB ile aynıdır. Yazmaç–yazmaç, yazmaç–bellek, bellek–yazmaç ve yazmaç ya da bellek–sabit birleşimleri geçerlidir. İki bellek işleneni olamaz; `CMP [SI], [DI]` yazılamaz. İki işlenen aynı boyutta olmalıdır. Bellek ile sabit karşılaştırılırken boyut belirtilmelidir: `CMP BYTE PTR [SI], 3`. Değişken adı kullanılıyorsa boyut değişkenin türünden anlaşılır: `cmp dizi[si], 3` bir bayt karşılaştırmasıdır, çünkü dizi DB ile tanımlıdır.

CMP'nin yürütme süresi işlenenlere göre değişir. Yazmaç–yazmaç 3, yazmaç–sabit 4 saat turu sürer. Bellek işleneni varsa önce etkin adres hesaplanır ve bir yol döngüsüyle bellek okunur: yazmaç–bellek 9 + EA, bellek–sabit 10 + EA tur. Karşılaştırma ile ardındaki koşullu atlama birlikte bir kararın bedelidir: `CMP AL, BL` + atlanmayan `JE` 3 + 4 = 7 tur, atlanan `JE` 3 + 16 = 19 tur.

Komut	Çıkarma	ZF	CF	SF	OF	JE / JZ
`CMP 05h, 02h`	5 − 2 = 3 → 03h	0	0	0	0	atlamaz
`CMP 07h, 07h`	7 − 7 = 0 → 00h	1	0	0	0	atlar

Tablo 7.7: İki basit CMP örneği.

Tablodaki yazım kavramsaldır; 8086'da iki sabiti karşılaştıran komut yoktur, işlenenlerden biri yazmaç ya da bellek olmalıdır. Birinci örnekte fark sıfır değildir ve ZF = 0 olur. İkinci örnekte fark sıfırdır: ZF = 1 olur ve `JE` ya da `JZ` atlar. Eşitlik denetiminin tamamı bu kadardır: iki sayı eşitse farkları sıfırdır.

7.4.2 CMP bayrakları adım adım

Eşitlik yalnız ZF'ye bakar. “Büyük” ve “küçük” kararları ise sayıların nasıl yorumlandığına bağlıdır. Aynı sekiz bit işaretsiz bir sayı olarak 0–255, işaretli bir sayı olarak −128…127 aralığında bir değer taşır. CMP bu ayrımı yapmaz; bitleri çıkarır ve her iki yorum için gereken bilgiyi bayraklara yazar.

İşaretsiz yorumun cevabı CF'de ve ZF'de, işaretli yorumun cevabı SF'de, OF'de ve ZF'de durur. Tablo 7.8, `CMP AL, BL` komutunun beş farklı girişte bayrakları nasıl ayarladığını gösterir.

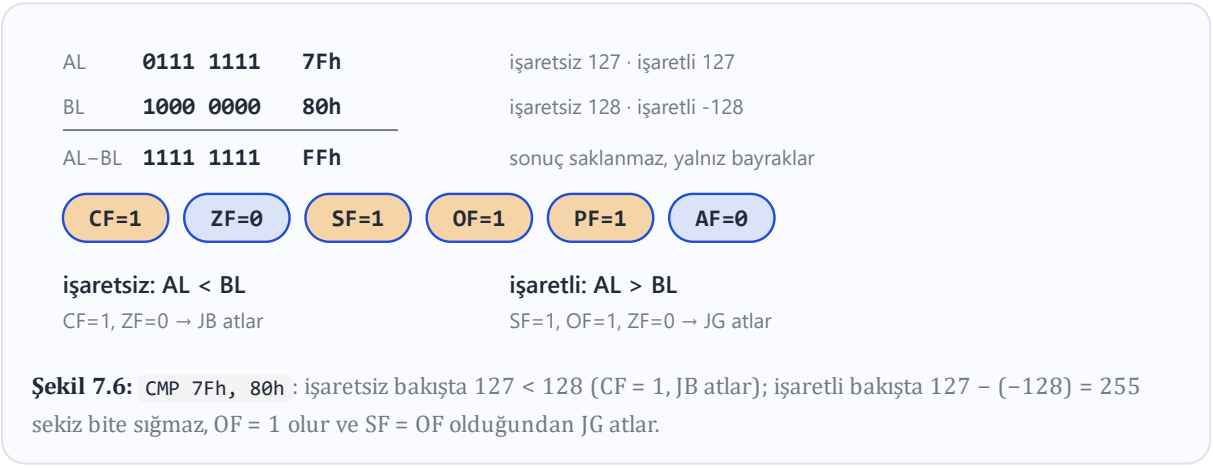
AL	BL	AL – BL	CF	ZF	SF	OF	PF	AF	İşaretsiz	İşaretli
05h (5)	02h (2)	03h	0	0	0	0	1	0	5 > 2: JA	5 > 2: JG
07h (7)	07h (7)	00h	0	1	0	0	1	0	eşit: JE	eşit: JE
05h (5)	07h (7)	FEh	1	0	1	0	0	1	5 < 7: JB	5 < 7: JL
7Fh (127)	80h (128 / −128)	FFh	1	0	1	1	1	0	127 < 128: JB	127 > −128: JG
FBh (251 / −5)	03h (3)	F8h	0	0	1	0	0	0	251 > 3: JA	−5 < 3: JL

Tablo 7.8: `CMP AL, BL` komutunun beş girişte ayarladığı bayraklar ve atlama kararları.

Birinci satırda fark pozitiftir ve PF dışında hiçbir bayrak kalkmaz; PF = 1 olur, çünkü 03h = 0000 0011b iki tane 1 içerir. İkinci satırda fark sıfırdır ve ZF = 1 olur. Hem işaretli hem işaretsiz bakışta sayılar eşittir.

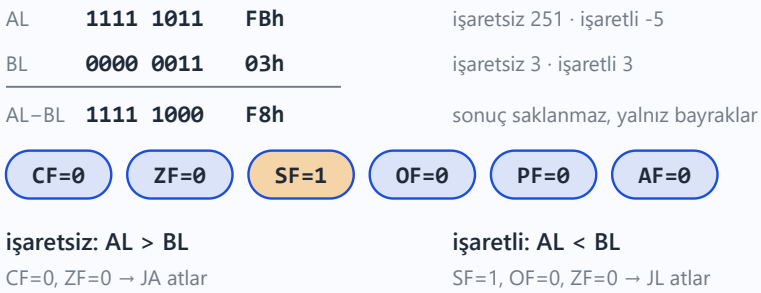
Üçüncü satırda küçük sayıdan büyük sayı çıkarılır. 5 − 7 işlemi bir ödünç gerektirir; CF = 1 olur. Sonuç FEh'dir: işaretsiz 254, işaretli −2. Sonucun en anlamlı biti 1 olduğu için SF = 1'dir. −2 sekiz bite sığdığı için taşma yoktur: OF = 0. Düşük dörtlükte de 5 − 7 için ödünç alındığından AF = 1'dir. FEh = 1111 1110b yedi tane 1 içerir; PF = 0'dır.

Dördüncü ve beşinci satırlar asıl ilginç durumlardır. Bu iki satırda işaretli ve işaretsiz yorum farklı cevap verir (Şekil 7.6 ve 7.7).



Şekil 7.6: `CMP 7Fh, 80h` : işaretsiz bakışta 127 < 128 (CF = 1, JB atlar); işaretli bakışta 127 − (−128) = 255 sekiz bite sığmaz, OF = 1 olur ve SF = OF olduğundan JG atlar.

7Fh − 80h işleminde işaretsiz yorum 127 − 128'dir; ödünç alınır ve CF = 1 olur. İşaretli yorum ise 127 − (−128) = +255'tir. Bu değer −128…127 aralığına sığmaz; OF = 1 olur. Sekiz bitlik sonuç FFh'dir ve en anlamlı biti 1 olduğundan SF = 1'dir. SF tek başına "sonuç negatif, yani AL küçük" derdi; bu yanlıştır. Doğru karar SF ile OF birlikte okunarak verilir.



Şekil 7.7: `CMP FBh, 03h` : FBh işaretsiz 251, işaretli −5'tir. İşaretsiz bakışta 251 > 3 (CF = 0, JA atlar); işaretli bakışta −5 < 3 (SF ≠ OF, JL atlar). Aynı bitler farklı karar verir.

FBh − 03h = F8h işleminde ödünç yoktur; CF = 0 olur. İşaretsiz yorumda 251 > 3'tür. İşaretli yorumda −5 − 3 = −8'dir. −8 sekiz bite sığar (F8h), taşma yoktur: OF = 0. Sonuç negatiftir: SF = 1. SF ≠ OF olduğundan işaretli yorumda AL küçüktür. Aynı CMP, verinin türüne göre iki zıt karar verir. Hangi kararın doğru olduğunu işlemci bilemez; bunu programcı, uygun atlama komutunu seçerek belirtir.

7.4.3 İşaretli sayılar için atlamalar

İşaretli karşılaştırma için kullanılan atlamalar "büyük" ve "küçük" sözcüklerinin İngilizcesiyle adlandırılır: **G**reater (büyük) ve **L**ess (küçük). Bu adlar, sayıların ikiye tümleyen gösterimde işaretli olduğu varsayımıyla kullanılır. Tablo 7.9 bu aileyi koşullarıyla verir.

Komut	Anlamı	Koşul	Zıt komut
`JE` , `JZ`	A = B	ZF = 1	`JNE` , `JNZ`
`JNE` , `JNZ`	A ≠ B	ZF = 0	`JE` , `JZ`
`JG` , `JNLE`	A > B (*greater*)	ZF = 0 ve SF = OF	`JNG` , `JLE`
`JL` , `JNGE`	A < B (*less*)	SF ≠ OF	`JNL` , `JGE`
`JGE` , `JNL`	A ≥ B	SF = OF	`JNGE` , `JL`
`JLE` , `JNG`	A ≤ B	ZF = 1 veya SF ≠ OF	`JNLE` , `JG`

Tablo 7.9: İşaretli sayılar için koşullu atlamalar (`CMP A, B` sonrası).

Koşullar bir bakışta karışık görünür, ama tek bir ilkeden türer: işaretli yorumda "A < B" bilgisi SF ≠ OF ile verilir. Geri kalanlar buradan çıkar. A ≥ B, A < B'nin değilidir: SF = OF. A > B, "A ≥ B ve A ≠ B" demektir: SF = OF ve ZF = 0. A ≤ B, A > B'nin değilidir: ZF = 1 veya SF ≠ OF. Eş anlamlılar da aynı mantıkla kurulur. JG, "küçük ya da eşit değil" anlamında JNLE diye de yazılır.

7.4.4 Neden SF = OF?

`CMP A, B` işlemcide A − B olarak hesaplanır. Taşma olmazsa sonucun işareti doğrudur: A < B ise fark negatiftir ve SF = 1 olur. Taşma varsa (OF = 1) fark sekiz ya da on altı bite sığmamıştır. Sığmayan bir sonucun işaret biti ters çıkar. Bunun nedeni işaretli sayıların bir çember üzerinde dizilmesidir (Şekil 7.8): aralığın bir ucunu aşan değer öbür uçtan girer. 127'ye 1 eklenince −128 olur; −128'den 1 çıkarılınca 127 olur.



Şekil 7.8: İşaretli bayt aralığı −128 ile 127 arasıdır. Taşma olunca değer aralığın öbür ucuna sarar ve işaret biti ters çıkar (OF = 1).

Gerçek işaret bu yüzden SF ile OF'nin dışlayan VEYA'sıdır: gerçek işaret = SF XOR OF. Taşma yoksa (OF = 0) gerçek işaret SF'nin kendisidir. Taşma varsa (OF = 1) gerçek işaret SF'nin tersidir. SF ≠ OF olduğunda XOR 1 verir: gerçek fark negatiftir ve A < B'dir. SF = OF olduğunda XOR 0 verir: gerçek fark negatif değildir ve A ≥ B'dir.

Teorem 7.1 (İşaretli karşılaştırma)

`CMP A, B` komutundan sonra, A ve B ikiye tümleyen gösterimde işaretli sayılar olarak yorumlanırsa A < B ⇔ SF ≠ OF ve A ≥ B ⇔ SF = OF'dir.

Gerekçe. Gerçek fark d = A − B, sekiz bitte −255…+255 aralığındadır. d −128…127 aralığındaysa sonuç doğru gösterilir, OF = 0 ve SF = (d < 0) olur. d > 127 ise sonuç d − 256 olarak görünür, negatiftir: SF = 1, OF = 1; gerçek fark pozitiftir. d < −128 ise sonuç d + 256 olarak görünür, pozitif ya da sıfırdır: SF = 0, OF = 1; gerçek fark negatiftir. Üç durumda da (d < 0) = SF XOR OF'dir.

A	B	A − B (8 bit)	Gerçek fark	SF	OF	Karar
5	7	FEh (−2)	−2	1	0	SF ≠ OF → 5 < 7
127 (7Fh)	−128 (80h)	FFh (−1)	+255, sığmaz	1	1	SF = OF → 127 > −128
−128 (80h)	1	7Fh (+127)	−129, sığmaz	0	1	SF ≠ OF → −128 < 1
−5 (FBh)	3	F8h (−8)	−8	1	0	SF ≠ OF → −5 < 3

Tablo 7.10: SF ve OF birlikte doğru kararı verir (8 bit).

Tablonun ikinci ve üçüncü satırları taşmalıdır. İkisinde de SF tek başına yanlış cevap verirdi: ikinci satırda “negatif, A küçük” derdi, üçüncü satırda “pozitif, A büyük” derdi. OF ile birlikte değerlendirildiğinde karar doğrudur. Donanım açısından bu iş çok ucuzdur: JL'nin koşul devresi tek bir XOR kapısıdır. JLE için bu XOR'un çıkışı ZF ile VEYA'lanır.

Sık yapılan hata

İşaretli karşılaştırmadan sonra `JS` ile "küçükse atla" yazmak. JS yalnız SF'ye bakar ve taşma olduğunda yanlış karar verir: `MOV AL, 80h` / `CMP AL, 1` sonrasında SF = 0'dır ve JS atlamaz, oysa −128 < 1'dir. Doğru komut `JL` 'dir; SF ≠ OF (0 ≠ 1) olduğundan atlar.

7.4.5 İşaretsiz sayılar için atlamalar

İşaretsiz karşılaştırma için kullanılan atlamalar "yukarıda" ve "aşağıda" sözcüklerinin İngilizcesiyle adlandırılır: **A**bove ve **B**elow. İşaretsiz yorumda karar çok daha basittir, çünkü taşma kavramı yoktur. A − B çıkarmasında A < B ise ödünç alınır ve CF = 1 olur. Demek ki "aşağıda" testi doğrudan elde bayrağıdır: JB ≡ JC.

Komut	Anlamı	Koşul	Zıt komut
`JE`, `JZ`	A = B	ZF = 1	`JNE`, `JNZ`
`JNE`, `JNZ`	A ≠ B	ZF = 0	`JE`, `JZ`
`JA`, `JNBE`	A > B (*above*)	CF = 0 ve ZF = 0	`JNA`, `JBE`
`JB`, `JNAE`, `JC`	A < B (*below*)	CF = 1	`JNB`, `JAE`, `JNC`
`JAE`, `JNB`, `JNC`	A ≥ B	CF = 0	`JNAE`, `JB`, `JC`
`JBE`, `JNA`	A ≤ B	CF = 1 veya ZF = 1	`JNBE`, `JA`

Tablo 7.11: İşaretsiz sayılar için koşullu atlamalar (`CMP A, B` sonrası).

JA'nın koşulu "ödünç yok ve fark sıfır değil" demektir; A hem B'den küçük değildir hem de ona eşit değildir. JBE bunun tersidir. JAE yalnız CF = 0'a bakar; eşitlikte de ödünç alınmadığı için "büyük ya da eşit" kendiliğinden karşılanır.

7.4.6 Above/Below mı, Greater/Less mi?

Altı karşılaştırma ilişkisinin her biri için iki komut vardır: biri işaretsiz, biri işaretli. Eşitlik ve eşitsizlikte iki aile aynı komutu kullanır, çünkü eşitlik bitlerin aynı olmasıdır ve yoruma bağlı değildir. Tablo 7.12 bu eşlemeyi tek bakışta verir.

Karşılaştırma	İşaretsiz (A/B)	İşaretli (G/L)
A = B	`JE` / `JZ`	`JE` / `JZ`
A ≠ B	`JNE` / `JNZ`	`JNE` / `JNZ`
A > B	`JA` / `JNBE`	`JG` / `JNLE`
A ≥ B	`JAE` / `JNB`	`JGE` / `JNL`
A < B	`JB` / `JNAE`	`JL` / `JNGE`

Karşılaştırma	İşaretsiz (A/B)	İşaretli (G/L)
A ≤ B	`JBE` / `JNA`	`JLE` / `JNG`

Tablo 7.12: İlişkiye ve veri tipine göre atlama seçimi.

Seçimi verinin türü belirler. Adresler, sayaçlar, dizi indisleri, uzunluklar ve ASCII kodları işaretsizdir; bunlarla A/B ailesi kullanılır. Sıcaklık, iki ölçüm arasındaki fark, bir konum değişimi gibi negatif olabilen değerler işaretlidir; bunlarla G/L ailesi kullanılır. Kolay bir hatırlatıcı şudur: **A**bove / **B**elow işaretsiz, **G**reater / **L**ess işaretli. Değerler 0–127 aralığında kaldığı sürece iki aile aynı sonucu verir; fark ancak en anlamlı biti 1 olan değerlerde ortaya çıkar. Bu yüzden yanlış aile seçimi küçük deneme verileriyle çoğu zaman fark edilmez.

7.4.7 Kod izleme: aynı bitler, JA ve JG

Aşağıdaki program aynı CMP'nin ardından JA ile JG'nin farklı karar verdiğini gösterir. Programın ikinci yarısı da taşmalı bir karşılaştırmada JG'nin doğru çalıştığını sınar. CL, CH ve DL yazmaçları işaret olarak kullanılır: bir atlama atlamazsa ilgili yazmaç 1 olur.

```
MOV  AL, 0FBh    ; 251 ya da -5
MOV  BL, 03h
CMP  AL, BL      ; FBh - 03h
JA   a1          ; işaretsiz: 251 > 3 ?
MOV  CL, 1       ; JA atlamasaydı
a1:  JG  g1      ; işaretli: -5 > 3 ?
MOV  CH, 1       ; JG atlamadı
g1:  MOV AL, 7Fh ; 127
CMP  AL, 80h     ; 127 - (-128)
JG   g2          ; SF = OF ?
MOV  DL, 1       ; JG atlamasaydı
g2:  RET
```

Kaynakta `0FBh` yazımına dikkat edilmelidir. Onaltılık sayı bir harfle başlıyorsa önüne 0 konur. `FBh` yazılırsa assembler onu bir ad sanar ve hata verir.

Adım	Adres	Komut	Kod	AL	CF ZF SF OF	Sonuç
1	0100h	`MOV AL, 0FBh`	`B0 FB`	FBh	0 0 0 0	—
2	0102h	`MOV BL, 03h`	`B3 03`	FBh	0 0 0 0	BL = 03h
3	0104h	`CMP AL, BL`	`3A C3`	FBh	0 0 1 0	F8h, saklanmaz
4	0106h	`JA a1`	`77 02`	FBh	0 0 1 0	atlar → 010Ah
5	010Ah	`JG g1`	`7F 02`	FBh	0 0 1 0	atlamaz → 010Ch
6	010Ch	`MOV CH, 1`	`B5 01`	FBh	0 0 1 0	CX = 0100h
7	010Eh	`MOV AL, 7Fh`	`B0 7F`	7Fh	0 0 1 0	—
8	0110h	`CMP AL, 80h`	`3C 80`	7Fh	1 0 1 1	FFh, saklanmaz

Adım	Adres	Komut	Kod	AL	CF ZF SF OF	Sonuç
9	0112h	`JG g2`	`7F 02`	7Fh	1 0 1 1	atlar → 0116h
10	0116h	`RET`	`C3`	7Fh	1 0 1 1	IP = 0000h

Tablo 7.13: JA ve JG programının izlemesi (CX ve DX başlangıçta 0000h).

Üçüncü adımda CMP, FBh − 03h = F8h hesaplar. Ödünç yoktur (CF = 0), sonuç sıfır değildir (ZF = 0), en anlamlı bit 1'dir (SF = 1), taşma yoktur (OF = 0). Dördüncü adımda `JA` CF = 0 ve ZF = 0 koşulunu sağlanmış bulur ve atlar; işaretsiz yorumda 251 > 3'tür. `MOV CL, 1` satırı atlanır ve CL = 0 kalır. Beşinci adımda `JG` aynı bayraklara bakar. ZF = 0'dır ama SF = 1, OF = 0'dır; SF ≠ OF olduğundan atlamaz. İşaretli yorumda −5 > 3 yanlıştır. Altıncı adımda CH = 1 olur ve CX = 0100h'dir. Aynı CMP iki farklı karar vermiştir.

Programın ikinci yarısında AL = 7Fh = 127 yapılır ve 80h ile karşılaştırılır. 7Fh − 80h = FFh'dir. İşaretsiz yorumda 127 < 128'dir ve ödünç alınır: CF = 1. İşaretli yorumda 127 − (−128) = 255 sığmaz: OF = 1. Sonucun işaret biti SF = 1'dir. PF = 1'dir, çünkü FFh sekiz tane 1 içerir. Dokuzuncu adımda `JG`, ZF = 0 ve SF = OF (1 = 1) koşulunu sağlanmış bulur ve atlar. 127 > −128 doğrudur. SF tek başına "küçük" derdi. DL = 0 kalır. Program bitince CL = 0, CH = 1, DL = 0'dır: işaretli yorumda −5, 3'ten büyük değildir; 127 ise −128'den büyüktür.



Şekil 7.9: JA ve JG izlemesinin dokuzuncu adımı: `CMP AL, 80h` sonrasında CF = 1, SF = 1, OF = 1. ZF = 0 ve SF = OF olduğundan JG atlar.

7.4.8 Kod izleme: CMP + JE ve CMP + JA

Gerçek programlarda karşılaştırmanın iki sonucu için iki ayrı blok bulunur. Aşağıdaki iki kısa parça bu kalıbın en basit hâlidir. Parçalar `RET` içermez, çünkü bir programın içine yerleştirilmek üzere yazılmıştır. Bloklardaki işlemler yerine yalnız açıklama yazılıdır.

```
mov ax, 5
mov bx, 5
cmp ax, bx
je  equal_message
jmp not_equal_message
equal_message:          ; eşitse
  jmp end_program
not_equal_message:      ; eşit değilse
end_program:
```

Assembler bu parçayı 0100h'den yerleştirdiğinde `cmp ax, bx` 0106h'de (`3B C3`), `je` 0108h'de (`74 02`), `jmp not_equal_message` 010Ah'de (`EB 02`), `jmp end_program` 010Ch'de (`EB 00`) durur. İzleme kısadır:

1. `mov ax, 5` ve `mov bx, 5`: AX = BX = 0005h.
2. `cmp ax, bx`: 5 − 5 = 0 → ZF = 1, CF = 0, SF = 0, OF = 0. PF = 1 olur (00h'de sıfır tane 1 vardır ve sıfır çifttir). AX değişmez.
3. `je equal_message`: ZF = 1 olduğundan atlar; IP = 010Ch.
4. `equal_message` bloğundaki işlemler yapılır, ardından `jmp end_program` öbür bloğun üzerinden atlar.

Son adımdaki JMP kritik önemdedir. Assembly'de bloklar yalnız etiketlerle ayrılır; bir bloğun bittiğini işlemciye söyleyen bir yapı yoktur. `jmp end_program` olmasaydı işlemci eşitlik bloğunu bitirince sıradaki satıra, yani eşitsizlik bloğuna geçerdi ve iki blok da yürütülürdü. Bu örnekte iki blok boş olduğundan `jmp end_program` kendinden hemen sonraki adrese gider ve kodu `EB 00` olur: uzaklık sıfırdır. Bloklar dolu olduğunda uzaklık blok uzunluğu kadar büyür. Sayılar eşit olmasaydı `je` atlamazdı; `jmp not_equal_message` akışı eşitsizlik bloğuna götürürdü.

İkinci parça aynı kalıbı işaretsiz "büyük mü" sorusuyla kurar:

```
mov ax, 5
mov bx, 7
cmp ax, bx
ja  jump_above
jmp not_jump_above
jump_above:             ; ax büyükse yapılacaklar
  jmp end_program
not_jump_above:         ; ax küçük veya eşitse
end_program:
```

Burada `cmp ax, bx` 5 − 7 = FFFEh hesaplar; işaretli yorumda −2'dir. Ödünç alındığından CF = 1, sonuç sıfır olmadığından ZF = 0, en anlamlı bit 1 olduğundan SF = 1'dir. Taşma yoktur: OF = 0. Düşük dörtlükte 5 − 7 için ödünç alındığından AF = 1 olur. `ja` için CF = 0 ve ZF = 0 gerekir; CF = 1 olduğun-

dan atlamaz. `jmp not_jump_above` akışı "küçük veya eşit" bloğuna götürür. Aynı kodda `ja` yerine `jg` yazılsaydı da sonuç değişmezdi: SF = 1, OF = 0, SF ≠ OF olduğundan JG de atlamazdı. İki değer de pozitif ve küçük olduğu için iki aile aynı cevabı verir.

7.4.9 Kod izleme: karşılaştır ve yaz

Aşağıdaki program iki sayıyı karşılaştırır. Sayılar eşitse ekrana `y`, değilse `n` yazar. Karakteri DOS'un `INT 21h` hizmetiyle yazar: AH = 02h iken DOS, DL'deki karakteri ekrana basar. Programın yapısı önemlidir: iki dal yalnız DL'ye farklı bir değer koyar ve sonra ortak `yaz` etiketinde birleşir. Yazma kodu iki kez yazılmaz.

```
org     100h
mov     al, 25          ; al = 25
mov     bl, 10          ; bl = 10
cmp     al, bl          ; al - bl karşılaştır
je      equal           ; al = bl ise (zf = 1) atla
mov     dl, 'n'         ; al <> bl: 'n' yazılacak
jmp     yaz             ; ortak yazma satırına git
equal:
mov     dl, 'y'         ; al = bl: 'y' yazılacak
yaz:
mov     ah, 2           ; DOS: DL'deki karakteri yaz
int     21h
ret                     ; DOS'a dön
```

Adım	Adres	Komut	Kod	AX	DL	ZF CF	Ekran
1	0100h	`mov al, 25`	`B0 19`	0019h	00h	0 0	
2	0102h	`mov bl, 10`	`B3 0A`	0019h	00h	0 0	
3	0104h	`cmp al, bl`	`3A C3`	0019h	00h	0 0	
4	0106h	`je equal`	`74 04`	0019h	00h	0 0	
5	0108h	`mov dl, 'n'`	`B2 6E`	0019h	6Eh	0 0	
6	010Ah	`jmp yaz`	`EB 02`	0019h	6Eh	0 0	
7	010Eh	`mov ah, 2`	`B4 02`	0219h	6Eh	0 0	
8	0110h	`int 21h`	`CD 21`	026Eh	6Eh	0 0	n
9	0112h	`ret`	`C3`	026Eh	6Eh	0 0	n

Tablo 7.14: Karşılaştır ve yaz programının izlemesi.

AL = 25 = 19h ve BL = 10 = 0Ah yüklenir. `cmp al, bl` 25 − 10 = 15 = 0Fh hesaplar; ZF = 0 olur. Düşük dörtlükte 9 − A için ödünç alındığından AF = 1, 0Fh dört tane 1 içerdiğinden PF = 1'dir. `je equal` atlamaz. `mov dl, 'n'` DL'ye 'n' karakterinin ASCII kodunu, 6Eh'yi koyar. `jmp yaz` 'y' satırını atlar; uzaklığı 2'dir, çünkü atlanan `mov dl, 'y'` iki bayttır. İki dal `yaz` etiketinde birleşir.

`mov ah, 2` AH'yi 02h yapar; AX = 0219h olur. `int 21h` ekrana `n` yazar. DOS bu hizmetten dönerken yazdığı karakteri AL'ye de koyar: AL = 6Eh, AX = 026Eh. Programın sonundaki `ret` zorunludur. O olmasaydı işlemci `int 21h`'den sonraki baytları komut sanıp yürütmeye devam ederdi; bu baytlar ne olduğu belli olmayan bellek içeriğidir.

Programda `mov dl, 'y'` hiç yürütülmez, ama yine bellekte yer kaplar. Sayılar eşit olsaydı (örneğin `mov bl, 25`) `je equal` atlar, DL = 79h ('y') olur, `jmp yaz` ise hiç yürütülmezdi. Karşılaştırma bloklarında her zaman bir dal "sıradan düşen" daldır, öbürü atlamayla gidilen daldır.

7.5 Yüksek düzey yapıların karşılıkları

7.5.1 if/else kalıbı

Yüksek düzeyli bir dildeki `if (koşul) A else B` yapısı assembly'de iki atlamayla kurulur. Akla ilk gelen yol koşulun kendisiyle atlamaktır: koşul doğruysa A bloğuna git. Ama bu yolda kodun sırası ters döner; koşul sağlanmadığında yürütülecek B bloğu, atlama komutunun hemen altına yazılmak zorunda kalır. Daha düzenli olan yol **koşulun tersiyle** atlamaktır. Koşul sağlanmıyorsa A bloğunun üzerinden B bloğuna atlanır. Koşul sağlanıyorsa atlama yapılmaz ve işlemci sıradan düşerek A bloğuna girer. Böylece assembly kodu, yüksek düzeyli koddaki A–B sırasını korur.

```
if (ax == bx) {
    ; A bloğu
} else {
    ; B bloğu
}
```

```
    cmp  ax, bx
    jne  degil       ; koşulun TERSİ
    ; A bloğu
    jmp  son
degil:
    ; B bloğu
son:
```

Kalıbın iki parçası vardır. Birincisi koşulun tersiyle yapılan atlamadır: `==` için `JNE`, `<` için `JGE` ya da `JAE`, `>` için `JLE` ya da `JBE`. İkincisi A bloğunun sonundaki `jmp son`'dur. Bu atlama unutulursa A bloğu bittiğinde işlemci B bloğuna da girer. Else kısmı olmayan bir if yapısında ikinci atlamaya gerek yoktur; ters koşulla doğrudan blok sonuna atlanır.

Yüksek düzey koşul	İşaretsiz veride atla	İşaretli veride atla
`a == b`	`JNE`	`JNE`
`a != b`	`JE`	`JE`

Yüksek düzey koşul	İşaretsiz veride atla	İşaretli veride atla
`a < b`	`JAE`	`JGE`
`a <= b`	`JA`	`JG`
`a > b`	`JBE`	`JLE`
`a >= b`	`JB`	`JL`

Tablo 7.15: if koşulları ve blok atlama için kullanılan ters atlamalar.

Bileşik koşullar da aynı yolla kurulur. `if (a >= -10 && a <= 10)` gibi bir VE koşulunda her alt koşul ayrı bir CMP ve ters atlamayla sınanır. Alt koşullardan biri sağlanmadığında doğrudan blok dışına atlanır. VEYA koşulunda ise alt koşullardan biri sağlandığında doğrudan bloğun içine atlanır. Bu, yüksek düzeyli dillerdeki "kısa devre" değerlendirmesinin makine düzeyindeki karşılığıdır: sonucu belli olan bir koşulun geri kalanı hiç sınanmaz.

7.5.2 while, do-while ve for kalıpları

Döngü, bir atlamanın geriye doğru yapılmasıdır. `while (koşul) gövde` yapısında koşul gövdeden önce sınanır; koşul baştan yanlışsa gövde hiç çalışmaz. Assembly karşılığında döngünün başına bir etiket konur, koşul ters atlamayla sınanır, gövdenin sonunda başa dönülür:

```
while (cx > 0) {     ; işaretsiz
    ; gövde
}
```

```
bas:  cmp  cx, 0
      jbe  bitti     ; cx <= 0 ise çık
      ; gövde
      jmp  bas
bitti:
```

`do { gövde } while (koşul)` yapısında koşul gövdeden sonra sınanır; gövde en az bir kez çalışır. Karşılığı daha kısadır, çünkü tek bir koşullu atlama hem sınamayı hem geri dönüşü yapar:

```
bas:  ; gövde
      cmp  cx, 0
      ja   bas       ; koşul doğruysa başa dön
```

Bu ikinci biçim her turda bir atlama daha az yürütür. while biçiminde her tur bir koşullu atlama (atlamaz, 4 tur) ve bir JMP (15 tur) içerir. do-while biçiminde her tur yalnız bir koşullu atlama (atlar, 16 tur) içerir. Bu yüzden derleyiciler while döngüsünü çoğunlukla "önce bir kez sına, sonra do-while olarak dön" biçimine çevirir.

`for (i = n; i > 0; i--)` gibi bir sayaçlı döngü, 8086'da en doğal biçimde CX ve `LOOP` komutuyla yazılır. LOOP, CX'i azaltma, sıfırla karşılaştırma ve geri atlama işlerini tek komutta yapar. Sayaç yukarı doğru sayılıyorsa (`for (i = 0; i < n; i++)`) sayaç başka bir yazmaçta tutulur. Her turda `INC` ile artırılır; `CMP` ve ardından `JB` ya da `JNE` ile sınanır.

7.5.3 Kod izleme: if/else ve while

Aşağıdaki program iki kalıbı arka arkaya kullanır. Önce `if (AX == BX) DX = 1; else DX = 2;` yapısı yürütülür. Ardından CX = 2 ile `while (CX > 0) { DX += CX; CX--; }` döngüsü çalışır.

```
MOV  AX, 12
MOV  BX, 30
CMP  AX, BX
JNE  degil       ; koşulun TERSİ
MOV  DX, 1       ; A bloğu: AX = BX
JMP  son
degil: MOV DX, 2 ; B bloğu: AX ≠ BX
son: MOV CX, 2   ; while (CX > 0)
bas: CMP CX, 0
JBE  bitti       ; CX ≤ 0 → çık
ADD  DX, CX      ; gövde
DEC  CX
JMP  bas
bitti: RET
```

Adım	Adres	Komut	Kod	CX	DX	ZF CF	Karar
1–2	0100 h	`MOV AX, 12` / `MOV BX, 30`	`B8 0C 00` / `BB 1E 00`	0000 h	0000 h	0 0	—
3	0106 h	`CMP AX, BX`	`3B C3`	0000 h	0000 h	0 1	12 − 30: eşit değil, ödünç
4	0108 h	`JNE degil`	`75 05`	0000 h	0000 h	0 1	atlar → 010Fh
5	010F h	`MOV DX, 2`	`BA 02 00`	0000 h	0002 h	0 1	B bloğu
6	0112 h	`MOV CX, 2`	`B9 02 00`	0002 h	0002 h	0 1	—
7–8	0115 h	`CMP CX, 0` / `JBE bitti`	`83 F9 00` / `76 05`	0002 h	0002 h	0 0	2 > 0: atlamaz
9–10	011A h	`ADD DX, CX` / `DEC CX`	`03 D1` / `49`	0001 h	0004 h	0 0	gövde, tur 1
11–13	011D h	`JMP bas` / `CMP` / `JBE`	`EB F6`	0001 h	0004 h	0 0	1 > 0: atlamaz
14–15	011A h	`ADD DX, CX` / `DEC CX`	—	0000 h	0005 h	1 0	gövde, tur 2

Adım	Adres	Komut	Kod	CX	DX	ZF CF	Karar
16–18	011Dh	`JMP bas` / `CMP` / `JBE`	—	0000h	0005h	1 0	0 = 0: atlar → 011Fh
19	011Fh	`RET`	`C3`	0000h	0005h	1 0	IP = 0000h

Tablo 7.16: if/else ve while programının izlemesi (AX = 000Ch, BX = 001Eh boyunca değişmez).

Üçüncü adımda `CMP AX, BX` 12 − 30 = −18 hesaplar; 16 bitlik sonuç FFEEh'dir. Sonuç sıfır değildir (ZF = 0), ödünç alınmıştır (CF = 1), işaret biti 1'dir (SF = 1). Dördüncü adımda `JNE degil` koşulun tersini sağlanmış bulur ve A bloğunun üzerinden atlar. `MOV DX, 1` ve `JMP son` yürütülmez. Beşinci adımda B bloğu DX = 2 yapar. B bloğu son bloktur ve doğrudan `son` etiketine düşer; bu yolda JMP gerekmez.

Altıncı adımda CX = 2 olur ve döngü başlar. `CMP CX, 0` 2 − 0 = 2 hesaplar: CF = 0, ZF = 0. `JBE bitti` için CF = 1 ya da ZF = 1 gerekir; atlamaz ve gövdeye girilir. Gövde DX = 2 + 2 = 4, CX = 1 yapar. `JMP bas` başa döner ve koşul yeniden sınanır: 1 > 0, atlamaz. İkinci turda DX = 4 + 1 = 5, CX = 0 olur. Bu turda `DEC CX` sonucu sıfırladığı için ZF = 1'dir, ama karar yine başta verilir. `CMP CX, 0` 0 − 0 = 0 hesaplar: ZF = 1, CF = 0. `JBE bitti` atlar. Sonuç DX = 5'tir: B bloğunun 2'si, artı 2, artı 1.

`CMP CX, 0` komutunun kodu `83 F9 00`'dır. 83h işlem kodu, 16 bitlik bir işleneni işaret genişletilmiş 8 bitlik bir sabitle karşılaştırır; böylece sabit 2 yerine 1 bayt tutar. Döngünün sonundaki `JMP bas` 011Dh'dedir ve iki bayttır: uzaklık 0115h − 011Fh = −0Ah = F6h, kod `EB F6`.

Bu döngü bir while döngüsüdür; koşul gövdeden önce sınanır. CX = 0 ile başlansaydı gövde hiç çalışmazdı. LOOP komutu ise koşulu gövdeden sonra sınar; iki yapı arasındaki önemli fark budur.

Tablo 7.16'daki ilk satırlarda CX = 0000h gösterilmiştir. Sunumdaki izleme de bu varsayımla yapılmıştır. Gerçekte emu8086, bir .COM programını yüklerken CX'e programın bayt sayısını yükler. Bu program 32 bayttır (0100h–011Fh), dolayısıyla emu8086'da program CX = 0020h ile başlar. Sonuç etkilenmez, çünkü CX ilk kez okunmadan önce `MOV CX, 2` ile yazılır. Genel kural şudur: başlangıçtaki yazmaç değerlerine güvenilmez; kullanılacak her yazmaç önce bir değerle yüklenir.

7.5.4 Menzil dışı koşullu atlama

8086'da koşullu atlamalar yalnız kısa biçimdedir: sonraki komuta göre −128…+127 bayt. Uzun bir if bloğunun üzerinden atlamak ya da uzun bir döngünün başına dönmek gerektiğinde bu menzil yetmeyebilir (Şekil 7.10).



Şekil 7.10: Koşullu atlama sonraki IP etrafında −128 ile +127 bayt arasına ulaşır. 200 bayt ötedeki hedef bu

pencerenin dışındadır; yakın JMP ise kesim içinde her yere gider.

Çözüm, koşulu ters çevirip kısa bir atlamayla hemen sonraki komutun üzerinden geçmek, uzak hedefe ise yakın JMP ile gitmektir:

```
; hatalı: hedef çok uzak
cmp al, 0
je  cok_uzak        ; out of range!
; ... 200 baytlık kod ...
cok_uzak:
```

```
; doğru: ters koşul + yakın JMP
cmp al, 0
jne devam           ; ters koşul, kısa
jmp cok_uzak        ; near JMP (E9)
devam:
; ... 200 baytlık kod ...
cok_uzak:
```

İkinci yazımda `jne devam` yalnız 3 bayt ileri atlar ve kısa menzile rahatça sığar. AL = 0 ise JNE atlamaz ve işlemci yakın JMP'ye düşer; o da kesim içinde her yere ulaşır. AL ≠ 0 ise JNE, JMP'nin üzerinden `devam` etiketine geçer. Davranış ilk yazımla aynıdır, yalnız komut çifti 2 yerine 5 bayttır.

Bu dönüşüm her koşullu atlamaya uygulanabilir, çünkü her koşulun bir tersi vardır ve işlem kodunun son biti değiştirilerek bulunur. Klasik assembler'lar (örneğin 8086 kipindeki MASM) menzil dışı bir Jcc görünce "*relative jump out of range*" hatası verir ve dönüşümü programcıya bırakır. emu8086'nın assembler'ı ise aynı dönüşümü kendisi yapar ve bir uyarıyla bildirir. Örneğin 0102h'deki `je uzak`, hedef 01CFh'de iken `75 03 E9 C8 00` olarak kodlanır: `JNE +3` (JMP'nin üç baytının üzerinden geçiş) ve `JMP 01CFh` (0107h + 00C8h = 01CFh). Hangi assembler kullanılırsa kullanılsın makine düzeyinde olan aynıdır. Programın uzunluğu da bu yüzden kaynakta göründüğünden 3 bayt fazla olur.

Sık yapılan hata

Menzil hatasını `JE` yerine eş anlamlısı `JZ` yazarak ya da `JE SHORT` ile düzeltmeye çalışmak. Eş anlamlı komut aynı makine koduna çevrilir ve menzili aynıdır. SHORT ise menzili büyütmez, yalnız kısa biçimi zorlar. `LOOPE` gibi bir döngü komutu da çözüm değildir: o da 2 bayt ve rel8'dir, üstelik CX'i değiştirir.

7.6 Döngü komutları

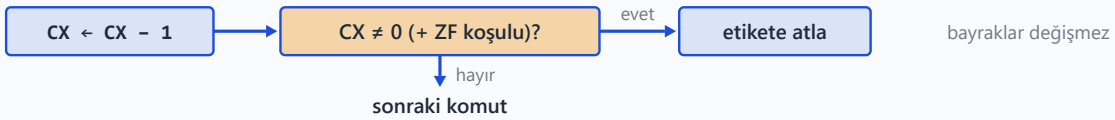
7.6.1 LOOP ailesi

Sayaçlı döngüler o kadar sık yazılır ki 8086 bunlar için ayrı komutlar sunar. Bu komutların hepsi sayaç olarak **CX** yazmacını kullanır. CX'in "sayaç yazmacı" (*count register*) diye anılmasının nedeni budur. Tablo 7.17 aileyi özetler.

Komut	Kod	İşlem ve atlama koşulu
`LOOP`	`E2`	CX'i 1 azalt; CX ≠ 0 ise etikete atla.
`LOOPE`	`E1`	CX'i 1 azalt; CX ≠ 0 ve eşitse (ZF = 1) atla.
`LOOPZ`	`E1`	LOOPE ile aynı: CX ≠ 0 ve ZF = 1 ise atla.
`LOOPNE`	`E0`	CX'i 1 azalt; CX ≠ 0 ve eşit değilse (ZF = 0) atla.
`LOOPNZ`	`E0`	LOOPNE ile aynı: CX ≠ 0 ve ZF = 0 ise atla.
`JCXZ`	`E3`	CX = 0 ise etikete atla (CX azaltılmaz).

Tablo 7.17: Döngü komutları (hepsi 2 bayt: işlem kodu + rel8).

Aile üç ortak özellik taşır. Birincisi, sayaç her zaman CX'tir; başka yazmaç seçilemez. İkincisi, LOOP ailesi CX'i azaltırken **bayrakları değiştirmez**. CX sıfıra inse bile ZF kalkmaz. Bu, aynı işi yapan `DEC CX` komutundan önemli bir farktır. Üçüncüsü, hepsi 2 bayttır ve rel8 kullanır; döngü gövdesi en çok 128 bayt geriye uzanabilir. Gövde daha uzunsa döngü `DEC CX` , `JZ` ve yakın `JMP` ile yazılır.



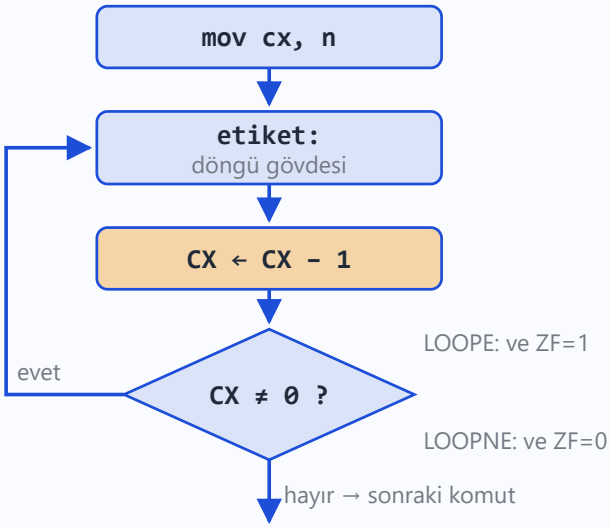
Şekil 7.11: LOOP ailesi önce CX'i bir azaltır. CX sıfır değilse (LOOPE/LOOPNE'de ZF koşulu da sağlanırsa) etikete atlar, değilse sıradaki komuta geçer. Bayraklar değişmez.

7.6.2 LOOP nasıl çalışır?

`LOOP etiket` komutunda işlem sırası önemlidir: önce CX azaltılır, *sonra* sıfır olup olmadığına bakılır. Bu yüzden LOOP bir do-while döngüsü kurar. Gövde, LOOP'a ulaşmadan önce en az bir kez yürütülür. CX = n ile başlayan bir döngüde gövde tam n kez çalışır: CX n − 1, n − 2, …, 1 değerlerinde geri atlanır, 0'da döngüden çıkılır.

```
      mov  cx, n
etiket:
      ; döngü gövdesi
      loop etiket
```

Etkisi bakımından LOOP, `DEC CX` ve `JNZ etiket` çiftine benzer; ama bayrakları korur. LOOPE ve LOOPNE ise döngüyü bir koşul daha sağlandığı sürece sürdürür. LOOPE (LOOPZ), CX ≠ 0 ve ZF = 1 olduğu sürece atlar; eşit öğeler bulundukça sürer, ilk farklılıkta durur. LOOPNE (LOOPNZ), CX ≠ 0 ve ZF = 0 olduğu sürece atlar; eşit olmayan öğeler oldukça sürer, aranan değer bulununca durur. Bu iki komut, "en çok n kez, ama koşul değişirse daha önce" biçimindeki arama döngüleri için tasarlanmıştır. ZF'yi döngü gövdesindeki bir CMP ayarlar.



Şekil 7.12: LOOP akış şeması: gövde yürütülür, CX bir azaltılır, CX sıfır değilse gövdeye dönülür, sıfırsa sıradaki komutla devam edilir. LOOPE ve LOOPNE'de dönüş için ZF koşulu da gerekir.

Bu sıranın bir tuzağı vardır. CX = 0 iken döngüye girilirse gövde bir kez çalışır, sonra LOOP CX'i azaltır. 0 − 1 = FFFFh'dir. CX sıfır olmadığı için geri atlanır. Döngü CX yeniden sıfıra inene kadar sürer ve gövde toplam 65.536 kez çalışır. Döngü sayısı çalışma anında hesaplanıyorsa ve sıfır olabiliyorsa bu durum özel olarak önlenmelidir. Önlemin adı JCXZ'dir.

Tanım 7.6

Döngü (*loop*), bir komut grubunun bir koşul sağlanana kadar yinelenmesidir. **Sayaçlı döngüde** yineleme sayısı bir sayaç yazmacında tutulur; 8086'nın LOOP ailesi sayaç olarak CX'i kullanır, CX'i her turda bir azaltır ve CX sıfırlandığında döngüden çıkar.

7.6.3 Kod izleme: LOOP

En basit LOOP örneği gövdesi boş bir döngüdür. Program yalnız LOOP'un CX üzerindeki etkisini gösterir.

```
org 100h
mov cx, 5      ; CX = tekrar sayısı (5)
dongu:
; döngü içeriği
loop dongu     ; CX ≠ 0 ise dongu'ya atla
jmp dur        ; döngü bitince dur'a atla
dur:
ret
```

`mov cx, 5` 0100h'de üç bayttır (`B9 05 00`). `dongu` etiketi 0103h'yi gösterir. Gövde boş olduğundan `loop dongu` da 0103h'dedir; kodu `E2 FE` 'dir. Uzaklık FEh = −2'dir: LOOP komutu kendi başına atlar. `jmp dur` 0105h'dedir ve kodu `EB 00` 'dır, çünkü `dur` zaten sonraki satırdır.

Tur	Komut	CX (sonra)	Karar	IP (sonra)
—	`mov cx, 5`	0005h	—	0103h
1	`loop dongu`	0004h	≠ 0, atlar	0103h
2	`loop dongu`	0003h	≠ 0, atlar	0103h
3	`loop dongu`	0002h	≠ 0, atlar	0103h
4	`loop dongu`	0001h	≠ 0, atlar	0103h
5	`loop dongu`	0000h	= 0, çıkar	0105h
—	`jmp dur`	0000h	—	0107h
—	`ret`	0000h	—	0000h

Tablo 7.18: LOOP örneğinin izlemesi.

Gövde (burada boş) beş kez çalışır ve LOOP beş kez yürütülür. İlk dört LOOP atlar, beşincisi atlamaz. Döngünün süresi Tablo 7.4'ten hesaplanabilir: 4 × 17 + 5 = 73 saat turu. Program bitince CX = 0000h'dir. Döngüden çıkışın ardından gelen `jmp dur` gereksizdir ama zararsızdır; uzaklığı 0 olan bir atlama yalnız 15 tur harcar ve kuyruğu boşaltır.

7.6.4 Kod izleme: iç içe döngüler

LOOP ailesinin sayacı tek bir yazmaç olduğundan iç içe iki LOOP aynı CX'i paylaşır. İç döngü CX'e kendi sayısını yüklediği anda dış döngünün sayacı kaybolur. Çözüm, dış sayacı iç döngüden önce yığına itmek ve iç döngüden sonra geri almaktır. Aşağıdaki program bu yöntemi kullanır. Dış döngü 5 kez döner; her dış turda ekrana bir `1` , ardından iç döngüde beş tane `2` yazılır. BX toplam adım sayısını tutar.

```
org 100h
mov bx, 0      ; toplam adım
mov cx, 5      ; dış döngü sayacı
k1:
 add bx, 1
 mov al, '1'
 mov ah, 0eh
 int 10h       ; '1' yaz
 push cx       ; dış sayacı sakla
 mov cx, 5     ; iç döngü sayacı
k2:
  add bx, 1
  mov al, '2'
  mov ah, 0eh
  int 10h      ; '2' yaz
  loop k2      ; iç döngü
 pop cx        ; dış sayacı geri al
loop k1        ; dış döngü
ret
```

Karakterler BIOS'un `INT 10h` hizmetiyle yazılır. AH = 0Eh iken BIOS, AL'deki karakteri imlecin bulunduğu yere yazar ve imleci bir ilerletir; bu hizmete daktilo biçimli yazım (*teletype*) denir. Hizmet BH'yi sayfa numarası olarak kullanır. BX bu programda 256'yı aşmadığından BH hep 0'dır ve yazım 0 numaralı sayfaya yapılır.

Adım	Komut	BX	CX	SP	Yığın	Ekran
başlangıç	`mov bx, 0` / `mov cx, 5`	0000h	0005h	FFFEh	[FFFEh] = 0000h	
dış 1	`add bx, 1` … `int 10h`	0001h	0005h	FFFEh	[FFFEh] = 0000h	1
dış 1	`push cx`	0001h	0005h	FFFCh	[FFFCh] = 0005h	1
iç 1	`mov cx, 5` … `int 10h`	0002h	0005h	FFFCh	[FFFCh] = 0005h	12
iç 1	`loop k2`	0002h	0004h	FFFCh	[FFFCh] = 0005h	12
iç 5	`loop k2` (çıkar)	0006h	0000h	FFFCh	[FFFCh] = 0005h	122222
dış 1	`pop cx`	0006h	0005h	FFFEh	[FFFEh] = 0000h	122222
dış 1	`loop k1`	0006h	0004h	FFFEh	[FFFEh] = 0000h	122222
dış 2 sonu	`loop k1`	000Ch	0003h	FFFEh	[FFFEh] = 0000h	122222 ×2
dış 3 sonu	`loop k1`	0012h	0002h	FFFEh	[FFFEh] = 0000h	122222 ×3
dış 4 sonu	`loop k1`	0018h	0001h	FFFEh	[FFFEh] = 0000h	122222 ×4
dış 5 sonu	`loop k1` (çıkar)	001Eh	0000h	FFFEh	[FFFEh] = 0000h	122222 ×5
son	`ret`	001Eh	0000h	0000h	boş	122222 ×5

Tablo 7.19: İç içe döngülerin izlemesi (seçilmiş adımlar).

Program başladığında SP = FFFEh'dir ve yığının tepesinde, DOS'un koyduğu dönüş adresi 0000h durur. İlk dış turda BX = 1 olur ve ekrana `1` yazılır. AL = '1' = 31h, AH = 0Eh'dir. `push cx` dış sayacın değerini, 5'i, yığına iter. SP iki azalır ve FFFCh olur; [FFFCh] = 0005h'dir. Little-endian düzende FFFCh'de 05h, FFFDh'de 00h baytı durur.

İç döngü CX'i yeniden 5 yapar. Dış sayacın değeri artık yalnız yığındadır. İç döngü beş kez döner, her turda BX'i artırır ve bir `2` yazar. Beşinci iç turun LOOP'u CX'i sıfırlar ve döngüden çıkar. Bu noktada BX = 6, CX = 0'dır. `pop cx` yığındaki 0005h'yi CX'e geri yükler ve SP = FFFEh olur. `loop k1` CX'i 4 yapar ve dış döngünün başına döner.

Her dış tur BX'e 1 + 5 = 6 ekler. Beş turdan sonra BX = 30 = 001Eh olur. Ekranda beş kez `122222` yazılıdır, yani 30 karakter. Son LOOP CX'i sıfırlar ve döngüden çıkılır. `push` ile `pop` sayısı eşit olduğu için yığın başlangıçtaki hâline dönmüştür. `ret` yığının tepesindeki 0000h'yi IP'ye alır, SP = 0000h olur ve program biter.

İç döngünün `loop k2` komutu 011Ch'dedir ve kodu `E2 F5`'tir: sonraki adres 011Eh, hedef k2 = 0113h, uzaklık −0Bh = F5h. Dış döngünün `loop k1` komutu 011Fh'dedir ve kodu `E2 E5`'tir: sonraki adres 0121h, hedef k1 = 0106h, uzaklık −1Bh = E5h. Dış döngünün gövdesi 27 bayttır ve rel8'in sınırına (128) daha çok uzaktır.

> **Sık yapılan hata**
>
> İç içe döngüde `push cx` / `pop cx` çiftini unutmak. İç döngü bitince CX = 0 olur. Dış döngünün LOOP'u bu sıfırı azaltır ve FFFFh yapar; dış döngü 5 yerine on binlerce kez döner. Çiftin yalnız birini yazmak daha da kötüdür. Yığın dengesi bozulur ve programın sonundaki `ret` yanlış bir değeri dönüş adresi sanır.

7.6.5 Kod izleme: LOOPNE ile arama

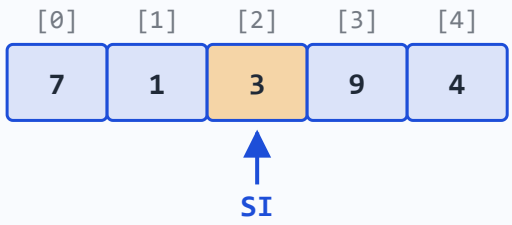
LOOPNE, bir dizide bir değeri aramak için biçilmiş kaftandır. Döngü iki durumdan birinde biter: değer bulunduğunda (ZF = 1) ya da bütün elemanlar incelendiğinde (CX = 0). Aşağıdaki program beş elemanlı bir dizide 3 değerini arar.

```
org 100h
mov cx, 5          ; eleman sayısı
mov si, -1         ; ilk inc ile 0 olur
ara:
inc si             ; sonraki eleman
cmp dizi[si], 3    ; aranan değer 3 mü?
loopne ara         ; CX≠0 ve ZF=0 ise sürdür
jne yok            ; ZF=0 → bulunamadı
mov al, 1          ; bulundu, konum SI'da
ret
yok:
mov al, 0          ; bulunamadı
ret
dizi db 7, 1, 3, 9, 4
```

SI'nın −1 (FFFFh) ile başlatılması bir inceliktir. Döngünün ilk komutu `inc si`'dir ve SI'yı 0 yapar. Böylece SI her turda incelenen elemanın indisini gösterir ve döngüden çıkıldığında bulunan elemanın konumunu taşır. `cmp dizi[si], 3` komutu 5 bayttır (`80 BC 16 01 03`): işlem kodu, ModR/M, dizinin 16 bitlik adresi (0116h) ve sabit. Dizi DB ile tanımlandığından karşılaştırma bayt boyutundadır.

Tur	SI	dizi[SI]	CMP sonrası ZF CF SF	LOOPNE sonrası CX	Karar
1	0000h	7	0 0 0	0004h	CX ≠ 0, ZF = 0 → sürdür
2	0001h	1	0 1 1	0003h	CX ≠ 0, ZF = 0 → sürdür
3	0002h	3	1 0 0	0002h	ZF = 1 → çık

Tablo 7.20: LOOPNE ile aramanın izlemesi (dizi = 7, 1, 3, 9, 4; aranan 3).



Şekil 7.13: Üçüncü turda SI = 2 ve dizi[2] = 3: karşılaştırma ZF = 1 yapar, LOOPNE döngüden çıkar.

Birinci turda `inc si` SI'yı FFFFh'den 0000h'ye getirir. Bu artırma sıfır sonuç verdiği için ZF = 1 yapar, ama hemen ardından gelen CMP bayrakları yeniden ayarlar. 7 − 3 = 4 sıfır değildir: ZF = 0. LOOPNE CX'i 4 yapar; CX ≠ 0 ve ZF = 0 olduğundan aramaya devam eder. İkinci turda dizi[1] = 1'dir. 1 − 3 = −2 (FEh) için ödünç alınır; CF = 1, SF = 1 olur. ZF yine 0'dır ve döngü sürer.

Üçüncü turda dizi[2] = 3'tür ve 3 − 3 = 0 ZF = 1 yapar. LOOPNE CX'i yine azaltır ve 2 yapar. CX sıfır değildir ama ZF = 1 olduğundan atlamaz; döngüden çıkılır. LOOPNE bayrakları değiştirmediği için ZF = 1 korunur. Döngüden sonraki `jne yok` tam bu bayrağa bakar. ZF = 1 olduğundan atlamaz ve `mov al,`

`1` "bulundu" sonucunu yazar. SI = 2, bulunan elemanın konumudur. Döngü beş yerine üç turda bitmiştir.

Döngüden sonra neden ZF'ye bakıldığı önemlidir. Döngü iki nedenle bitebilir ve CX'e bakmak bu nedenleri ayırt etmez: değer son elemanda bulunursa CX de sıfıra iner. Son karşılaştırmanın sonucu ise ZF'de durur ve LOOPNE onu bozmaz. Dizi 3 içermeseydi beş tur sonunda CX = 0, ZF = 0 olurdu; `jne yok` atlar ve AL = 0 olurdu. Bu yapı, LOOP ailesinin bayrakları korumasının doğrudan bir yararıdır.

7.6.6 JCXZ; LOOP mu, DEC–JNZ mi?

`JCXZ etiket` komutu CX = 0 ise atlar; CX'i azaltmaz ve bayraklara bakmaz. Başlıca kullanımı sıfır turlu döngüye karşı korumadır. Döngü sayısı çalışma anında hesaplanıyorsa döngünün önüne bir JCXZ konur:

```
      mov  cx, n      ; n çalışma anında
      jcxz atla       ; CX = 0 ise girme
tekrar:
      ; gövde
      loop tekrar
atla:
```

JCXZ olmadan CX = 0 iken LOOP önce CX'i FFFFh yapar ve gövde 65.536 kez çalışır. JCXZ ile bu durumda döngünün üzerinden atlanır. Komut, CX'e başka bir komutla bakmadan CX'in sıfır olup olmadığını sınar; CX'i sıfırla karşılaştıran bir `CMP CX, 0` bayrakları bozardı, JCXZ bozmaz.

Bir sayaçlı döngü LOOP yerine `DEC` ile `JNZ` çiftiyle de yazılabilir. İki yazımın farkları Tablo 7.21'de toplanmıştır.

Özellik	`LOOP`	`DEC CX + JNZ`
Boyut	2 bayt	1 + 2 = 3 bayt
Bayraklar	değişmez	ZF, SF, OF, PF, AF değişir (CF korunur)
Sayaç	yalnız CX	herhangi bir yazmaç
Ek koşul	LOOPE / LOOPNE	farklı Jcc seçilebilir
8086'da süre (atlarsa / atlamazsa)	17 / 5 tur	2 + 16 = 18 / 2 + 4 = 6 tur

Tablo 7.21: LOOP ile DEC CX + JNZ yazımlarının karşılaştırması.

8086'da LOOP hem daha kısa hem biraz daha hızlıdır. On turluk bir döngüde LOOP, dokuz atlama ve bir çıkışla 9 × 17 + 5 = 158 tur harcar. DEC–JNZ çifti 10 × 2 + 9 × 16 + 4 = 168 tur harcar. Öte yandan DEC–JNZ daha esnektir: sayaç olarak CX'ten başka bir yazmaç kullanılabilir. Bu, CX'i başka bir iş için (örneğin bir kaydırma sayısı ya da dizgi komutunun sayacı olarak) kullanan döngülerde önemlidir. Sonraki x86 işlemcilerde LOOP komutu yavaşladı ve derleyiciler onu kullanmaz oldu; ama 8086 için durum tersidir.

Bayrak farkı da önemlidir. `DEC`, CF dışındaki durum bayraklarını sonuca göre ayarlar. Döngü boyunca CF'de saklanan bir bilgi (örneğin çok sözcüklü bir toplamanın eldesi) DEC'ten etkilenmez. Ama ZF ya da SF'de saklanan bir bilgi bozulur. LOOP hiçbir bayrağa dokunmaz. Çok sözcüklü toplama gibi eldenin turdan tura taşındığı döngülerde iki yazım da CF'yi korur; bu yüzden ikisi de kullanılabilir.

7.6.7 Kod izleme: JCXZ koruması ve DEC–JNZ

Aşağıdaki program iki döngü içerir. Birinci döngü n = 0 ile çalıştırılır ve JCXZ onu korur. İkinci döngü n = 3 ile DEC–JNZ yazımıyla çalışır ve sayaç olarak BX'i kullanır. İkinci döngüden önce CF = 1 yapılır; döngü boyunca CF'nin korunduğu gözlenir.

```
MOV  CX, 0        ; n = 0
MOV  AX, 0
JCXZ atla         ; CX = 0 → girme
tekrar: INC AX
LOOP tekrar
atla: MOV BX, 3   ; n = 3, sayaç BX
STC               ; CF = 1
tekrar2: INC AX
DEC  BX           ; CF korunur
JNZ  tekrar2
RET
```

Adım	Komut	AX	BX	CX	ZF	CF	Karar
1–2	`MOV CX, 0` / `MOV AX, 0`	0000h	0000h	0000h	0	0	—
3	`JCXZ atla` (`E3 03`)	0000h	0000h	0000h	0	0	CX = 0 → atlar, 010Bh
4	`MOV BX, 3`	0000h	0003h	0000h	0	0	—
5	`STC`	0000h	0003h	0000h	0	1	—
6–8	`INC AX` / `DEC BX` / `JNZ`	0001h	0002h	0000h	0	1	atlar (tur 1)
9–11	`INC AX` / `DEC BX` / `JNZ`	0002h	0001h	0000h	0	1	atlar (tur 2)
12–14	`INC AX` / `DEC BX` / `JNZ`	0003h	0000h	0000h	1	1	atlamaz (tur 3)
15	`RET`	0003h	0000h	0000h	1	1	IP = 0000h

Tablo 7.22: JCXZ ve DEC–JNZ programının izlemesi.

Üçüncü adımda JCXZ, CX = 0 olduğunu görür ve döngünün üzerinden atlar. `INC AX` ve `LOOP tekrar` hiç yürütülmez; AX = 0 kalır. Koruma olmasaydı gövde 65.536 kez çalışır ve AX sonunda yeniden 0000h olurdu (65.536 = 1 0000h, 16 bite sığmaz). Sonuç aynı görünürdü, ama işlemci boşuna 65.536 tur dönerdi. Gövde AX'i değil bir bellek dizisini değiştirseydi sonuç felaket olurdu.

İkinci döngüde sayaç BX'tir; LOOP bunu yapamazdı. `STC` CF'yi 1 yapar. Her turda `INC AX` ve `DEC BX` bayrakları ayarlar, ama ikisi de CF'ye dokunmaz. Birinci turda BX = 2, ZF = 0 olur ve `JNZ` geri atlar. İkinci turda BX = 1'dir. Üçüncü turda `DEC BX` BX'i 0 yapar ve ZF = 1 olur. `JNZ` atlamaz ve döngü biter.

Sonuçta AX = 3'tür ve CF = 1 korunmuştur. Üçüncü turda `INC AX` AX'i 3 yaptığında PF = 1 olur (0000 0011b iki tane 1 içerir); bu da DEC ile yeniden ayarlanır. LOOP da bayrak değiştirmez, ama yalnız CX ile çalışır.

7.7 Örnek programlar

7.7.1 Dizi elemanlarının toplamı

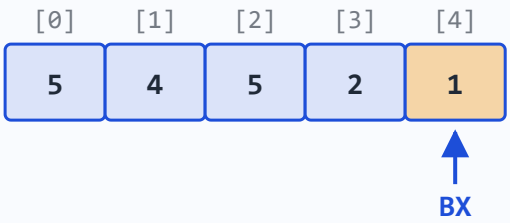
Bir dizinin elemanlarını toplamak, sayaçlı döngünün en tipik kullanımıdır. Aşağıdaki program beş baytlık bir diziyi toplar ve sonucu `m` değişkenine yazar. CX eleman sayısını, BX indisi, AL toplamı tutar.

```
org 100h
mov cx, 5             ; eleman sayısı
mov al, 0             ; toplam (AL) = 0
mov bx, 0             ; BX = indis
next:
add al, vector[bx]    ; elemanı topla
inc bx                ; sonraki eleman
loop next             ; CX = 0 olana kadar dön
mov m, al             ; sonucu m'ye yaz
ret
vector db 5, 4, 5, 2, 1
m db 0
```

Veriler `ret`'ten sonra tanımlanmıştır. Program `ret` ile bittiği için işlemci veri baytlarına hiç ulaşmaz ve onları komut sanmaz. Assembler kodu yerleştirdiğinde vector 0113h'de, m 0118h'de durur. `add al, vector[bx]` komutu 4 bayttır (`02 87 13 01`): ModR/M baytı 87h, "[BX] + 16 bitlik yer değiştirme" biçimini seçer ve yer değiştirme vector'ün adresidir. `loop next` 010Dh'dedir; kodu `E2 F9`, uzaklık 0108h − 010Fh = −7'dir.

Tur	Eklenen	AL (onluk)	AL	BX	CX	LOOP
başlangıç	—	0	00h	0000h	0005h	—
1	vector[0] = 5	5	05h	0001h	0004h	atlar
2	vector[1] = 4	9	09h	0002h	0003h	atlar
3	vector[2] = 5	14	0Eh	0003h	0002h	atlar
4	vector[3] = 2	16	10h	0004h	0001h	atlar
5	vector[4] = 1	17	11h	0005h	0000h	çıkar

Tablo 7.23: Dizi toplamının izlemesi (her satır bir turun sonu).



Şekil 7.14: Dizi toplamının son turu: BX = 4 iken vector[4] = 1 eklenir ve AL = 11h olur.

Her turda üç komut yürütülür. `add al, vector[bx]` elemanı AL'ye ekler; etkin adres 0113h + BX'tir ve kesim DS'dir. `inc bx` indisi bir artırır, çünkü elemanlar birer bayttır. Sözcük dizisinde indis ikişer artırılırdı. `loop next` CX'i azaltır ve sıfır değilse başa döner. Beşinci turdan sonra CX = 0 olur ve döngüden çıkılır. `mov m, al` sonucu 0118h adresine yazar: m = 11h = 17 = 5 + 4 + 5 + 2 + 1.

Döngü gövdesindeki ADD bayrakları her turda yeniden ayarlar, ama LOOP onları hiç okumaz ve değiştirmez. Örneğin dördüncü turda 0Eh + 02h = 10h toplamasında düşük dörtlükten elde çıkar ve AF = 1 olur. Bu bilgiyi hiçbir komut kullanmaz. Toplamın bir bayta sığmadığı durumu ise CF bildirir. Bu dizide toplam 17'dir ve hiçbir turda CF = 1 olmaz. Elemanlar daha büyük olsaydı AL taşabilirdi; bu durum alıştırmalardan birinde incelenir.

7.7.2 Ekrana Hello yazdırma

Aşağıdaki program, sonu 0 baytıyla işaretlenmiş bir dizgiyi karakter karakter ekrana yazar. Döngü sayısı önceden bilinmez; döngü 0 baytına rastlayınca biter. Bu yüzden LOOP yerine CMP, JE ve JMP kullanılır. Bu yapı bir while döngüsüdür: koşul her turun başında sınanır.

```
org     100h
jmp     start
msg     db 'Hello', 0
start:
mov     si, 0           ; kaynak indisi = 0
next_char:
mov     al, msg[si]     ; karakteri al
cmp     al, 0           ; 0 ile karşılaştır
je      stop            ; 0 ise yazmayı durdur
mov     ah, 0eh         ; teletype işlevi
int     10h             ; AL'deki karakteri yaz
inc     si              ; sonraki karakter
jmp     next_char       ; başa dön
stop:
ret
```

Bu programda veri kodun arasındadır. msg, programın ilk komutunun hemen arkasında, 0102h'den başlayarak 48h 65h 6Ch 6Ch 6Fh 00h baytlarını tutar. İlk komut `jmp start` (`EB 06`) bu altı baytın üzerinden 0108h'ye atlar. JMP olmasaydı işlemci 0102h'deki 48h baytını bir komut olarak çözerdi.

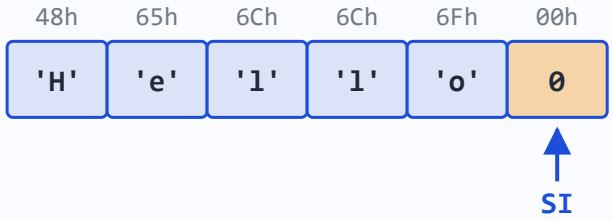
48h, `DEC AX` komutunun kodudur; arkasından gelen baytlar da başka komutlar gibi çözülür ve programın ne yapacağı öngörülemez.

Tur	SI (başta)	AL = msg[SI]	ZF	je stop	Ekran
1	0000h	48h 'H'	0	atlamaz	H
2	0001h	65h 'e'	0	atlamaz	He
3	0002h	6Ch 'l'	0	atlamaz	Hel
4	0003h	6Ch 'l'	0	atlamaz	Hell
5	0004h	6Fh 'o'	0	atlamaz	Hello
6	0005h	00h	1	atlar	Hello

Tablo 7.24: Hello programının izlemesi (her satır bir turun sonu).

Her turda `mov al, msg[si]` (`8A 84 02 01`) sıradaki karakteri AL'ye alır. `cmp al, 0` (`3C 00`) onu sıfırla karşılaştırır. Karakter sıfır değilse ZF = 0 olur ve `je stop` atlamaz. AH = 0Eh ve `INT 10h` ile karakter ekrana yazılır. `inc si` indisi artırır ve `jmp next_char` (`EB F1`) başa döner. Bu JMP 0118h'dedir; uzaklık 010Bh − 011Ah = −0Fh = F1h'dir.

Altıncı turda msg[5] = 0 okunur. 0 − 0 = 0 olduğundan ZF = 1 olur ve `je stop` atlar. Ekranda `Hello` yazılıdır, SI = 5'tir. `ret` programı bitirir. Program sonunda AX = 0E00h'dir: AH, son turdaki 0Eh değerini korur, AL ise son okunan 00h'dir.



Şekil 7.15: Hello dizgisi bellekte: altı bayt, sonuncusu 00h. SI = 5 iken okunan 0 baytı döngüyü bitirir.

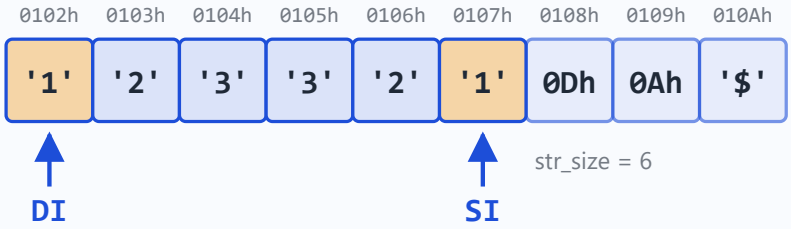
Sonu sıfırla işaretlenen dizgilere **sıfır sonlu dizgi** denir. Bu düzende uzunluk ayrıca saklanmaz; dizgi 0 baytına kadar sürer. Aynı düzeni C dili de kullanır. DOS'un `INT 21h` /09h hizmeti ise farklı bir işaret kullanır: dizginin sonu '$' karakteridir. İki düzenin ortak yanı, döngünün sayaçla değil bir koşulla bitmesidir.

Programdaki her tur yedi komut yürütür; bunların iki tanesi akış kontrol komutudur. Atlanmayan `je` 4, atlanan `jmp` 15 saat turu harcar. Döngü do-while biçiminde yeniden yazılsaydı (karşılaştırma ve koşullu geri atlama sonda) her turda bir JMP eksik olurdu. Kısa bir dizgide fark önemsizdir; çok uzun bir metinde ise toplam sürenin önemli bir kısmını oluşturur.

7.7.3 Palindrom kontrolü: fikir

Palindrom, baştan ve sondan okunuşu aynı olan dizidir: `123321`, `kayak`, `ey edip adanada pide ye` (boşluklar sayılmazsa). Bir dizinin palindrom olup olmadığını anlamak için iki işaretçi kullanılır. DI dizinin başını, SI sonunu gösterir. Her adımda iki karakter karşılaştırılır; sonra DI bir ileri, SI bir geri gider. Karakterlerden biri farklıysa dizi palindrom değildir. Bütün çiftler eşleşirse palindromdur.

Uzunluğun yarısı kadar karşılaştırma yeterlidir. Altı karakterlik bir dizide 3 çift vardır: (1., 6.), (2., 5.), (3., 4.). Daha fazla karşılaştırma aynı çiftleri ters sırayla yeniden sınar. Tek uzunlukta ortadaki karakterin bir eşi yoktur ve karşılaştırılmaz; 5 karakterde 2 çift sınanır. Yarıya bölme işlemi `shr cx, 1` ile yapılır: bir bit sağa kaydırma, işaretsiz bir sayıyı 2'ye böler ve kalanı atar.



Şekil 7.16: `str` dizisi bellekte 0102h'den başlar; DI baştan, SI sondan ilerler. Dizinin ardında satır sonu (0Dh, 0Ah) ve '$' durur.

Programda dizinin uzunluğu bir sabittir: `str_size = $ - msg`. Burada `$`, assembler'ın o satırdaki konum sayacıdır, yani dizinin hemen arkasındaki adrestir. msg etiketi ile str aynı adresi gösterir; fark dizinin uzunluğu olan 6'dır. `=` direktifi EQU gibi bir sabit tanımlar ve bellekte yer kaplamaz. Dizi değişirse uzunluk kendiliğinden güncellenir.

Program dizinin kendisini ve sonucu DOS'un `INT 21h`/09h hizmetiyle yazar. Bu hizmet DS:DX adresindeki dizgiyi '$' karakterine kadar ekrana basar; '$' yazılmaz. Dizinin ardındaki 0Dh ve 0Ah baytları satır başı ve yeni satır karakterleridir. Sonuç mesajı bu sayede bir alt satıra yazılır. Şekildeki adresler `org 100h` ve iki baytlık kısa `jmp start` (0100h) varsayımıyla verilmiştir: str 0102h'den başlar.

7.7.4 Palindrom: hazırlık

```
org 100h
jmp start
msg:
str db '123321'
str_size = $ - msg     ; = 6
db 0Dh,0Ah,'$'
start:
mov ah, 9              ; '$'e kadar yaz
mov dx, offset str
int 21h
lea di, str            ; DI = str adresi
mov si, di
add si, str_size       ; SI = son + 1
dec si                 ; SI = son karakter
mov cx, str_size
cmp cx, 1              ; tek karakter mi?
je is_palindrome       ; tek karakter palindromdur
shr cx, 1              ; CX = uzunluk / 2
```

Programın başındaki `jmp start` (`EB 09`) dokuz baytlık veriyi (altı karakter, 0Dh, 0Ah, '$') atlar ve 010Bh'ye gider. `mov ah, 9` ve `mov dx, offset str` DOS'un dizgi yazma hizmetini hazırlar: DX = 0102h. `int 21h` ekrana `123321` ve satır sonunu yazar. DOS bu hizmetten AL = 24h ('$') ile döner; AX = 0924h olur.

`lea di, str` DI'ya str'nin adresini, 0102h'yi yükler. Burada etkin adres doğrudan bir sabittir; emu8086 bu komutu `mov di, offset str` ile aynı 3 baytlık koda (`BF 02 01`) çevirir. `mov si, di` SI'yı da 0102h yapar. `add si, str_size` 6 ekler: SI = 0108h. Bu, dizinin son karakterinin bir sonrasıdır (0Dh baytı). `dec si` SI'yı 0107h'ye, son karakter '1''e getirir. `mov cx, str_size` CX'e 6 yükler.

`cmp cx, 1` ve `je is_palindrome` tek karakterlik dizi için bir kısa yoldur. Tek karakterli dizi her zaman palindromdur; ama asıl neden başkadır. Uzunluk 1 iken `shr cx, 1` CX'i 0 yapar ve LOOP'lu döngüye CX = 0 ile girilirse karşılaştırma 65.536 kez çalışırdı. Burada CX = 6'dır: 6 − 1 ≠ 0, ZF = 0, CF = 0; JE atlamaz. `shr cx, 1` CX'i 0110b'den 0011b'ye kaydırır: CX = 3. Dışarı kayan bit 0'dır, bu yüzden CF = 0 olur.

Adres	Komut	Kod	Sonuç
0100h	`jmp start`	`EB 09`	IP = 010Bh
010Bh	`mov ah, 9`	`B4 09`	AH = 09h
010Dh	`mov dx, offset str`	`BA 02 01`	DX = 0102h
0110h	`int 21h`	`CD 21`	ekran: 123321 ↵; AL = 24h
0112h	`lea di, str`	`BF 02 01`	DI = 0102h
0115h	`mov si, di`	`8B F7`	SI = 0102h
0117h	`add si, str_size`	`83 C6 06`	SI = 0108h

Adres	Komut	Kod	Sonuç
011Ah	`dec si`	`4E`	SI = 0107h
011Bh	`mov cx, str_size`	`B9 06 00`	CX = 0006h
011Eh	`cmp cx, 1`	`83 F9 01`	ZF = 0, CF = 0
0121h	`je is_palindrome`	`74 0E`	atlamaz
0123h	`shr cx, 1`	`D1 E9`	CX = 0003h, CF = 0

Tablo 7.25: Palindrom programının hazırlık kısmının izlemesi.

7.7.5 Palindrom: karşılaştırma döngüsü

```
next_char:
mov al, [di]          ; baştaki karakter
mov bl, [si]          ; sondaki karakter
cmp al, bl
jne not_palindrome    ; farklıysa değil
inc di                ; baştan bir ileri
dec si                ; sondan bir geri
loop next_char
is_palindrome:
mov ah, 9             ; "palindrome!" yaz
mov dx, offset msg1
int 21h
jmp stop
not_palindrome:
mov ah, 9             ; "not palindrome!" yaz
mov dx, offset msg2
int 21h
stop:
ret
msg1 db "palindrome!$"
msg2 db "not palindrome!$"
```

Döngü `next_char` etiketiyle 0125h'den başlar. Her turda `mov al, [di]` baştaki, `mov bl, [si]` sondaki karakteri okur. `cmp al, bl` ikisini karşılaştırır. Farklılarsa `jne not_palindrome` döngüden hemen çıkar. Eşitlerse `inc di` ve `dec si` işaretçileri birbirine yaklaştırır; `loop next_char` CX'i azaltır ve başa döner.

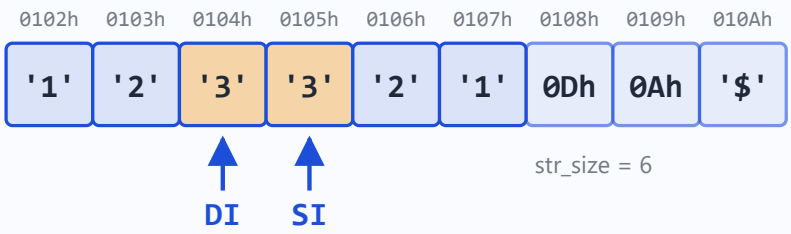
Tur	DI	SI	AL	BL	CMP sonrası ZF	INC/DEC sonrası DI, SI, ZF	LOOP sonrası CX
1	0102h	0107h	31h '1'	31h '1'	1	0103h, 0106h, 0	0002h
2	0103h	0106h	32h '2'	32h '2'	1	0104h, 0105h, 0	0001h
3	0104h	0105h	33h '3'	33h '3'	1	0105h, 0104h, 0	0000h

Tablo 7.26: Palindrom karşılaştırma döngüsünün izlemesi ('123321').

Her turda CMP eşitliği bulur ve ZF = 1 yapar; JNE atlamaz. Ardından gelen `inc di` ve `dec si` de bayrakları ayarlar. Sonuçları sıfır olmadığından ZF yeniden 0 olur. Bu, döngünün doğruluğunu etkilemez, çünkü karar CMP'nin hemen arkasındaki JNE'de verilmiştir. Döngü kararı ise LOOP'a aittir ve LOOP yalnız CX'e bakar. Üçüncü turdan sonra CX = 0 olur ve döngüden çıkılır: üç çiftin hepsi eşleşmiştir. Bu noktada DI = 0105h, SI = 0104h'dir; işaretçiler birbirini geçmiştir.

Döngüden çıkan işlemci sıradan düşerek `is_palindrome` bloğuna girer. AH = 09h ve DX = offset msg1 ile DOS'un dizgi yazma hizmeti çağrılır. msg1, kodun hemen ardından 0142h'de durur; DX = 0142h olur. Ekrana `palindrome!` yazılır ve AL yine 24h ile döner. `jmp stop` (`EB 07`) `not_palindrome` bloğunun üzerinden atlar ve `ret` programı bitirir.

Veriler `stop:` etiketinin hemen ardından gelir. `ret` olmasaydı işlemci `palindrome!$` baytlarını komut sanıp yürütmeye başlardı. Programın uzunluğu 94 bayttır (0100h–015Dh): msg1 12, msg2 16 bayt tutar.



Şekil 7.17: Üçüncü turun karşılaştırması: DI = 0104h ve SI = 0105h ortadaki iki '3' karakterini gösterir.

7.7.6 Palindrom: farklı girişler

Programın farklı girişlerde nasıl davrandığını görmek, yazılan koşulların gerekli olup olmadığını anlamanın en iyi yoludur. Tablo 7.27 beş girişi karşılaştırır.

str	Uzunluk	CX (shr sonrası)	Karşılaştırmalar	Çıktı
`'123321'`	6	3	1=1, 2=2, 3=3	palindrome!
`'123421'`	6	3	1=1, 2=2, 3≠4 → jne	not palindrome!
`'12321'`	5	2	1=1, 2=2 (ortadaki 3 atlanır)	palindrome!
`'7'`	1	—	cmp cx,1 → je is_palindrome	palindrome!
`'ab'`	2	1	a≠b → jne	not palindrome!

Tablo 7.27: Palindrom programının farklı girişlerdeki davranışı.

İkinci girişte üçüncü turda '3' ile '4' karşılaştırılır; ZF = 0 olur ve `jne not_palindrome` döngüden çıkar. LOOP'a hiç ulaşılmaz ve CX = 1'de kalır. Üçüncü girişte uzunluk 5'tir; `shr` 5'i 2'ye böler, kalan 1 atılır (CF = 1 olur). İki çift karşılaştırılır, ortadaki karakter tek başına kalır. Dördüncü girişte `cmp cx,`

`1` denetimi işe yarar: uzunluk 1 iken `shr cx, 1` CX'i 0 yapardı ve LOOP'lu döngü 65.536 kez çalışırdı. Uzunluk 0 da aynı tehlikeyi taşır. Boş bir dizi için CX = 0'dır, `cmp cx, 1` eşit bulmaz, `shr` 0'ı yine 0 yapar. Bu durum döngüden önce bir `jcxz is_palindrome` ile önlenebilir.

7.8 Sık yapılan hatalar

Akış kontrolündeki hataların çoğu birkaç kalıba indirgenebilir. Aşağıdaki liste bu kalıpları ve önlemlerini toplar.

Sık yapılan hata

Yanlış atlama ailesi. İşaretli veride JA/JB, işaretsiz veride JG/JL kullanmak. AL = FBh, BL = 03h için JA atlar (251 > 3), JG atlamaz (−5 < 3). Hata yalnız en anlamlı biti 1 olan değerlerde ortaya çıkar ve küçük deneme verileriyle gizli kalır. Önlem: her karşılaştırmada verinin türü belirlenir ve aile buna göre seçilir.

Sık yapılan hata

Unutulan JMP. if/else kalıbında A bloğunun sonundaki `JMP son` yazılmazsa A bloğu bittiğinde işlemci B bloğuna da girer. Bloklar etiketlerle ayrılır; etiket bir duvar değildir, yalnız bir adrestir.

Sık yapılan hata

CMP ile atlama arasına bayrak değiştiren komut girmesi. `CMP AL, BL` / `INC SI` / `JE eşit` yazımında JE, CMP'nin değil INC'in sonucuna bakar. Araya girmesi gereken bir komut varsa bayrak değiştirmeyen bir komut (MOV, LEA, PUSH, POP) seçilir ya da CMP atlamanın hemen önüne taşınır.

Sık yapılan hata

CX = 0 ile LOOP. LOOP önce azaltır, sonra sınar. CX = 0 iken girilen döngü 65.536 kez döner. Önlem: döngü sayısı çalışma anında belirleniyorsa döngünün önüne `JCXZ` konur.

Sık yapılan hata

İç içe döngüde CX'i saklamamak. İç döngünün `MOV CX, n` komutu dış sayacı siler. Önlem: iç döngüden önce `PUSH CX`, sonra `POP CX`; ya da dış döngüde DEC–JNZ ile başka bir sayaç yaz-

macı.

Sık yapılan hata

Veriyi kod gibi yürütmek. Veriler kodun önündeyse `JMP start` unutulur, kodun arkasındaysa son `RET` unutulur. İki durumda da işlemci veri baytlarını komut sanar. 'H' karakteri (48h) `DEC AX` olarak yürütülür; ardından gelen baytlar rastgele komutlara dönüşür.

Sık yapılan hata

Menzil dışı Jcc'yi eş anlamlıyla düzeltmek. JE yerine JZ yazmak ya da SHORT eklemek menzili değiştirmez. Doğru çözüm ters koşulla kısa atlama ve yakın JMP'dir.

Sık yapılan hata

Döngü gövdesinde sayacı değiştirmek. LOOP'lu bir döngünün gövdesinde CX'i değiştiren bir komut (örneğin `DEC CX`) döngü sayısını bozar. CX = 3 ile başlayan ve gövdesinde `DEC CX` bulunan bir döngüde CX her turda iki azalır. LOOP sonrasında CX tek değerlerden geçer: 1, FFFFh, FFFDh, … CX sıfırı hiç LOOP'tan sonra görmediği için döngü hiç bitmez.

7.9 Alıştırmalar ve çözümleri

Sunumun alıştırmaları ve çözümleri aşağıdadır. Çözümler ara adımlarıyla verilmiştir.

1. **0120h adresindeki JNE komutu 0100h adresine atlamak istiyor. Makine kodu nedir?**

   JNE'nin işlem kodu 75h'dir ve komut 2 bayttır. Sonraki IP = 0122h. Uzaklık = 0100h − 0122h = −22h. −22h'nin ikiye tümleyeni: 22h = 0010 0010b, tersi 1101 1101b, artı 1: 1101 1110b = DEh. Kod `75 DE` 'dir. Sağlama: 0122h + FFDEh = 1 0100h → IP = 0100h.

2. `mov al, 80h` / `cmp al, 1` **sonrasında JA ve JG atlar mı?**

   80h − 01h = 7Fh. Ödünç yoktur (CF = 0), sonuç sıfır değildir (ZF = 0): JA atlar; işaretsiz yorumda 128 > 1'dir. İşaretli yorumda −128 − 1 = −129 sekiz bite sığmaz: OF = 1. Sonucun işaret biti 0'dır: SF = 0. SF ≠ OF olduğundan JG atlamaz; −128 < 1'dir. Düşük dörtlükte 0 − 1 ödünç gerektirdiği için AF = 1, 7Fh yedi tane 1 içerdiği için PF = 0 olur.

3. `mov cx, 0` / `d: loop d` **kaç kez döner? Nasıl önlenir?**

LOOP önce CX'i azaltır: 0 − 1 = FFFFh ≠ 0, geri atlar. CX FFFFh'den 1'e kadar her değerde atlar, 0'da çıkar. LOOP toplam 65.536 kez yürütülür. Önlemek için döngüden önce `jcxz` ile CX = 0 durumu atlanır.

4. **AX'teki işaretsiz sayı 100'den büyükse BX = 1, değilse BX = 0 yapan kod.**

```
    mov  bx, 0
    cmp  ax, 100
    jbe  son        ; AX <= 100 ise atla (işaretsiz)
    mov  bx, 1
son:
```

Koşulun tersi (≤) işaretsiz aileden seçilir: JBE. Denendiğinde AX = 0, 99 ve 100 için BX = 0; AX = 101 ve FFFFh (65.535) için BX = 1 bulunur. JLE kullanılsaydı FFFFh işaretli −1 sayılır ve BX = 0 kalırdı.

5. **Dizi toplamı örneğinde elemanlar 200 ve 100 olsaydı AL ve CF ne olurdu? Çözüm önerin.**

200 + 100 = 300 = 12Ch. AL'ye yalnız düşük bayt sığar: AL = 2Ch = 44. Dokuzuncu bit CF'ye düşer: CF = 1 (elde). Sonuç yanlıştır. Çözüm toplamı 16 bitte tutmaktır. Döngüde `add al, vector[bx]` komutunun arkasına `adc ah, 0` eklenir ve döngüden önce AX sıfırlanır. ADC, AH'ye 0 ile birlikte CF'yi de ekler; her elde AH'ye taşınır. Bu çözümle AX = 012Ch = 300 bulunur. LOOP bayrak değiştirmediği için eldeyi bozmaz; ama ADD her turda CF'yi yeniden ayarladığı için ADC hemen ADD'in arkasında durmalıdır.

6. **Hello örneğinde `jmp start` satırı silinirse ne olur?**

Program 0100h'den başlar ve orada msg'nin ilk baytı, 48h ('H') durur. İşlemci bu baytı `DEC AX` komutu olarak yürütür. Arkasından gelen 65h, 6Ch, 6Ch, 6Fh, 00h baytları da komut olarak çözülür ve yürütülür. Programın davranışı öngörülemez olur; ekrana Hello yazılmaz. Assembler bu hatayı yakalamaz, çünkü veri ile kod arasındaki ayrım yalnız programcının kafasındadır.

7.10 Özet

- **IP ve kuyruk**: işlemci komutları CS:IP'den sırayla yürütür; IP her komuttan sonra komutun uzunluğu kadar artar. BIU, sıradaki baytları altı baytlık önden getirme kuyruğuna önceden alır. Atlama kuyruğu boşaltır; bu yüzden atlanan bir atlama 15–19 saat turu sürer.
- **JMP** koşulsuzdur ve bayrakları değiştirmez. Kısa (EB rel8, −128…+127), yakın (E9 rel16, kesim içi), uzak (EA ofs seg, CS:IP) ve dolaylı (FF /4, yazmaç ya da bellek) biçimleri vardır.
- **Uzaklık** = hedef − (komut adresi + komut uzunluğu). Negatif uzaklık ikiye tümleyenle saklanır; işlemci onu işaret genişletip IP'ye ekler.
- **Etiket** bellekte yer kaplamaz; assembler onu bir bağıl konuma çevirir. Rakamla başlayamaz, boşluk içeremez, komut ya da yazmaç adı olamaz.

- **Jcc** bayraklara bakar, bayrakları değiştirmez. 8086'da hepsi 2 bayttır (70h–7Fh + rel8); işlem kodunun son biti koşulu tersler. Eş anlamlılar aynı koda derlenir.
- **CMP** sonucu saklamayan bir SUB'dır; ardından gelen Jcc kararı verir.
- **İşaretsiz** karşılaştırma: JA, JAE, JB, JBE → CF ve ZF. JB ≡ JC.
- **İşaretli** karşılaştırma: JG, JGE, JL, JLE → ZF, SF ve OF. A < B ⇔ SF ≠ OF; taşmada işaret biti ters çıktığı için SF tek başına yetmez.
- **if/else**: koşulun tersiyle A bloğu atlanır, A bloğunun sonunda JMP ile B bloğu atlanır. **while**: koşul başta sınanır, gövde sonunda JMP ile başa dönülür.
- **Menzil dışı Jcc**: ters koşulla kısa atlama + yakın JMP. emu8086 bu dönüşümü kendisi yapar.
- **LOOP ailesi**: CX azalır, bayraklar korunur; LOOPE/LOOPNE ZF'ye de bakar; JCXZ CX = 0 ise atlar. Hepsi 2 bayt ve rel8'dir.
- **Tuzaklar**: CX = 0 ile LOOP (65.536 tur), iç içe döngüde CX'i saklamamak, unutulan JMP, CMP ile Jcc arasına bayrak değiştiren komut, veriyi kod gibi yürütmek.

7.11 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

akış kontrolü *flow control* · komut işaretçisi (IP) *instruction pointer* · önden getirme kuyruğu *prefetch queue* · yol arayüz birimi (BIU) *bus interface unit* · yürütme birimi (EU) *execution unit* · atlama *jump* · koşulsuz / koşullu atlama *unconditional / conditional jump* · kısa / yakın / uzak atlama *short / near / far jump* · uzaklık *jump displacement* · göreceli adresleme *relative addressing* · etiket *label* · bağıl konum *offset* · işlem kodu *opcode* · ikiye tümleyen *two's complement* · işaret genişletme *sign extension* · bayrak *flag* · sıfır bayrağı (ZF) *zero flag* · elde bayrağı (CF) *carry flag* · işaret bayrağı (SF) *sign flag* · taşma bayrağı (OF) *overflow flag* · eşlik bayrağı (PF) *parity flag* · ödünç *borrow* · karşılaştırma (CMP) *comparison* · işaretli / işaretsiz sayı *signed / unsigned number* · büyük / küçük, işaretsiz *above / below* · büyük / küçük, işaretli *greater / less* · döngü *loop* · iç içe döngü *nested loop* · sayaç yazmacı *counter register* · dallanma *branch* · yığın *stack* · dizgi *string* · saat turu *clock cycle* · PSP (program kesimi öneki) *program segment prefix*

7.12 Kendini Dene

1. 0150h adresindeki kısa JMP komutu 0140h'ye atlıyor. Makine kodu nedir? Komut 0300h'de olsay-

dı hangi biçim kullanılırdı?

Cevap

Sonraki IP = 0152h; uzaklık 0140h − 0152h = −12h → EEh. Kod `EB EE`. 0300h'den uzaklık −1C2h (−450) olurdu ve kısa menzile sığmazdı. Yakın biçim seçilir: sonraki IP 0303h, uzaklık FE3Dh, kod `E9 3D FE`.

2. AX = 8000h ve BX = 0001h iken `CMP AX, BX` hangi bayrakları ayarlar? JB ve JL atlar mı?

Cevap

8000h − 0001h = 7FFFh. CF = 0 (ödünç yok), ZF = 0, SF = 0 (sonucun en anlamlı biti 0), OF = 1 (işaretli −32.768 − 1 sığmaz). JB atlamaz: işaretsiz yorumda 32.768 > 1. JL atlar: SF ≠ OF, işaretli yorumda −32.768 < 1.

3. JL'nin işlem kodu 7Ch'dir. Tersi olan komutun işlem kodu nedir? 0108h'deki bir JL'yi 0100h'ye geri atlatmak için gereken kod nedir?

Cevap

Son bit değişir: 7Dh, yani JGE (JNL). Geri atlama: sonraki IP = 010Ah, uzaklık 0100h − 010Ah = −0Ah = F6h; kod `7C F6`.

4. Aşağıdaki döngünün gövdesinde yanlışlıkla `dec cx` bırakılmıştır. CX = 4 ile başlarsa program bittiğinde AX nedir? CX = 3 ile başlarsa ne olur?

```
    mov  cx, 4
    mov  ax, 0
d:  inc  ax
    dec  cx
    loop d
```

Cevap

CX = 4: tur 1'de DEC 3, LOOP 2 (atlar); tur 2'de DEC 1, LOOP 0 (çıkar). AX = 2. CX = 3: tur 1'de DEC 2, LOOP 1 (atlar); tur 2'de DEC 0, LOOP FFFFh (atlar); sonra CX hep tek değerlerden geçer ve LOOP'tan sonra hiç 0 olmaz. Döngü sonsuzdur.

5. AL'deki işaretli sayı negatifse BL = 1, değilse BL = 0 yapan kod CMP ve tek bir koşullu atlamayla

nasıl yazılır? Hangi atlama seçilmelidir?

Cevap

```
    mov  bl, 0
    cmp  al, 0
    jge  son        ; AL >= 0 ise atla
    mov  bl, 1
son:
```

Koşulun tersi (AL ≥ 0) işaretli aileden seçilir: JGE. AL = 80h ve FFh için BL = 1; 00h, 05h ve 7Fh için BL = 0 olur. Sıfırla karşılaştırmada taşma olamayacağı için `JNS` da doğru çalışır.

6. On turluk bir döngü LOOP ile ve DEC CX + JNZ ile yazılıyor. 8086'da yalnız döngü denetimi kaç saat turu sürer?

Cevap

LOOP: 9 atlama × 17 + 1 çıkış × 5 = 158 tur. DEC CX + JNZ: 10 × 2 (DEC) + 9 × 16 + 1 × 4 = 168 tur. 8086'da LOOP hem 1 bayt kısa hem 10 tur hızlıdır.

7. 0102h'deki `je uzak` komutunun hedefi 01CFh'dedir. emu8086 bu komutu nasıl kodlar? Toplam kaç bayt olur?

Cevap

Uzaklık 01CFh − 0104h = CBh = 203 > 127; kısa biçim yetmez. Koşul terslenir: `75 03` (JNE; koşul sağlanırsa JMP'nin üç baytının üzerinden geçer) ve `E9 C8 00` (yakın JMP; sonraki IP 0107h, 01CFh − 0107h = 00C8h). Toplam 5 bayt.

8. LOOPNE ile arama örneğinde dizi `7, 1, 8, 9, 3` olsaydı döngü kaç turda biter? Döngüden sonra CX, ZF ve AL nedir?

Cevap

3 son elemandadır. Beşinci turda CMP ZF = 1 yapar; LOOPNE CX'i 1'den 0'a indirir ve iki nedenle de çıkar. CX = 0, ZF = 1 olur. `jne yok` ZF'ye baktığı için atlamaz ve AL = 1, SI = 4 bulunur. CX'e bakılsaydı "bulunamadı" sanılırdı.

7.13 İleri okuma

- B. B. Brey, *The Intel Microprocessors* (8. baskı), Bölüm 6 (Program Control Instructions): JMP biçimleri, koşullu atlamalar, LOOP ailesi ve yüksek düzey yapıların assembly karşılıkları; bu notun en yakın kaynağıdır.
- Intel, *The 8086 Family User's Manual* (1979): komut kodlama tabloları ve saat turu sayıları birinci elden buradadır; önden getirme kuyruğunun işleyişi de anlatılır.
- K. R. Irvine, *Assembly Language for x86 Processors* (7. baskı), Bölüm 6 (Conditional Processing): CMP, işaretli ve işaretsiz atlamalar, bileşik koşullar ve döngü örnekleri.
- emu8086 belgeleri, *8086 assembler tutorial for beginners*, Part 7 (Program Flow Control): sunumdaki örneklerin kaynağı; menzil dışı koşullu atlama hilesi de burada anlatılır.

7.14 Kaynaklar

1. emu8086 belgeleri: *8086 Assembler Tutorial for Beginners* (Part 7: Program Flow Control).
2. Intel, *The 8086 Family User's Manual*, 1979: komut kümesi ve kodlama tabloları.
3. Barry B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson, 2009: Bölüm 6 “Program Control Instructions”.
4. Kip R. Irvine, *Assembly Language for x86 Processors*, 7. baskı, Pearson, 2014: koşullu işlem ve döngüler.

KISIM III · BÖLÜM 8

Yığın ve Prosedürler

8086 yığını SS:SP çifti ile adreslenir ve bellekte aşağı doğru büyüyen bir LIFO yapısıdır. Bölümde PUSH ve POP komutları, SP'nin değişimi, yazmaçların saklanması ve takası, PUSHF ve POPF ile bayrakların korunması ve yığın güvenliği anlatılır. Prosedürler kısmında PROC ve ENDP ile tanımlama, CALL ve RET sırasında dönüş adresinin yığına yazılıp geri alınması, yazmaçlar, bellek ve yığın üzerinden parametre geçirme ve BP ile kurulan yığın çerçevesi işlenir. Özyineleme örnekleri ve sık yapılan hatalarla bölüm tamamlanır.

Kod (8086): `github.com/sercankulcu/microprocessors-8086/Bolum08`

BU BÖLÜMÜN NOTLARI

8.1 Yığın

8.2 Prosedürler

KISIM III · BÖLÜM 8.1

Yığın

Bir program çalışırken pek çok değerin kısa bir süre için bir kenara konması gerekir. Bir yazmacın eski değeri, bir prosedürün dönüş adresi, kesilen programın bayrakları bunların başında gelir. 8086 bu işler için belleğin bir bölümünü yığın olarak kullanır. Yığına en son konan değer ilk geri alınır. Not, yığın kavramıyla ve son giren ilk çıkar düzeniyle başlar. Ardından yığının SS ve SP yazmaçlarıyla nasıl adreslendiğini, neden aşağı doğru büyüdüğünü ve 8086'nın yığına hangi yol döngüleriyle eriştiğini anlatır. Sonra PUSH, POP, PUSHF ve POPF komutlarının yazmaçlara, bayraklara ve belleğe etkisini tek tek ele alır. Sunumdaki bütün örnekler yazmaçlar ve yığınla birlikte adım adım izlenir: değer saklama, takas, üç yazmacın döndürülmesi ve bir dizinin yığınla ters çevrilmesi. Son kesim dengesiz PUSH ve POP, yığın taşması ve alt taşma gibi hatalara ve bunlardan korunma yollarına ayrılmıştır.

ÖĞRENME ÇIKTILARI

Yığın modelini açıklamak
Yığının son giren ilk çıkar (LIFO) düzenini, 8086 yığınının SS:SP ile nasıl adreslendiğini ve neden aşağı doğru büyüdüğünü açıklamak; yığının üstünün fiziksel adresini hesaplamak.

PUSH ve POP'u izlemek
PUSH ve POP için geçerli işlenen türlerini bilmek, her komutta SP'nin ve yığın belleğinin nasıl değiştiğini bayt düzeyinde adım adım göstermek.

Yazmaçları saklamak ve takas etmek
Yığını bir yazmacın değerini korumak, geri yüklemek, iki yazmacı takas etmek ve bir diziyi ters çevirmek için doğru sırayla kullanmak.

Bayrakları korumak
PUSHF ve POPF ile bayrak yazmacını yığında saklamak, bir işlemin bozduğu bayrakları geri yüklemek ve bayrak sözcüğünü bit bit yorumlamak.

Yığın hatalarını tanımak
Dengesiz PUSH ve POP, yanlış sıralama, yığın taşması ve alt taşma hatalarının belirtilerini tanımak ve bunları önleyen yazım kurallarını uygulamak.

emu8086 ile yığını izlemek
Bir programı tek adım çalıştırırken yazmaç panelini, yığın penceresini ve bellek görünümünü yorumlamak.

8.1.1 Yığın kavramı

8.1.1.1 Yığın nedir?

8086'nın sekiz genel amaçlı yazmacı vardır. Bir program çoğu zaman bu yazmaçların hepsini kullanır ve bir noktada yer darlığı başlar. Örneğin AX bir toplamı tutarken AX'e başka bir iş için de gerek duyulur. Toplamı bir değişkene yazıp sonra geri okumak bir çözümdür. Ama her ara değer için ayrı bir değişken tanımlamak programı kalabalıklaştırır. Ayrıca kaç değerin saklanacağı her zaman önceden bilinmez. Bir prosedür başka bir prosedürü, o da bir başkasını çağırabilir. Her çağrının dönüş adresi bir yerde durmalıdır ve çağrı derinliği programın akışına bağlıdır.

Bu ihtiyaçların ortak bir özelliği vardır: en son saklanan değer ilk geri istenir. En içteki prosedür önce biter, dönüş adresi en son saklanmış olan adrestir. Bir yazmaç geçici olarak kullanıldığında da en son saklanan değer ilk geri yüklenir. 8086 bu düzene uyan bir bellek bölgesini donanım düzeyinde destekler.

Tanım 8.1.1

Yığın (*stack*), geçici verileri saklamak için ayrılmış ve son giren ilk çıkar düzeninde kullanılan bir bellek alanıdır. 8086'da yığının yeri SS (yığın kesimi) ve SP (yığın işaretçisi) yazmaçlarıyla belirlenir. Yığına değer koymaya **itme** (*push*), yığından değer almaya **çekme** ya da **alma** (*pop*) denir.

Programcı yığına iki komutla erişir. `PUSH` 16 bitlik bir değeri yığına koyar. `POP` 16 bitlik bir değeri yığından alır ve bir hedefe yazar. Yığın yalnız programcının değil, işlemcinin de kullandığı bir alandır. `CALL` komutu bir prosedürü çağırırken dönüş adresini yığına koyar. Prosedürün sonundaki `RET` bu adresi yığından alır ve oraya döner. `INT` bir kesmeyi çağırdığında benzer bir iş yapılır. Bayrak yazmacı, kod kesimi (CS) ve komut işaretçisi (IP) yığına saklanır; `IRET` ile kesmeden dönülür ve üç değer geri yüklenir. İşletim sisteminin ve BIOS'un kesme yordamları da yazmaçları korumak için aynı yığını kullanır.

Yığının bu kadar çok kullanıcısı olması bir sorumluluk da getirir. Programcının yığına koyduğu her değer, işlemcinin dönüş adresleriyle aynı alanda durur. Programcı bir değeri yığında unutursa ya da fazladan bir değer alırsa, işlemcinin dönüş adresi kayar ve program yanlış bir adrese döner. Yığın güvenliği bu yüzden ayrı bir kesimde ele alınır.

8086'da yığının birimi sözcüktür. Yığına her zaman 2 bayt yazılır ve yığından her zaman 2 bayt okunur. Bu yüzden `PUSH AL` gibi 8 bitlik bir işlenen geçersizdir. Bir baytı saklamak gerekirse onu içeren 16 bitlik yazmaç itilir; `PUSH AX` hem AL'yi hem de AH'yi yığına koyar. Yığının bir yönü de vardır. Yığın yüksek adresten düşük adrese doğru, yani aşağı doğru büyür. Bu iki kural, PUSH ve POP komutlarının bütün ayrıntısını belirler.

Kullanan	Komut	Yığına konan	Geri alan
programcı	`PUSH`, `PUSHF`	yazmaç, kesim yazmacı, bellek sözcüğü, bayraklar	`POP`, `POPF`
işlemci (prosedür)	`CALL`	dönüş adresi (yakın çağrıda IP, uzak çağrıda CS ve IP)	`RET`, `RETF`
işlemci (kesme)	`INT n`, donanım kesmesi	FLAGS, CS ve IP	`IRET`
işletim sistemi, BIOS	kesme yordamları	yordamın kullandığı yazmaçlar	yordamın sonundaki POP'lar

Tablo 8.1.1: 8086'da yığını kullananlar ve yığına koydukları.

Tablo 8.1.1 yığının dört kullanıcısını özetler. Bu notun konusu ilk satırdır: programcının PUSH, POP, PUSHF ve POPF ile yığını kullanması. CALL ve RET'in yığına etkisi yalnız yığının bakış açısından, kısa olarak anılır. Prosedürlerin tanımı, parametre geçirme ve yığın çerçevesi Prosedürler notunun konusudur.

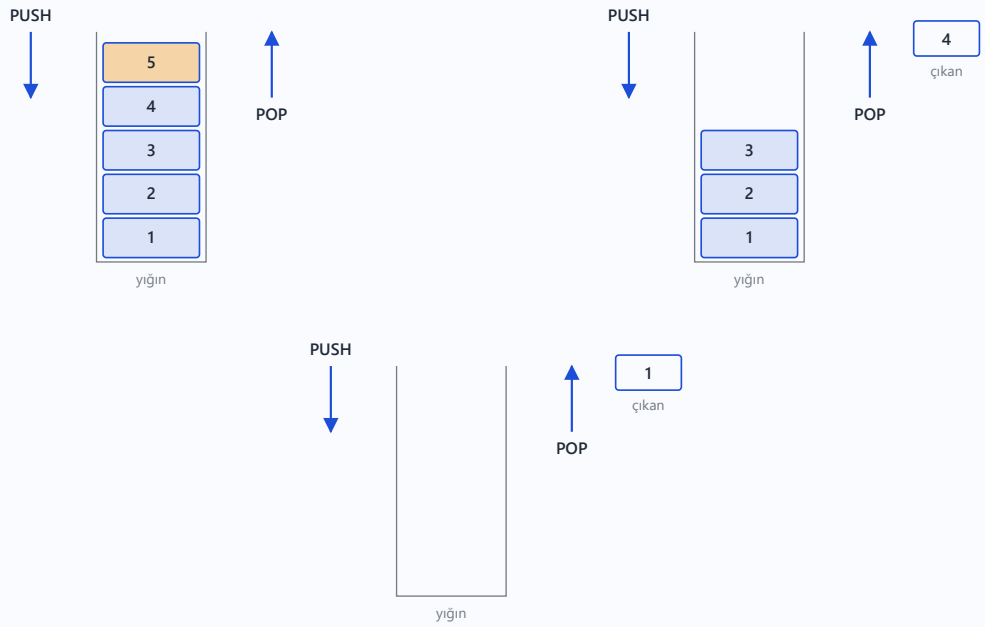
8.1.1.2 Son giren ilk çıkar (LIFO)

Yığın kelimesi günlük hayattaki anlamıyla da bu düzeni iyi anlatır. Masaya üst üste konan tabaklar düşünülebilir. Yeni bir tabak her zaman en üste konur. Bir tabak alınacağı zaman da en üstteki alınır. Ortadaki bir tabağa ulaşmak için önce üstündekilerin hepsi kaldırılmalıdır. En son konan tabak ilk alınır, ilk konan tabak en son alınır.

Tanım 8.1.2

LIFO (*last in, first out*), "son giren ilk çıkar" düzenidir. Bir yapıya eklenen son öğe, yapıdan çıkan ilk öğe olur. Yığın LIFO düzeninde çalışır; yalnız en üstteki öğeye erişilir. En üstteki öğeye **yığının üstü** (*top of stack*) denir.

LIFO'nun tersi FIFO'dur (*first in, first out*): ilk giren ilk çıkar. Bir bankadaki sıra ya da klavye tamponu FIFO düzeninde çalışır. İki düzenin farkı en kolay bir sayı dizisiyle görülür. Yığına sırasıyla 1, 2, 3, 4 ve 5 konsun. İlk alınan değer 5 olur. Ardından 4, 3, 2 alınır ve en son 1 alınır. Aynı değerler bir FIFO kuyruğuna konsaydı 1, 2, 3, 4, 5 sırasıyla çıkardı.



Şekil 8.1.1: LIFO düzeni. Solda 1'den 5'e kadar beş değer itilmiş, 5 yığının üstündedir. Ortada iki POP'tan sonra 5 ve 4 çıkmış, 3 üste gelmiştir. Sağda beş POP'tan sonra yığın boştur; çıkış sırası 5, 4, 3, 2, 1 olmuştur.

Şekil 8.1.1 bu süreci üç anda gösterir. Şekildeki yığın kavramsal bir çizimdir; bellekteki adresler henüz yoktur. Önemli olan, her POP'un o anda en üstte duran değeri almasıdır. Üçüncü POP 3'ü alır, çünkü 5 ile 4 daha önce alınmıştır ve yığında 2 ile 1 kalır.

LIFO düzeninin programlamadaki iki sonucu vardır. Birincisi, yığın değerleri ters sırada geri verir. Bu özellik bilerek kullanılabilir: bir diziyi yığına sırayla itip sonra çekmek, diziyi ters çevirir. İkincisi, birkaç yazmaç saklanıp sonra eski yerlerine geri yüklenecekse POP sırası PUSH sırasının tersi olmalıdır. AX, BX ve CX bu sırayla itildiyse geri yükleme sırası CX, BX, AX'tir. Bu kurala **ayna sırası** denir. Kod alt alta yazıldığında PUSH'lar ile POP'lar bir aynadaki görüntü gibi durur.

> **Sık yapılan hata**
>
> Yazmaçları `PUSH AX`, `PUSH BX` sırasıyla itip `POP AX`, `POP BX` sırasıyla geri almak. Bu dizi yazmaçları korumaz, değerlerini takas eder: BX en son itildiği için ilk POP onu AX'e yazar. Hata derleme sırasında fark edilmez, çünkü dört komut da geçerlidir. Program yalnız yanlış sonuç üretir.

8.1.1.3 Yığın kesimi: SS ve SP

8086'nın belleği kesimlere ayrılmıştır. Her bellek erişiminde 16 bitlik bir kesim değeri ile 16 bitlik bir bağıl konum birleştirilir. Kod için CS ve IP, veri için çoğu zaman DS ve bir adresleme ifadesi kullanılır. Yığın için de ayrılmış bir yazmaç çifti vardır.

Tanım 8.1.3

SS (*stack segment*, yığın kesimi), yığının bulunduğu 64 KB'lık kesimin başlangıcını tutan kesim yazmacıdır. **SP** (*stack pointer*, yığın işaretçisi), yığının üstündeki sözcüğün bu kesim içindeki bağıl konumunu tutan 16 bitlik yazmaçtır. SS:SP her zaman yığının üstünü, yani en son konan sözcüğü gösterir. Yığının üstünün fiziksel adresi SS × 10h + SP'dir.

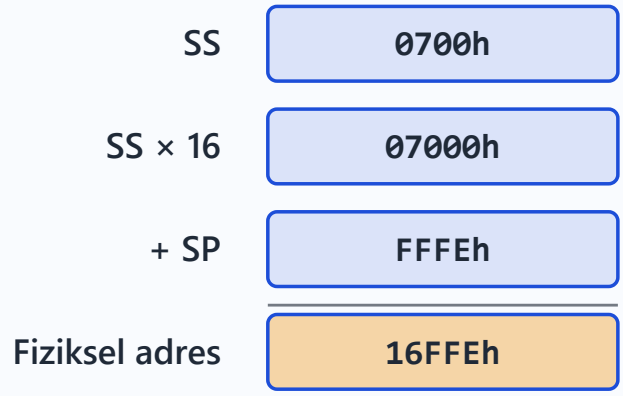
Fiziksel adres hesabı, kod ve veri erişimindeki hesapla aynıdır. Kesim değeri 16 ile çarpılır; onaltılık tabanda bu, sayının sağına bir 0 eklemek demektir. Sonra bağıl konum eklenir ve 20 bitlik adres bulunur.

Örnek 8.1.1

emu8086 bir .COM programını yüklediğinde bütün kesim yazmaçlarını 0700h yapar ve SP'yi FFFEh'ye ayarlar. Yığının üstünün fiziksel adresi şöyle bulunur:

1. Yığın kesimi: SS = 0700h.
2. Kesim değeri 16 ile çarpılır: 0700h × 10h = 07000h.
3. Bağıl konum eklenir: 07000h + FFFEh = 16FFEh.

Yığının üstü 16FFEh fiziksel adresindedir. Toplamada elde oluşur: 7000h + FFFEh = 16FFEh, yani sonuç 16 biti aşar. Adres yolu 20 bit olduğu için bu taşma kaybolmaz; 20 bitlik adreste bit 16 olarak yer alır.



Şekil 8.1.2: Yığının üstünün fiziksel adresi. SS = 0700h 16 ile çarpılır (07000h), SP = FFFEh eklenir ve 20 bitlik 16FFEh adresi bulunur.

SS ve SP'nin başlangıç değerlerini çoğu zaman programcı değil, programı belleğe yükleyen işletim sistemi belirler (Şekil 8.1.2). Bir .COM programında kod, veri ve yığın tek bir 64 KB'lık kesimi paylaşır. Yükleyici SS'yi CS ile aynı yapar ve SP'yi kesimin sonuna, FFFEh'ye ayarlar. Bir .EXE programında ise yığın ayrı bir kesimdir. Kesimin yeri ve SP'nin başlangıç değeri .EXE dosyasının başlığında yazılıdır; yükleyici bu değerleri SS ve SP'ye koyar. Kaynak kodda `.STACK 100h` gibi bir direktif 256 baytlık bir yığın kesimi ayırır. emu8086'da `.stack 100h`, `.data` ve `.code` sırasıyla yazılmış bir .EXE programı SS = 0710h ve SP = 0100h ile başlar. Yığının üstü bu durumda 07100h + 0100h = 07200h fiziksel adresidir; ilk PUSH 071FEh'ye yazar.

Yığına yalnız SP ile erişilmez. BP (taban işaretçisi) de varsayılan olarak SS kesimini kullanır. `MOV AX, [BP+4]` gibi bir komut, kesim öneki yazılmadığı hâlde SS:BP+4 adresini okur. Bu yüzden BP yığının içindeki verilere, örneğin bir prosedüre yığından geçirilen parametrelere erişmek için uygun yazmaçtır. SP ise 8086'da adresleme ifadesinde kullanılamaz: `MOV AX, [SP]` geçersizdir. Yığının üstündeki değeri silmeden okumak için SP önce BX ya da BP'ye kopyalanır.

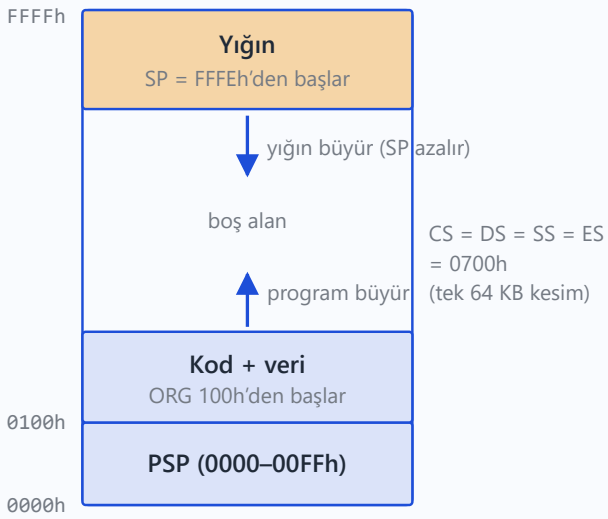
Programın yığını kendisi taşıması da mümkündür. Bunun için SS ve SP'ye yeni değerler yüklenir. Bu iki yükleme arasında bir kesme gelirse, işlemci FLAGS, CS ve IP'yi yarı eski, yarı yeni bir adrese yazar. 8086 bu tehlikeye karşı bir önlem içerir. Bir kesim yazmacına `MOV` ya da `POP` ile yazan komuttan sonra, bir sonraki komut bitene kadar kesme kabul edilmez. Bu yüzden yığın değiştirilirken önce SS, hemen ardından SP yüklenir:

```
MOV  AX, 2000h
MOV  SS, AX        ; kesme bir komut ertelenir
MOV  SP, 0100h     ; SS:SP = 2000:0100, tutarlı
```

İki satırın sırası değiştirilirse koruma işe yaramaz. Erteleme yalnız kesim yazmacına yazan komuttan sonraki komutu kapsar.

8.1.1.4 Yığın aşağı doğru büyür

Bir .COM programında 64 KB'lık tek kesimin içinde üç bölge bulunur. Kesimin ilk 256 baytı (0000h–00FFh) program kesim önekidir (*program segment prefix*, PSP). DOS bu alanı programı yüklerken doldurur; komut satırı ve dönüş bilgileri buradadır. Program, `ORG 100h` direktifinin istediği gibi 0100h adresinden başlar; kod ve veri yukarı doğru yerleşir. Yığın ise kesimin en üstünden, FFFEh'den başlar ve aşağı doğru büyür.



Şekil 8.1.3: .COM programında 64 KB'lık kesimin yerleşimi. PSP en altta, kod ve veri 0100h'den yukarı, yığın FFFEh'den aşağı doğru yerleşir. Aradaki boş alanı ikisi de kullanır.

Bu yerleşimin mantığı Şekil 8.1.3'te açıkça görülür. Program ile yığın kesimin iki ucundan birbirine doğru büyür. Programın ne kadar yer kaplayacağı derleme anında bellidir. Yığının ne kadar derinleşeceği ise çalışma anında, çağrıların ve PUSH'ların sayısına göre belirlenir. İki bölgeyi karşılıklı uçlara koymak, aradaki boş alanın tamamını hangisi gerekirse onun kullanmasını sağlar. Yığın yukarı doğru büyüseydi, programın sonuna sabit bir boyut ayırmak gerekirdi ve bu boyut ya fazla ya da yetersiz kalırdı.

Yığının aşağı doğru büyümesinin komutlara yansıması basittir. Her PUSH, SP'yi 2 azaltır. Her POP, SP'yi 2 artırır. SP'nin değeri küçüldükçe yığın derinleşir. Yığındaki sözcük sayısı da SP'den hesaplanır. Örneğin başlangıçta SP = FFFEh, şu anda SP = FFF8h ise yığına (FFFEh − FFF8h) / 2 = 3 sözcük itilmiştir.

Bir .COM programında yığının ne kadar derinleşebileceği de hesaplanabilir. Yığınla diziyi ters çeviren örnek program 26 bayttır ve 0100h–0119h adreslerini kaplar. Programdan sonraki ilk boş bayt 011Ah'dir. Yığın FFFEh'den başlar; n PUSH'tan sonra SP = FFFEh − 2n olur. Yığının programın üzerine yazmaması için SP ≥ 011Ah kalmalıdır. Buradan n ≤ (FFFEh − 011Ah) / 2 = (65.534 − 282) / 2 = 32.626 bulunur. Bu programda yığına 32.626 sözcüğe kadar değer konabilir. Pratikte bu sınıra pek yaklaşılmaz, ama sonsuz bir döngü ya da sonsuz özyineleme bu sınırı kısa sürede aşar.

Sık yapılan hata

"Yığının üstü" ifadesini "en yüksek adres" olarak anlamak. Yığının üstü, en son konan sözcüğün bulunduğu ve SP'nin gösterdiği yerdir. Yığın aşağı doğru büyüdüğü için bu yer, yığındaki öğelerin *en düşük* adreslisidir. Şekillerde küçük adresler yukarıya çizildiğinde üst kelimesi görsel olarak da doğru olur; bu notun yığın şekilleri bu düzeni kullanır.

8.1.1.5 Yığın erişimi donanımda

PUSH ve POP, yazılım açısından tek bir komuttur. Donanım açısından ise 8086'nın iki birimi birlikte çalışır. Yürütme birimi (*execution unit*, EU) komutu çözer ve SP'yi kendi aritmetik-mantık biriminde 2 azaltır ya da artırır. Yol arayüz birimi (*bus interface unit*, BIU) ise fiziksel adresi hesaplar ve belleğe erişir. BIU'daki adres toplayıcı SS'yi 4 bit sola kaydırıp SP'yi ekler. Bulunan 20 bitlik adres, bir bellek yazma (PUSH) ya da okuma (POP) yol döngüsüyle adres/veri hatlarına çıkarılır.

8086'nın veri yolu 16 bittir. Bir sözcük çift bir adreste duruyorsa tek yol döngüsüyle, yani dört saat turunda (T1–T4) taşınır. Bu çevrimde A0 = 0 ve BHE = 0 olur; düşük bayt D0–D7, yüksek bayt D8–D15 hatlarından geçer. SP çift olduğu sürece her PUSH tek bir yazma çevrimi, her POP tek bir okuma çevrimidir. SP tek bir değere düşerse sözcük iki ayrı çevrimle taşınır ve her erişim dört saat turu uzar. Yükleyiciler bu yüzden SP'yi hep çift bir değerle başlatır ve programlar SP'yi tek sayı kadar değiştirmez.

Yığın erişimini dış devreye bildiren bir ayrıntı daha vardır. 8086, her yol döngüsünün T2–T4 aralığında A16/S3–A19/S6 hatlarına durum bitleri çıkarır. S4 ve S3 bitleri o çevrimde hangi kesim yazmacının kullanıldığını söyler. S4 = 0 ve S3 = 1 ise o çevrim SS kesimine, yani yığına yapılan bir erişimdir. Bir tasarımcı bu bitleri çözerek yığın erişimlerini ayrı bir bellek bankasına yönlendirebilir ve böylece 1 MB'tan geniş bir bellek alanı kullanabilir. Sıradan PC'lerde bu bitler kullanılmaz, ama mantık analizöründe bir programın yığına ne zaman eriştiğini görmek için işe yararlar.

S4	S3	Kullanılan kesim	Örnek erişim
0	0	ES (ek kesim)	dizgi komutlarının hedefi, ES önekli erişim
0	1	SS (yığın kesimi)	PUSH, POP, CALL, RET, BP tabanlı erişim
1	0	CS ya da kesim yok	komut getirme, G/Ç ve kesme onayı çevrimleri
1	1	DS (veri kesimi)	sıradan değişken erişimi

Tablo 8.1.2: 8086'nın S4 ve S3 durum bitleri (yol döngüsünün T2–T4 aralığında).

8088'de durum farklıdır. 8088'in dış veri yolu 8 bittir; her sözcük iki ayrı bayt çevrimiyle taşınır. Bu yüzden 8088'de her PUSH ve POP, SP çift olsa bile iki bellek çevrimi sürer. Programlama açısından iki işlemci aynıdır; yalnız süreler değişir. Bu notun bütün örnekleri 8086 içindir.

8.1.2 PUSH ve POP

8.1.2.1 PUSH komutu

`PUSH kaynak` komutu 16 bitlik kaynağın bir kopyasını yığının üstüne koyar. Kaynak değişmez; yığında onun bir kopyası oluşur. Kaynak dört türden olabilir: genel amaçlı ya da işaretçi yazmacı, kesim yazmacı, bellekteki bir sözcük ve 80186'dan itibaren anlık değer. Tablo 8.1.3 bu türleri örnekleriyle verir.

Biçim	Geçerli işlenenler	Örnek
PUSH yazmaç	AX, BX, CX, DX, SI, DI, BP, SP	`PUSH AX`
PUSH kesim yazmacı	CS, DS, SS, ES	`PUSH DS`
PUSH bellek	`[BX]`, `[BX+SI+7]`, 16 bitlik değişken	`PUSH [BX]`, `PUSH sayi`
PUSH anlık değer	5, –24, 3Fh, 10001101b	`PUSH 3Fh` (8086'da yok)

Tablo 8.1.3: PUSH komutunun işlenen türleri.

Tablonun son satırı bir uyarı taşır. Anlık değerle PUSH, 8086 ve 8088'in komut kümesinde yoktur; 80186 ile gelmiştir. emu8086 bu komutu kabul eder ve çalıştırır. Ama gerçek bir 8086 için yazılan programda bir sabit önce bir yazmaca yüklenir, sonra yazmaç itilir:

```
MOV  AX, 5
PUSH AX          ; 8086 için PUSH 5 yerine
```

Bellek işlenenli PUSH'ta işlenenin boyutu sözcük olmalıdır. `sayi DW 0BEEFh` olarak tanımlanmış bir değişken için `PUSH sayi` geçerlidir; değişkenin iki baytı yığına kopyalanır. `DB` ile tanımlanmış bir bayt değişkeni ise itilemez. Aynı kural yazmaçlar için de geçerlidir: `PUSH AL` ve `PUSH byte_degisken` derleme hatası verir. Bayrak yazmacı ise sıradan bir işlenen olarak yazılamaz; onun için ayrı bir komut, `PUSHF` vardır.

SP'nin kendisini itmek ilginç bir ayrıntı taşır. `PUSH SP` komutu SP'yi hem azaltır hem de yığına yazar; hangisinin önce olduğu sonucu değiştirir. 8086 önce SP'yi azaltır, sonra *azaltılmış* değeri yazar. SP = FFFEh iken `PUSH SP` yığına FFFCh koyar. 80286 ve sonraki işlemciler ise azaltmadan önceki değeri, FFFEh'yi yazar. Bu fark, eski programlarda işlemcinin 8086 mı yoksa 80286 mı olduğunu anlamak için kullanılmıştır. Taşınabilir bir programda `PUSH SP`'ye güvenilmez; SP'nin değeri gerekiyorsa önce başka bir yazmaca kopyalanır.

> **Sık yapılan hata**
>
> emu8086'da `PUSH 5` çalıştı diye bu komutu 8086 komutu sanmak. emu8086 bazı 80186 komutlarını kabul eder ve yalnız uyarı verir. Aynı kod gerçek bir 8086 ya da 8088 üzerinde çalıştırıldığında işlemci bu baytları başka bir komut gibi yorumlar ve program beklenmedik biçimde davranır.

8.1.2.2 POP komutu

`POP hedef` komutu yığının üstündeki sözcüğü okur, hedefe yazar ve yığının üstünü bir sözcük yukarı taşır. Hedef türleri PUSH'un kaynak türlerine benzer, ama iki eksik vardır (Tablo 8.1.4).

Biçim	Geçerli işlenenler	Örnek
POP yazmaç	AX, BX, CX, DX, SI, DI, BP, SP	`POP BX`

Biçim	Geçerli işlenenler	Örnek
POP kesim yazmacı	DS, SS, ES (CS hariç)	`POP DS`
POP bellek	`[BX]`, `[BX+SI+7]`, 16 bitlik değişken	`POP sonuc`

Tablo 8.1.4: POP komutunun işlenen türleri.

Neden POP CS yok? CS, komutların getirildiği kesimi belirler. `POP CS` CS'yi değiştirip IP'yi olduğu gibi bıraksaydı, bir sonraki komut bambaşka bir kesimin aynı bağıl konumundan getirilirdi. Program akışı rastgele bir yere atlardı. Bu yüzden CS yalnız IP ile birlikte değişir: uzak `JMP` ve `CALL`, `RETF`, `INT` ve `IRET` CS'yi ve IP'yi aynı komutta yükler. Assembler'lar `POP CS` yazımını kabul etmez. İlk 8086'larda bu komutun kodu (0Fh) belgelenmemiş biçimde çalışıyordu; Intel bu kodu 80286'da iki baytlık yeni komutların öneki olarak kullandı.

Neden POP sabit yok? `POP 5` anlamsızdır. POP yığından bir değer alır ve bu değerin yazılacağı bir yer gerekir. Bir sabit, değer yazılabilecek bir yer değildir. Yığının üstündeki değer gerekmiyorsa ve yalnız atılmak isteniyorsa iki yol vardır. Değer kullanılmayan bir yazmaca alınır ya da SP'ye doğrudan 2 eklenir (`ADD SP, 2`). İkinci yol bayrakları değiştirir; birincisi değiştirmez.

Bayraklar. PUSH ve POP bayraklara dokunmaz. `POP AX` ile AX'e 0000h gelse bile ZF değişmez; ZF'yi değiştiren bir aritmetik ya da mantık işlemidir, veri taşıma değil. Bu kuralın tek istisnası `POPF`'tir. POPF yığından aldığı sözcüğü bayrak yazmacının tamamına yükler.

Kesim yazmaçları arasında kopyalama. 8086'da bir kesim yazmacından diğerine doğrudan `MOV` yoktur; `MOV DS, CS` geçersizdir. Bu kopyalama iki yolla yapılır. Ya değer genel amaçlı bir yazmaç üzerinden aktarılır (`MOV AX, CS` ve `MOV DS, AX`) ya da yığın kullanılır:

```
PUSH CS          ; CS'nin değeri yığına
POP  DS          ; DS ← CS
PUSH DS          ; DS'nin değeri yığına
POP  ES          ; ES ← DS
```

Yığınla kopyalamanın üstünlüğü, hiçbir genel amaçlı yazmacı bozmamasıdır. AX'in değeri korunmak isteniyorsa ilk yol AX'i ayrıca saklamayı gerektirir; ikinci yol gerektirmez. `PUSH CS` ile `POP DS` deyimi, verisi kod kesiminde duran programlarda ve kesme yordamlarının başında sık görülür. emu8086'da DS = 1234h yapılıp bu dört komut çalıştırıldığında ES = 1234h ve DS = 0700h olur; SP yeniden FFFEh'ye döner.

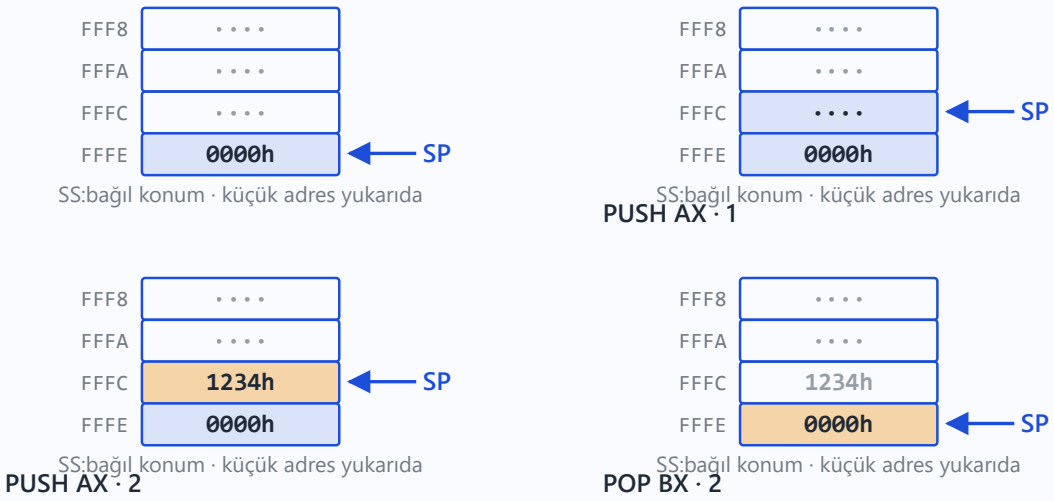
8.1.2.3 PUSH ve POP nasıl çalışır?

Her iki komut da iki adımdan oluşur. Adımların sırası, SP'nin her an neyi gösterdiğini belirler.

Tanım 8.1.4

PUSH kaynak: (1) SP ← SP − 2; (2) [SS:SP] ← kaynak. **POP hedef**: (1) hedef ← [SS:SP]; (2) SP ← SP + 2. PUSH önce azaltır sonra yazar; POP önce okur sonra artırır. Böylece SP her an dolu olan en üst hücreyi gösterir.

Bu sıralamaya “dolu yığın” düzeni denir. SP'nin gösterdiği hücrede her zaman geçerli bir değer vardır. Bazı işlemciler ters düzeni kullanır: SP bir sonraki boş hücreyi gösterir, PUSH önce yazar sonra azaltır. Hangi düzen seçilirse seçilsin, PUSH ile POP birbirinin tam tersi olmalıdır. 8086'da PUSH “azalt, yaz”, POP ise “oku, artır” biçimindedir.



Şekil 8.1.4: AX = 1234h iken PUSH AX ve POP BX. Sol üstte başlangıç: SP = FFFEh, orada DOS'un koyduğu 0000h vardır. Sağ üstte PUSH'un birinci adımı: SP = FFFCh, yeni hücre henüz yazılmamıştır. Sol altta ikinci adım: [FFFCh] ← 1234h. Sağ altta POP BX bitmiştir: BX = 1234h, SP = FFFEh; FFFCh'deki 1234h silinmemiş, yalnız yığının dışında kalmıştır.

Şekil 8.1.4 iki komutu dört anda gösterir. Başlangıçta SP = FFFEh'dir ve AX = 1234h'dir. FFFEh adresinde DOS'un .COM programı için koyduğu 0000h sözcüğü durur; bu sözcüğün görevi program sonunda anlatılır.

1. **PUSH AX, adım 1.** SP ← FFFEh − 2 = FFFCh. Yığının üstü bir sözcük aşağı iner. FFFCh'deki hücrede şimdilik eski, anlamsız bir değer vardır.
2. **PUSH AX, adım 2.** [SS:FFFCh] ← 1234h. Yeni üst hücreye AX'in kopyası yazılır. AX değişmez.
3. **POP BX, adım 1.** BX ← [SS:FFFCh] = 1234h. Değer yığının üstünden okunur.
4. **POP BX, adım 2.** SP ← FFFCh + 2 = FFFEh. Yığının üstü bir sözcük yukarı çıkar.

Son adımın önemli bir sonucu vardır. POP, okuduğu değeri bellekten silmez. FFFCh'deki 1234h orada durmaya devam eder; yalnız artık SP'nin üstünde, yani yığının dışında kalmıştır. Şekilde bu hücre soluk çizilmiştir. Bir sonraki PUSH aynı hücrenin üzerine yazar. Bu yüzden yığından alınmış bir değe-

re bellekten yeniden erişmeye çalışmak güvenilir değildir. Arada bir kesme gelirse, işlemci FLAGS, CS ve IP'yi tam o hücrelere yazar ve eski değer kaybolur.

Sık yapılan hata

POP'tan sonra değerin bellekten silindiğini, yerine 0 yazıldığını düşünmek. POP yalnız okur ve SP'yi artırır. emu8086'nın bellek penceresinde eski değerler görünmeye devam eder. Bu değerler "yığında" değildir; yığın yalnız SP'den FFFEh'ye kadar olan bölgedir.

8.1.2.4 Yığında bayt sırası

Yığın sözcüklerle çalışır, bellek ise baytlarla adreslenir. Bir sözcük belleğe yazılırken iki baytına iki ayrı adres verilir. 8086 bu işte little-endian (*little-endian*) düzeni kullanır: sözcüğün düşük baytı düşük adrese, yüksek baytı bir sonraki adrese yazılır. Kural yığın için de aynıdır.



Şekil 8.1.5: PUSH AX (AX = 1234h) sonrasında yığın. Solda sözcük görünümü: SS:FFFC'de 1234h. Sağda bayt görünümü: [FFFCh] = 34h (AL), [FFFDh] = 12h (AH); FFFEh ve FFFFh'de DOS'un 0000h sözcüğü.

AX = 1234h iken `PUSH AX` çalıştığında SP = FFFCh olur. Sözcüğün düşük baytı 34h (AL) FFFCh adresine, yüksek baytı 12h (AH) FFFDh adresine yazılır (Şekil 8.1.5). SP her zaman sözcüğün düşük baytını, yani daha küçük adresi gösterir. emu8086'nın yığın penceresi sözcükleri gösterir ve bu satırda 1234h yazar. Bellek penceresi ise baytları sırayla gösterir: önce 34, sonra 12. İki pencere aynı veriyi farklı biçimde sunar. Bellek penceresinde değerin "ters" görünmesi bir hata değil, little-endian düzenin doğal sonucudur.

Bu düzen bir programla da sınanabilir. Yığının üstündeki baytları okumak için SP bir taban yazmacına kopyalanır, çünkü 8086'da `[SP]` adresleme ifadesi yoktur:

```
ORG 100h
MOV  AX, 1234h
PUSH AX
MOV  BX, SP        ; BX = FFFCh
MOV  CL, [BX]      ; CL = 34h, düşük bayt
MOV  CH, [BX+1]    ; CH = 12h, yüksek bayt
POP  DX            ; DX = 1234h
```

Program şöyle çalışır. `PUSH AX` SP'yi FFFCh yapar ve 34h ile 12h baytlarını FFFCh ve FFFDh'ye yazar. `MOV BX, SP` BX'i FFFCh yapar. `[BX]` varsayılan olarak DS kesimini kullanır. .COM programında DS = SS olduğu için DS:FFFCh, yığının üstüyle aynı fiziksel adrestir: 07000h + FFFCh = 16FFCh. `MOV CL, [BX]` buradan 34h'yi, `MOV CH, [BX+1]` 16FFDh'den 12h'yi okur. CX = 1234h olur. Son olarak `POP DX` iki baytı yeniden sözcük olarak okur: DX = 1234h. Bir .EXE programında DS ile SS farklı olabilir. O zaman `[BX]` yığını değil veri kesimini okur; bu durumda `SS:[BX]` yazmak ya da BX yerine BP kullanmak gerekir.

SP'nin çift kalması da bayt düzeyindeki bu yapıyla ilgilidir. SP çift olduğu sürece her yığın sözcüğü çift bir adreste başlar ve 8086 onu tek yol döngüsüyle taşır. SP tek olsaydı sözcüğün düşük baytı tek, yüksek baytı çift adrese düşer ve iki ayrı çevrim gerekirdi. Program yine doğru çalışır, yalnız yavaşlar.

8.1.2.5 Ne değişir, ne değişmez?

Yığını kullanan bütün komutlar SP'yi değiştirir. Belleğe yazıp yazmadıkları, bir hedefi değiştirip değiştirmedikleri ve bayraklara etkileri ise farklıdır. Tablo 8.1.5 bu etkileri yan yana verir.

Komut	SP	Bellek	Hedef	Bayraklar
`PUSH kaynak`	−2	[SS:SP] ← kaynak	—	değişmez
`POP hedef`	+2	okunur, silinmez	hedef ← [SS:SP]	değişmez
`PUSHF`	−2	[SS:SP] ← FLAGS	—	değişmez
`POPF`	+2	okunur	FLAGS ← [SS:SP]	hepsi yüklenir
`CALL` (yakın)	−2	[SS:SP] ← dönüş IP'si	IP ← hedef	değişmez
`RET` (yakın)	+2	okunur	IP ← [SS:SP]	değişmez
`INT n`	−6	FLAGS, CS, IP	CS:IP ← vektör	IF = TF = 0
`IRET`	+6	okunur	IP, CS, FLAGS	hepsi yüklenir

Tablo 8.1.5: Yığın komutlarının SP'ye, belleğe, hedefe ve bayraklara etkisi.

Tablonun en önemli sütunu sonuncusudur. PUSH, POP ve PUSHF bayraklara dokunmaz. Bu sayede bir karşılaştırmanın sonucu, yani bayraklar, PUSH ve POP arasında güvenle taşınabilir. Aşağıdaki kodda `CMP` ZF'yi 1 yapar. Ardından AX yığına konur, değiştirilir ve geri alınır. Bu üç komut ZF'ye dokunmadığı için `JE` yine atlar:

```
MOV  AX, 5
CMP  AX, 5          ; ZF = 1
PUSH AX
MOV  AX, 7          ; MOV bayrakları değiştirmez
POP  AX             ; AX = 5, ZF hâlâ 1
JE   esit           ; atlar
```

Arada bayrakları değiştiren bir komut, örneğin bir `ADD` olsaydı, karşılaştırmanın sonucu kaybolurdu. O durumda bayrakları saklamak için `PUSHF` ve `POPF` kullanılır. CALL ve RET'in yığın etkisi PUSH ve POP'unkine çok benzer: CALL dönüş adresini iter, RET onu çeker. Fark, hedefin bir yazmaç değil IP olmasıdır. INT ise üç sözcük, yani 6 bayt iter ve bu komutlar arasında bayrakları değiştiren tek itme komutudur: kesme yordamına girerken IF ve TF sıfırlanır.

8.1.2.6 Makine kodları ve saat turları

Yığın komutları 8086'nın en kısa komutları arasındadır. Bir yazmacı itmek ya da çekmek tek bayt tutar. İşlem kodunun beş biti komutu, kalan üç biti yazmacı belirler. `PUSH` için kod 50h + r, `POP` için 58h + r biçimindedir. Yazmaç numaraları 8086'nın genel sırasını izler: AX = 0, CX = 1, DX = 2, BX = 3, SP = 4, BP = 5, SI = 6, DI = 7. Bu yüzden `PUSH AX` = 50h, `PUSH BX` = 53h, `POP BX` = 5Bh ve `POP DI` = 5Fh'dir.

Kesim yazmaçları da tek bayttır. İtme kodu 06h + 8 × s, çekme kodu 07h + 8 × s biçimindedir; s, ES = 0, CS = 1, SS = 2, DS = 3 numarasıdır. `PUSH ES` = 06h, `PUSH CS` = 0Eh, `PUSH SS` = 16h, `PUSH DS` = 1Eh; `POP ES` = 07h, `POP SS` = 17h, `POP DS` = 1Fh'dir. Bu düzende `POP CS` 'nin kodu 0Fh olurdu; bu kod POP CS ile birlikte kullanım dışı kaldı. Bellek işlenenli biçimler ModR/M baytı kullanır: `PUSH bellek` = FFh /6, `POP bellek` = 8Fh /0. Örneğin 0109h adresindeki bir değişken için `PUSH sayi` = FF 36 09 01, dört bayttır. `PUSHF` = 9Ch ve `POPF` = 9Dh'dir.

Komut	Kod	Bayt	Saat turu
`PUSH yazmaç`	50h + r	1	11
`PUSH kesim`	06h + 8s	1	10
`PUSH bellek`	FFh /6	2–4	16 + EA
`POP yazmaç`	58h + r	1	8
`POP kesim`	07h + 8s	1	8
`POP bellek`	8Fh /0	2–4	17 + EA
`PUSHF`	9Ch	1	10
`POPF`	9Dh	1	8

Tablo 8.1.6: Yığın komutlarının kodları ve 8086 saat turları (Intel 8086 kılavuzu; EA = etkin adres hesabının süresi).

Tablo 8.1.6'daki süreler sözcüklerin çift adreste durduğu durum içindir. Bir PUSH 11 saat turu sürer; 5 MHz'lik bir 8086'da bir saat turu 200 ns olduğundan bu 2,2 µs eder. Bu sürenin dört turu bellek yazma çevrimidir; geri kalanı komutun çözülmesi ve SP'nin azaltılmasıdır. Bellek işlenenli biçimlerde iki bellek erişimi vardır: biri değişkene, biri yığına. Süre bu yüzden daha uzundur ve etkin adres hesabının süresi de eklenir. 8088'de her sözcük erişimine dört tur daha eklenir.

8.1.3 Örnekler: kod izleme

Bu kesimdeki programların hepsi emu8086'da .COM programı olarak çalıştırılmıştır. Başlangıçta CS = DS = SS = ES = 0700h, IP = 0100h ve SP = FFFEh'dir; aksi belirtilmedikçe diğer yazmaçlar 0000h'dir. İzleme tablolarında her satır, o komut yürütüldükten *sonraki* durumu verir. O satırda değişen değerler vurgulanmıştır. Yığın sütunu yalnız SP'nin gösterdiği ve onun altındaki dolu hücreleri gösterir; FFFEh'deki 0000h, DOS'un .COM programı için koyduğu sözcüktür.

8.1.3.1 İlk örnek: PUSH AX / POP BX

İlk program bir değeri AX'ten BX'e yığın üzerinden kopyalar. Bu iş bir `MOV BX, AX` ile de yapılabilirdi; amaç yığının işleyişini en sade biçimde görmektir.

```
ORG   100h
MOV   AX, 42      ; AX'e 42 ata
PUSH  AX          ; AX'i yığına koy
POP   BX          ; yığından BX'e al
RET               ; işletim sistemine dön
END
```

Program dört komuttan ve 6 bayttan oluşur: `MOV AX, 42` 3 bayt (B8 2A 00, 0100h–0102h), `PUSH AX` 1 bayt (50, 0103h), `POP BX` 1 bayt (5B, 0104h) ve `RET` 1 bayttır (C3, 0105h).

`MOV AX, 42` satırında sayının sonunda h yoktur. Assembler onu onluk sayı olarak okur: 42 = 2 × 16 + 10 = 2Ah. AX = 002Ah olur, 0042h değil. Bu küçük ayrıntı emu8086'da yeni başlayanların en sık şaşırdığı noktalardan biridir. `PUSH AX` SP'yi FFFEh − 2 = FFFCh yapar ve [FFFCh] ← 002Ah yazar; bayt düzeyinde [FFFCh] = 2Ah, [FFFDh] = 00h olur. `POP BX` FFFCh'deki 002Ah'yi BX'e alır ve SP'yi FFFEh'ye geri getirir. Değer AX'ten BX'e yığın üzerinden kopyalanmıştır; AX de değerini korur.

Komut	IP (sonraki)	AX	BX	SP	Yığın (üstten)
başlangıç	0100h	0000h	0000h	FFFEh	0000h
`MOV AX, 42`	0103h	**002Ah**	0000h	FFFEh	0000h
`PUSH AX`	0104h	002Ah	0000h	**FFFCh**	**002Ah, 0000h**
`POP BX`	0105h	002Ah	**002Ah**	**FFFEh**	0000h

Komut	IP (sonraki)	AX	BX	SP	Yığın (üstten)
`RET`	**0000h**	002Ah	002Ah	**0000h**	(boş)

Tablo 8.1.7: PUSH AX / POP BX programının izlemesi.

Son satır yığının programın sonundaki rolünü gösterir (Tablo 8.1.7). `RET` yığının üstündeki sözcüğü dönüş adresi olarak alır. Yığında yalnız DOS'un koyduğu 0000h vardır; IP = 0000h ve SP = FFFEh + 2 = 0000h olur. SP 16 bitte sarar. İşlemci 0700:0000 adresine, yani PSP'nin başına gider. PSP'nin ilk iki baytı CD 20'dir; bu, programı sonlandıran `INT 20h` komutudur. Program böylece düzgün biter. Bu düzenek ancak yığın dengeliyse çalışır: RET'e gelindiğinde yığının üstünde DOS'un 0000h'si durmalıdır.

8.1.3.2 LIFO sırası: 1, 2, 3 girer, 3, 2, 1 çıkar

İkinci program LIFO düzenini gerçek yazmaçlarla gösterir. AX'e sırayla 1, 2 ve 3 yüklenir ve her değer yığına itilir. Ardından üç kez `POP BX` yapılır.

```
MOV     AX, 1       ; AX'e 1 ata
PUSH    AX          ; 1'i yığına koy
MOV     AX, 2       ; AX'e 2 ata
PUSH    AX          ; 2'yi yığına koy
MOV     AX, 3       ; AX'e 3 ata
PUSH    AX          ; 3'ü yığına koy
POP     BX          ; BX ← 3
POP     BX          ; BX ← 2
POP     BX          ; BX ← 1
```

Komut	AX	BX	SP	Yığın (üstten, DOS sözcüğü hariç)
başlangıç	0000h	0000h	FFFEh	(boş)
`MOV AX, 1`	**0001h**	0000h	FFFEh	(boş)
`PUSH AX`	0001h	0000h	**FFFCh**	**0001h**
`MOV AX, 2`	**0002h**	0000h	FFFCh	0001h
`PUSH AX`	0002h	0000h	**FFFAh**	**0002h, 0001h**
`MOV AX, 3`	**0003h**	0000h	FFFAh	0002h, 0001h
`PUSH AX`	0003h	0000h	**FFF8h**	**0003h, 0002h, 0001h**
`POP BX`	0003h	**0003h**	**FFFAh**	0002h, 0001h
`POP BX`	0003h	**0002h**	**FFFCh**	0001h
`POP BX`	0003h	**0001h**	**FFFEh**	(boş)

Tablo 8.1.8: LIFO sırası programının izlemesi.

İzlemede iki nokta dikkat çeker (Tablo 8.1.8). Birincisi, AX'e yeni bir değer yüklendiğinde eski değer kaybolmaz; bir önceki PUSH onu yığına kopyalamıştır. AX = 0002h olduğunda 0001h yığında güvendedir. İkincisi, BX'e gelen değerler giriş sırasının tersidir: önce 3, sonra 2, sonra 1. Üç PUSH ve üç POP yapıldığı için SP başladığı yere, FFFEh'ye döner. Programın sonunda AX = 0003h kalır, çünkü AX'e son yüklenen değer 3'tür ve POP'lar yalnız BX'i değiştirir.

8.1.3.3 Bir yazmacın değerini saklamak

Yığının en yaygın kullanımı bir yazmacı geçici olarak başka bir iş için kullanıp sonra eski değerine döndürmektir. Program önce AX'e 1234h yükler. Sonra AX'i saklar, AX'e başka bir değer yazar ve saklanan değeri geri yükler.

```
ORG   100h
MOV   AX, 1234h
PUSH  AX        ; AX'in değerini sakla
MOV   AX, 5678h ; AX'i değiştir
POP   AX        ; eski değeri geri yükle
RET
END
```

Komut	AX	SP	[FFFCh]	Açıklama
`MOV AX, 1234h`	**1234h**	FFFEh	—	korunacak değer
`PUSH AX`	1234h	**FFFCh**	**1234h**	AX'in kopyası yığında
`MOV AX, 5678h`	**5678h**	FFFCh	1234h	AX başka bir iş için kullanılıyor
`POP AX`	**1234h**	**FFFEh**	1234h	eski değer geri geldi; hücre silinmedi
`RET`	1234h	**0000h**	1234h	IP ← 0000h, PSP'deki INT 20h

Tablo 8.1.9: Değer saklama programının izlemesi.

Program 9 bayttır: iki üç baytlık MOV, birer baytlık PUSH, POP ve RET. Tablo 8.1.9'un dördüncü sütunu POP'un belleği silmediğini bir kez daha gösterir: POP'tan sonra da [FFFCh] = 1234h'dir. Bu desen, prosedürlerin girişinde kullandıkları yazmaçları PUSH ve çıkışta POP etmesinin aynısıdır. Bir prosedür AX'i bozacaksa girişte `PUSH AX`, çıkışta `POP AX` yazar. Prosedürü çağıran program AX'in değiştiğini fark etmez. Bu yönteme yazmaçların korunması denir.

Örnek 8.1.2

AX'te bir döngünün sayacı, BX'te bir dizinin adresi duruyor. Döngünün ortasında ekrana bir karakter yazmak için AH = 02h ve DL = karakter yüklenip `INT 21h` çağrılacaktır. INT 21h'nin 02h işlevi AL'yi de değiştirir: işlev bittiğinde AL, yazılan karakterdir. AX'in iki yarısı da bozulacağı için yazdırma şöyle sarılır:

```
PUSH AX          ; sayacı sakla
PUSH DX          ; DX de bozulacak
MOV  AH, 2
MOV  DL, '*'
INT  21h         ; AL = '*' olur
POP  DX          ; ayna sırası: önce DX
POP  AX          ; sayaç geri geldi
```

PUSH sırası AX, DX olduğu için POP sırası DX, AX'tir. BX'e dokunulmadığı için saklanmaz. Saklanan her yazmaç iki komut ve 19 saat turu (11 + 8) tutar; yalnız gerçekten bozulan yazmaçlar saklanır.

8.1.3.4 Yığınla iki yazmacı takas etmek

Ayna sırasının tersi, yani aynı sırayla POP, değerleri takas eder. Bu bazen bir hata, bazen de bilerek kullanılan bir yöntemdir.

```
ORG   100h
MOV   AX, 1212h  ; AX ← 1212h
MOV   BX, 3434h  ; BX ← 3434h
PUSH  AX         ; AX'i yığına koy
PUSH  BX         ; BX'i yığına koy
POP   AX         ; AX ← BX'in eski değeri
POP   BX         ; BX ← AX'in eski değeri
RET
END
```

emu8086 bu programı yüklediğinde CX'te 000Bh görülür. Bu değer programın bayt sayısıdır: iki MOV 3'er bayt, dört yığın komutu ve RET birer bayt, toplam 3 + 3 + 1 + 1 + 1 + 1 + 1 = 11 bayt. emu8086 yükleme sonrasında CX'e .COM programının boyutunu yazar; DOS'un DEBUG programı da yüklediği dosyanın uzunluğunu BX:CX'e yazar. Bu bir gelenektir, bir kural değildir. Program CX'in başlangıç değerine güvenmemeli, CX'i kullanmadan önce kendisi yüklemelidir. Bu notun diğer izlemelerinde CX'in başlangıç değeri 0000h alınmıştır; takas programı CX'i kullanmadığı için sonuç değişmez.

Komut	AX	BX	SP	Yığın (üstten)
başlangıç	0000h	0000h	FFFEh	0000h
`MOV AX, 1212h`	**1212h**	0000h	FFFEh	0000h
`MOV BX, 3434h`	1212h	**3434h**	FFFEh	0000h
`PUSH AX`	1212h	3434h	**FFFCh**	**1212h, 0000h**
`PUSH BX`	1212h	3434h	**FFFAh**	**3434h, 1212h, 0000h**
`POP AX`	**3434h**	3434h	**FFFCh**	1212h, 0000h
`POP BX`	3434h	**1212h**	**FFFEh**	0000h

Komut	AX	BX	SP	Yığın (üstten)
`RET`	3434h	1212h	**0000h**	(boş); IP = 0000h

Tablo 8.1.10: Takas programının izlemesi (CX = 000Bh baştan sona değişmez).

Takasın sırrı LIFO'dur (Tablo 8.1.10). BX en son itildiği için yığının üstündedir ve ilk `POP AX` onu alır. Ardından `POP BX` sıradaki değeri, yani AX'in eski değeri 1212h'yi alır. Bir an için AX ile BX'in ikisi de 3434h olur; yığın bu sırada 1212h'yi güvenle tutar. Ayna sırasıyla POP edilseydi, yani önce `POP BX` sonra `POP AX` yazılsaydı, değerler yerinde kalırdı.

8.1.3.5 Üç yazmaç: aynı sırayla POP

Üç yazmaçla aynı deney yapıldığında sonuç daha ilginçtir. Bu program sunumda bir .BIN dosyası olarak yüklenmiştir. emu8086 .BIN dosyasını CS = SS = 0100h ve IP = 0000h ile başlatır; SP yine FFFEh'dir. Kesim değeri farklı olduğu için fiziksel adresler değişir, ama bağıl konumlar ve SP değerleri .COM programıyla aynıdır.

```
mov ax, 2
mov bx, 4
mov cx, 8
push ax
push bx
push cx
pop  ax
pop  bx
pop  cx
```

Komut	AX	BX	CX	SP	Yığın (üstten)
`mov ax, 2`	**0002h**	0000h	0000h	FFFEh	(boş)
`mov bx, 4`	0002h	**0004h**	0000h	FFFEh	(boş)
`mov cx, 8`	0002h	0004h	**0008h**	FFFEh	(boş)
`push ax`	0002h	0004h	0008h	**FFFCh**	**0002h**
`push bx`	0002h	0004h	0008h	**FFFAh**	**0004h, 0002h**
`push cx`	0002h	0004h	0008h	**FFF8h**	**0008h, 0004h, 0002h**
`pop ax`	**0008h**	0004h	0008h	**FFFAh**	0004h, 0002h
`pop bx`	0008h	0004h	0008h	**FFFCh**	0002h
`pop cx`	0008h	0004h	**0002h**	**FFFEh**	(boş)

Tablo 8.1.11: Üç yazmaçlı yığın programının izlemesi.

Sonuçta AX = 0008h, BX = 0004h ve CX = 0002h olur (Tablo 8.1.11). AX ile CX yer değiştirmiş, BX yerinde kalmıştır. Bunun nedeni, ortadaki değerin hem itme hem de çekme sırasında ortada olmasıdır. Genel olarak n yazmaç aynı sırayla itilip aynı sırayla çekilirse yazmaç dizisi ters çevrilir: birinci yazmaç sonuncunun değerini, sonuncu birincinin değerini alır. Üç yazmaçta ortadaki, sabit noktadır. `pop bx` satırında BX'e yine 0004h yazılır; değer değişmediği için tabloda vurgulanmamıştır, ama komut yürütülmüş ve SP artmıştır.

8.1.3.6 Yığınla diziyi ters çevirmek

LIFO'nun ters çevirme özelliği bir dizgiye de uygulanabilir. Program 'ABC' dizgisinin karakterlerini sırayla yığına iter, sonra yığından çekerek ekrana yazar. Ekranda CBA görünür.

```
ORG  100h
MOV  SI, OFFSET metin
MOV  CX, 3
koy: MOV  AL, [SI]
     PUSH AX       ; karakteri sakla
     INC  SI
     LOOP koy
MOV  CX, 3
yaz: POP  DX       ; DL = karakter
     MOV  AH, 2
     INT  21h      ; DL'yi ekrana yaz
     LOOP yaz
RET
metin DB 'ABC'   ; 0117h
```

Önce programın bellekteki yerleşimi çıkarılır, çünkü `metin` değişkeninin adresi koddan sonra gelir. Tablo 8.1.12 her komutun adresini ve makine kodunu verir. Kod 0100h'den 0116h'ye kadar 23 bayt tutar. Ardından gelen `metin` 0117h–0119h adreslerindedir; 41h, 42h ve 43h baytları 'A', 'B' ve 'C' karakterlerinin ASCII kodlarıdır. Program toplam 26 bayttır.

Satır	Adres	Kod	Komut
2	0100h	BE 17 01	`MOV SI, OFFSET metin`
3	0103h	B9 03 00	`MOV CX, 3`
4	0106h	8A 04	`koy: MOV AL, [SI]`
5	0108h	50	`PUSH AX`
6	0109h	46	`INC SI`
7	010Ah	E2 FA	`LOOP koy`
8	010Ch	B9 03 00	`MOV CX, 3`
9	010Fh	5A	`yaz: POP DX`
10	0110h	B4 02	`MOV AH, 2`
11	0112h	CD 21	`INT 21h`

Satır	Adres	Kod	Komut
12	0114h	E2 F9	`LOOP yaz`
13	0116h	C3	`RET`
14	0117h	41 42 43	`metin DB 'ABC'`

Tablo 8.1.12: Ters çevirme programının adresleri ve makine kodları.

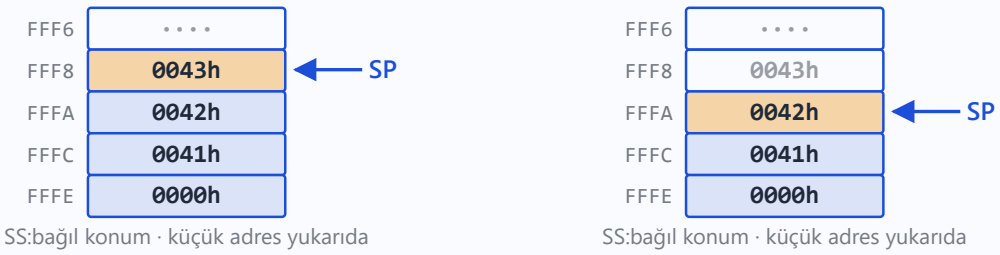
LOOP komutlarının ikinci baytı bağıl atlama uzaklığıdır. `LOOP koy` 010Ah'dedir ve iki bayttır; sonraki komut 010Ch'dedir. 0106h'ye dönmek için uzaklık 0106h − 010Ch = −6 olmalıdır; −6'nın 8 bitlik ikiye tümleyeni FAh'dir. `LOOP yaz` için uzaklık 010Fh − 0116h = −7 = F9h'dir.

Birinci döngü: karakterleri yığına itmek. SI dizgiyi, CX döngü sayacını tutar. Her turda `MOV AL, [SI]` bir karakteri AL'ye okur. Yığına bayt itilemediği için `PUSH AX` AL'yi AH ile birlikte sözcük olarak iter. Programın başında AH = 00h olduğu için yığına 0041h, 0042h ve 0043h gider. `INC SI` sonraki karakteri gösterir; `LOOP koy` CX'i 1 azaltır ve CX ≠ 0 ise başa döner. Üç tur sonunda SI = 011Ah ve SP = FFF8h olur. Yığının üstünde son itilen 'C' vardır.

Satır	Komut	AX	CX	SI	SP	Yığın (üstten, DOS sözcüğü hariç)
2	`MOV SI, OFFSET metin`	0000h	0000h	**0117h**	FFFEh	(boş)
3	`MOV CX, 3`	0000h	**0003h**	0117h	FFFEh	(boş)
4	`MOV AL, [SI]`	**0041h**	0003h	0117h	FFFEh	(boş)
5	`PUSH AX`	0041h	0003h	0117h	**FFFCh**	**0041h**
6	`INC SI`	0041h	0003h	**0118h**	FFFCh	0041h
7	`LOOP koy`	0041h	**0002h**	0118h	FFFCh	0041h
4	`MOV AL, [SI]`	**0042h**	0002h	0118h	FFFCh	0041h
5	`PUSH AX`	0042h	0002h	0118h	**FFFAh**	**0042h, 0041h**
6	`INC SI`	0042h	0002h	**0119h**	FFFAh	0042h, 0041h
7	`LOOP koy`	0042h	**0001h**	0119h	FFFAh	0042h, 0041h
4	`MOV AL, [SI]`	**0043h**	0001h	0119h	FFFAh	0042h, 0041h
5	`PUSH AX`	0043h	0001h	0119h	**FFF8h**	**0043h, 0042h, 0041h**
6	`INC SI`	0043h	0001h	**011Ah**	FFF8h	0043h, 0042h, 0041h
7	`LOOP koy`	0043h	**0000h**	011Ah	FFF8h	0043h, 0042h, 0041h

Tablo 8.1.13: Birinci döngünün izlemesi (DX = 0000h).

Birinci döngüde bayraklarla ilgili küçük bir ayrıntı da vardır. `INC SI` CF dışındaki aritmetik bayrakları günceller. SI = 0118h olduğunda düşük bayt 18h = 0001 1000b iki tane 1 biti içerir; PF = 1 olur. SI = 0119h ve 011Ah'de düşük baytlardaki 1 sayısı tektir ve PF = 0'dır. Bu bayraklar programın sonucunu etkilemez, çünkü `LOOP` bayraklara bakmaz; yalnız CX'i sınar.



Şekil 8.1.6: Solda birinci döngü bitince yığın: 0043h ('C') üstte, SP = FFF8h. Sağda ikinci döngünün ilk POP DX'inden sonra: 'C' DX'e alınmış, SP = FFFAh; FFF8h'deki 0043h artık yığının dışındadır.

İkinci döngü: yığından çekip yazmak. CX yeniden 3 yapılır. Her turda `POP DX` yığının üstündeki sözcüğü DX'e alır; DL karakteri, DH ise 00h'dir. `MOV AH, 2` DOS'un karakter yazma işlevini seçer ve `INT 21h` DL'deki karakteri ekrana yazar (Şekil 8.1.6). INT 21h'nin 02h işlevi bir yan etki taşır: işlev bittiğinde AL, yazılan karakterdir. Bu yüzden AX her INT 21h'den sonra değişir. İlk turda AL zaten 43h olduğu için fark görünmez, sonraki turlarda AL = 42h ve AL = 41h olur. Yığından ilk çıkan 'C', sonra 'B', en son 'A''dır.

Satır	Komut	AX	CX	DX	SP	Ekran
8	`MOV CX, 3`	0043h	**0003h**	0000h	FFF8h	
9	`POP DX`	0043h	0003h	**0043h**	**FFFAh**	
10	`MOV AH, 2`	**0243h**	0003h	0043h	FFFAh	
11	`INT 21h`	0243h	0003h	0043h	FFFAh	**C**
12	`LOOP yaz`	0243h	**0002h**	0043h	FFFAh	C
9	`POP DX`	0243h	0002h	**0042h**	**FFFCh**	C
10	`MOV AH, 2`	0243h	0002h	0042h	FFFCh	C
11	`INT 21h`	**0242h**	0002h	0042h	FFFCh	**CB**
12	`LOOP yaz`	0242h	**0001h**	0042h	FFFCh	CB
9	`POP DX`	0242h	0001h	**0041h**	**FFFEh**	CB
10	`MOV AH, 2`	0242h	0001h	0041h	FFFEh	CB
11	`INT 21h`	**0241h**	0001h	0041h	FFFEh	**CBA**
12	`LOOP yaz`	0241h	**0000h**	0041h	FFFEh	CBA
13	`RET`	0241h	0000h	0041h	**0000h**	CBA

Tablo 8.1.14: İkinci döngünün izlemesi (SI = 011Ah).

Tablo 8.1.14'ün son satırında SP = 0000h ve IP = 0000h olur; RET, DOS'un 0000h sözcüğünü almıştır. Program toplam 28 komut yürütür ve PSP'deki INT 20h ile biter. İki döngü aynı sayıda tur döndüğü için üç PUSH'a üç POP karşılık gelir ve yığın dengelidir. Döngü sayıları farklı olsaydı, örneğin ikinci döngü yanlışlıkla CX = 2 ile başlasaydı, 'A' yığında kalır ve RET onu dönüş adresi sanardı: IP = 0041h. Program PSP'nin ortasına atlar ve düzgün sonlanmazdı.

Aynı yöntem dizgiyi ekrana yazmadan, bellekte yerinde ters çevirmek için de kullanılabilir. İkinci döngüde karakter `POP AX` ile alınır ve `MOV [SI], AL` ile dizgiye baştan itibaren yeniden yazılır; bunun için SI önce dizginin başına geri getirilir. Yığın burada bir ara bellek görevi görür. Ayrı bir tampon dizi tanımlamak gerekmez, ama her karakter yığında 2 bayt yer kaplar.

> **Sık yapılan hata**
>
> INT 21h çağrısından sonra AX'in değişmediğini varsaymak. AH = 02h işlevi AL'yi yazılan karaktere eşitler; AH = 09h işlevi AL'ye 24h ('$') bırakır. Döngü sayacı ya da ara sonuç AL'de tutuluyorsa çağrıdan önce `PUSH AX`, sonra `POP AX` yazılır.

8.1.3.7 Yazmaçlar ne olur?

Sunumdaki tahmin sorusu, PUSH ve POP'ların karışık bir sırayla kullanıldığı bir koddur. Başlangıçta SP = FFFEh'dir.

```
MOV AX, 1111h
MOV BX, 2222h
MOV CX, 3333h
PUSH AX
PUSH BX
PUSH CX
POP AX
POP CX
PUSH AX
POP BX
POP BX
```

Çözüm yığının içeriğini adım adım yazarak bulunur. Üç PUSH'tan sonra yığın üstten alta 3333h, 2222h, 1111h'dir ve SP = FFF8h'dir. `POP AX` üstteki 3333h'yi alır: AX = 3333h. `POP CX` sıradaki 2222h'yi alır: CX = 2222h, SP = FFFCh. Yığında yalnız 1111h kalmıştır. `PUSH AX` 3333h'yi yeniden iter ve `POP BX` onu hemen alır: BX = 3333h. Son `POP BX` yığındaki 1111h'yi alır ve BX'in az önceki değerinin üzerine yazar: BX = 1111h.

Sonuç AX = 3333h, BX = 1111h, CX = 2222h ve SP = FFFEh'dir. Dört PUSH ve dört POP yapıldığı için yığın dengelidir. Değerler yazmaçlar arasında bir tur dönmüştür: AX, CX'in; CX, BX'in; BX de AX'in eski değerini almıştır. Bu tür sorularda en güvenilir yöntem, yığını bir kâğıt üzerinde sütun olarak çizmek ve her komutta üstteki hücreyi işaretlemektir.

8.1.3.8 Takas yöntemlerinin karşılaştırması

İki yazmacı takas etmenin üç yolu vardır. Yığınla takas dört komut tutar: `PUSH AX`, `PUSH BX`, `POP AX`, `POP BX`. Geçici bir yazmaçla takas üç `MOV` tutar: `MOV CX, AX`, `MOV AX, BX`, `MOV BX, CX`. Üçüncü yol 8086'nın takas komutudur: `XCHG AX, BX`.



Şekil 8.1.7: Üç takas yönteminin kod boyutu: yığınla 4 bayt, geçici yazmaçla 6 bayt, XCHG ile 1 bayt.

Yöntem	Bayt	Bellek erişimi	Saat turu	Yan etki ve kapsam
yığınla	4	4	2 × 11 + 2 × 8 = 38	hiçbir yazmaç bozulmaz; kesim yazmaçları için de çalışır
geçici yazmaçla	6	0	3 × 2 = 6	CX'in değeri bozulur
`XCHG AX, BX`	1 (93h)	0	3	en kısa ve en hızlı; kesim yazmaçlarıyla kullanılamaz

Tablo 8.1.15: Takas yöntemlerinin karşılaştırması (8086 saat turları).

Tablo 8.1.15 açık bir sonuç verir: iki genel amaçlı yazmacı takas etmenin en iyi yolu `XCHG`'dir. Komut tek bayttır. AX ile başka bir 16 bitlik yazmacı takas eden `XCHG` için 90h + r biçiminde kısa kodlar vardır; BX'in numarası 3 olduğundan kod 93h'dir. 5 MHz'de yığınla takas 38 × 200 ns = 7,6 µs, XCHG ise 0,6 µs sürer. Yığın yönteminin asıl değeri takas değil, saklamadır. Yazmaçları bir prosedür ya da kesme süresince korumak, ara sonuçları geçici olarak tutmak ve kesim yazmaçlarını kopyalamak (`PUSH DS`, `POP ES`) yığının asıl görevleridir. Kesim yazmaçları `XCHG` ile takas edilemediği için onların takası da yığınla yapılır: `PUSH DS`, `PUSH ES`, `POP DS`, `POP ES`.

8.1.3.9 Etkileşimli yığın simülatörü

Sunumdaki etkileşimli simülatör AX, BX, CX ve SP yazmaçlarını ve SS:SP çevresindeki yığın hücrelerini gösterir. Yazmaç değerleri onaltılık olarak düzenlenebilir. Altı düğme `PUSH AX`, `PUSH BX`, `PUSH CX`, `POP AX`, `POP BX` ve `POP CX` komutlarını çalıştırır ve çalıştırılan komutlar bir listede birikir. Simülatörle bu notun bütün sıralama örnekleri yeniden denenebilir: ayna sırası, takas, üç yazmacın döndürülmesi.

Simülatörün gösterdiği önemli bir durum boş yığından POP'tur. Gerçek 8086 bunu engellemez. Yığın boşken POP yapılırsa SP yine 2 artar ve kesimde o adreste ne varsa okunur. İşlemcinin yığının boş ya da dolu olduğunu bilmesinin bir yolu yoktur; SP yalnız bir sayıdır. Yığının sınırlarını izlemek tamamen programcının işidir.

8.1.4 Bayrakları saklamak: PUSHF ve POPF

8.1.4.1 Bayrak sözcüğü

8086'nın bayrak yazmacı (FLAGS) 16 bitlik bir yazmaçtır, ama bitlerinin yalnız dokuzu kullanılır. Altı durum bayrağı son aritmetik ya da mantık işleminin sonucunu anlatır: CF (elde, bit 0), PF (eşlik, bit 2), AF (yardımcı elde, bit 4), ZF (sıfır, bit 6), SF (işaret, bit 7) ve OF (taşma, bit 11). Üç denetim bayrağı işlemcinin davranışını belirler: TF (tuzak, bit 8), IF (kesme, bit 9) ve DF (yön, bit 10). Kalan bitler kullanılmaz. 8086'da bit 1 ve bit 12–15 her zaman 1 okunur; bit 3 ve bit 5 ise 0'dır.

Tanım 8.1.5

PUSHF, bayrak yazmacının 16 bitlik sözcüğünü yığına iter: SP ← SP − 2, [SS:SP] ← FLAGS. **POPF**, yığının üstündeki sözcüğü bayrak yazmacına yükler: FLAGS ← [SS:SP], SP ← SP + 2. PUSHF bayrakları değiştirmez; POPF ise bütün bayrakları birden değiştirebilir.

Bayraklar programda tek tek okunur: `JC` CF'ye, `JZ` ZF'ye bakar. Bayrak yazmacının tamamına doğrudan erişen bir `MOV` ise yoktur. `LAHF` yalnız düşük baytı (SF, ZF, AF, PF, CF) AH'ye kopyalar; OF, DF, IF ve TF'ye ulaşmaz. Bayrak yazmacının tamamını saklamanın ve geri yüklemenin tek yolu yığındır. Bayrak sözcüğünü bir yazmaca almak için de yığın kullanılır: `PUSHF` ve ardından `POP AX`. Bir .COM programının başında bu iki komut AX'e F202h getirir. F000h dört sabit 1 bitidir, 0200h IF = 1'dir, 0002h ise sabit 1 olan bit 1'dir. `STC` 'den sonra aynı iki komut F203h verir; fark CF'nin bit 0'ıdır.

8.1.4.2 PUSHF ve POPF: kod izleme

Aşağıdaki program bir toplamanın bayraklarını saklar, araya bayrakları bozan başka bir toplama koyar ve saklanan bayrakları geri yükler.

```
ORG   100h
MOV   AX, 0FFFFh
ADD   AX, 1       ; CF=1, ZF=1
PUSHF             ; bayrakları sakla
MOV   BX, 5
ADD   BX, 3       ; CF=0, ZF=0
POPF              ; bayraklar geri
RET
```

İlk satırdaki sayı 0FFFFh biçiminde yazılmıştır. Onaltılık sayı bir harfle başlıyorsa assembler onu bir ad sanmasın diye başına 0 eklenir; FFFFh yazmak hata verir. `ADD AX, 1` FFFFh + 1 = 1 0000h hesaplar. Sonuç 17 bittir; AX'e düşük 16 bit, 0000h yazılır ve dışarı taşan 1 CF'ye gider. Sonuç sıfır olduğu

için ZF = 1'dir. 0000h'nin düşük baytında 1 biti yoktur, yani çift sayıda 1 vardır ve PF = 1 olur. Düşük dörtlüde F + 1 = 10h elde ürettiği için AF = 1'dir. SF = 0 ve OF = 0'dır: −1 + 1 = 0 işaretli olarak da doğrudur.

`PUSHF` bu anda bayrak sözcüğünü iter. Sözcüğün değeri bitlerden hesaplanır: sabit bitler F000h + 0002h, IF için 0200h, ZF için 0040h, AF için 0010h, PF için 0004h ve CF için 0001h. Toplam F257h'dir. [FFFCh] = 57h ve [FFFDh] = F2h olur. `MOV BX, 5` bayrakları değiştirmez. `ADD BX, 3` 5 + 3 = 8 hesaplar: elde yok, sonuç sıfır değil. CF, ZF, PF ve AF'nin dördü de 0 olur. 8 = 0000 1000b'de tek bir 1 biti vardır, bu yüzden PF = 0'dır. İlk toplamanın bayrakları kaybolmuştur. `POPF` F257h'yi bayrak yazmacına yükler ve CF = 1, ZF = 1, PF = 1, AF = 1 geri gelir. AX ve BX'e dokunulmaz; AX = 0000h, BX = 0008h kalır.

Komut	AX	BX	SP	Bayraklar	Yığın (üstten)
`MOV AX, 0FFFFh`	**FFFFh**	0000h	FFFEh	odItszapc	0000h
`ADD AX, 1`	**0000h**	0000h	FFFEh	**odItsZAPC**	0000h
`PUSHF`	0000h	0000h	**FFFCh**	odItsZAPC	**F257h, 0000h**
`MOV BX, 5`	0000h	**0005h**	FFFCh	odItsZAPC	F257h, 0000h
`ADD BX, 3`	0000h	**0008h**	FFFCh	**odItszapc**	F257h, 0000h
`POPF`	0000h	0008h	**FFFEh**	**odItsZAPC**	0000h
`RET`	0000h	0008h	**0000h**	odItsZAPC	(boş); IP = 0000h

Tablo 8.1.16: PUSHF/POPF programının izlemesi (IF = 1 baştan sona; bayraklar ODITSZAPC sırasıyla, büyük harf = 1).

Programın son durumu Tablo 8.1.16'da görülür. POPF yerine `POP BX` yazılsaydı ne olurdu? Yığından yine F257h alınırdı, ama bu kez BX'e yazılırdı: BX = F257h. Bayraklar ikinci toplamadaki gibi CF = 0 ve ZF = 0 kalırdı. Bu deney POPF ile POP arasındaki farkı açıkça gösterir. İki komut yığından aynı sözcüğü alır; yalnız hedefleri farklıdır.

Program 15 bayttır. `ADD BX, 3` üç baytla kodlanır (83 C3 03): 83h işlem kodu, 8 bitlik anlık değerin işaret genişletilerek 16 bite tamamlandığını söyler. `ADD AX, 1` ise AX'e özel kısa biçimle 05 01 00 olarak kodlanır. PUSHF (9Ch) ve POPF (9Dh) birer bayttır.

Sık yapılan hata

PUSHF ile saklanan bayrakları geri yüklemeden önce yığına başka bir değer itip POPF ile o değeri bayrak yazmacına almak. `PUSHF , PUSH AX , ..., POPF , POP AX` dizisinde POPF, AX'in değerini bayrak olarak yükler ve AX'e eski bayrak sözcüğü gelir. Bayraklar ve yazmaçlar da ayna sırasına uyar: `PUSH AX , PUSHF , ..., POPF , POP AX .`

8.1.4.3 Bayrak sözcüğünü değiştirmek

CF, DF ve IF için doğrudan komutlar vardır: `STC` / `CLC` / `CMC`, `STD` / `CLD`, `STI` / `CLI`. TF'yi değiştiren bir komut ise yoktur. TF'yi 1 yapmanın tek yolu yığından geçer. Bayrak sözcüğü yığına itilir, bir yazmaca alınır, bit 8 değiştirilir, sözcük yeniden itilir ve POPF ile bayrak yazmacına yüklenir:

```
PUSHF
POP  AX
OR   AX, 0100h     ; bit 8 = TF
PUSH AX
POPF               ; TF = 1
```

Hata ayıklayıcılar programı tek adım çalıştırmak için bu kalıbı kullanır. TF = 1 olduğunda işlemci her komuttan sonra bir tip 1 kesmesi üretir. Aynı yöntemle OF de değiştirilebilir; OF'yi doğrudan kuran bir komut da yoktur. Bu kalıpta AX bozulur; AX korunacaksa kalıp `PUSH AX` ile `POP AX` arasına alınır.

Bayrak sözcüğünü yığında değiştirmenin bir başka kullanımı da kesme yordamlarıdır. Bir kesme yordamı çağıran programa CF ile bir hata bilgisi döndürmek isterse, yığındaki FLAGS kopyasının bit 0'ını değiştirir. IRET bu kopyayı geri yüklediğinde çağıran program CF'yi görür. DOS işlevlerinin çoğu hatayı bu yolla, CF = 1 olarak bildirir.

8.1.4.4 Yığını otomatik kullanan komutlar

PUSHF ve POPF'in yaptığını bazı komutlar kendiliğinden yapar. `INT n` önce bayrak sözcüğünü, sonra CS'yi ve IP'yi iter; ardından IF ve TF'yi sıfırlar. `IRET` IP'yi, CS'yi ve bayrak sözcüğünü geri yükler. Kesme yordamları bu yüzden bayrakları kendileri saklamaz; INT bu işi otomatik yapar. Normal kodda ise bir sonucu korumak için PUSHF ve POPF gerekir, çünkü CALL bayrakları saklamaz.

`CALL` ve `RET` de yığını otomatik kullanır. Yakın bir CALL, kendisinden sonraki komutun bağıl konumunu, yani dönüş adresini iter ve SP'yi 2 azaltır. RET bu adresi IP'ye çeker ve SP'yi 2 artırır. Uzak bir CALL önce CS'yi, sonra IP'yi iter; RETF ikisini ters sırada geri alır. Bu komutlar, yığın açısından bakıldığında, hedefi IP olan PUSH ve POP'lardır. Prosedürün içindeki her PUSH'a bir POP karşılık gelmezse RET dönüş adresini değil, unutulan değeri alır. Yığın güvenliği kurallarının hepsi bu basit gözleme dayanır.

Komut	İttiği (sırayla)	Çeken komut	SP farkı
`CALL` (yakın)	IP	`RET`	2 bayt
`CALL` (uzak)	CS, IP	`RETF`	4 bayt
`INT n`, donanım kesmesi	FLAGS, CS, IP	`IRET`	6 bayt

Tablo 8.1.17: Yığını kendiliğinden kullanan komutların ittiği ve çektiği sözcükler.

8.1.5 Yığın güvenliği

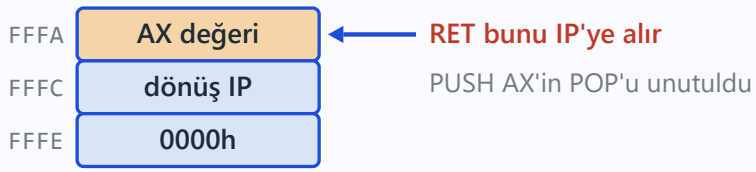
8.1.5.1 Denge kuralı ve .COM programının sonu

PUSH ve POP komutları programın çalışması boyunca geçici verileri saklar. Yığının bütünlüğü, eşit sayıda PUSH ve POP ile korunur. Bu kural programın yalnız bir yolunda değil, her yolunda, her dalında sağlanmalıdır. Eşit sayıda PUSH ve POP yapılmazsa yığın bozulur ve işletim sistemine geri dönülemeyebilir.

Tanım 8.1.6

Bir kod parçası, girişindeki SP değeri ile çıkışındaki SP değeri aynıysa **yığın açısından dengelidir**. Dengeli bir parçada her PUSH'a (ve PUSHF'e) bir POP (ya da POPF) karşılık gelir. Parçanın birden çok çıkışı varsa denge her çıkış için ayrı ayrı sağlanmalıdır.

Dengenin neden bu kadar önemli olduğu .COM programının sonunda görülür. RET yığının üstünde bir dönüş adresi bekler. DOS bir .COM programını başlatırken yığına 0000h koyar. Program dengeliyse son RET bu sözcüğü alır ve CS:0000h'ye, yani PSP'nin başına gider. Orada duran `INT 20h` programı sonlandırır. Program dengeli değilse RET başka bir sözcüğü alır.



Şekil 8.1.8: Fazla PUSH: yığının üstünde dönüş adresi yerine AX'in değeri kalır ve RET onu IP'ye alır.

Yığın hatalarının üç biçimi vardır ve üçü de farklı belirtiler verir (Şekil 8.1.8). **Fazla PUSH**'ta RET, dönüş adresi yerine son konan veriyi alır ve rastgele bir adrese atlar. **Fazla POP**'ta dönüş adresi bir yazmaca "yenir"; RET ondan sonraki sözcüğü alır ve yine yanlış yere döner. **Yanlış sıra**da ise yığın dengelidir ve program doğru yere döner, ama değerler karışır. Program sessizce yanlış çalışır. Üçüncüsü bulunması en zor olanıdır, çünkü program çökmez.

8.1.5.2 Hata örneği: dengesiz PUSH

Aşağıdaki programda AX yığına konur, ama hiç geri alınmaz.

```
ORG   100h
MOV   AX, 1234h
PUSH  AX          ; POP unutuldu!
RET               ; IP ← ?
```

Komut	IP (sonraki)	AX	SP	Yığın (üstten)
başlangıç	0100h	0000h	FFFEh	0000h
`MOV AX, 1234h`	0103h	**1234h**	FFFEh	0000h
`PUSH AX`	0104h	1234h	**FFFCh**	**1234h, 0000h**
`RET`	**1234h**	1234h	**FFFEh**	0000h

Tablo 8.1.18: Dengesiz PUSH programının izlemesi.

RET yığının üstündeki sözcüğü dönüş adresi sanar (Tablo 8.1.18). IP = 1234h olur ve SP FFFEh'ye çıkar. İşlemci 0700:1234h adresindeki baytları komut olarak çalıştırmaya başlar. Bu program yalnız 5 bayttır; 1234h adresinde program yoktur, belleğin o bölgesinde ne kaldıysa o vardır. Program çöker ya da donar. Garip olan, DOS'un 0000h sözcüğünün hâlâ yığında durmasıdır. Dönüş için gereken bilgi kaybolmamış, yalnız bir sözcük altta kalmıştır.

Düzeltme kolaydır: RET'ten önce `POP AX` eklenir ya da PUSH hiç yazılmaz. Asıl zorluk, dengesizliğin çoğu zaman bir dalda gizlenmesidir. Aşağıdaki kodda AX iki dalda da itilir, ama yalnız bir dalda geri alınır:

```
        PUSH AX
        CMP  BX, 0
        JE   bitti        ; bu dalda POP yok!
        ; … AX'i kullanan işlem …
        POP  AX
bitti:  RET
```

BX ≠ 0 olduğu sürece program doğru çalışır. BX = 0 olduğu ilk anda RET, AX'in değerine atlar. Bu tür hatalar testte kolay yakalanmaz. Kural şudur: her dalın (JMP, Jcc) PUSH ve POP sayısı ayrı ayrı sayılır. En güvenli çözüm, POP'u tek bir çıkış etiketinin altına koyup bütün dalları oraya yönlendirmektir. Bu örnekte düzeltme, `JE bitti` yerine POP'tan önceki bir etikete atlamaktır.

8.1.5.3 Fazla POP ve yanlış sıra

Fazla POP'un etkisi de bir izlemeyle görülebilir:

```
ORG  100h
MOV  AX, 5
PUSH AX
POP  BX
POP  CX            ; fazladan POP
RET
```

`PUSH AX` ile `POP BX` dengelidir; BX = 0005h ve SP = FFFEh olur. Fazladan `POP CX`, DOS'un 0000h sözcüğünü CX'e alır ve SP'yi FFFEh + 2 = 0000h yapar. SP 16 bitte sarmıştır. RET şimdi SS:0000h'deki sözcüğü okur. Bu adres PSP'nin başıdır ve ilk iki baytı CD 20'dir. Little-endian düzende bu sözcük 20CDh'dir. IP = 20CDh olur ve işlemci programla ilgisi olmayan bir adrese atlar. Dönüş adresi, yani 0000h, CX'e "yenmiştir".

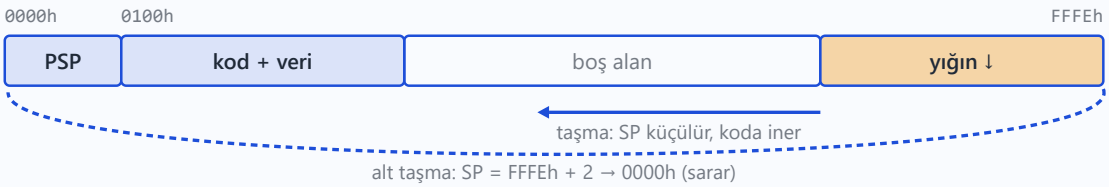
Yanlış sıra ise farklı bir belirti verir. `PUSH AX`, `PUSH BX`, ..., `POP AX`, `POP BX` dizisi dengelidir; RET doğru yere döner. Ama AX ile BX'in değerleri takas olmuştur. Prosedürü çağıran program, AX'te kendi bıraktığı değeri bulacağını sanar ve yanlış bir değerle devam eder. Hata, ancak bu değer kullanıldığında ve çoğu zaman prosedürden çok uzakta ortaya çıkar.

8.1.5.4 Yığın taşması ve alt taşma

Dengesizlik bir kez değil de sürekli tekrarlanırsa yığın sınırlarını aşar. 8086 bu sınırları denetlemez; ortada bir hata kesmesi yoktur, yalnız bozulma vardır.

Tanım 8.1.7

Yığın taşması (*stack overflow*), yığına ayrılan alandan fazla değer itilmesi ve SP'nin yığın alanının altına inmesidir. **Yığın alt taşması** (*stack underflow*), yığına itilmemiş bir değerin çekilmesi ve SP'nin yığın alanının üstüne çıkmasıdır.



Şekil 8.1.9: .COM programında PSP, kod ve veri kesimin altında; yığın FFFEh'den aşağı büyür. Taşmada SP küçülerek koda iner; alt taşmada SP FFFEh'den 0000h'ye sarar.

Taşmanın tipik nedenleri sonsuz özyineleme ve döngü içinde POP'suz PUSH'tur. SP küçüldükçe yığın aşağı iner ve .COM programında aynı kesimdeki veriyi, sonra kodu ezer (Şekil 8.1.9). Kod ezildikten sonra işlemci bir sonraki turda yığın verisini komut diye çalıştırır. Belirti, çoğu zaman taşmanın başladığı yerden çok uzakta görülür.

Döngüde POP'suz PUSH'un küçük bir örneği şöyledir:

```
        ORG  100h
        MOV  CX, 3
        MOV  AX, 0100h
tekrar: PUSH AX
        INC  AX
        LOOP tekrar
        RET
```

Döngü üç tur döner ve yığına 0100h, 0101h ve 0102h'yi iter. SP = FFF8h olur. RET yığının üstündeki 0102h'yi alır ve IP = 0102h olur. Bu adres programın içindedir, ama bir komutun başı değildir: `MOV CX, 3` 0100h–0102h'yi kaplar ve 0102h onun son baytıdır. İşlemci 00h baytını yeni bir komutun başı sayar ve anlamsız bir komut dizisini yürütmeye başlar. Üç tur yerine CX = 0 ile başlayan bir döngü 65.536 tur döner. Bu durumda yığına 128 KB kadar değer itilir. 64 KB'lık kesim bu kadarını tutamaz: SP sarar ve yığın programın kodunun üzerine yazar.

Alt taşma PUSH edilmemiş bir değeri POP etmektir. SP = FFFEh iken POP yapılırsa SP = 0000h olur; 16 bitlik toplama sarar. Okunan değer çöptür ya da .COM programında DOS'un 0000h'sidir. Bundan sonraki PUSH, SP'yi FFFEh'ye indirir ve DOS'un sözcüğünün üzerine yazar. Program bu durumda RET ile düzgün sonlanamaz, çünkü dönüş için gereken 0000h ezilmiştir.

Taşmayı önlemenin yolu yığın alanını yeterli büyüklükte seçmektir. Yığın alanının boyutu programın en derin çağrı zincirine ve en çok PUSH sayısına göre seçilir. Her CALL 2 bayt (uzak çağrıda 4 bayt), her PUSH 2 bayt ve her kesme 6 bayt tutar. Kesmeler programın herhangi bir anında gelebilir; donanım kesmesi yordamları da aynı yığını kullanır ve kendi kaydettikleri yazmaçlar için yer ister. .EXE programlarında `.STACK 100h` gibi bir direktifle ayrı ve yeterli bir yığın kesimi ayrılır. Ayrı kesimin bir üstünlüğü de vardır: yığın taşsa bile kodun üzerine yazmaz, yalnız kendi kesiminin altındaki belleği bozar.

Örnek 8.1.3

Bir program en çok dört düzey iç içe yakın CALL yapar. Her prosedür girişte üç yazmacı iter. Programın her anında bir donanım kesmesi gelebilir ve kesme yordamı dört yazmaç iter. Yığının en çok kaç bayt derinleşeceği şöyle bulunur:

1. Çağrılar: 4 düzey × (2 bayt dönüş adresi + 3 × 2 bayt yazmaç) = 4 × 8 = 32 bayt.
2. Kesme: 6 bayt (FLAGS, CS, IP) + 4 × 2 bayt yazmaç = 14 bayt.
3. Toplam: 32 + 14 = 46 bayt.

Yığın en çok 46 bayt derinleşir. `.STACK 100h` ile ayrılan 256 bayt bu program için fazlasıyla yeterlidir; aradaki pay sonradan eklenecek prosedürlere ve BIOS yordamlarının kendi kullanımına kalır.

8.1.5.5 İyi uygulamalar

Yığın hatalarının çoğu birkaç yazım alışkanlığıyla önlenir. Sunumdaki altı kural şunlardır:

1. **Ayna sırası.** `PUSH AX`, `PUSH BX`, `PUSH CX` sırasının tersi `POP CX`, `POP BX`, `POP AX`'tir. Kod alt alta yazıldığında ayna görüntüsü gibi görünmelidir.
2. **Tek çıkış.** Prosedürde birden çok RET yerine tek bir çıkış etiketi kullanılır. POP'lar bir kez yazılır ve bütün dallar oradan geçer.
3. **SP'ye dokunmamak.** SP'yi doğrudan değiştirmek yalnız iki iş için yapılır: yığından geçirilen parametreleri temizlemek (`ADD SP, n`) ve yerel değişkenlere yer ayırmak (`SUB SP, n`). n her zaman çift olmalıdır.
4. **BP çerçevesi.** Yığındaki verilere `[SP+n]` ile erişilemez, çünkü 8086'da SP taban yazmacı değildir. `MOV BP, SP` yapılır ve `[BP+n]` kullanılır; BP varsayılan olarak SS kesimini adresler.
5. **Kesimleri korumak.** DS ya da ES'yi değiştiren yordamlar onları da PUSH ve POP etmelidir. Çağıran program kendi veri kesiminin değiştiğini bilemez.
6. **emu8086 ile denetlemek.** Yığın penceresi açılır ve RET'ten önce SP'nin girişteki değerine döndüğü denetlenir.



Şekil 8.1.10: Ayna sırası. PUSH AX, BX, CX sırasının tersi POP CX, BX, AX'tir; çizgiler aynı yazmacı birleştirir ve birbirini kesmez.

Şekil 8.1.10 ayna sırasını görsel olarak verir. PUSH'lar ile POP'lar arasındaki çizgiler iç içe geçmiş yaylar oluşturur. Çizgiler birbirini kesiyorsa sıra yanlıştır. Bu görüntü, açılış ve kapanış parantezlerinin iç içe dizilişine benzer: en son açılan parantez ilk kapanır.

Aşağıdaki iskelet bu kuralların hepsini bir arada gösterir:

```
yordam PROC
       PUSH AX          ; kullanılan yazmaçlar
       PUSH BX
       PUSH DS          ; kesim de değişecek
       ; … iş …
       JC   cikis         ; hata dalı da çıkıştan geçer
       ; … iş …
cikis: POP  DS            ; ayna sırası
       POP  BX
       POP  AX
       RET
yordam ENDP
```

8.1.5.6 emu8086 ile yığını izlemek

Bu notun izleme tablolarının hepsi emu8086'da adım adım elde edilebilir. Yöntem dört adımdan oluşur.

1. **Derle ve yükle.** Program *compile* ile derlenir, *emulate* ile öykünücüye yüklenir. Yazmaç panelinde CS, IP, SS ve SP'nin başlangıç değerleri görülür: .COM programı için 0700h, 0100h, 0700h ve FFFEh.
2. **Tek adım.** *single step* düğmesi her basışta bir komut çalıştırır ve değişen yazmaçlar vurgulanır. *step back* son adımı geri alır; bir PUSH'un etkisini tekrar görmek için kullanışlıdır.
3. **Yığın penceresi.** Alttaki *stack* düğmesi SS:SP çevresindeki sözcükleri gösteren pencereyi açar. SP'nin gösterdiği satır işaretlidir. Her PUSH'ta işaret bir satır yukarı (daha küçük adrese), her POP'ta bir satır aşağı kayar.
4. **Bellek görünümü.** Öykünücünün ortasındaki liste baytları onaltılık, onluk ve ASCII olarak gösterir. `B8 12 12` baytları `MOV AX, 1212h` komutudur. Little-endian düzen burada görülür: `MOV AX, 1234h` B8 34 12 olarak kodlanır.

Yüklemeden sonra CX programın bayt sayısını gösterir; takas örneğinde bu değer 000Bh, yani 11 bayttır. Bir program bu değere güvenmemeli, CX'i kullanmadan önce kendisi yüklemelidir. emu8086'nın yığın penceresinde FFFEh satırında görülen 0000h da bir rastlantı değildir; program sonundaki RET'in dönüş adresidir. Programın her RET'inden önce bu satıra bakmak, yığın dengesini denetlemenin en kısa yoludur.

8.1.6 Alıştırmalar ve çözümleri

Sunumdaki beş alıştırma burada ara adımlarıyla çözülmüştür.

1. SS = 2000h, SP = 0100h iken PUSH AX sonrası yazılan baytların fiziksel adresleri nedir?

PUSH önce SP'yi azaltır: SP = 0100h − 2 = 00FEh. Yığının yeni üstü 2000h × 10h + 00FEh = 20000h + 00FEh = 200FEh'dir. Little-endian düzende AL 200FEh'ye, AH 200FFh'ye yazılır. AX = 1234h olsaydı [200FEh] = 34h ve [200FFh] = 12h olurdu.

2. MOV AX,1 / MOV BX,2 / MOV CX,3 / PUSH AX / PUSH BX / PUSH CX / POP BX / POP CX / POP AX sonunda AX, BX, CX nedir?

Üç PUSH'tan sonra yığın üstten alta 3, 2, 1'dir. `POP BX` 3'ü, `POP CX` 2'yi, `POP AX` 1'i alır. Sonuç AX = 1, BX = 3, CX = 2'dir. AX yerinde kalır, BX ile CX takas edilir. AX hem ilk itilen hem de son çekilen yazmaç olduğu için sabit kalmıştır.

3. Yalnız PUSH ve POP ile DS'nin değerini ES'ye kopyalayın.

`PUSH DS` ve ardından `POP ES`. İki komut iki bayttır (1Eh, 07h) ve hiçbir genel amaçlı yazmacı bozmaz. Yığın dengeli kalır.

4. SP = FFFEh iken PUSH AX / PUSH BX / PUSHF / POP CX yürütülüyor. SP kaç olur, CX'e ne gelir?

Üç itme SP'yi 6 azaltır, bir çekme 2 artırır: FFFEh − 6 + 2 = FFFAh. CX'e en son itilen sözcük, yani PUSHF anındaki bayrak sözcüğü gelir. Bir .COM programının başında bu değer F202h'dir. Yığında AX ve BX'in kopyaları kalır; program bu hâliyle RET ile bitirilirse BX'in değerine döner.

5. AX'i ve bayrakları bir işlem boyunca koruyan PUSH/POP dizisini yazın.

`PUSH AX`, `PUSHF`, … işlem …, `POPF`, `POP AX`. Bayraklar en son itildiği için ilk çekilir; dizi ayna sırasına uyar. Örneğin AX = 0000h, CF = ZF = 1 iken sarmalın içinde `MOV AX, 7` ve `ADD AX, 1` çalıştırılsın. Sarmalın sonunda AX = 0000h, CF = 1, ZF = 1 ve SP = FFFEh olur.

8.1.7 Özet

- **Yığın**, geçici verileri saklamak için ayrılmış ve son giren ilk çıkar (**LIFO**) düzeninde çalışan bir bellek alanıdır. Programcı (PUSH, POP), işlemci (CALL, INT) ve işletim sistemi aynı yığını kullanır.
- Yığının üstü **SS:SP**'dir; fiziksel adresi SS × 10h + SP'dir. .COM programında SS = CS ve SP = FFFEh, .EXE programında başlıktaki değerler kullanılır. BP de varsayılan olarak SS kesimini adresler; SP adresleme ifadesinde kullanılamaz.
- Yığın **aşağı doğru büyür**: PUSH SP'yi 2 azaltır, POP 2 artırır. Birim her zaman sözcüktür; `PUSH AL` geçersizdir. .COM programında kod yukarı, yığın aşağı büyür ve aradaki boş alanı paylaşır.
- **PUSH**: önce SP ← SP − 2, sonra [SS:SP] ← kaynak. **POP**: önce hedef ← [SS:SP], sonra SP ← SP + 2. POP belleği silmez. SP her an dolu olan en üst hücreyi gösterir.
- PUSH yazmaç, kesim yazmacı ve bellek sözcüğü alır; anlık değerli PUSH yalnız 80186 ve sonrasında vardır. POP CS ve POP sabit yoktur. 8086'da `PUSH SP` azaltılmış değeri yazar.
- Sözcük yığına **little-endian** yazılır: düşük bayt SP'nin gösterdiği adrese, yüksek bayt bir üstüne. SP çift tutulur; tek SP her erişimi iki yol döngüsüne çıkarır.

- PUSH, POP ve PUSHF **bayrakları değiştirmez**; POPF bütün bayrakları yükler. Bir CMP sonucu PUSH ve POP arasında güvenle taşınır.
- Yazmaç saklarken POP sırası PUSH sırasının tersidir (**ayna sırası**); aynı sırayla POP değerleri takas eder ya da ters çevirir. Yığınla bir dizi ters çevrilebilir. İki yazmacın takası için en iyi yol XCHG'dir; yığının asıl görevi saklamadır.
- **PUSHF/POPF** bayrak yazmacının tamamını saklar ve geri yükler. Bayrak sözcüğü bir yazmaca PUSHF ve POP ile alınır; TF bu yolla değiştirilir. INT bayrakları kendiliğinden iter, CALL itmez.
- Her yolda ve her dalda **eşit sayıda PUSH ve POP** yapılır. .COM programında son RET, DOS'un koyduğu 0000h'yi alır ve PSP'deki INT 20h programı bitirir. Fazla PUSH'ta RET veriye, fazla POP'ta yığının ötesine döner.
- 8086 **taşmayı ve alt taşmayı** denetlemez: SP 16 bitte sarar ve yığın kodu ezebilir. Yığın alanı en derin çağrı zinciri, en çok PUSH ve kesmeler hesaba katılarak seçilir.

8.1.8 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

yığın *stack* · LIFO (son giren ilk çıkar) *last in, first out* · yığına itme / yığından alma (PUSH / POP) *push / pop* · yığın kesimi (SS) *stack segment* · işaretçi yazmacı (SP, BP) *pointer register* · taban işaretçisi (BP) *base pointer* · bayrak yazmacı (FLAGS) *flags register* · yığın taşması / alt taşması *stack overflow / underflow* · little-endian / big-endian · kesim:bağıl konum (mantıksal adres) *segment:offset* · fiziksel adres *physical address* · varsayılan kesim *default segment* · dönüş adresi *return address* · yazmaçların korunması *register preservation* · veri taşıma komutu *data transfer instruction*

8.1.9 Kendini Dene

1. SS = 3000h ve SP = 0000h iken `PUSH AX` çalışıyor. SP kaç olur? AX'in iki baytı hangi fiziksel adreslere yazılır?

   Cevap

   SP = 0000h − 2 = FFFEh; 16 bitlik çıkarma sarar. Yığının üstü 30000h + FFFEh = 3FFFEh'dir. AL 3FFFEh'ye, AH 3FFFFh'ye yazılır. SP = 0000h başlangıç değeri, yığının kesimin en üstünden başlaması için kullanılabilir; ilk PUSH kesimin son sözcüğüne yazar.

2. `MOV AX, 1A2Bh`, `PUSH AX`, `MOV BX, SP`, `MOV DL, [BX+1]`, `MOV DH, [BX]`, `POP AX` dizisinden

sonra (.COM programı, SP = FFFEh ile başlıyor) BX ve DX nedir?

Cevap

BX = FFFCh. [FFFCh] = 2Bh (düşük bayt), [FFFDh] = 1Ah (yüksek bayt). DL = 1Ah, DH = 2Bh; DX = 2B1Ah olur. Baytlar ters yerlere okunduğu için DX, AX'in baytları yer değiştirmiş hâlidir.

3. AX = AAAAh, BX = BBBBh, CX = CCCCh iken `PUSH AX`, `PUSH BX`, `PUSH CX`, `POP BX`, `POP AX`, `POP CX` çalışıyor. Sonuçta üç yazmaç nedir?

Cevap

Yığın üstten alta CCCCh, BBBBh, AAAAh'dir. BX = CCCCh, AX = BBBBh, CX = AAAAh olur. Her yazmaç bir başkasının değerini almıştır; yığın dengelidir.

4. `STC`, `PUSHF`, `CLC`, `POPF` dizisinden sonra CF kaçtır? `POPF` yerine `POP AX` yazılsaydı CF ve AX ne olurdu (programın başında IF = 1, diğer bayraklar 0)?

Cevap

POPF, STC sonrası saklanan sözcüğü geri yükler: CF = 1. POP AX ile CF, CLC'nin bıraktığı gibi 0 kalır ve AX = F203h olur (sabit bitler F002h, IF 0200h, CF 0001h).

5. Bir .COM programı `MOV AX, 5`, `PUSH AX`, `PUSH AX`, `POP BX`, `RET` ile bitiyor. RET'ten sonra IP kaçtır? Program düzgün sonlanır mı?

Cevap

İki PUSH'a bir POP karşılık gelir; yığının üstünde ikinci 0005h kalır. RET bu değeri alır: IP = 0005h. İşlemci PSP'nin içindeki bir adrese atlar; program INT 20h'ye ulaşmaz ve düzgün sonlanmaz.

6. Yığınla diziyi ters çeviren 26 baytlık program 0100h–0119h'yi kaplıyor. SP = FFFEh'den başlayan yığına, programın üzerine yazmadan en çok kaç sözcük itilebilir?

Cevap

n itmeden sonra SP = FFFEh − 2n olur ve SP ≥ 011Ah kalmalıdır. n ≤ (FFFEh − 011Ah) / 2 = (65.534 − 282) / 2 = 32.626. En çok 32.626 sözcük itilebilir.

7. Bir öğrenci `CMP AX, BX` ile `JE` arasına `PUSH CX`, `INC CX`, `POP CX` koyuyor. JE karşılaştırmanın

sonucuna göre mi atlar?

Cevap

Hayır. PUSH ve POP bayrakları değiştirmez, ama `INC CX` ZF'yi CX'in yeni değerine göre yeniden kurar. JE artık CMP'nin değil INC'in sonucuna bakar. Çözüm, araya PUSHF/POPF koymak ya da INC'i karşılaştırmadan önceye almaktır.

8. 8086'da yığınla takas (iki PUSH, iki POP) ile `XCHG AX, BX` kaç saat turu sürer? 5 MHz'de fark kaç µs'dir?

Cevap

Yığınla: 2 × 11 + 2 × 8 = 38 tur. XCHG AX, BX: 3 tur. Fark 35 tur; 5 MHz'de bir tur 200 ns olduğundan 35 × 200 ns = 7 µs'dir.

8.1.10 İleri okuma

- Intel, *The 8086 Family User's Manual* (1979): PUSH, POP, PUSHF, POPF, CALL, RET, INT ve IRET'in kesin tanımları, kodlamaları ve saat turu sayıları; S3–S6 durum bitlerinin anlamı. Tablo 8.1.2 ve 8.1.6'daki değerler bu kaynaktandır.
- B. B. Brey, *The Intel Microprocessors* (8. baskı), Bölüm 4: veri taşıma komutları arasında yığın komutları, yığının başlatılması ve PUSH/POP'un ayrıntılı örnekleri.
- K. R. Irvine, *Assembly Language for x86 Processors* (7. baskı), yığın işlemleri bölümü: çalışma anı yığını, yazmaçların korunması ve yığınla dizgi ters çevirme örneği.

8.1.11 Kaynaklar

1. Intel, *The 8086 Family User's Manual*, 1979 — PUSH, POP, CALL, RET, INT, IRET komut tanımları.
2. B. B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson, 2009 — Bölüm 4 ve 6.
3. K. R. Irvine, *Assembly Language for x86 Processors*, 7. baskı, Pearson, 2014 — yığın işlemleri ve prosedürler.
4. emu8086 yardım belgeleri: 8086 Assembler Tutorial, Part 9 (The Stack).

KISIM III · BÖLÜM 8.2

Prosedürler

Bir programda aynı iş birçok yerde gerekir: ekrana bir metin yazmak, bir sayıyı onluk basamaklara ayırmak, bir dizide değer aramak. Bu işin kodunu her yerde yeniden yazmak hem belleği doldurur hem de hatayı çoğaltır. Prosedür, böyle bir işi bir kez yazıp istenen her yerden çağırmayı sağlar. 8086 bunu iki komutla yapar: CALL dönüş adresini yığına yazıp prosedüre atlar, RET o adresi yığından alıp geri döner. Not önce prosedürün nasıl tanımlandığını ve CALL ile RET'in yazmaçlara, yığına ve makine koduna etkisini anlatır. Ardından yakın ve uzak çağrıları, dolaylı çağrıyı ve çağrının saat turu maliyetini ele alır. Sonra parametrelerin yazmaçla, bellekle, adresle ve yığınla nasıl geçirildiğine, yazmaçların korunmasına, yığın çerçevesine ve yerel değişkenlere geçer. Son kesimler örnek programları, özyinelemeyi ve prosedürlerde sık yapılan hataları işler. Her kod parçası yazmaçlar, bayraklar ve yığınla birlikte adım adım izlenir.

ÖĞRENME ÇIKTILARI

Prosedür tanımlamak
PROC/ENDP ile prosedür yazmak, prosedürü programda doğru yere koymak ve assembler direktifini işlemci komutundan ayırmak.

CALL ve RET'i izlemek
Dönüş adresinin yığına nasıl itildiğini ve RET ile nasıl geri alındığını IP, SP ve yığın değerleriyle adım adım göstermek; CALL'ın makine kodundaki bağıl uzaklığı hesaplamak.

Çağrı türlerini ayırt etmek
Yakın, uzak ve dolaylı çağrıların yığına ne ittiğini, RET, RETF ve RET n'in ne çektiğini açıklamak; prosedür türü ile dönüş komutunun uyuşması gerektiğini göstermek.

Parametre geçirmek
Yazmaçlar, bellek değişkenleri, adresler ve yığın yoluyla prosedüre veri göndermek; sonucu yazmaçla, bayrakla ya da bellekle geri almak; değerle ve adresle geçirmeyi karşılaştırmak.

Yığın çerçevesi kurmak
BP ile parametrelere [BP+4], [BP+6] gibi sabit uzaklıklarla erişmek, SUB SP ile yerel değişken alanı açmak ve çerçeveyi doğru sırayla kapatmak.

Güvenli prosedür yazmak
Yazmaçları PUSH/POP ile korumak, yığını dengede tutmak, tek çıkış noktası kullanmak, özyinelemenin yığın ihtiyacını hesaplamak ve sık yapılan hatalardan kaçınmak.

8.2.1 Prosedür kavramı ve tanımlama

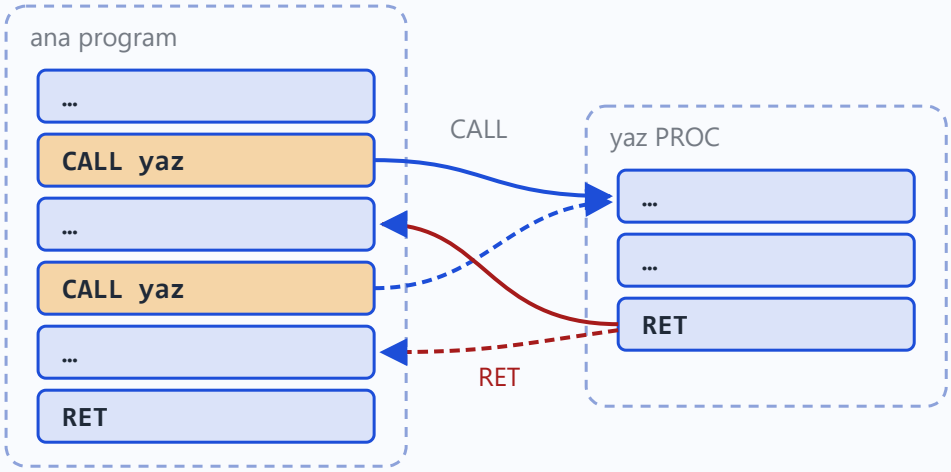
8.2.1.1 Prosedür nedir?

Assembly programları kısa sürede uzar. Ekrana bir dizgi yazmak, bir sayıyı onluk basamaklarına ayırmak ya da bir dizide değer aramak gibi işler programın birçok yerinde gerekir. Aynı komut dizisini her gerektiği yerde yeniden yazmak iki sorun doğurur. Birincisi, program gereksiz yere büyür; 8086 tabanlı küçük sistemlerde bellek sınırlı bir kaynaktır. İkincisi, kodun bir kopyasında bulunan hata bütün kopyalarda vardır ve her biri ayrı ayrı düzeltilmelidir. Yüksek düzeyli dillerdeki fonksiyon ve alt program kavramı bu sorunları çözer. Assembly dilindeki karşılığı prosedürdür.

Tanım 8.2.1

Prosedür (*procedure*, *subroutine*), belirli bir görevi yapan ve programın istenen yerinden çağrılabilen kod parçasıdır. Prosedür `CALL` komutuyla çağrılır. Görevini bitirince `RET` komutuyla, çağrıldığı komutun hemen arkasındaki komuta döner. Çağrı anında yığına yazılan bu adrese **dönüş adresi** (*return address*) denir.

Tanımdaki en önemli kavram dönüş adresidir. Bir prosedür programın farklı yerlerinden çağrılabilir. Her çağrıda dönülecek yer başkadır, ama prosedürün kodu aynıdır. Prosedür, nereye döneceğini kendi kodunda tutamaz. Bu bilgiyi her çağrıda CALL komutu yığına yazar, RET komutu da oradan okur. Böylece aynı RET komutu bir çağrıda bir adrese, başka bir çağrıda başka bir adrese döner.



Şekil 8.2.1: Ana program `yaz` prosedürünü iki kez çağırır. Her RET, kendisini başlatan çağrının hemen arkasındaki komuta döner.

Şekil 8.2.1 bu durumu gösterir. Ana program `yaz` prosedürünü iki ayrı yerden çağırır. Birinci çağrının dönüşü birinci CALL'ın arkasına, ikinci çağrının dönüşü ikinci CALL'ın arkasına yapılır. Prosedürün belleğe tek bir kopyası yüklenmiştir.

Prosedür kullanmanın üç faydası vardır. **Tekrar kullanım**: kod bir kez yazılır, istenen sayıda çağrılır; kod tekrarı ve bellek kullanımı azalır. **Okunabilirlik**: program, her biri adı konmuş küçük görevlere bölünür. `CALL sayi_yaz` satırı, yerine gelecek on beş komuttan çok daha kolay okunur. **Bakım**: hata tek yerde düzeltilir ve prosedür programın geri kalanından ayrı olarak denenebilir. Bu özellikler programı daha yapılandırılmış (*structured*) ve anlaşılır kılar.

Prosedürün bir bedeli de vardır. Her çağrıda işlemci CALL ve RET komutlarını fazladan yürütür, dönüş adresini belleğe yazar ve geri okur. Bu yük saat turu cinsinden hesaplanabilir. Yük küçüktür, ama çok sık çağrılan çok kısa işlerde fark edilir.

8.2.1.2 PROC ve ENDP ile tanımlama

8086 assembly dilinde prosedür, adıyla başlayan bir `PROC` satırı ile aynı adla biten bir `ENDP` satırı arasına yazılır. Genel biçim şöyledir:

```
ad PROC     ; başlangıç (varsayılan NEAR)
    ; prosedür kodu buraya yazılır
    RET
ad ENDP     ; bitiş
```

Bu kalıbın üç kuralı vardır. Birincisi, `ad` prosedürün adıdır ve başta ve sonda aynı yazılmalıdır. `p1 PROC` ile başlayıp `p2 ENDP` ile biten bir tanımı assembler hata sayar. İkincisi, ad etiket kurallarına uyar: harfle ya da alt çizgiyle başlar, rakamla başlayamaz, bir komut adıyla aynı olamaz. Assembler

adı, prosedürün ilk komutunun adresine bağlar. `CALL ad` yazıldığında bu adres kullanılır. Üçüncüsü, prosedürün içinde en az bir `RET` bulunmalıdır. İşlemci ENDP'yi görmez. RET yoksa akış prosedürün son komutundan sonra bellekte ne varsa onunla devam eder.

PROC satırına türü de yazılabilir: `ad PROC NEAR` ya da `ad PROC FAR`. Tür yazılmazsa prosedür **yakın** (NEAR) sayılır. Yakın prosedür, onu çağıran kodla aynı kod kesimindedir. Bu notun örneklerinin çoğu tek kesimli .COM programlarıdır ve bütün prosedürleri yakındır. Uzak prosedürün yığına etkisi CALL ve RET türleriyle birlikte ele alınır.

Prosedürün programdaki yeri de önemlidir. İşlemci komutları bellekteki sırasıyla yürütür ve bir PROC satırında durmaz. Prosedür ana programın ortasına yazılırsa ana program oraya geldiğinde prosedürün komutlarını, çağrılmamış olduğu hâlde yürütür. Bu duruma **prosedüre düşmek** denir. Güvenli yer, ana programın son RET'inden sonrasıdır:

```
ORG  100h
     CALL p1
     RET        ; ana program biter
p1   PROC       ; RET'ten sonra:
     ; ...      ; sırayla yürütülmez
     RET
p1   ENDP
END
```

Burada ana program iki komuttan oluşur. `CALL p1` prosedürü çağırır, ardından gelen `RET` programı bitirir. Prosedürün komutları ana programın RET'inden sonra durduğu için akış onlara yalnız CALL ile ulaşır. Prosedür ana programdan önce yazılmak istenirse ana programın başına, prosedürün üzerinden atlayan bir `JMP` konur. emu8086 örneklerinde ikinci biçim de sık görülür: `JMP start`, ardından prosedürler, ardından `start:` etiketi.

Sık yapılan hata

Ana programın sonundaki RET'i unutmak. Aşağıdaki programda `CALL ikiyle` 'den sonra RET yoktur:

```
ORG 100h
MOV AX, 2
CALL ikiyle
ikiyle PROC
ADD AX, AX
RET
ikiyle ENDP
```

CALL dönüş adresi olarak 0106h'yi iter, prosedür AX'i 4 yapar ve RET 0106h'ye döner. Oysa 0106h prosedürün kendi ilk komutudur. İşlemci prosedüre bir kez daha "düşer", AX = 8 olur ve bu kez RET yığındaki 0000h'yi alarak programı bitirir. Prosedür bir kez çağrıldığı hâlde iki kez çalışmış, sonuç 4 yerine 8 çıkmıştır. Assembler bu hatayı yakalayamaz, çünkü kod söz dizimi bakımından doğrudur.

8.2.1.3 Direktifler ve komutlar

Prosedür tanımında iki farklı türden öğe yan yana durur. `PROC`, `ENDP` ve `END` assembler'a yöneliktir. `CALL` ve `RET` ise işlemcinin yürüttüğü komutlardır. Bu ayrım prosedürlerin nasıl çalıştığını anlamanın anahtarıdır.

Tanım 8.2.2

Direktif (*directive*), assembler'a çeviri sırasında ne yapacağını söyleyen satırdır. Direktif makine koduna çevrilmez ve bellekte yer kaplamaz. **Komut** (*instruction*) ise bir ya da birkaç bayta çevrilen ve çalışma anında işlemcinin yürüttüğü işlemdir.

Öğe	Türü	Görevi
`PROC` / `ENDP`	assembler direktifi	Prosedürün başlangıç adresini ve türünü (NEAR/FAR) assembler'a bildirir. Makine koduna çevrilmez.
`CALL`	işlemci komutu	Dönüş adresini yığına iter, prosedürün adresine atlar. Yakın doğrudan biçimin kodu E8'dir.
`RET`	işlemci komutu	Dönüş adresini yığından alır ve oraya atlar. Yakın biçimin kodu C3'tür. Hem prosedürden hem .COM programından dönmek için kullanılır.
`END`	assembler direktifi	Kaynak dosyanın bittiğini bildirir.

Tablo 8.2.1: Prosedür tanımında kullanılan öğeler.



Şekil 8.2.2: PROC, ENDP ve END yalnız derleme sırasında kullanılır ve bayt üretmez; CALL ve RET işlemcinin çalıştırdığı komutlardır.

Tablo 8.2.1 ve Şekil 8.2.2 bu ayrımı özetler. Sonucu şudur: çalışma anında "prosedür" diye bir nesne yoktur. Bellekte yalnız komut baytları vardır. Prosedürün adı, başı ve sonu yalnız kaynak kodda ve assembler'ın sembol tablosunda yaşar. İşlemci için bir prosedür, CALL ile atlanan ve RET ile terk edilen bir adres aralığından ibarettir. Bu yüzden ENDP satırı bir "bitiş" işi yapmaz. Prosedürü bitiren, içindeki RET komutudur.

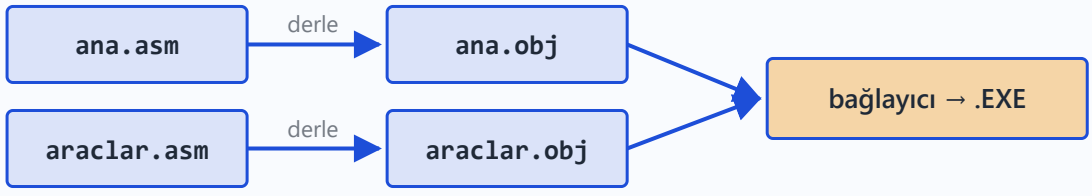
RET'in ikinci kullanımı da bu bakışla anlaşılır. Bir .COM programı da DOS'un çağırdığı bir prosedür gibi biter. DOS programı başlatmadan önce programın kesimindeki ilk 256 bayta, yani PSP'ye (*program segment prefix*, program kesim öneki) `INT 20h` komutunu yazar. Sonra yığına 0000h sözcüğünü iter ve programı 0100h adresinden başlatır. Programın son RET'i yığındaki 0000h'yi IP'ye alır. İşlemci CS:0000h'deki `INT 20h`'yi yürütür ve program sonlanır. emu8086'da bir .COM programı başladığında SP = FFFEh'dir ve [FFFEh] = 0000h'dir. Bu notun bütün izleme tablolarında yığının dibindeki 0000h, DOS'un ittiği bu sözcüktür.

8.2.1.4 Modüler programlama

Büyük programlarda prosedürler de bir kaynak dosyaya sığmaz. Program, her biri birbiriyle ilgili prosedürleri toplayan kaynak dosyalarına bölünür. Bu dosyalara **modül** denir. Bir modüldeki prosedürü başka bir modülden çağırmak için iki direktif kullanılır:

```
; ana.asm
EXTRN yaz:NEAR      ; başka dosyada
      ...
      CALL yaz
; araclar.asm
PUBLIC yaz          ; dışarıya açık
yaz   PROC NEAR
      ...
      RET
yaz   ENDP
```

`PUBLIC yaz`, `yaz` adını başka modüllere açar. `EXTRN yaz:NEAR` ise assembler'a bu adın başka bir modülde tanımlı olduğunu ve türünün NEAR olduğunu söyler. Assembler ana.asm'yi çevirirken `yaz`'ın adresini bilmez. CALL komutunun adres alanını boş bırakır ve nesne dosyasına "buraya yaz'ın adresi gelecek" diye bir not düşer.



Şekil 8.2.3: İki kaynak dosya ayrı ayrı derlenip nesne dosyalarına dönüşür; bağlayıcı bunları tek programda birleştirir.

Her modül ayrı derlenir ve bir nesne dosyası (.OBJ) olur (Şekil 8.2.3). Bağlayıcı (*linker*) nesne dosyalarını art arda yerleştirir, her prosedürün son adresini bulur ve CALL komutlarındaki boş alanları doldurur. Sık kullanılan nesne dosyaları bir kütüphane (.LIB) dosyasında toplanabilir. Bağlayıcı kütüpha-

neden yalnız programın gerçekten çağırdığı prosedürleri alır. Bu yöntemin faydası, değişmeyen modüllerin yeniden derlenmemesi ve aynı kütüphanenin birçok programda kullanılmasıdır.

emu8086 ayrı derleme ve bağlama yapmaz. Ortak kod bu ortamda `INCLUDE` direktifiyle eklenir. INCLUDE, adı verilen dosyanın metnini kaynağa olduğu gibi katar ve program tek parça hâlinde derlenir. emu8086 ile gelen `emu8086.inc` dosyası bu yolla kullanılır. İki yöntemin farkı şudur: bağlayıcı nesne kodunu birleştirir, INCLUDE ise kaynak metni birleştirir.

8.2.2 CALL ve RET

8.2.2.1 CALL ve RET adım adım

CALL ve RET'in işi kısaca "çağır ve dön"dür. İşlemcinin içinde bu iş birkaç küçük adıma ayrılır. Adımları görmek için en küçük örnek kullanılır: ana program `m1` prosedürünü çağırır, m1 BX'e 5 yazar ve döner, ana program AX'e 2 yazar ve biter.

```
ORG     100h
CALL    m1          ; 0100h: E8 04 00
MOV     AX, 2       ; 0103h: B8 02 00
RET                 ; 0106h: C3 → DOS
m1 PROC
MOV     BX, 5       ; 0107h: BB 05 00
RET                 ; 010Ah: C3 → çağırana
m1 ENDP
END
```

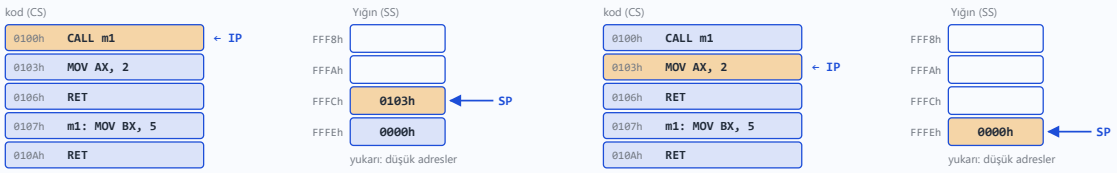
Açıklamalardaki adresler ve baytlar programın bellekteki görünüşüdür. `ORG 100h` ilk komutun 0100h adresine konacağını söyler. CALL 3 bayttır ve 0100h–0102h'yi kaplar. Sonraki komut 0103h'de başlar. Prosedürün ilk komutu, `MOV BX, 5`, 0107h'dedir; dolayısıyla m1 etiketinin değeri 0107h'dir.

> **Tanım 8.2.3**
>
> **Yakın CALL** iki iş yapar: (1) SP ← SP − 2 ve [SS:SP] ← IP; burada IP, CALL'dan sonraki komutun adresidir. (2) IP ← hedef adres. **Yakın RET** bunun tersini yapar: IP ← [SS:SP] ve SP ← SP + 2. İki komut da bayraklara dokunmaz.

Tanımdaki "IP, CALL'dan sonraki komutun adresidir" ifadesi işlemcinin çalışma düzeninden gelir. 8086'da yol arayüz birimi (BIU) komut baytlarını bellekten getirir ve 6 baytlık komut kuyruğuna koyar. Yürütme birimi (EU) komutu kuyruktan alır ve çözer. Komut çözüldüğünde IP, o komutun baytlarının hemen arkasını gösterir. CALL'ın yürütülmeye başladığı anda IP = 0103h'dir. Dönüş adresi diye yığına yazılan değer budur. İşlemci ayrıca bir toplama yapmaz.

Adım	İşlem	IP	SP	Yığın
0	başlangıç, sıradaki komut `CALL m1`	0100h	FFFEh	[FFFEh] = 0000h
1	CALL çözüldü: IP sonraki komutu gösterir	0103h	FFFEh	değişmedi
2	SP ← SP − 2, [SP] ← IP	0103h	**FFFCh**	**[FFFCh] = 0103h**
3	IP ← 0103h + 0004h	**0107h**	FFFCh	değişmedi
4	`MOV BX, 5` yürütüldü	010Ah	FFFCh	değişmedi
5	`RET` : IP ← [SP], SP ← SP + 2	**0103h**	**FFFEh**	[FFFCh]'deki 0103h artık geçersiz
6	`MOV AX, 2` yürütüldü	0106h	FFFEh	[FFFEh] = 0000h

Tablo 8.2.2: `CALL m1` ve `RET` 'in iç adımları (SS = 0700h, başlangıçta SP = FFFEh).



Şekil 8.2.4: Solda CALL dönüş adresini ittikten sonra: [FFFCh] = 0103h, SP = FFFCh. Sağda RET'ten sonra: IP = 0103h, SP = FFFEh; yığın çağrıdan önceki hâline döndü.

Tablo 8.2.2 adımları sırayla verir. Üç noktaya dikkat etmek gerekir. Birincisi, 8086'nın yığını aşağı doğru büyür. CALL SP'yi önce azaltır, sonra yazar. Dönüş adresi bu yüzden FFFEh'nin bir sözcük altına, FFFCh'ye yazılır. İkincisi, adım 3'teki hedef adres bir toplamla bulunur: 0103h + 0004h = 0107h. 0004h değeri CALL komutunun içindeki bağıl uzaklıktır; makine kodu incelenirken ayrıntılı açıklanır. Üçüncüsü, RET yığından okuduğu değeri silmez. 0103h FFFCh adresinde durmaya devam eder, ama SP artık onun üstünü gösterdiği için geçersiz sayılır. Sonraki ilk PUSH ya da CALL bu sözcüğün üstüne yazar.

Bu adımların donanımdaki karşılığı da izlenebilir. CALL'ın yığına yazması bir bellek yazma yol döngüsüdür. Fiziksel adres SS × 16 + SP ile bulunur: 07000h + FFFCh = 16FFCh. Adres çift olduğu için 16 bitlik sözcük tek bir yol döngüsünde yazılır. Bu çevrimde A0 = 0 ve BHE = 0'dır, iki veri baytı D0–D7 ve D8–D15 hatlarından birlikte geçer. SP tek bir değerde olsaydı aynı sözcük iki yol döngüsü gerektirirdi. Bu yüzden SP her zaman çift tutulur. RET aynı adresten bir okuma yol döngüsü yapar.

CALL ve RET'in bir donanım bedeli daha vardır. BIU, CALL'dan sonraki baytları, yani 0103h'deki `MOV AX, 2` 'nin baytlarını kuyruğa çoktan getirmiş olabilir. IP 0107h'ye değişince bu baytlar işe yaramaz. İşlemci kuyruğu boşaltır ve BIU getirmeye yeni adresten başlar. Sıradaki komut, baytları bellekten gelene kadar bekler. RET'te de aynı şey olur. Atlama ve çağrı komutlarının saat turu sayısının yüksek olmasının bir nedeni budur.

8.2.2.2 Basit prosedür: kod izleme

Aynı program şimdi baştan sona komut komut izlenir. Tablo 8.2.3'ün her satırı, o komut yürütüldükten sonraki durumu gösterir. Programın emu8086'da çalıştırılmasıyla elde edilen değerlerle aynıdır.

Satır	Komut	IP (sonraki)	SP	AX	BX	Yığın
—	başlangıç	0100h	FFFEh	0000h	0000h	[FFFEh] = 0000h
2	`CALL m1`	**0107h**	**FFFCh**	0000h	0000h	[FFFCh] = 0103h
6	`MOV BX, 5`	010Ah	FFFCh	0000h	**0005h**	[FFFCh] = 0103h
7	`RET`	**0103h**	**FFFEh**	0000h	0005h	[FFFEh] = 0000h
3	`MOV AX, 2`	0106h	FFFEh	**0002h**	0005h	[FFFEh] = 0000h
4	`RET` (DOS'a)	**0000h**	**0000h**	0002h	0005h	boş

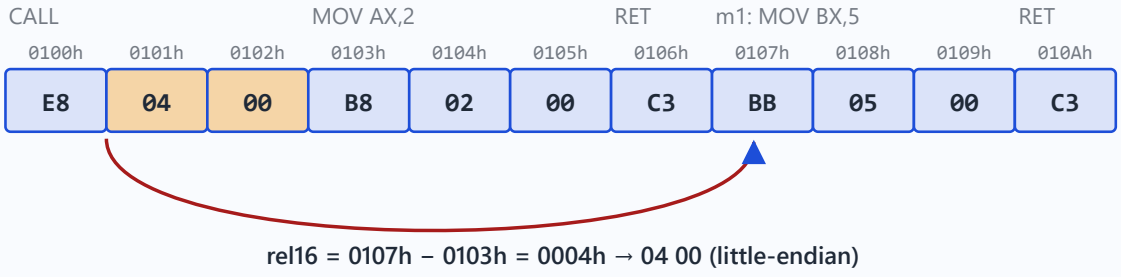
Tablo 8.2.3: m1 programının izlemesi (her satır yürütüldükten sonraki durum).

Satır sırası 2, 6, 7, 3, 4'tür. Akış kaynak koddaki sırayı izlemez: CALL prosedüre atlar, RET geri getirir. Son satırda ana programın RET'i yığındaki 0000h'yi IP'ye alır ve SP, FFFEh + 2 = 10000h'nin 16 bite sığan kısmı olan 0000h olur. İşlemci 0700:0000'daki `INT 20h`'yi yürütür ve program biter. Sonuç AX = 0002h, BX = 0005h'dir.

Tabloda yazmaçların değişmediği satırlara da bakmak öğreticidir. CALL ve RET, AX ve BX'e dokunmaz. Prosedürün yazdığı BX, dönüşten sonra da 0005h'dir. Prosedürle ana program aynı yazmaç kümesini paylaşır; prosedürün özel yazmaçları yoktur. Bu durum parametre geçirmeyi kolaylaştırır. Ama prosedürün çağıranın yazmaçlarını bozma tehlikesi de buradan doğar. Parametre geçirme ve yazmaçların korunması bu iki yüzü ele alır.

8.2.2.3 Bellekteki makine kodu

Assembler kaynak kodu baytlara çevirdiğinde m1 programı bellekte yalnız 11 bayt tutar: `E8 04 00 B8 02 00 C3 BB 05 00 C3`. ORG, PROC, ENDP ve END'den hiçbir iz kalmaz.



Şekil 8.2.5: 0100h–010Ah arasındaki makine kodu baytları. CALL komutunun 16 bitlik uzaklık alanı (rel16) 0004h'dir ve little-endian düzende 04 00 olarak saklanır.

Şekil 8.2.5'teki ilk üç bayt CALL komutudur. E8, "yakın, doğrudan CALL" işlem kodudur. Arkasındaki iki bayt hedefin mutlak adresi değildir. Hedefin, CALL'dan sonraki komuta göre uzaklığıdır.

Tanım 8.2.4

Yakın doğrudan CALL'ın **bağıl uzaklığı** (rel16), hedef adres ile CALL'dan sonraki komutun adresi arasındaki farktır: rel16 = hedef − (CALL'ın adresi + 3). İşlemci çalışırken IP ← IP + rel16 hesabını yapar. Fark 16 bitlik ikiye tümleyen sayıdır; hedef geride ise rel16 negatiftir.

m1 için rel16 = 0107h − (0100h + 3) = 0107h − 0103h = 0004h'dir. 8086 çok baytlı değerleri little-endian düzende saklar: önce düşük bayt (04), sonra yüksek bayt (00). Bu yüzden komut `E8 04 00` olur.

Uzaklık kullanmanın bir faydası vardır. Kod bellekte başka bir yere taşınırsa CALL ile hedef arasındaki fark değişmez ve komut doğru çalışmaya devam eder. Bu özelliğe **konumdan bağımsız kod** denir. Mutlak adres saklansaydı kodun her taşınmasında adresin düzeltilmesi gerekirdi.

Örnek 8.2.1

İki CALL'ın makine kodunu bulalım: (a) `m2` örneğinde 0107h'deki ikinci `CALL m2`, m2 = 010Bh; (b) 0120h'deki `CALL p`, p = 0100h.

1. **Sonraki komut.** (a) 0107h + 3 = 010Ah. (b) 0120h + 3 = 0123h. Bu adresler aynı zamanda yığına itilecek dönüş adresleridir.
2. **Fark.** (a) 010Bh − 010Ah = 0001h. (b) 0100h − 0123h = −23h. 16 bitlik ikiye tümleyende −23h = 10000h − 23h = FFDDh.
3. **Baytlar.** Düşük bayt önce yazılır. (a) `E8 01 00`. (b) `E8 DD FF`.
4. **Sağlama.** (b) için işlemci IP ← 0123h + FFDDh hesabını yapar. Toplam 10100h'dir; 16 bite sığmayan elde atılır ve IP = 0100h olur.

Prosedürün adı da makine kodunda yer almaz. `m1` yalnız assembler'ın sembol tablosunda vardır; makine kodunda ondan geriye yalnız 0107h adresinin uzaklık olarak saklanmış hâli kalır. emu8086'nın hata ayıklayıcısı adları gösterebilir, çünkü sembol tablosunu ayrıca saklar.

8.2.2.4 CALL ve RET türleri

8086'nın birkaç CALL ve RET biçimi vardır. Biçimler iki soruya göre ayrılır. Hedef aynı kod kesiminde mi (yakın), başka bir kesimde mi (uzak)? Hedef komutun içinde mi (doğrudan), bir yazmaçta ya da bellekte mi (dolaylı)?

Komut	Kod	Bayt	Yığına / yığından	Açıklama
CALL yakın	E8 rel16	3	IP	Aynı kesimde doğrudan çağrı (varsayılan).
CALL yazmaç/bellek	FF /2	2–4	IP	Dolaylı yakın çağrı: `CALL BX`, `CALL [tablo+SI]`.
CALL FAR	9A ofs seg	5	CS, sonra IP	Başka kesimdeki prosedür; CS:IP birlikte değişir.
RET	C3	1	IP	Yakın dönüş.

Komut	Kod	Bayt	Yığına / yığından	Açıklama
RETF	CB	1	IP, sonra CS	Uzak dönüş (FAR prosedür).
RET n	C2 n16	3	IP; SP += n	Dönüşte yığındaki n baytlık parametreyi de atar.

Tablo 8.2.4: CALL ve RET biçimleri.

Tablo 8.2.4'teki “FF /2” gösterimi, işlem kodunun FF olduğunu ve ardından gelen ModR/M baytının orta alanının 2 olduğunu söyler. ModR/M baytı işlenenin bir yazmaç mı, yoksa bir bellek adresi mi olduğunu belirtir. Bu yüzden dolaylı CALL'ın uzunluğu işlenene göre 2 ile 4 bayt arasında değişir. `CALL BX` 2 bayttır (FF D3). `CALL tablo[SI]` 16 bitlik bir yer değiştirme taşıdığı için 4 bayttır (FF 94 10 01). Uzak RET'in bir de `RETF n` biçimi vardır (CA n16).

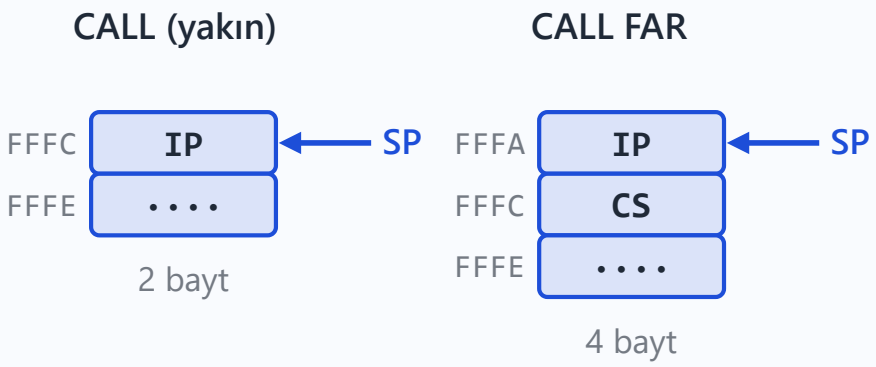
Assembler, prosedür içindeki `RET`'i prosedürün türüne göre kodlar. `ad PROC NEAR` ya da türü yazılmamış bir prosedürde RET, C3 olur. `ad PROC FAR` içinde aynı RET kelimesi CB, yani RETF olarak kodlanır. Programcı türü bir kez PROC satırında yazar, dönüş komutlarını assembler uydurur. CALL da hedefin türüne bakar: hedef FAR bir prosedürse assembler 9A ile başlayan 5 baytlık uzak çağrıyı üretir.



Şekil 8.2.6: Yakın CALL yığına yalnız IP iter (SP − 2); uzak CALL önce CS'yi, sonra IP'yi iter (SP − 4).

8.2.2.5 Uzak prosedür ve uzak dönüş: kod izleme

Uzak (FAR) prosedür, çağıranın kod kesiminden başka bir kod kesimindedir. Ona ulaşmak için hem CS hem IP değişmelidir. Geri dönebilmek için de ikisinin eski değeri saklanmalıdır. Uzak CALL bu yüzden yığına iki sözcük iter: önce CS'yi, sonra IP'yi. RETF bunları ters sırayla çeker: önce IP'yi, sonra CS'yi. Yığındaki sıra RETF'in çekme sırasıyla uyumludur. En son itilen IP yığının tepesindedir ve ilk çekilen odur.



Şekil 8.2.7: Yakın CALL'dan sonra yığında yalnız IP vardır (2 bayt). CALL FAR'dan sonra tepede IP, altında CS durur (4 bayt).

Hangi çağrının gerektiği programın yapısına bağlıdır. Bir .COM programı tek bir 64 KB'lık kesimde çalışır; CS, DS, ES ve SS aynı değeri taşır. Böyle bir programda yakın çağrı yeterlidir. Birden çok kod kesimi olan büyük .EXE programlarında ise başka kesimdeki prosedürlere uzak çağrı gerekir. Uzak çağrı daha uzundur (5 bayt) ve daha yavaştır, çünkü yığına iki sözcük yazar ve CS'yi değiştirir.

Uzak dönüşün yığınla ilişkisini görmek için tek kesimli bir programda küçük bir hile kullanılabilir. CALL FAR'ın iki itmesi iki ayrı komutla yapılır: önce `PUSH CS`, sonra yakın bir CALL. Yığın uzak çağrıdan sonraki düzene girer ve prosedür RETF ile dönebilir.

```
ORG 100h
MOV AX, 5
PUSH CS         ; uzak çağrının ilk yarısı
CALL uzak       ; IP itilir
ADD AX, 1
RET
uzak PROC
SHL AX, 1
RETF            ; önce IP, sonra CS
uzak ENDP
```

Kodun bayt düzeni şöyledir: `MOV AX, 5` 3 bayt (0100h–0102h), `PUSH CS` 1 bayt (0103h), `CALL uzak` 3 bayt (0104h–0106h), `ADD AX, 1` 3 bayt (0107h–0109h), `RET` 1 bayt (010Ah). Prosedür 010Bh'de başlar: `SHL AX, 1` 2 bayt, `RETF` 1 bayt (010Dh, kodu CB). `uzak` PROC satırında tür yazılmadığı için NEAR'dır; programcı içine bilerek RETF yazmıştır.

Satır	Komut	AX	CS:IP (sonraki)	SP	Yığın (tepeden)
2	`MOV AX, 5`	0005h	0700:0103	FFFEh	0000h
3	`PUSH CS`	0005h	0700:0104	**FFFCh**	[FFFCh] = 0700h, 0000h
4	`CALL uzak`	0005h	**0700:010B**	**FFFAh**	[FFFAh] = 0107h, 0700h, 0000h

Satır	Komut	AX	CS:IP (sonraki)	SP	Yığın (tepeden)
8	`SHL AX, 1`	**000Ah**	0700:010D	FFFAh	0107h, 0700h, 0000h
9	`RETF`	000Ah	**0700:0107**	**FFFEh**	0000h
5	`ADD AX, 1`	**000Bh**	0700:010A	FFFEh	0000h

Tablo 8.2.5: PUSH CS + CALL ile RETF izlemesi (her satır yürütüldükten sonraki durum).

İzlemede (Tablo 8.2.5) iki satır önemlidir. **Satır 4'ten sonra** yığının tepesinde dönüş IP'si (0107h), onun altında CS (0700h) vardır. Bu, gerçek bir CALL FAR'ın bırakacağı düzenin aynısıdır. **Satır 9'da** RETF iki sözcük çeker: IP = 0107h, CS = 0700h. SP 4 artarak FFFEh olur. CS zaten 0700h olduğu için değişiklik görünmez, ama işlemci CS'yi yığından yeniden yüklemiştir. Program `ADD AX, 1`'den sürer ve AX = 000Bh ile biter. Gerçek bir uzak çağrı (9A, 5 bayt) aynı iki itmeyi tek komutla yapar.



Şekil 8.2.8: Solda yakın CALL'dan sonra: tepede IP (0107h), altında PUSH CS'nin ittiği 0700h. Sağda RETF'ten sonra: iki sözcük çekildi, SP = FFFEh.

Sık yapılan hata

Prosedürün türü ile dönüş komutunu uyuşturmamak. Yakın CALL ile çağrılan prosedür RETF ile biterse RETF, yığında olmayan bir CS'yi çeker. CALL FAR ile çağrılan prosedür yakın RET ile biterse CS yığında kalır. İki durumda da yığın 2 bayt kayar ve dönüş yanlış adrese yapılır. Bu hata, sık yapılan hatalar arasında adım adım izlenir. Türü PROC satırında doğru yazmak ve dönüş komutu olarak hep `RET` kullanmak bu hatayı önler; assembler RET'i türe göre kodlar.

8.2.2.6 Dolaylı CALL: kod izleme

Şimdiye kadarki CALL'larda hedef, komutun içine yazılmıştı. Hangi prosedürün çağrılacağı program yazılırken belliydi. Bazı durumlarda bu karar çalışma anında verilir. Örneğin bir menüde kullanıcının seçtiği işlem çağrılır. Bu iş için hedef adres bir yazmaçtan ya da bellekten okunur. Böyle çağrıya **dolaylı çağrı** denir.

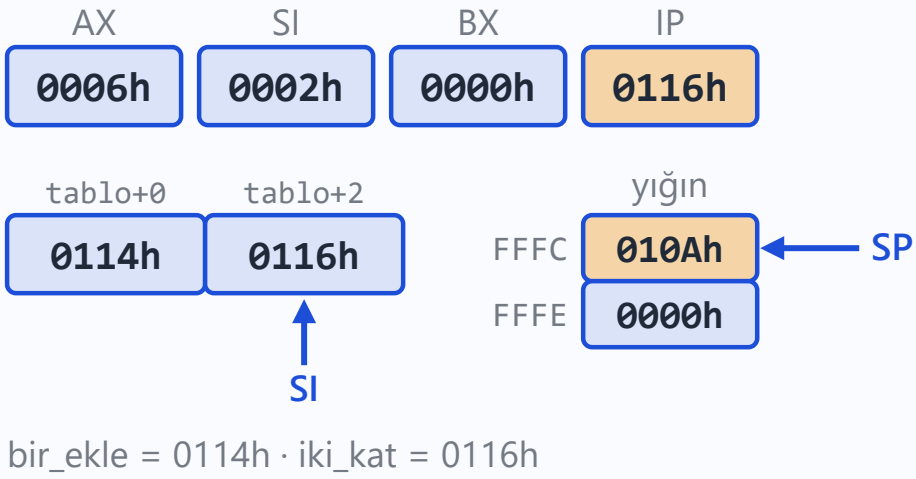
```
MOV AX, 6
MOV SI, 2               ; 2. işlem (× 2)
CALL tablo[SI]          ; bellekle dolaylı
MOV BX, OFFSET bir_ekle
CALL BX                 ; yazmaçla dolaylı
RET
tablo DW bir_ekle, iki_kat
bir_ekle: INC AX
RET
iki_kat: SHL AX, 1
RET
```

`tablo`, iki prosedürün adresini tutan bir sözcük dizisidir. Assembler `DW bir_ekle, iki_kat` satırına iki etiketin adresini yazar: 0114h ve 0116h. Tablo 0110h'de başlar; ilk eleman 0110h'de, ikinci eleman 0112h'dedir. Her eleman 2 bayt olduğu için i indisli elemanın uzaklığı 2 × i'dir (indis 0'dan başlar). SI = 2, tablonun ikinci elemanını, yani `iki_kat`'ın adresini seçer.

`CALL tablo[SI]` etkin adresi tablo + SI = 0110h + 2 = 0112h olarak hesaplar ve oradaki sözcüğü okur: 0116h. Dönüş adresini (010Ah) yığına iter ve IP'ye 0116h'yi yükler. Burada iki bellek erişimi vardır: hedefi okumak için DS kesiminden bir okuma, dönüş adresini yazmak için SS kesimine bir yazma. `CALL BX` ise hedefi doğrudan BX'ten alır ve bellekten hedef okumaz.

Satır	Komut	AX	BX	IP (sonraki)	SP	[FFFCh]
1	`MOV AX, 6`	0006h	0000h	0103h	FFFEh	—
2	`MOV SI, 2`	0006h	0000h	0106h	FFFEh	—
3	`CALL tablo[SI]`	0006h	0000h	**0116h**	**FFFCh**	**010Ah**
10	`iki_kat: SHL AX, 1`	**000Ch**	0000h	0118h	FFFCh	010Ah
11	`RET`	000Ch	0000h	**010Ah**	**FFFEh**	(geçersiz)
4	`MOV BX, OFFSET bir_ekle`	000Ch	**0114h**	010Dh	FFFEh	—
5	`CALL BX`	000Ch	0114h	**0114h**	**FFFCh**	**010Fh**
8	`bir_ekle: INC AX`	**000Dh**	0114h	0115h	FFFCh	010Fh
9	`RET`	000Dh	0114h	**010Fh**	**FFFEh**	(geçersiz)

Tablo 8.2.6: Dolaylı CALL izlemesi (her satır yürütüldükten sonraki durum).



Şekil 8.2.9: `CALL tablo[SI]` anında: hedef [tablo+2] = 0116h'den okunur, dönüş adresi 010Ah yığına itilir.

Sonuçta AX = 13 (000Dh) olur: önce 6 × 2 = 12, sonra 12 + 1 = 13. İzlemede (Tablo 8.2.6) iki çağrının dönüş adresleri farklıdır: birincisi 010Ah, ikincisi 010Fh. İkisi de aynı yığın sözcüğüne, [FFFCh]'ye yazılır, çünkü iki çağrı arasında SP FFFEh'ye dönmüştür. Hangi prosedürün çağrılacağı çalışma anında SI ya da BX ile seçilmiştir. SI = 0 olsaydı ilk çağrı `bir_ekle`'ye giderdi ve aynı kod başka bir sonuç üretirdi.

Kodun bir ayrıntısı daha vardır. `bir_ekle` ve `iki_kat` PROC/ENDP ile değil, sıradan etiketlerle tanımlanmıştır. İşlemci için fark yoktur: CALL bir adrese atlar, RET geri döner. PROC/ENDP, programcıya ve assembler'a düzen ve tür bilgisi sağlar.

8.2.2.7 Atlama tablosuyla menü

Dolaylı çağrının en yaygın kullanımı atlama tablosudur. Bir menüde her seçenek bir prosedürdür ve adresleri bir tabloda sırayla durur. Kullanıcının bastığı tuş tablo indisine çevrilir ve ilgili prosedür tek bir dolaylı CALL ile çağrılır.

```
MOV AH, 1
INT 21h           ; AL = '1'..'3'
SUB AL, '1'       ; 0..2
CMP AL, 2
JA gecersiz       ; aralık denetimi
MOV BL, AL
MOV BH, 0
SHL BX, 1         ; her adres 2 bayt
CALL islem[BX]
gecersiz: RET
islem DW topla, cikar, carp
```

Kod satır satır şöyle çalışır. `INT 21h`'nin 01h işlevi klavyeden bir karakter okur, ekrana yansıtır ve ASCII kodunu AL'ye koyar. `SUB AL, '1'` karakteri sayıya çevirir: '1' (31h) → 0, '2' → 1, '3' → 2. `CMP AL, 2` ve `JA gecersiz` aralık denetimidir; değer 2'den büyükse çağrı yapılmaz. Sonraki üç komut 8 bitlik indisi 16 bitlik BX'e taşır ve 2 ile çarpar. Tablodaki adresler 2 bayt olduğu için indis i'nin bayt uzaklığı 2i'dir. Sola bir bit kaydırmak 2 ile çarpmanın en hızlı yoludur. Son olarak `CALL islem[BX]` seçilen prosedürü çağırır.

Aralık denetiminin bir inceliği vardır. JA işaretsiz karşılaştırma yapar. '0' tuşuna basılırsa `SUB AL, '1'` sonucu 30h − 31h = FFh olur. İşaretli bakışla bu −1'dir ve 2'den küçüktür. İşaretsiz bakışla ise 255'tir ve 2'den büyüktür, bu yüzden JA atlar ve '0' da reddedilir. Tek bir işaretsiz denetim hem alt hem üst sınırı kapatır. '9' tuşu 8 verir ve o da reddedilir.

Tuş	AL (SUB'dan sonra)	BX	Çağrılan
'1'	00h	0000h	`topla`
'2'	01h	0002h	`cikar`
'3'	02h	0004h	`carp`
'0'	FFh	—	hiçbiri (RET)
'9'	08h	—	hiçbiri (RET)

Tablo 8.2.7: Tuşa göre BX ve çağrılan prosedür.

Tablo 8.2.7'deki değerler, üç prosedür kısa gövdelerle eklenip emu8086'da beş tuşla çalıştırıldığında elde edilenlerle aynıdır. Aralık denetimi olmasaydı '9' tuşu BX = 16 verir ve CALL, tablonun 16 bayt ötesindeki iki baytı adres sanıp oraya giderdi. Orada program kodu ya da veri bulunur; işlemci anlamsız bir adrese atlar. Atlama tablosu kullanan her programda indis, tablo boyutuna göre denetlenmelidir.

Atlama tablosunun karşılığı, ardışık CMP ve JE komutlarıyla kurulan bir karar zinciridir. Seçenek sayısı arttıkça zincir uzar ve son seçeneğe ulaşmak için bütün karşılaştırmalar yürütülür. Tablo ise seçenek sayısından bağımsız olarak sabit sayıda komutla hedefe ulaşır. Yüksek düzeyli dillerdeki `switch` yapısını derleyiciler çoğu zaman bu yolla çevirir.

8.2.2.8 CALL ve RET'in süresi

Prosedür çağırmanın bir zaman maliyeti vardır. 8086'nın komut süreleri saat turu (*clock cycle*) cinsinden Intel kılavuzunda verilir. Tablo 8.2.8 prosedürlerle ilgili komutların sürelerini toplar.

Komut	Saat turu	Komut	Saat turu
CALL yakın	19	RET	8
CALL yazmaç	16	RETF	18
CALL FAR	28	RET n	12

Komut	Saat turu	Komut	Saat turu
PUSH yazmaç	11	POP yazmaç	8

Tablo 8.2.8: Çağrıyla ilgili komutların 8086'daki süreleri (saat turu).

Tablodaki değerler yazmaç ve doğrudan biçimler içindir. Bellek işlenenli biçimlere etkin adresin hesap süresi eklenir. Örneğin `CALL tablo[SI]` için kılavuz 21 + EA turu verir; EA, yer değiştirmeli indisli adreslemede 9 turdur. CALL'ın RET'ten pahalı olmasının nedeni, hem yığına yazması hem de hedef adresi hesaplamasıdır. İki komutta da komut kuyruğu boşaltılır ve yeniden doldurulur.

Örnek 8.2.2

İki yazmacı saklayan bir prosedürün, asıl işi dışındaki yükü ne kadardır? 5 MHz'lik bir 8086'da prosedür 1000 kez çağrılırsa bu yük ne kadar sürer?

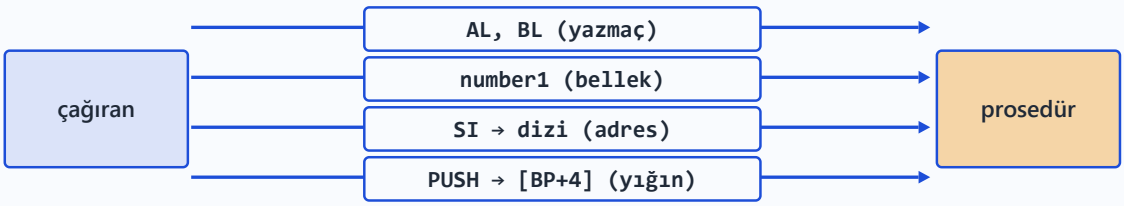
1. **Komutlar.** Her çağrıda bir CALL (19), bir RET (8), iki PUSH (2 × 11) ve iki POP (2 × 8) yürütülür.
2. **Toplam.** 19 + 8 + 2 × (11 + 8) = 27 + 38 = 65 tur.
3. **Süre.** 5 MHz'de bir tur 1 / 5 MHz = 200 ns'dir. 65 × 200 ns = 13.000 ns = 13 µs.
4. **1000 çağrı.** 1000 × 13 µs = 13 ms.

13 ms küçük görünür, ama prosedürün asıl işi birkaç komutluk bir işse yük işin kendisini geçebilir. Bu noktada makro bir seçenektir. Makro, assembler'ın her kullanım yerine gövdeyi kopyaladığı bir metin kalıbıdır. CALL ve RET olmadığı için çağrı yükü taşımaz, ama her kullanımda kod yeniden yer kaplar. Pratik kural şudur: kısa ve çok sık çalışan işler (tek karakter yazmak gibi) makroya, uzun ya da seyrek çağrılan işler prosedüre uygundur.

8.2.3 Parametre geçirme

8.2.3.1 Dört yol

Bir prosedür çoğu zaman bir girdiyle çalışır ve bir sonuç üretir. Ekrana yazan prosedür yazılacak metni, toplayan prosedür toplanacak sayıları bilmelidir. Çağıranın prosedüre verdiği bu değerlere **parametre**, değerlerin prosedüre ulaştırılmasına da **parametre geçirme** (*parameter passing*) denir. Yüksek düzeyli dillerde derleyici bu işi kendisi düzenler. Assembly dilinde yolu programcı seçer ve hem çağıranda hem prosedürde aynı kurala uyar.



Şekil 8.2.10: Parametre prosedüre dört yoldan geçer: yazmaç, bellek değişkeni, adres (işaretçi) ve yığın.

Şekil 8.2.10'daki dört yolun her birinin bir yeri vardır.

1. **Yazmaçlarla.** En kolay ve en hızlı yoldur; parametre zaten işlemcinin içindedir ve bellek erişimi gerekmez. Örneğin AL ve BL iki parametreyi, AX sonucu taşır. Yazmaç sayısı sınırlı olduğu için birkaç parametreden fazlasına yetmez.
2. **Bellek değişkenleriyle.** Prosedür `number1`, `result` gibi ortak değişkenleri okur ve yazar. Kolaydır, ama prosedür o değişkenlere bağımlı kalır. Başka değişkenlerle çalışması için kodunun değişmesi gerekir.
3. **Adres (işaretçi) ile.** Verinin kendisi değil, başlangıç adresi SI, DI ya da BX'te gönderilir. Uzun bir dizi ya da dizgi kopyalanmadan işlenir. Prosedür hangi diziye adresi verilirse onunla çalışır.
4. **Yığınla.** Parametreler CALL'dan önce PUSH ile yığına itilir. Prosedür onları BP yazmacı üzerinden okur. Dönüşte `RET n` ya da çağıranda `ADD SP, n` ile yığından atılır. C ve Pascal gibi yüksek düzeyli dillerin derleyicileri bu yolu kullanır.

Hangi yol seçilirse seçilsin, prosedürün başına bir açıklama bloğu yazılır. Blokta prosedürün girdileri, çıktıları ve bozduğu yazmaçlar belirtilir. Assembly dilinde parametrelerin adı, türü ya da sayısı için derleyici denetimi yoktur. Bu açıklama, prosedür ile çağıran arasındaki sözleşmenin tek yazılı kaydıdır.

8.2.3.2 Değerle mi, adresle mi?

Parametre geçirmenin yolundan bağımsız ikinci bir soru, prosedüre ne verildiğidir: verinin kopyası mı, yoksa verinin yeri mi?

Tanım 8.2.5

Değerle geçirmede (*pass by value*) prosedüre verinin bir kopyası verilir. Prosedür kopyayı değiştirse de bellekteki asıl değişken değişmez. **Adresle geçirmede** (*pass by reference*) prosedüre verinin adresi verilir. Prosedür bu adres üzerinden asıl değişkeni okur ve değiştirebilir.

```
MOV  AX, a          ; değer: AX'te a'nın kopyası
CALL iki_kat        ; AX iki katına çıkar, a aynı kalır
LEA  SI, a          ; adres: SI = OFFSET a
CALL iki_kat_m      ; [SI] iki katına çıkar, a değişir
```

İlk iki satırda `a`'nın içeriği AX'e kopyalanır. `iki_kat` AX'i iki katına çıkarır; sonuç AX'tedir ve `a` bellekte eski değeriyle durur. Son iki satırda `a`'nın adresi SI'ya yüklenir. `iki_kat_m` bellekteki `[SI]` sözcüğünü iki katına çıkarır; değişen, `a`'nın kendisidir.

Değerle geçirme tek sayı gibi küçük veriler için hızlı ve güvenlidir. Prosedür çağıranın değişkenini yanlışlıkla bozamaz. Adresle geçirme üç durumda gerekir: dizi ve dizgi gibi bir yazmaca sığmayan veriler, prosedürün değiştirmesi gereken değişkenler ve birden çok sonucun belleğe yazılması gereken durumlar.

`LEA SI, a` ile `MOV SI, OFFSET a` aynı sonucu verir: SI, `a`'nın bağıl konumunu taşır. emu8086 doğrudan adresli bu LEA'yı MOV ile aynı 3 baytlık koda (`BE` + 16 bitlik adres) çevirir. Bu notun adres hesaplarında LEA 3 bayt sayılmıştır. Gerçek 8086 assembler'ları bu biçimi 4 baytlık `8D 36` kodlamasıyla da üretebilir. `MOV SI, a` ise bambaşka bir komuttur: adresi değil, `a`'nın içindeki sözcüğü SI'ya yükler. Bu karışıklık adresle geçirmenin en sık hatasıdır.

8.2.3.3 Adresle takas: kod izleme

Adresle geçirmenin tipik örneği iki değişkenin değerini değiştiren (takas eden) prosedürdür. Prosedür iki değeri değil, iki yeri alır. Değer alsaydı yalnız kendi kopyalarını değiştirebilirdi.

```
ORG 100h
LEA SI, a
LEA DI, b
CALL takas
RET
takas PROC        ; Girdi: SI, DI (adres)
PUSH AX
MOV AX, [SI]
XCHG AX, [DI]
MOV [SI], AX
POP AX
RET
takas ENDP
a DW 1234h
b DW 00FFh
```

Önce programın bellekteki yerleşimi çıkarılır. İki LEA 3'er bayttır (0100h ve 0103h). `CALL takas` 0106h'dedir ve 3 bayttır, dolayısıyla dönüş adresi 0109h'dir. Ana programın RET'i 0109h'yi, prosedür 010Ah'yi kaplar. Prosedürün komutları sırasıyla 1, 2, 2, 2, 1 ve 1 bayttır ve 0112h'deki RET ile biter. Veri hemen arkasından gelir: `a` = 0113h, `b` = 0115h.

Prosedür AX'i ara değişken olarak kullanır, ama AX onun girdisi ya da çıktısı değildir. Bu yüzden girişte `PUSH AX` ile saklanır ve çıkışta `POP AX` ile geri yüklenir. `XCHG AX, [DI]` takasın kalbidir: AX ile bellekteki sözcüğü tek komutta yer değiştirir. Üç komut (MOV, XCHG, MOV) takası tamamlar.

Satır	Komut	SI	DI	AX	SP	a [0113h]	b [0115h]
2–3	`LEA SI, a` · `LEA DI, b`	**0113h**	**0115h**	0000h	FFFEh	1234h	00FFh
4	`CALL takas`	0113h	0115h	0000h	**FFFCh**	1234h	00FFh
7	`PUSH AX`	0113h	0115h	0000h	**FFFAh**	1234h	00FFh
8	`MOV AX, [SI]`	0113h	0115h	**1234h**	FFFAh	1234h	00FFh
9	`XCHG AX, [DI]`	0113h	0115h	**00FFh**	FFFAh	1234h	**1234h**
10	`MOV [SI], AX`	0113h	0115h	00FFh	FFFAh	**00FFh**	1234h
11	`POP AX`	0113h	0115h	**0000h**	**FFFCh**	00FFh	1234h
12	`RET`	0113h	0115h	0000h	**FFFEh**	00FFh	1234h

Tablo 8.2.9: Adresle takas izlemesi (her satır yürütüldükten sonraki durum).



Şekil 8.2.11: Solda `XCHG AX, [DI]`'den sonra: b = 1234h, AX = 00FFh. Sağda POP ve RET'ten sonra: AX eski değerine döndü, a ile b yer değiştirmiş olarak kaldı.

İzlemede (Tablo 8.2.9) dört noktaya bakılır. SI ve DI baştan sona değişmez; prosedüre gönderilen adreslerdir. AX önce `a`'nın değerini (1234h), XCHG'den sonra `b`'nin eski değerini (00FFh) taşır. Bellekte önce `b`, sonra `a` değişir. Son olarak POP AX, AX'i çağrıdan önceki değerine (0000h) döndürür. Çağıran için tek görünür etki, `a` ile `b`'nin yer değiştirmesidir.

XCHG'nin bellek işlenenli biçimi 8086'da bir okuma ve bir yazma yol döngüsü yapar. Aynı işi iki MOV ile yapmak bir yazmaç daha gerektirirdi. Okuma ile yazma arasında başka bir yol yöneticisinin araya girmemesi gereken durumlarda komutun önüne LOCK öneki konur. Önek, komut süresince 8086'nın LOCK pinini etkin tutar. Çok işlemcili sistemlerde semafor gibi paylaşılan değişkenler bu yüzden `LOCK XCHG` ile değiştirilir.

8.2.3.4 Yazmaçlarla parametre: kod izleme

Yazmaçla parametre geçirmenin örneği olarak iki sayıyı çarpan `m2` prosedürü kullanılır. AL ve BL iki parametreyi temsil eder; m2 bunları çarpar ve sonucu AX'e koyar.



Şekil 8.2.12: AL ve BL girdidir; m2 prosedürü `MUL BL` yapar, sonuç AX'tedir.

Prosedürün tek komutu MUL'dur. MUL işaretsiz çarpma yapar ve işlenen boyutuna göre iki biçimde çalışır (Tablo 8.2.10). Kaynak bir yazmaç ya da bellek işleneni olabilir, sabit olamaz: `MUL 2` geçersizdir. Çarpımın üst yarısı (AH ya da DX) sıfır değilse CF = OF = 1 olur; bu bayraklar sonucun alt yarıya sığıp sığmadığını bildirir. SF, ZF, AF ve PF MUL'dan sonra tanımsızdır.

Kaynak	İşlem	Sonuç
8 bit	AL × kaynak	AX
16 bit	AX × kaynak	DX:AX

Tablo 8.2.10: MUL komutunun iki biçimi.

```
ORG 100h
MOV AL, 1       ; 1. parametre
MOV BL, 2       ; 2. parametre
CALL m2         ; 0104h: E8 04 00
CALL m2         ; 0107h: E8 01 00
RET             ; 010Ah: DOS'a dön
m2 PROC         ; 010Bh
MUL BL          ; AX = AL * BL
RET             ; çağırana dön
m2 ENDP
END
```

`MOV AL, 1` ve `MOV BL, 2` 8 bitlik sabit yüklemeleridir ve 2'şer bayttır (B0 01, B3 02). İki CALL 0104h ve 0107h'dedir. Birincinin dönüş adresi 0107h, ikincininki 010Ah'dir. m2 010Bh'de başlar. İki CALL'ın makine kodları farklıdır (E8 04 00 ve E8 01 00), çünkü uzaklık her CALL'ın kendi konumuna göre hesaplanır. Hedef ikisinde de 010Bh'dir.

Satır	Komut	IP (sonraki)	SP	AX	BL	CF	[FFFCh]
2	`MOV AL, 1`	0102h	FFFEh	0001h	00h	0	—
3	`MOV BL, 2`	0104h	FFFEh	0001h	02h	0	—
4	`CALL m2` (1.)	**010Bh**	**FFFCh**	0001h	02h	0	**0107h**
8	`MUL BL`	010Dh	FFFCh	**0002h**	02h	0	0107h
9	`RET`	**0107h**	**FFFEh**	0002h	02h	0	(geçersiz)

Satır	Komut	IP (sonraki)	SP	AX	BL	CF	[FFFCh]
5	`CALL m2` (2.)	**010Bh**	**FFFCh**	0002h	02h	0	**010Ah**
8	`MUL BL`	010Dh	FFFCh	**0004h**	02h	0	010Ah
9	`RET`	**010Ah**	**FFFEh**	0004h	02h	0	(geçersiz)
6	`RET` (DOS'a)	**0000h**	**0000h**	0004h	02h	0	—

Tablo 8.2.11: m2 programının izlemesi (her satır yürütüldükten sonraki durum).

İzlemenin (Tablo 8.2.11) en öğretici yanı, aynı prosedürün iki farklı dönüş noktasına dönmesidir. Prosedürün RET'i iki çağrıda da aynı komuttur (010Dh). Birinci çağrıda yığında 0107h, ikincide 010Ah bulur ve oraya döner.

İkinci önemli nokta MUL'un AL'yi değiştirmesidir. Birinci çağrıda AX = 1 × 2 = 0002h olur. AL artık 02h'dir. İkinci çağrı AL'yi yeniden yüklemeden yapıldığı için birinci çağrının sonucu ikinci çağrının birinci parametresi olur: AX = 2 × 2 = 0004h. Program bu davranışı bilerek kullanır, ama aynı durum farkında olunmadan bir hataya da yol açabilir. Prosedür açıklamasında “Çıktı: AX” yazması, AL'nin de değiştiğini gösterir. Her iki çarpımda da AH = 0 olduğu için CF = OF = 0'dır.

8.2.3.5 Sonucu bayrakla döndürmek: kod izleme

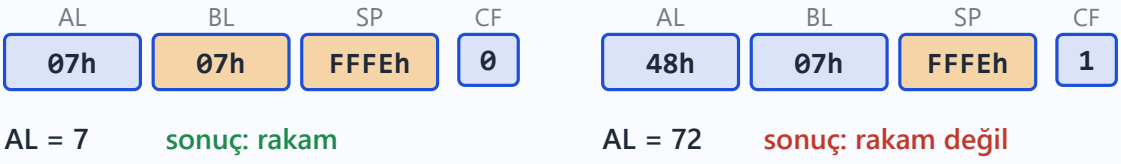
Bazı prosedürlerin sonucu bir sayı değil, bir evet/hayır cevabıdır: dosya açılabildi mi, karakter rakam mı, değer bulundu mu? Böyle bir cevap için bir yazmaç harcamak yerine bir bayrak kullanılır. Elde bayrağı (CF) bu iş için en uygun bayraktır. STC, CLC ve CMC komutlarıyla doğrudan kurulabilir, JC ve JNC ile sınanabilir. DOS hizmetleri de hatayı CF = 1 ile bildirir.

```
MOV AL, '7'
CALL rakam       ; CF = 0 → AL = değer
MOV BL, AL
MOV AL, 'x'
CALL rakam       ; CF = 1 → rakam değil
RET
rakam PROC       ; Girdi AL, Çıktı CF
SUB AL, '0'
CMP AL, 10       ; 0..9 → ödünç, CF = 1
CMC              ; rakamsa CF = 0
RET
rakam ENDP
```

`rakam` prosedürü AL'deki ASCII karakteri sayıya çevirir ve karakterin bir rakam olup olmadığını CF ile bildirir. `SUB AL, '0'` karakterden 30h çıkarır. Karakter '0'–'9' aralığındaysa sonuç 0–9'dur. `CMP AL, 10` işaretsiz karşılaştırma yapar. AL < 10 ise çıkarma ödünç ister ve CF = 1 olur. AL ≥ 10 ise CF = 0 olur. Bu, istenen anlamın tersidir. `CMC` (*complement carry*) CF'yi tersine çevirir: rakam için CF = 0, rakam olmayan için CF = 1. '0'dan küçük bir karakterde SUB sonucu D0h–FFh aralığına sarar ve 10'dan büyük sayılır; o da reddedilir.

Satır	Komut	AL	BL	CF	SP	Açıklama
1	`MOV AL, '7'`	37h	00h	0	FFFEh	'7' = 37h
2, 8	`CALL rakam` · `SUB AL, '0'`	**07h**	00h	0	FFFCh	dönüş 0105h; 37h − 30h = 7
9	`CMP AL, 10`	07h	00h	**1**	FFFCh	7 − 10 ödünç ister
10	`CMC`	07h	00h	**0**	FFFCh	"rakamdır"
11, 3	`RET` · `MOV BL, AL`	07h	**07h**	0	FFFEh	değer saklandı
4, 5, 8	`MOV AL, 'x'` · `CALL` · `SUB`	**48h**	07h	0	FFFCh	78h − 30h = 48h (72); dönüş 010Ch
9	`CMP AL, 10`	48h	07h	0	FFFCh	72 ≥ 10, ödünç yok
10	`CMC`	48h	07h	**1**	FFFCh	"rakam değil"
11	`RET`	48h	07h	1	**FFFEh**	sonuç CF'de

Tablo 8.2.12: Bayrakla dönüş izlemesi (her satır yürütüldükten sonraki durum).



Şekil 8.2.13: Solda birinci dönüşten sonra: CF = 0, BL = 7. Sağda ikinci dönüşten sonra: CF = 1, AL = 48h kullanılmamalıdır.

İzlemeden (Tablo 8.2.12) iki sonuç çıkar. Birincisi, RET bayraklara dokunmaz. Prosedürün içinde kurulan CF, dönüşten sonra çağıranda da geçerlidir. Çağıran `JC hata` ya da `JNC tamam` ile sonucu sınayabilir. İkincisi, CF = 1 durumunda AL bozulmuştur (48h). Prosedürün sözleşmesi şudur: AL'deki değer yalnız CF = 0 ise anlamlıdır. Çağıran önce bayrağa bakmalı, sonra yazmacı kullanmalıdır.

Sık yapılan hata

Sonucu CF ile döndüren bir prosedürde STC, CLC ya da CMC'den sonra bayrakları değiştiren bir komut yürütmek. Örneğin CMC'den sonra çıkışta `ADD SP, 2` ya da `CMP` yapılırsa CF yeniden hesaplanır ve sonuç kaybolur. POP ve RET bayraklara dokunmaz; bayrağı kuran komut, RET'ten önceki son bayrak değiştiren komut olmalıdır.

8.2.3.6 Birden çok sonuç döndürmek

Bir prosedür birden fazla sonuç üretebilir. Bölme işlemi hem bölüm hem kalan verir; arama hem "bulundu mu" cevabını hem de konumu verir. Bunun için üç yol vardır.

- **Yazmaç çifti.** 32 bitlik sonuç DX:AX'te döner (16 bitlik MUL gibi). Bölüm ile kalan AL ve AH'de döner (8 bitlik DIV gibi).
- **Bayrak ve yazmaç.** CF başarıyı, yazmaç değeri bildirir. Çağıran önce JC ile bayrağa bakar.

- **Bellek.** Sonuçlar adresi verilen bir alana yazılır. Sonuç sayısı yazmaç sayısıyla sınırlı değildir.

Aşağıdaki `bolkalan` prosedürü ilk iki yolu birlikte kullanır. AX'i BL'ye böler, bölümü AL'de, kalanı AH'de döndürür. Bölen sıfırsa bölme yapmaz ve CF = 1 ile hata bildirir.

```
bolkalan PROC      ; Girdi: AX, BL (bölen)
CMP BL, 0          ; Çıktı: AL = bölüm
JE hata            ;        AH = kalan
DIV BL             ; CF = 1: bölen 0
CLC
RET
hata: STC
RET
bolkalan ENDP
```

8 bitlik `DIV BL`, AX'teki 16 bitlik sayıyı BL'ye böler. Bölümü AL'ye, kalanı AH'ye yazar. DIV'den sonra bütün bayraklar tanımsızdır. Bu yüzden başarı durumunda CF açıkça `CLC` ile sıfırlanır. Prosedürde iki RET vardır. Yığına hiçbir şey itilmediği için bu güvenlidir: iki RET de tepede dönüş adresini bulur.

Girdi	Sonuç	Açıklama
AX = 100, BL = 7	AL = 0Eh (14), AH = 02h, CF = 0	100 = 7 × 14 + 2; AX = 020Eh
AX = 100, BL = 0	CF = 1, AX = 0064h (değişmez)	DIV hiç yürütülmez
AX = 1000, BL = 2	bölme hatası (INT 0)	bölüm 500, AL'ye sığmaz

Tablo 8.2.13: bolkalan prosedürünün üç girdiyle sonuçları.

Tablo 8.2.13'ün son satırı prosedürün bir açığını gösterir. Bölen sıfır olmadığı hâlde 1000 / 2 = 500 sonucu 8 bitlik AL'ye sığmaz (en büyük değer 255). 8086 bu durumda da tip 0 kesmesini (bölme hatası) üretir. Prosedür yalnız bölenin sıfır olmasını denetlediği için bu durumu yakalayamaz. Tam bir denetim için AH < BL koşulu da sınanmalıdır; bölüm ancak bu koşulda AL'ye sığar.

İki RET'li yapı yalnız yığına bir şey itilmemişse güvenlidir. PUSH yapan bir prosedürde bütün yollar, POP'ların bulunduğu tek bir çıkış etiketinde birleştirilmelidir. Bu kural, tek çıkış noktalı bir arama prosedürünün izlemesiyle gösterilir.

8.2.4 Yazmaçların korunması

8.2.4.1 Sorun ve belgeleme

8086'nın sekiz genel amaçlı yazmacı vardır ve prosedür ile çağıran bunları paylaşır. Prosedür, çağıranın kullandığı bir yazmacı habersizce değiştirirse çağıran kod bozulur. En tipik örnek döngü sayacıdır. Çağıran CX ile bir döngü kurar ve döngünün içinden bir prosedür çağırır. Prosedür de kendi döngüsü için CX kullanırsa, dönüşte CX'te çağıranın sayacı değil prosedürün bıraktığı değer bulunur.

Sorunun ilk çaresi belgelemedir. Her prosedürün başına girdisi, çıktısı ve bozduğu yazmaçlar yazılır:

```
; m2: AL * BL hesaplar
; Girdi : AL, BL (işaretsiz)
; Çıktı : AX = AL * BL
; Bozar : CF, OF
```

Bu açıklamayı okuyan biri, m2'yi çağırdıktan sonra AX'teki eski değerin kaybolacağını ve CF ile OF'nin değişeceğini bilir. Başka hiçbir yazmacın değişmediği de bu açıklamanın bir vaadidir. Vaat tutulmazsa, yani prosedür açıklamada yazmayan bir yazmacı değiştirirse, hata çoğu zaman prosedürde değil, onu çağıran kodda ortaya çıkar. Hatanın yeri ile nedeninin farklı olması onu bulmayı zorlaştırır.

Tanım 8.2.6

Yazmaçların korunması (*register preservation*), bir prosedürün kullandığı yazmaçları girişte yığına itmesi ve çıkışta ters sırayla geri almasıdır. Böylece bu yazmaçlar çağırana hiç değişmemiş görünür. Sonuç döndüren yazmaçlar korunmaz.

```
yaz PROC
    PUSH AX     ; kullanacaklarını
    PUSH SI     ; sakla
    ; ... AX ve SI kullanılır
    POP SI      ; TERS sırayla
    POP AX      ; geri yükle
    RET
yaz ENDP
```

Ters sıra yığının son giren ilk çıkar (LIFO) yapısından gelir. AX önce, SI sonra itildiği için yığının tepesinde SI'nın değeri durur. İlk POP o değeri alır ve SI'ya yazmalıdır. Sıra karıştırılırsa, yani önce `POP AX` yazılırsa, AX'e SI'nın, SI'ya AX'in eski değeri yüklenir. Yığın yine dengelidir ve RET doğru döner, ama iki yazmaç birbirinin değerini almıştır.

Tanımın son cümlesi de önemlidir. Sonucunu AX'te döndüren bir prosedür AX'i girişte itip çıkışta geri alırsa, hesapladığı sonucu kendi eliyle siler. Yığın dengeli olduğu için program çökmez; yalnız sonuç yanlış çıkar. Kural şudur: prosedür kullandığı yazmaçları korur, sonuç yazmaçlarını korumaz.

8.2.4.2 PUSH CX ile sayaç koruma: kod izleme

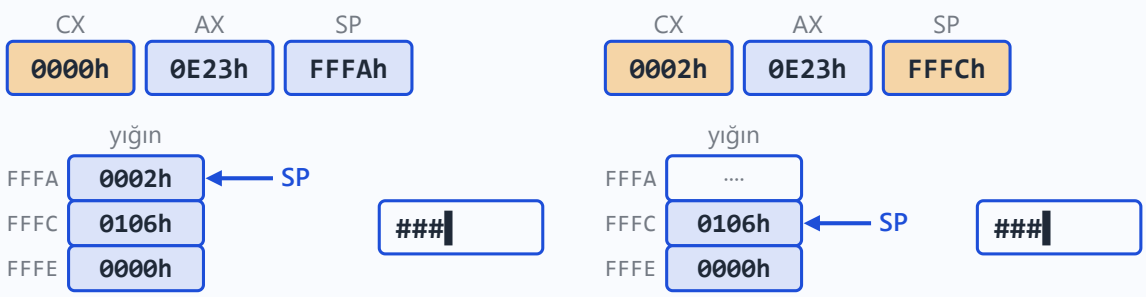
Aşağıdaki program ekrana iki satır yazmak ister. Ana döngü CX = 2 ile döner ve her turda `yildiz` prosedürünü çağırır. Prosedür de üç `#` yazmak için CX'i iç döngü sayacı olarak kullanır. Prosedürün ilk ve son komutları çağıranın sayacını korur.

```
MOV CX, 2          ; ana döngü: 2 satır
satir: CALL yildiz
LOOP satir
RET
yildiz PROC
PUSH CX            ; çağıranın sayacı
MOV CX, 3          ; iç sayaç
MOV AX, 0E23h      ; AH = 0Eh, AL = '#'
ic: INT 10h
LOOP ic
POP CX             ; sayaç geri
RET
yildiz ENDP
```

`MOV AX, 0E23h` tek komutla iki yazmaç yarısını doldurur. AH = 0Eh, BIOS ekran hizmetinin (INT 10h) "teletype" işlevidir: AL'deki karakteri imlecin bulunduğu yere yazar ve imleci ilerletir. AL = 23h, '#' karakterinin ASCII kodudur. LOOP komutu önce CX'i 1 azaltır, sonra CX ≠ 0 ise etikete atlar. İç döngü bu yüzden tam üç tur döner.

Satır	Komut	CX	SP	Yığın (tepeden)	Ekran
1	`MOV CX, 2`	0002h	FFFEh	0000h	
2	`CALL yildiz`	0002h	**FFFCh**	0106h, 0000h	
6	`PUSH CX`	0002h	**FFFAh**	0002h, 0106h, 0000h	
7–8	`MOV CX, 3` · `MOV AX, 0E23h`	**0003h**	FFFAh	0002h, 0106h, 0000h	
9–10	iç döngü, 3 tur	**0000h**	FFFAh	0002h, 0106h, 0000h	###
11	`POP CX`	**0002h**	**FFFCh**	0106h, 0000h	###
12	`RET`	0002h	**FFFEh**	0000h	###
3	`LOOP satir`	**0001h**	FFFEh	0000h	###
2–12	ikinci çağrı	0001h	FFFEh	0000h	######
3	`LOOP satir`	**0000h**	FFFEh	0000h	######

Tablo 8.2.14: PUSH CX ile sayaç koruma izlemesi (her satır yürütüldükten sonraki durum).



Şekil 8.2.14: Solda iç döngü bittiğinde: CX = 0, yığında çağıranın sayacı (2) ve dönüş adresi. Sağda POP CX'ten sonra: CX = 2, tepede dönüş adresi.

İzlemede (Tablo 8.2.14) CX'in yolculuğu açıkça görülür. Çağıranın sayacı (2) PUSH ile yığına gider. İç döngü CX'i 3'ten 0'a indirir. POP CX çağıranın değerini geri getirir. Ana LOOP bu değeri 1'e indirir ve ikinci satır için prosedürü yeniden çağırır. İkinci çağrıda yığına 1 itilir ve geri gelir. Son LOOP CX'i 0 yapar ve döngü biter. Ekranda 6 karakter vardır.

Prosedürün içinde yığının en derin anında (iç döngü sırasında) yığında iki sözcük vardır: dönüş adresi ve saklanan CX. SP = FFFAh'dir. Bu düzen, çıkıştaki RET'in neden doğru çalıştığını da açıklar: POP CX saklanan değeri aldıktan sonra tepede yine dönüş adresi durur.

> **Sık yapılan hata**
>
> CX'i kullanan bir prosedürü LOOP ile kurulmuş bir döngünün içinden, CX'i korumadan çağırmak. PUSH CX / POP CX olmasaydı prosedür CX = 0 ile dönerdi. Ana döngüdeki LOOP önce CX'i 1 azaltır: 0 − 1 = FFFFh. FFFFh sıfır olmadığı için atlar. Her çağrı CX'i yeniden 0 yapacağı için döngü bir daha bitmez ve ekrana durmadan karakter yazılır. Çözüm iki yerden birine konur: ya prosedürde PUSH CX / POP CX, ya da çağıranda `PUSH CX / CALL yildiz / POP CX`.

8.2.4.3 Çağıran mı korur, çağrılan mı?

Yazmaçları saklama görevi iki yere verilebilir: prosedürün kendisine (çağrılan korur) ya da onu çağıran koda (çağıran korur). İki yaklaşımın güçlü ve zayıf yanları Tablo 8.2.15'tedir.

Ölçüt	Çağrılan korur (prosedürde PUSH/POP)	Çağıran korur (CALL'ın çevresinde)
Kod nerede?	prosedürde bir kez	her çağrı noktasında
Gereksiz saklama	çağıran o yazmacı kullanmasa da saklanır	yalnız gerçekten gereken yazmaçlar
Unutma riski	düşük: tek yerde yazılır	yüksek: her çağrıda tekrar gerekir
Uygun olduğu durum	kütüphane prosedürleri	çıktı yazmacındaki eski değerin gerektiği yer

Tablo 8.2.15: Çağrılan korur ve çağıran korur yaklaşımlarının karşılaştırması.

Çağıranın koruması gereken durumun örneği şöyledir. `sayi_oku` prosedürü klavyeden okuduğu sayıyı CX'te döndürür. CX onun çıktısıdır, prosedür onu koruyamaz. Çağıran ise CX'i döngü sayacı olarak kullanır. Bu durumda sayacı saklamak çağıranın işidir:

```
dongu: PUSH CX        ; çağıran: sayaç
       CALL sayi_oku  ; sonuç CX'te
       ADD toplam, CX
       POP CX         ; sayaç geri
       LOOP dongu
```

PUSH CX ile POP CX arasında CX iki farklı anlam taşır. Önce okunan sayıdır ve `toplam`'a eklenir; sonra yığından gelen döngü sayacıdır. Uygulamadaki kural iki yaklaşımı birleştirir: prosedür kullandığı yazmaçları korur, çıktı yazmaçlarını korumaz. Çıktı yazmacındaki eski değer gerekiyorsa onu çağı-

ran saklar.

8.2.4.4 Tek çıkış noktası: kod izleme

Yazmaç saklayan bir prosedürün birden çok dönüş yolu varsa bir tehlike doğar. Yollardan biri POP'ları atlayıp doğrudan RET'e giderse RET, yığının tepesinde dönüş adresi yerine saklanmış bir yazmaç değeri bulur. Bu yüzden PUSH yapan prosedürlerde bütün yollar, POP'ların bulunduğu tek bir çıkış etiketinde birleşir.

```
ORG 100h
LEA SI, dizi
MOV CX, 5
MOV AL, 7
CALL bul          ; CF = 1: bulundu
RET
bul PROC          ; Girdi: SI, CX, AL
PUSH CX
PUSH SI
XOR BX, BX        ; Çıktı: BX = indis
ara: CMP [SI], AL
JE buldu
INC SI
INC BX
LOOP ara
CLC               ; bulunamadı
JMP son
buldu: STC
son: POP SI       ; tek çıkış noktası
POP CX
RET
bul ENDP
dizi DB 3, 9, 7, 1, 4
```

`bul` prosedürü, SI'nın gösterdiği bayt dizisinde CX eleman içinde AL'deki değeri arar. Sonucu iki yolla döndürür: CF = 1 ise değer bulunmuştur ve BX onun indisidir. CF = 0 ise değer dizide yoktur. Prosedür arama sırasında SI'yı ve CX'i değiştirir. İkisi de girdidir ve çağıran onları bir sonraki çağrıda kullanmak isteyebilir, bu yüzden korunur. BX ise çıktıdır ve korunmaz. `XOR BX, BX` bir yazmacı sıfırlamanın kısa yoludur (2 bayt).

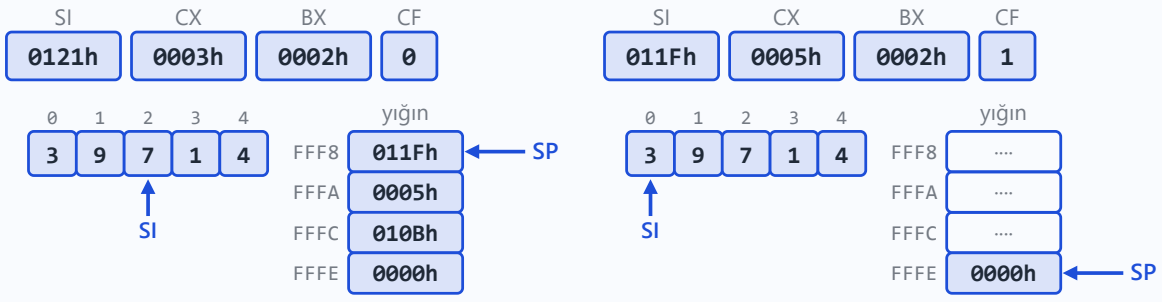
Programın yerleşimi şöyledir: LEA 3 bayt (0100h), `MOV CX, 5` 3 bayt (0103h), `MOV AL, 7` 2 bayt (0106h), CALL 3 bayt (0108h). Dönüş adresi 010Bh'dir. Prosedür 010Ch'de başlar ve 011Eh'deki RET ile biter. Dizi 011Fh'dedir; SI'nın başlangıç değeri budur.

Satır	Komut	SI	CX	BX	CF	SP	Yığın (tepeden)
2–4	LEA, MOV, MOV	011Fh	0005h	0000h	0	FFFEh	0000h
5	`CALL bul`	011Fh	0005h	0000h	0	**FFFCh**	010Bh

Satır	Komut	SI	CX	BX	CF	SP	Yığın (tepeden)
8–10	PUSH CX, PUSH SI, XOR	011Fh	0005h	0000h	0	**FFF8h**	011Fh, 0005h, 010Bh
11–15	dizi[0] = 3 ≠ 7	**0120h**	**0004h**	**0001h**	1	FFF8h	011Fh, 0005h, 010Bh
11–15	dizi[1] = 9 ≠ 7	**0121h**	**0003h**	**0002h**	0	FFF8h	011Fh, 0005h, 010Bh
11–12	dizi[2] = 7: ZF = 1, JE atlar	0121h	0003h	0002h	0	FFF8h	011Fh, 0005h, 010Bh
18	`buldu: STC`	0121h	0003h	0002h	**1**	FFF8h	011Fh, 0005h, 010Bh
19–20	`POP SI` · `POP CX`	**011Fh**	**0005h**	0002h	1	**FFFCh**	010Bh
21	`RET`	011Fh	0005h	0002h	1	**FFFEh**	0000h

Tablo 8.2.16: Tek çıkış noktası izlemesi (her satır grubu yürütüldükten sonraki durum).

Tablodaki CF sütununun ilk turda 1, sonra 0 olması dikkat çekebilir (Tablo 8.2.16). Bunun nedeni CMP'dir. `CMP [SI], AL` 3 − 7 hesabını yapar ve ödünç istediği için CF = 1 olur. İkinci turda 9 − 7 ödünç istemez, CF = 0 olur. LOOP ve INC CF'ye dokunmaz. Arama sırasındaki CF değerleri bir anlam taşımaz. Anlamlı olan, prosedürün çıkışta STC ya da CLC ile kurduğu değerdir.



Şekil 8.2.15: Solda değer bulunduğu anda: döngü erken bitiyor, ama yığında hâlâ iki saklı değer var. Sağda RET'ten sonra: SI ve CX eski değerlerinde, CF = 1, BX = 2.

Kritik an, değerin bulunduğu üçüncü turdur. JE `buldu`'ya atlar ve döngü erken biter. O anda yığının tepesinde dönüş adresi değil, saklanmış SI (011Fh) durur (Şekil 8.2.15). `buldu` STC ile CF'yi kurar ve akış `son` etiketine geçer. Bulunamama yolu da (CLC ve JMP son) aynı etikete gelir. İki yol da POP SI ve POP CX'ten geçtiği için RET her durumda tepede 010Bh'yi bulur.

Programcı JE'nin hedefinde doğrudan bir RET kullanmış olsaydı, yani `buldu: STC` / `RET` yazsaydı, RET tepedeki 011Fh'yi dönüş adresi sanırdı. İşlemci, dizinin başladığı 011Fh adresindeki 03h, 09h, 07h baytlarını komut olarak yürütmeye çalışırdı. Sonuç öngörülemez olurdu. Program ancak arama başarısız olduğunda doğru çalışırdı ve hata yalnız bazı girdilerde ortaya çıkardı.

8.2.5 Yığınla parametre ve yığın çerçevesi

8.2.5.1 Yığınla parametre geçirme: kod izleme

Yazmaçla parametre geçirmenin iki sınırı vardır: yazmaç sayısı azdır ve her prosedür hangi yazmacın ne taşıdığını ayrıca belgelemelidir. Yığınla parametre geçirme ikisini de aşar. Çağıran parametreleri CALL'dan önce sırayla yığına iter. Prosedür onları yığındaki sabit yerlerinden okur. Parametre sayısı yığının büyüklüğüyle sınırlıdır ve kural her prosedür için aynıdır.

Prosedür parametreleri nasıl bulur? SP'yi kullanamaz: 8086'da SP bir adresleme yazmacı değildir, `MOV AX, [SP+4]` geçersizdir. Ayrıca SP, prosedür her PUSH ve POP yaptığında değişir; aynı parametrenin uzaklığı prosedürün farklı yerlerinde farklı olurdu. Çözüm BP yazmacıdır. BP'yi içeren adresler varsayılan olarak SS kesimini kullanır. Prosedür girişte BP'ye SP'nin değerini kopyalar ve böylece yığında sabit bir dayanak noktası kurar.

```
ORG 100h
MOV AX, 7
PUSH AX          ; 1. parametre
MOV AX, 3
PUSH AX          ; 2. parametre
CALL TOPLA
ADD SP, 4        ; parametreleri at
RET
TOPLA PROC
PUSH BP
MOV BP, SP
MOV AX, [BP+6]   ; 1. parametre
ADD AX, [BP+4]   ; 2. parametre
POP BP
RET
TOPLA ENDP
```

Programın yerleşimi izleme tablosundaki IP değerlerini açıklar. Ana program 0100h–010Eh'yi kaplar: `MOV AX, 7` (3 bayt), `PUSH AX` (1), `MOV AX, 3` (3), `PUSH AX` (1), `CALL TOPLA` (3, 0108h'de), `ADD SP, 4` (3, 010Bh'de) ve `RET` (1). TOPLA 010Fh'de başlar. `PUSH BP` 1, `MOV BP, SP` 2, iki bellek işlenenli komut 3'er, `POP BP` ve `RET` 1'er bayttır.

Satır	Komut	AX	BP	SP	IP (sonraki)	Yığına yazılan
2	`MOV AX, 7`	0007h	0000h	FFFEh	0103h	
3	`PUSH AX`	0007h	0000h	**FFFCh**	0104h	[FFFCh] = 0007h
4	`MOV AX, 3`	0003h	0000h	FFFCh	0107h	
5	`PUSH AX`	0003h	0000h	**FFFAh**	0108h	[FFFAh] = 0003h
6	`CALL TOPLA`	0003h	0000h	**FFF8h**	**010Fh**	[FFF8h] = 010Bh

Satır	Komut	AX	BP	SP	IP (sonraki)	Yığına yazılan
10	`PUSH BP`	0003h	0000h	**FFF6h**	0110h	[FFF6h] = 0000h
11	`MOV BP, SP`	0003h	**FFF6h**	FFF6h	0112h	
12	`MOV AX, [BP+6]`	**0007h**	FFF6h	FFF6h	0115h	
13	`ADD AX, [BP+4]`	**000Ah**	FFF6h	FFF6h	0118h	
14	`POP BP`	000Ah	**0000h**	**FFF8h**	0119h	
15	`RET`	000Ah	0000h	**FFFAh**	**010Bh**	
7	`ADD SP, 4`	000Ah	0000h	**FFFEh**	010Eh	
8	`RET` (DOS'a)	000Ah	0000h	**0000h**	**0000h**	

Tablo 8.2.17: Yığınla parametre geçirme izlemesi (her satır yürütüldükten sonraki durum).

Tablo 8.2.17'deki ilk beş satır çağıranın hazırlığıdır. İki PUSH yığına 7 ve 3'ü yazar, CALL bunların üstüne dönüş adresini (010Bh) koyar. Prosedüre girildiğinde yığın, SP'den yukarı doğru şöyledir: [FFF8h] dönüş adresi, [FFFAh] ikinci parametre (3), [FFFCh] birinci parametre (7).

Prosedürün ilk iki komutu, `PUSH BP` ve `MOV BP, SP`, çerçeveyi kurar. Çağıranın BP'si yığına saklanır, çünkü çağıran da BP'yi kendi çerçevesi için kullanıyor olabilir. Sonra BP = SP = FFF6h olur. Bu andan sonra her parametrenin BP'ye göre uzaklığı sabittir:

- `[BP+0]` = [FFF6h]: saklanan eski BP (0000h),
- `[BP+2]` = [FFF8h]: dönüş adresi (010Bh),
- `[BP+4]` = [FFFAh]: son itilen parametre (3),
- `[BP+6]` = [FFFCh]: ilk itilen parametre (7).

Parametrelerin BP'ye göre uzaklığı, prosedürün içinde SP ne kadar değişirse değişsin aynı kalır. Prosedür arada başka yazmaçlar itse de `[BP+6]` hep birinci parametredir. `MOV AX, [BP+6]` fiziksel adresi SS × 16 + BP + 6 = 07000h + FFFCh = 16FFCh olarak hesaplar. Kesim öneki gerekmez, çünkü BP'li adresler varsayılan olarak SS'yi kullanır. Aynı adres DS ile hesaplansaydı tek kesimli .COM programında yine doğru çıkardı, ama DS ≠ SS olan .EXE programlarında yanlış olurdu.



Şekil 8.2.16: Solda `MOV AX, [BP+6]` anında: BP = FFF6h, parametreler [BP+4] ve [BP+6]'da. Sağda prosedürün RET'inden sonra: SP = FFFAh, parametreler hâlâ yığında.

Çıkış girişin tersidir. `POP BP` çağıranın BP'sini geri getirir ve SP = FFF8h olur; tepede dönüş adresi vardır. RET 010Bh'ye döner ve SP = FFFAh olur. Bu noktada iki parametre hâlâ yığındadır (Şekil 8.2.16). Çağıranın `ADD SP, 4` komutu onları atar: SP, PUSH'lardan önceki değerine, FFFEh'ye döner. Yığın dengelenmiştir ve ana programın RET'i doğru olarak 0000h'yi bulur. AX = 000Ah = 10 sonucu taşır.

8.2.5.2 Parametreleri kim atar? RET n

Yığına itilen parametrelerin bir yerde atılması gerekir; yoksa her çağrı yığında birkaç sözcük bırakır. Bu işi ya çağıran ya da prosedür yapar.

Tanım 8.2.7

RET n, dönüş adresini yığından alıp IP'ye yükledikten sonra SP'ye ayrıca n ekleyen dönüş komutudur: IP ← [SP], SP ← SP + 2 + n. n, komutun içinde 16 bitlik bir sabit olarak saklanır (C2 n16, 3 bayt) ve parametre baytlarının sayısıdır.

TOPLA prosedüründe son komut `RET 4` olsaydı çağıranın `ADD SP, 4` yazmasına gerek kalmazdı. RET 4 çalışırken SP toplam 6 artar: dönüş adresi için 2, iki parametre için 4. SP, PUSH'lardan önceki değerine (FFFEh) tek komutta döner.

İki yöntemin farkı, kimin parametre sayısını bildiğidir. RET n'de sayı prosedürün içine sabitlenmiştir. Prosedür her zaman aynı sayıda parametre almalıdır. Pascal derleyicileri bu yöntemi kullanır. ADD SP, n'de ise sayıyı çağıran bilir. Aynı prosedür farklı çağrılarda farklı sayıda parametre alabilir. C dilinin değişken sayıda parametre alan `printf` gibi fonksiyonları bu yüzden çağıranın temizlediği yöntemle çalışır. Hangi yöntem seçilirse seçilsin, çağıran ile prosedür aynı kurala uymalıdır. İkisi birden temizlerse yığın 4 bayt fazla geri verilir; hiçbiri temizlemezse 4 bayt yığında kalır.

Örnek 8.2.3

Çağıran SP = FFFEh iken üç parametre itiyor ve yakın CALL yapıyor. Prosedür `PUSH BP` ve `MOV BP, SP` ile başlıyor. BP kaç olur, ilk itilen parametre nerededir, prosedür nasıl dönmelidir?

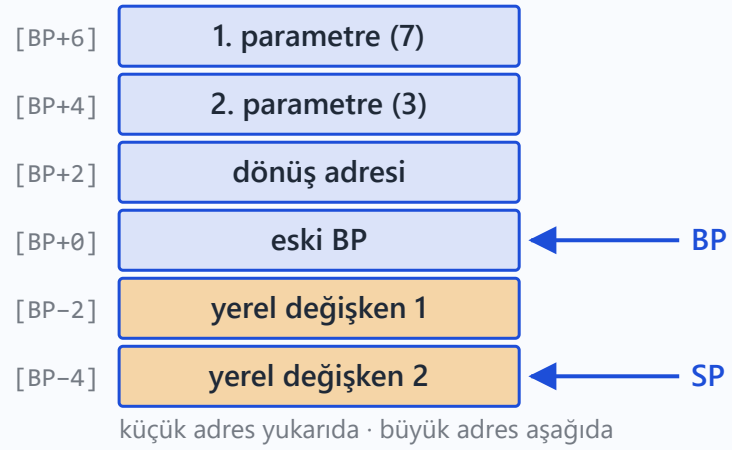
1. **İtmeler.** Üç parametre 6 bayt, CALL 2 bayt, PUSH BP 2 bayt: FFFEh − 10 = FFF4h. BP = FFF4h olur.
2. **Uzaklıklar.** [BP] eski BP, [BP+2] dönüş adresi, [BP+4] son itilen (üçüncü) parametre, [BP+6] ikinci, [BP+8] ilk itilen parametredir.
3. **Dönüş.** POP BP'den sonra `RET 6` : SP = FFF6h + 2 + 6 = FFFEh. Yığın çağrıdan önceki hâline döner.

8.2.5.3 Yığın çerçevesi ve yerel değişkenler

Prosedür çalışırken ara sonuçlarını saklayacak bir yere ihtiyaç duyabilir. Yazmaçlar yetmeyebilir. Ara sonuçları genel bellek değişkenlerinde tutmak ise prosedürü o değişkenlere bağlar ve özyinelemeli çağrıda her düzeyin aynı değişkeni bozmasına yol açar. Doğru yer yine yığındır.

Tanım 8.2.8

Yığın çerçevesi (*stack frame*), bir prosedür çağrısına ait yığın bölgesidir: parametreler, dönüş adresi, saklanan eski BP ve **yerel değişkenler** (*local variables*). Yerel değişken, yalnız prosedür çalışırken yığında yer alan değişkendir. BP çerçevenin sabit dayanak noktasıdır: parametreler BP'nin üstünde pozitif, yerel değişkenler altında negatif uzaklıktadır.



Şekil 8.2.17: İki parametreli ve iki yerel değişkenli bir prosedürün yığın çerçevesi. Küçük adresler yukarıda, büyük adresler aşağıdadır.

Çerçeveyi kuran ve yıkan komutlar her prosedürde aynı kalıbı izler:

```
PUSH BP
MOV  BP, SP
SUB  SP, 4        ; yerel alan
MOV  [BP-2], AX   ; yerel 1
...
MOV  SP, BP       ; yerelleri at
POP  BP
RET  4            ; 2 parametreyi at
```

`SUB SP, 4` yığında iki sözcüklük yer açar. Bu alana bir şey yazılmaz; yalnız SP aşağı çekilir. Böylece prosedürün sonraki PUSH'ları ve çağrıları yerel alanın üstüne yazmaz. İki yerel değişken `[BP−2]` ve `[BP−4]` adreslerindedir (Şekil 8.2.17). Açılan alanın ilk değeri, o adreste daha önce ne kaldıysa odur. Yerel değişkenler kullanılmadan önce bir değer almalıdır.

Çıkışta `MOV SP, BP` yerel alanı tek komutla geri verir. Kaç yerel değişken açıldığı önemli değildir; SP, BP'nin kurulduğu andaki değerine döner. Ardından POP BP ve RET n gelir. Bu sıra bozulamaz: yerel alan kapatılmadan yapılan POP BP, eski BP yerine bir yerel değişkeni BP'ye yükler.

Yerel değişkenin ömrü çağrının ömrüdür. Prosedür döndükten sonra o yığın sözcükleri bir sonraki PUSH ya da CALL ile ezilir. Bu yüzden bir prosedür, yerel değişkeninin adresini sonuç olarak döndürmemelidir. Aynı prosedür kendini çağırırsa her çağrı kendi çerçevesini, dolayısıyla kendi yerel değişkenlerini kurar. Düzeyler birbirinin değişkenine dokunmaz.

80186 ve sonraki işlemcilerde bu kalıbın iki komutluk kısaltması vardır: `ENTER` çerçeveyi kurar, `LEAVE` `MOV SP, BP` ile `POP BP`'yi birlikte yapar. 8086'da bu komutlar yoktur.

8.2.5.4 Yerel değişkenli çerçeve: kod izleme

Aşağıdaki prosedür AX'teki x değerinden x·x + x'i hesaplar. MUL, AX'in üzerine yazdığı için x'in kopyası bir yerel değişkende saklanır.

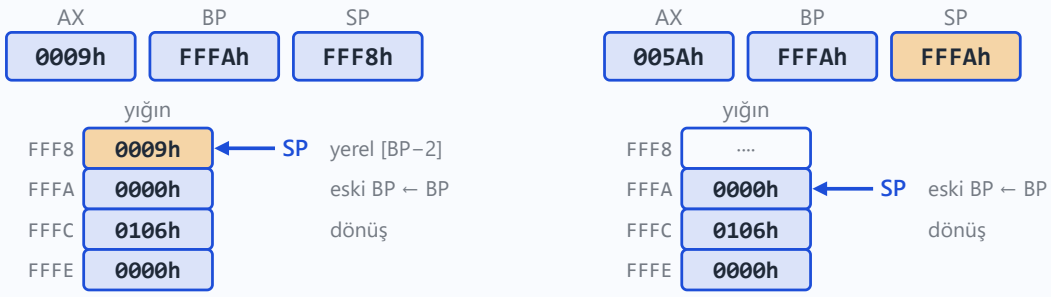
```
MOV AX, 9
CALL kare_art       ; AX = x·x + x
RET
kare_art PROC
PUSH BP
MOV BP, SP
SUB SP, 2           ; yerel alan [BP-2]
MOV [BP-2], AX      ; yerel = x
MUL AX              ; AX = x·x
ADD AX, [BP-2]      ; + x
MOV SP, BP          ; yereli at
POP BP
RET
kare_art ENDP
```

Satır	Komut	AX	DX	BP	SP	Yığın
1	`MOV AX, 9`	0009h	0000h	0000h	FFFEh	
2	`CALL kare_art`	0009h	0000h	0000h	**FFFCh**	[FFFCh] = 0106h
5	`PUSH BP`	0009h	0000h	0000h	**FFFAh**	[FFFAh] = 0000h
6	`MOV BP, SP`	0009h	0000h	**FFFAh**	FFFAh	
7	`SUB SP, 2`	0009h	0000h	FFFAh	**FFF8h**	[FFF8h] = yerel alan
8	`MOV [BP-2], AX`	0009h	0000h	FFFAh	FFF8h	**[FFF8h] = 0009h**
9	`MUL AX`	**0051h**	0000h	FFFAh	FFF8h	
10	`ADD AX, [BP-2]`	**005Ah**	0000h	FFFAh	FFF8h	
11	`MOV SP, BP`	005Ah	0000h	FFFAh	**FFFAh**	yerel alan bırakıldı
12	`POP BP`	005Ah	0000h	**0000h**	**FFFCh**	

Satır	Komut	AX	DX	BP	SP	Yığın
13	`RET`	005Ah	0000h	0000h	**FFFEh**	IP = 0106h

Tablo 8.2.18: Yerel değişkenli çerçeve izlemesi (her satır yürütüldükten sonraki durum).

İzlemede (Tablo 8.2.18) `SUB SP, 2`'den sonra BP = FFFAh, SP = FFF8h'dir. Yerel değişken [BP−2] = [FFF8h]'dedir; SP onu gösterir ve sonraki bir PUSH onun altına yazar. `MOV [BP-2], AX` x = 9'u yerele kopyalar. 16 bitlik `MUL AX` DX:AX ← AX × AX hesaplar: 9 × 9 = 81 = 0051h, DX = 0000h. AX'teki x kaybolmuştur, ama yerelde durur. `ADD AX, [BP-2]` 81 + 9 = 90 = 005Ah verir.



Şekil 8.2.18: Solda yerel değişkene x yazıldıktan sonra: [BP−2] = 9. Sağda `MOV SP, BP`'den sonra: SP = BP = FFFAh, yerel alan bırakıldı.

Çıkışta üç komut çerçeveyi ters sırayla yıkar. `MOV SP, BP` SP'yi FFFAh'ye çeker ve yerel alan bırakılır. [FFF8h]'de 0009h hâlâ yazılıdır, ama artık yığının dışında kalmıştır. `POP BP` çağıranın BP'sini (0000h) geri getirir. RET 0106h'ye döner. Sonuç AX = 90'dır.

Bu prosedürün belgelemesinde bir eksik vardır. 16 bitlik MUL her durumda DX'e de yazar: x = 9 için DX = 0000h olur. Çağıranın DX'te tuttuğu değer bu yüzden kaybolur. Bu izlemede DX zaten 0 olduğu için fark görünmez. x ≥ 256 olduğunda çarpım 16 bite sığmaz ve DX sıfırdan farklı bir değer alır. Açıklamaya "Bozar: DX" yazılmalı ya da DX prosedürde PUSH/POP ile korunmalıdır.

8.2.6 Örnek programlar

8.2.6.1 Merhaba Dünya: print_me

İlk örnek, sıfırla biten bir dizgiyi ekrana karakter karakter yazan `print_me` prosedürüdür. Parametre, dizginin adresidir ve SI ile geçirilir. Dizginin uzunluğu ayrıca verilmez; prosedür sıfır baytını görünce durur. C dilindeki dizgiler de bu düzende saklanır.

```
ORG 100h
LEA SI, msg       ; SI = msg adresi
CALL print_me
RET               ; DOS'a dön
print_me PROC
next_char:
CMP b.[SI], 0     ; sıfır mı?
JE stop           ; evet: bitir
MOV AL, [SI]      ; sıradaki ASCII karakter
MOV AH, 0Eh       ; teletype işlevi
INT 10h           ; AL'yi ekrana yaz
ADD SI, 1         ; sonraki karakter
JMP next_char     ; başa dön
stop:
RET               ; çağırana dön
print_me ENDP
msg DB 'Hello World!', 0
END
```

Ana program üç komuttur. `LEA SI, msg` dizginin adresini SI'ya yükler; prosedürün parametresi budur. `CALL print_me` prosedürü çağırır, `RET` programı bitirir. Programın yerleşiminde LEA 3 bayttır (BE 18 01), CALL 0103h'dedir ve dönüş adresi 0106h'dir. Prosedür 0107h'de başlar ve 0117h'deki RET ile biter. Dizgi 0118h'dedir: 12 karakter ve bir sıfır baytı.

Prosedürün ilk komutu `CMP b.[SI], 0`'dır. `b.`, emu8086'nın `BYTE PTR` kısaltmasıdır. `CMP [SI], 0` yazılsaydı assembler işlemin boyutunu bilemezdi: [SI] bir bayt da, bir sözcük de olabilir, 0 sabiti de boyut belirtmez. Boyut bu yüzden açıkça yazılır. MASM'da aynı satır `CMP BYTE PTR [SI], 0` biçiminde yazılır.

Döngü her turda altı komut yürütür: karşılaştırma, koşullu atlama, karakteri AL'ye alma, AH'ye 0Eh yazma, INT 10h ve SI'yı ilerletme; sonra JMP başa döner. `MOV AH, 0Eh` döngünün dışına alınabilirdi, ama INT 10h'nin AH'yi koruyacağı kesin değilse içeride kalması güvenlidir.

An	SI	AL	ZF	SP	Ekran
`LEA SI, msg`	msg+0	—	0	FFFEh	
`CALL print_me`	msg+0	—	0	**FFFCh**	
1. tur, CMP: 'H' ≠ 0	msg+0	—	0	FFFCh	
1. tur, INT 10h	msg+0	**48h**	0	FFFCh	H
1. tur, ADD SI, 1	**msg+1**	48h	0	FFFCh	H
2. tur sonu	msg+2	65h	0	FFFCh	He
5. tur sonu	msg+5	6Fh	0	FFFCh	Hello
11. tur sonu	msg+11	64h	0	FFFCh	Hello World
12. tur sonu	msg+12	21h	0	FFFCh	Hello World!
CMP: [SI] = 0, JE stop	msg+12	21h	**1**	FFFCh	Hello World!

An	SI	AL	ZF	SP	Ekran
prosedürün `RET`'i	msg+12	21h	1	**FFFEh**	Hello World!
ana programın `RET`'i	msg+12	21h	1	**0000h**	Hello World!

Tablo 8.2.19: print_me izlemesi (seçilmiş anlar; msg = 0118h).

Döngü 12 tur döner ve her turda bir karakter yazar (Tablo 8.2.19). Altıncı karakter boşluktur (20h); boşluk da bir karakterdir ve ekrana yazılır. On üçüncü kez yapılan karşılaştırmada [SI] = 0'dır, ZF = 1 olur ve JE `stop`'a atlar. Sıfır baytı ekrana yazılmaz. Bütün döngü boyunca SP = FFFCh'dir: prosedür yığına bir şey itmez, yığında yalnız dönüş adresi vardır.

Prosedür SI'yı değiştirir ve korumaz. Dönüşte SI, dizginin sonundaki sıfır baytını gösterir (msg + 12 = 0124h). Bu bir yan etkidir ve açıklamaya “Bozar: SI, AX” diye yazılmalıdır. Aynı prosedür başka bir metin için yalnız SI değiştirilerek çağrılabilir: `LEA SI, baska_metin` / `CALL print_me`.

8.2.6.2 İki değişkeni toplamak

İkinci örnek, parametrelerini ve sonucunu bellek değişkenlerinde bulan bir prosedürdür.

```
ORG 100h
start:
CALL AddNumbers     ; prosedürü çağır
RET                 ; DOS'a dön
AddNumbers PROC
MOV AX, number1     ; AX = ilk sayı
ADD AX, number2     ; AX = AX + ikinci sayı
MOV result, AX      ; sonucu sakla
RET                 ; prosedürü bitir
AddNumbers ENDP
number1 DW 5        ; ilk sayı 5
number2 DW 3        ; ikinci sayı 3
result DW ?         ; sonucu tutar
```

Programın ilk komutu CALL'dır ve 0100h'dedir; dönüş adresi 0103h'dir. Prosedür 0104h'de başlar. Komutların uzunlukları birbirinden farklıdır. `MOV AX, number1` AX'e özel kısa kodlamayla 3 bayttır (A1 + adres). `ADD AX, number2` 4 bayttır. `MOV result, AX` yine AX'e özel kodlamayla 3 bayttır. Değişkenler prosedürün arkasındadır: number1 = 010Fh, number2 = 0111h, result = 0113h.

Satır	Komut	AX	SP	ZF	CF	result
3	`CALL AddNumbers`	0000h	**FFFCh**	0	0	?
6	`MOV AX, number1`	**0005h**	FFFCh	0	0	?
7	`ADD AX, number2`	**0008h**	FFFCh	0	0	?
8	`MOV result, AX`	0008h	FFFCh	0	0	**0008h**
9	`RET`	0008h	**FFFEh**	0	0	0008h

Satır	Komut	AX	SP	ZF	CF	result
4	`RET` (DOS'a)	0008h	**0000h**	0	0	0008h

Tablo 8.2.20: İki değişken toplama izlemesi (her satır yürütüldükten sonraki durum).

Tablo 8.2.20'de ADD'in bayrakları da gösterilmiştir. 5 + 3 = 8 sıfır değildir (ZF = 0) ve 16 bitte elde üretmez (CF = 0). MOV bayraklara dokunmaz. `result` bellekte `08 00` olarak, little-endian düzende saklanır. `DW ?` başlangıç değerinin belirsiz olduğunu söyler. emu8086 bu alana 0 koyar, ama program bu değere güvenmemelidir.

Bu prosedür kısa ve anlaşılırdır, ama yalnız `number1`, `number2` ve `result` ile çalışır. Başka iki sayıyı toplamak için ya önce o sayılar bu değişkenlere kopyalanır ya da yeni bir prosedür yazılır. Bellek değişkeniyle parametre geçirmenin zayıf yanı budur.

8.2.6.3 Dizi toplamı prosedürü

Üçüncü örnek bir dizinin elemanlarını toplar. Parametre olarak iki şey alır: dizinin adresi (SI) ve eleman sayısı (CX). Sonucu AL'de döndürür. Prosedür hangi dizinin adresi verilirse onunla çalışır.

```
ORG 100h
LEA SI, vector     ; parametre 1: adres
MOV CX, 5          ; parametre 2: adet
CALL topla
MOV m, AL          ; dönüş değeri AL'de
RET
topla PROC         ; Girdi: SI, CX
MOV AL, 0          ; Çıktı: AL = toplam
dongu:
ADD AL, [SI]
INC SI
LOOP dongu
RET
topla ENDP
vector DB 5, 4, 5, 2, 1
m DB 0
```

Satır	Komut / tur	SI	CX	AL	SP	m
2–3	`LEA SI, vector` · `MOV CX, 5`	0115h	0005h	00h	FFFEh	00h
4	`CALL topla`	0115h	0005h	00h	**FFFCh**	00h
8	`MOV AL, 0`	0115h	0005h	00h	FFFCh	00h
10–12	1. tur: +5	0116h	0004h	**05h**	FFFCh	00h
10–12	2. tur: +4	0117h	0003h	**09h**	FFFCh	00h
10–12	3. tur: +5	0118h	0002h	**0Eh**	FFFCh	00h
10–12	4. tur: +2	0119h	0001h	**10h**	FFFCh	00h

Satır	Komut / tur	SI	CX	AL	SP	m
10–12	5. tur: +1	011Ah	0000h	**11h**	FFFCh	00h
13	`RET`	011Ah	0000h	11h	**FFFEh**	00h
5	`MOV m, AL`	011Ah	0000h	11h	FFFEh	**11h**

Tablo 8.2.21: Dizi toplamı izlemesi (vector = 0115h, m = 011Ah).

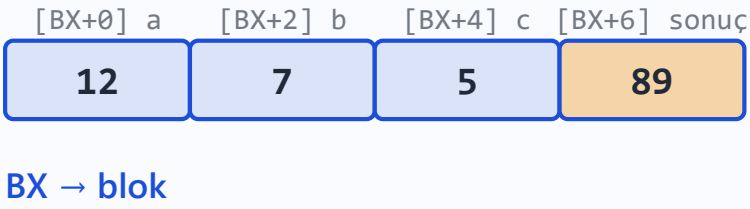
Toplam 5 + 4 + 5 + 2 + 1 = 17 = 11h'dir (Tablo 8.2.21). Son turdan sonra SI = 011Ah olur ve dizinin hemen arkasındaki `m` değişkenini gösterir. LOOP CX'i 0'a indirdiğinde atlamaz ve akış RET'e geçer. Çağıran AL'deki sonucu `m`'ye yazar.

Prosedürün iki sınırı açıklamasına yazılmalıdır. Birincisi, toplam 8 bitlik AL'de tutulur. Elemanların toplamı 255'i aşarsa sonuç taşar ve yalnız alt 8 biti kalır. İkincisi, prosedür SI ve CX'i değiştirir. Dönüşte SI dizinin sonunu, CX ise 0'ı gösterir. Aynı prosedür başka bir dizi için yalnız SI ve CX değiştirilerek çağrılabilir.

8.2.6.4 Parametre bloğu

Prosedürün çok sayıda parametresi varsa yazmaçlar yetmez. Bir çözüm, parametreleri bellekte art arda duran bir bloğa yazmak ve prosedüre yalnız bloğun adresini vermektir. Tek bir yazmaç istenen sayıda parametreyi taşır.

```
LEA BX, blok
CALL hesap            ; sonuç = a × b + c
RET
hesap PROC            ; Girdi: BX = blok adresi
PUSH AX
PUSH DX               ; MUL DX'i de yazar
MOV AX, [BX]          ; a
MUL WORD PTR [BX+2]   ; × b
ADD AX, [BX+4]        ; + c
MOV [BX+6], AX        ; sonuç
POP DX
POP AX
RET
hesap ENDP
blok DW 12, 7, 5, ?
```



Şekil 8.2.19: Parametre bloğu: [BX+0] a = 12, [BX+2] b = 7, [BX+4] c = 5, [BX+6] sonuç = 89.

Blok dört sözcükten oluşur (Şekil 8.2.19). Prosedür her parametreyi BX'e göre sabit bir uzaklıktan okur: a [BX], b [BX+2], c [BX+4]. Sonucu da bloğun dördüncü sözcüğüne, [BX+6]'ya yazar. `MUL WORD PTR [BX+2]` satırında boyut açıkça yazılır; MUL'un tek işleneni bellekte olduğu için assembler boyutu başka yoldan bilemez.

Hesap şöyle ilerler: AX = 12, MUL ile DX:AX = 12 × 7 = 84 = 0054h, ADD ile AX = 84 + 5 = 89 = 0059h. Sonuç [BX+6]'ya yazılır. emu8086'da çalıştırıldığında blok 0117h'de başlar ve sonuç 011Dh–011Eh baytlarına `59 00` olarak yazılır. Prosedür AX ve DX'i kullandığı hâlde ikisini de korur, çünkü sonucunu yazmaçta değil bellekte döndürür. Dönüşte AX ve DX çağrıdan önceki değerlerindedir.

Aynı prosedür başka bir blokla, yalnız BX değiştirilerek çağrılır. Blok düzeni, yani hangi uzaklıkta hangi parametrenin durduğu, prosedürün açıklamasında belgelenmelidir. DOS'un dosya ve program yükleme hizmetleri de bu yöntemi kullanır: çağıran bir parametre bloğu hazırlar ve adresini bir yazmaçta verir.

8.2.6.5 Prosedürü test etmek

Bir prosedür yalnız olağan girdiyle denenirse gizli hataları ortaya çıkmaz. Sınama, sınır durumlarını da kapsamalıdır: ilk ve son eleman, bulunmayan değer, tek elemanlık ve boş girdi. Tek çıkış noktalı `bul` prosedürü (dizi: 3, 9, 7, 1, 4) bu durumlarla çalıştırıldığında Tablo 8.2.22'deki sonuçlar elde edilir.

Durum	AL	CX	CF	BX	Değerlendirme
ilk eleman	3	5	1	0	doğru
son eleman	4	5	1	4	doğru
dizide yok	8	5	0	5	doğru; BX anlamsızdır, yalnız CF'ye bakılır
tek elemanlık arama	9	1	0	1	doğru: 9 ikinci eleman, aranan aralığın dışında
boş dizi	7	0	1	2	yanlış: CX = 0 iken LOOP 65.536 tur döner

Tablo 8.2.22: bul prosedürünün sınır durumlarıyla sınanması.

Bütün durumlarda dönüşte CX ve SI girdideki değerlerindedir; koruma doğru çalışır. Hata yalnız boş dizide ortaya çıkar. Nedeni LOOP'un çalışma biçimidir: LOOP önce CX'i 1 azaltır, sonra sıfır olup olmadığına bakar. CX = 0 ile girilen döngüde ilk LOOP CX'i 0 − 1 = FFFFh yapar ve atlar. Döngü 65.536 tur

dönebilir. Prosedür hiç eleman taramaması gerekirken dizinin baytlarını taramaya başlar ve üçüncü baytta 7'yi "bulur".

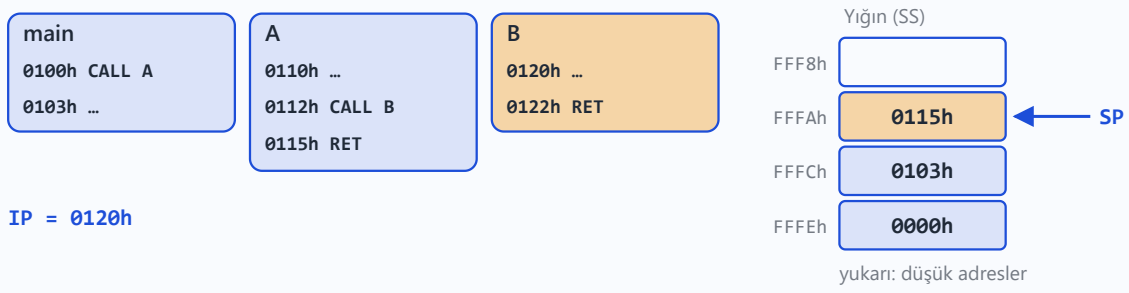
Düzeltme, döngüye girmeden önce CX'i sınamaktır. `JCXZ` (*jump if CX zero*) komutu CX = 0 ise atlar ve bayraklara bakmaz:

```
bul PROC
PUSH CX
PUSH SI
XOR BX, BX
JCXZ yok          ; boş dizi: aramadan çık
ara: CMP [SI], AL
JE buldu
INC SI
INC BX
LOOP ara
yok: CLC          ; bulunamadı
JMP son
buldu: STC
son: POP SI
POP CX
RET
bul ENDP
```

Düzeltilmiş prosedür CX = 0 için CF = 0 ve BX = 0 döndürür. CX = 5 ile 7 arandığında önceki gibi CF = 1, BX = 2 verir. İki sonuç da emu8086'da çalıştırılarak elde edilmiştir. Bu örnek, sınamanın her prosedür için bir liste hâlinde hazırlanmasının nedenini gösterir: hata, olağan girdilerle yapılan hiçbir denemede görünmezdi.

8.2.6.6 İç içe çağrılar

Bir prosedür başka bir prosedürü çağırabilir. Bu durumda yığında aynı anda birden çok dönüş adresi bulunur. Yığının LIFO yapısı, dönüşlerin doğru sırada yapılmasını kendiliğinden sağlar: son çağrılan ilk döner.



Şekil 8.2.20: main, A'yı; A da B'yi çağırmıştır. Yığında iki dönüş adresi vardır: tepede 0115h (A'ya dönüş), altında 0103h (main'e dönüş).

An	IP	SP	Yığın (tepeden)
main, CALL A'dan önce	0100h	FFFEh	0000h
`CALL A` (0100h)	0110h	**FFFCh**	0103h, 0000h
A içinden `CALL B` (0112h)	0120h	**FFFAh**	0115h, 0103h, 0000h
B'nin `RET`'i	0115h	**FFFCh**	0103h, 0000h
A'nın `RET`'i	0103h	**FFFEh**	0000h

Tablo 8.2.23: İç içe çağrılarda yığın.

Tabloda (Tablo 8.2.23) B'nin RET'i yığının tepesindeki 0115h'yi alır ve A'ya döner. A'nın işi bittiğinde onun RET'i bir alttaki 0103h'yi alır ve main'e döner. Her RET yalnız tepedeki adresi bilir; doğru adresin tepede olmasını yığının yapısı sağlar. Bu yüzden prosedürler, kaç düzey iç içe çağrıldıklarını bilmek zorunda değildir.

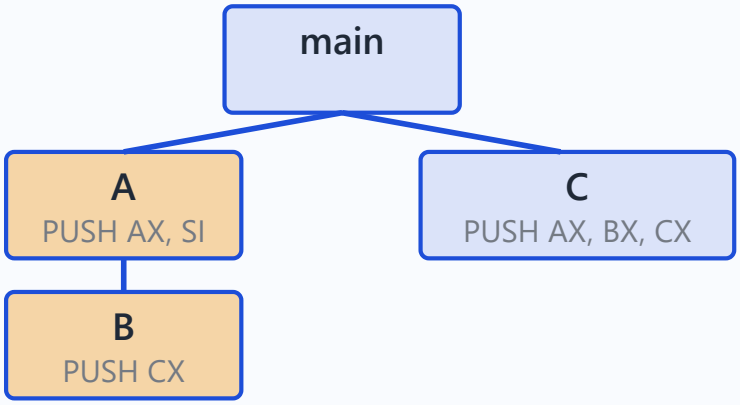
Aynı düzen daha somut bir programla da izlenebilir. main AX = 1 ile A'yı çağırır; A, AX'e 2 ekler, B'yi çağırır, sonra AX'e 4 ekler; B, AX'i iki katına çıkarır. Sıra 1 → 3 → 6 → 10'dur ve program AX = 000Ah ile biter. Yığında en çok iki dönüş adresi birikir.

8.2.6.7 Yığın ihtiyacı hesabı

İç içe çağrılarda yığın, en derin çağrı zinciri boyunca biriken baytlar kadar büyür. Bir programın ne kadar yığına ihtiyaç duyduğunu bilmek, yığın için yeterli yer ayırmak ve yığının veriyle ya da kodla çakışmasını önlemek için gerekir.

Tanım 8.2.9

Bir programın **yığın gereksinimi**, çağrı ağacındaki en derin yol boyunca biriken baytların toplamıdır. Her yakın CALL 2 bayt dönüş adresi, her uzak CALL 4 bayt, her PUSH 2 bayt ve her yerel alan kendi boyutu kadar ekler.



Şekil 8.2.21: Çağrı ağacı: main, A ve C'yi; A da B'yi çağırır. A iki, C üç, B bir yazmaç saklar.

Yol	Hesap	Bayt
main → A → B	2 (CALL A) + 4 (PUSH AX, SI) + 2 (CALL B) + 2 (PUSH CX)	10
main → C	2 (CALL C) + 6 (PUSH AX, BX, CX)	8

Tablo 8.2.24: Çağrı yollarının yığın ihtiyacı.

En derin yol main → A → B'dir ve 10 bayt gerektirir (Tablo 8.2.24). SP = FFFEh ile başlayan programda SP en çok FFFEh − 10 = FFF4h'ye iner. C üç yazmaç sakladığı için A'dan daha fazla PUSH yapar, ama derinliği daha azdır. A ve C hiçbir zaman aynı anda yığında değildir: C çağrıldığında A'nın baytları çoktan geri alınmıştır. Bu yüzden gereksinim yolların toplamı değil, en büyüğüdür.

Hesaba bir pay eklenmelidir. Donanım kesmeleri programın herhangi bir anında, yığının en derin anında da gelebilir. Her kesme yığına 6 bayt iter (FLAGS, CS ve IP) ve kesme hizmet yordamı kendi saklamalarını ekler. Hesaplanan değerin üstünde bu yüzden her zaman pay bırakılır. .COM programında yığın kesimin en üstünden aşağı büyür, kod ve veri en alttan yukarı. İkisi arasındaki boşluk bittiğinde yığın veriyi ezer. Buna **yığın taşması** (*stack overflow*) denir ve işlemci bunu bildirmez.

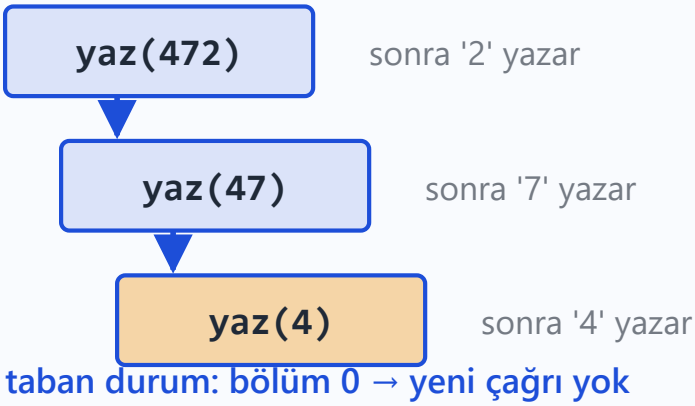
8.2.7 Özyineleme

8.2.7.1 Kendini çağıran prosedür

Bir prosedürün başka bir prosedürü çağırabildiği görüldü. Aynı düzenek, prosedürün kendini çağırmasına da izin verir. İşlemci için CALL'ın hedefinin hangi prosedür olduğu önemli değildir; dönüş adresini iter ve atlar.

> **Tanım 8.2.10**
>
> **Özyineleme** (*recursion*), bir prosedürün kendini doğrudan ya da başka bir prosedür üzerinden çağırmasıdır. Her çağrı yığına yeni bir dönüş adresi ve kendi sakladığı değerleri koyar; her düzeyin değerleri birbirinden ayrı kalır. Kendini yeniden çağırmadan dönülen duruma **taban durum** (durma koşulu) denir.

Özyineleme, problemin her adımda küçülerek kendine benzediği durumlarda doğal bir çözümdür. Bir sayıyı onluk basamaklarıyla yazmak buna iyi bir örnektir. 472'yi yazmak, önce 47'yi yazmak, sonra '2'yi yazmak demektir. 47'yi yazmak da önce 4'ü, sonra '7'yi yazmaktır. 4 tek basamaklıdır; onu yazmak için başka bir şey yazmaya gerek yoktur. Bu, taban durumdur.



Şekil 8.2.22: yaz(472), yaz(47)'yi; o da yaz(4)'ü çağırır. Dönüşte sırasıyla 4, 7, 2 yazılır.

Taban durum şarttır. Taban durumu olmayan bir prosedür kendini sonsuza kadar çağırır. Her çağrı yığına en az 2 bayt ekler. SP sürekli azalır, yığın programın verisinin ve kodunun üzerine kadar büyür ve program çöker. Sayı yazdırmanın yanında faktöriyel hesabı, ağaçta gezinme ve dizinde gezinme özyinelemenin tipik örnekleridir.

8.2.7.2 Özyinelemeli sayı yazdırma: kod izleme

```
ORG 100h
MOV AX, 472
CALL yaz
RET
yaz PROC        ; Girdi: AX (işaretsiz)
XOR DX, DX
MOV BX, 10
DIV BX          ; AX = bölüm, DX = basamak
PUSH DX
OR AX, AX       ; başka basamak var mı?
JZ son
CALL yaz        ; önce üst basamaklar
son: POP AX
ADD AL, '0'
MOV AH, 0Eh
INT 10h
RET
yaz ENDP
```

Prosedür AX'teki işaretsiz sayıyı ekrana yazar. İlk üç komut sayıyı 10'a böler. 16 bitlik `DIV BX`, DX:AX'teki 32 bitlik sayıyı böler; bu yüzden önce `XOR DX, DX` ile DX sıfırlanır. Bölümden sonra AX = sayı / 10, DX = sayı mod 10 olur. Kalan, sayının son basamağıdır ve `PUSH DX` ile yığına saklanır. `OR`

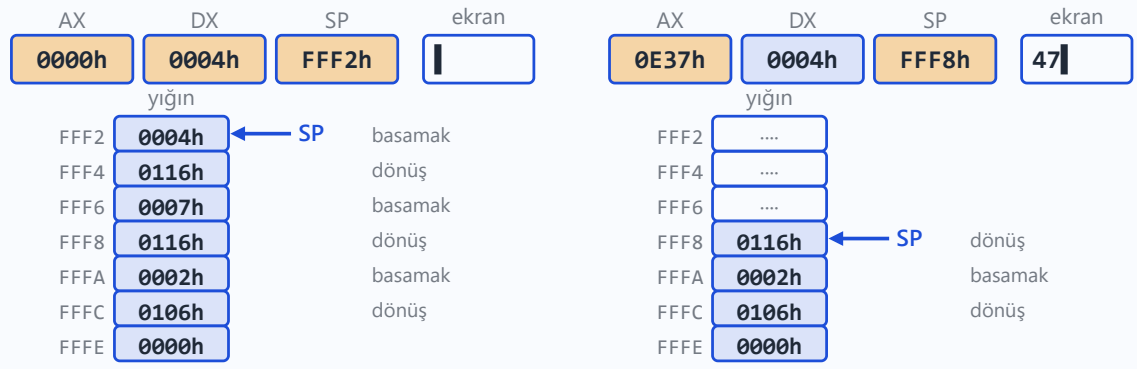
`AX, AX` AX'i değiştirmeden ZF'yi kurar: bölüm sıfırsa yazılacak başka basamak yoktur ve JZ taban duruma, `son`'a atlar. Bölüm sıfır değilse prosedür, bölümü yazmak için kendini çağırır. Dönüşte `son`'daki dört komut saklanan basamağı alır, ASCII'ye çevirir ve yazar.

Programın yerleşiminde CALL'lar iki yerdedir. Ana programın `CALL yaz`'ı 0103h'dedir ve dönüş adresi 0106h'dir. Prosedürün içindeki `CALL yaz` 0113h'dedir ve dönüş adresi 0116h'dir, yani `son` etiketi. Prosedür kendini çağırdığında bu CALL'ın makine kodu `E8 F1 FF` olur: 0107h − 0116h = −0Fh = FFF1h. Hedef geride olduğu için uzaklık negatiftir.

Satır	Olay	AX	DX	SP	Yığın (tepeden)	Ekran
2–3	AX = 472, `CALL yaz`	01D8h	0000h	**FFFCh**	0106h	
6–9	1. düzey: 472 / 10, PUSH DX	**002Fh**	**0002h**	**FFFAh**	2, 0106h	
10–12	AX ≠ 0: `CALL yaz`	002Fh	0002h	**FFF8h**	0116h, 2, 0106h	
6–12	2. düzey: 47 / 10, PUSH, CALL	**0004h**	**0007h**	**FFF4h**	0116h, 7, 0116h, 2, 0106h	
6–11	3. düzey: 4 / 10, PUSH; AX = 0, JZ son	**0000h**	**0004h**	**FFF2h**	4, 0116h, 7, 0116h, 2, 0106h	
13–16	POP AX = 4, yaz	**0E34h**	0004h	**FFF4h**	0116h, 7, 0116h, 2, 0106h	4
17, 13–16	RET → 0116h; POP AX = 7, yaz	**0E37h**	0004h	**FFF8h**	0116h, 2, 0106h	47
17, 13–16	RET → 0116h; POP AX = 2, yaz	**0E32h**	0004h	**FFFCh**	0106h	472
17	son RET → 0106h	0E32h	0004h	**FFFEh**	0000h	472

Tablo 8.2.25: Özyinelemeli sayı yazdırma izlemesi (her satır grubu yürütüldükten sonraki durum).

İzleme (Tablo 8.2.25) iki evreye ayrılır. **İniş evresinde** prosedür her düzeyde bir basamak ayırır ve kendini yeniden çağırır. Yığına her düzeyde iki sözcük eklenir: basamak ve dönüş adresi. Ama basamaklar sondan başa doğru ayrılır: önce 2, sonra 7, sonra 4. Üçüncü düzeyde bölüm 0 olur ve taban duruma ulaşılır. Bu anda SP en derin noktasındadır: FFF2h. Yığında üç basamak ve üç dönüş adresi vardır (Şekil 8.2.23).



Şekil 8.2.23: Solda taban durumda: SP = FFF2h, yığında üç basamak ve üç dönüş adresi. Sağda ikinci düzeye dönüldükten sonra: '4' ve '7' yazıldı, SP = FFF8h.

Çıkış evresinde her düzey kendi basamağını yığından alır ve yazar. Son itilen basamak (4) ilk çıkar. Böylece basamaklar ayrıldıkları sıranın tersiyle, yani doğru sırada yazılır. Üçüncü düzeyin RET'i 0116h'ye, ikinci düzeyin `son` satırına döner. İkinci düzey 7'yi yazar ve onun RET'i de 0116h'ye, bu kez birinci düzeye döner. Aynı dönüş adresinin yığında iki kez bulunması doğaldır: iki ayrı çağrı aynı CALL komutundan yapılmıştır. Birinci düzey 2'yi yazar ve son RET 0106h'ye, ana programa döner. SP = FFFEh olur ve yığın dengelenmiştir.

İzlemenin son satırı, prosedürün en dıştaki RET'inden hemen sonraki durumdur: IP = 0106h, SP = FFFEh. Ardından ana programın RET'i 0000h'yi alır ve SP = 0000h ile program biter.

Her düzey kendi basamağını yığında sakladığı için ek bir değişken ya da dizi gerekmemiştir. Yığın, düzeylerin her birine ayrı bir saklama alanı sağlamıştır. Döngüyle yazılan bir çözüm de basamakları yığına itip sonra çekerek ters çevirir; özyineleme bu işi CALL ve RET'in doğal düzeniyle yapar.

Örnek 8.2.4

AX = 0 ile çağrıldığında prosedür ne yazar? SP en çok kaça iner?

1. **Bölme.** 0 / 10: AX = 0, DX = 0. Basamak 0 yığına itilir.
2. **Taban durum.** AX = 0, JZ `son`'a atlar; yeni çağrı yapılmaz.
3. **Yazma.** POP AX = 0, ADD AL, '0' → 30h; ekrana '0' yazılır.
4. **Yığın.** Ana programın CALL'ı (2 bayt) ve bir basamak (2 bayt): SP en çok FFFEh − 4 = FFFAh'ye iner.

Taban durum denetimi bölmeden sonra yapıldığı için 0 sayısı da ek kod gerektirmeden '0' olarak yazılır.

8.2.7.3 Özyineleme mi, döngü mü?

Aynı iş bir döngüyle de yapılabilir: sayı art arda 10'a bölünür, basamaklar yığına itilir ve bir sayaçla sayılır; ikinci bir döngü basamakları çekip yazar. İki çözümün karşılaştırması Tablo 8.2.26'dadır.

Ölçüt	Özyineleme (yaz)	Döngü (bölme + LOOP)
Kod	kısa; problemin tanımına yakın	sayaç ve ayrı bir yazdırma döngüsü gerekir
Yığın	her düzeyde dönüş adresi + basamak: 4 bayt	basamak başına 2 bayt
Süre	her düzeyde CALL + RET (19 + 8 tur)	çağrı yükü yok
Risk	derinlik sınırsız büyürse yığın taşar	yığın kullanımı sınırlı

Tablo 8.2.26: Özyinelemeli ve döngülü sayı yazdırmanın karşılaştırması.

Örnek 8.2.5

`yaz` prosedürü 16 bitlik en büyük işaretsiz sayı olan 65.535 ile çağrılırsa yığın en çok ne kadar derinleşir?

1. **Düzey sayısı.** Düzey sayısı basamak sayısına eşittir: 65.535 beş basamaklıdır, beş düzey.
2. **Bayt.** Her düzey bir dönüş adresi (2 bayt) ve bir basamak (2 bayt) ekler: 5 × 2 + 5 × 2 = 20 bayt.
3. **SP.** FFFEh − 20 = FFFEh − 14h = FFEAh.
4. **Süre.** Beş CALL ve beş RET: 5 × (19 + 8) = 135 tur, 5 MHz'de 27 µs çağrı yükü.

Bu prosedürün derinliği 5 ile sınırlıdır, çünkü AX en çok beş basamak taşır. emu8086'da 65.535 ile çalıştırıldığında SP'nin en küçük değeri FFEAh'dir.

Taban durumun yeri de önemlidir. Denetim yeni CALL'dan önce yapılmalıdır. `yaz`'da önce bölünür, sonra bölümün sıfır olup olmadığına bakılır ve ancak sıfır değilse CALL yapılır. Denetim CALL'dan sonraya konsaydı, CALL'a hiçbir koşul bağlanmamış olurdu ve prosedür kendini durmadan çağırırdı.

8.2.8 Sık yapılan hatalar

8.2.8.1 Dengesiz yığın: kod izleme

Prosedür hatalarının çoğunun kökü aynıdır: RET yürütüldüğünde yığının tepesinde dönüş adresi yoktur. Bunun en sık nedeni, bir PUSH'un POP'unun unutulmasıdır.

```
ORG 100h
MOV AX, 1234h      ; 0100h: B8 34 12
CALL hatali        ; 0103h: dönüş = 0106h
RET                ; 0106h
hatali PROC
PUSH AX            ; AX saklandı...
; ... işlemler ...
RET                ; POP AX unutuldu!
hatali ENDP
```

Satır	Komut	AX	IP (sonraki)	SP	Yığın (tepeden)
2	`MOV AX, 1234h`	1234h	0103h	FFFEh	0000h
3	`CALL hatali`	1234h	0107h	**FFFCh**	0106h, 0000h
6	`PUSH AX`	1234h	0108h	**FFFAh**	1234h, 0106h, 0000h
8	`RET`	1234h	**1234h**	**FFFCh**	0106h, 0000h

Tablo 8.2.27: Dengesiz yığın izlemesi (her satır yürütüldükten sonraki durum).

RET yığının tepesinde ne varsa onu dönüş adresi sayar (Tablo 8.2.27). Tepede AX'in saklanan değeri, 1234h, vardır. RET'ten sonra IP = 1234h olur. İşlemci CS:1234h adresindeki baytları komut olarak yürütmeye başlar. O adreste programın kodu yoktur; bellekte ne kaldıysa o yürütülür. Program çöker ya da öngörülemez davranır. Asıl dönüş adresi 0106h ise yığında, SP = FFFCh'nin gösterdiği yerde, hiç kullanılmadan kalır.

Kural şudur: prosedürden çıkarken SP, girişteki değerine eşit olmalıdır. Her PUSH'un ters sırada bir POP'u bulunmalıdır, üstelik prosedürün her yolunda. Hata, PUSH ile RET arasındaki kod uzadıkça ve dallar çoğaldıkça daha kolay yapılır. Tek çıkış noktası bu yüzden önerilir.

8.2.8.2 Tür uyuşmazlığı: kod izleme

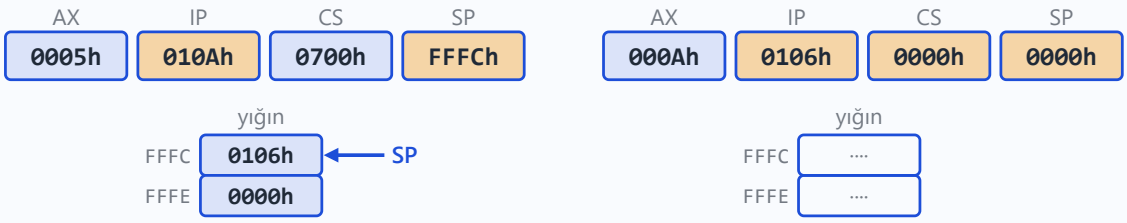
İkinci tipik hata, çağrının türü ile dönüşün türünün uyuşmamasıdır. Aşağıdaki programda `uzak` prosedürü RETF ile biter, ama yakın CALL ile çağrılır.

```
ORG 100h
MOV AX, 5
CALL uzak      ; yakın CALL: yalnız IP
ADD AX, 1
RET
uzak PROC
SHL AX, 1
RETF           ; IP ve CS çeker!
uzak ENDP
```

Satır	Komut	AX	CS:IP (sonraki)	SP	Yığın (tepeden)
2	`MOV AX, 5`	0005h	0700:0103	FFFEh	0000h
3	`CALL uzak`	0005h	0700:010A	**FFFCh**	0106h, 0000h
7	`SHL AX, 1`	**000Ah**	0700:010C	FFFCh	0106h, 0000h
8	`RETF`	000Ah	**0000:0106**	**0000h**	boş

Tablo 8.2.28: Yakın CALL, uzak dönüş izlemesi (her satır yürütüldükten sonraki durum).

Yakın CALL yığına yalnız IP'yi (0106h) iter. RETF ise iki sözcük çeker. İlki IP olur: 0106h, bu doğrudur. İkincisi CS olur. Yığında CS bulunmadığı için RETF bir alttaki sözcüğü, DOS'un .COM programı için ittiği 0000h'yi alır. SP 4 artar ve 0000h olur. İşlemci 0000:0106h adresine atlar (Tablo 8.2.28).



Şekil 8.2.24: Solda yakın CALL'dan sonra: yığında yalnız IP (2 bayt). Sağda RETF'ten sonra: CS = 0000h, IP = 0106h, SP = 0000h.

0000:0106h fiziksel adresi 00106h'dir. Bu adres kesme vektör tablosunun içindedir (00000h–003FFh); tip 41h'nin vektörü 00104h–00107h baytlarındadır. Orada program kodu değil, kesme yordamlarının adresleri vardır. İşlemci bu baytları komut sanıp yürütür ve program çöker. `ADD AX, 1` hiç çalışmaz. Ters uyuşmazlıkta, yani CALL FAR ile çağrılan prosedür yakın RET ile bittiğinde, CS yığında kalır ve dönüş yine yanlış olur.

8.2.8.3 Hata listesi

Prosedürlerde en sık yapılan hatalar Tablo 8.2.29'da toplanmıştır. Çoğu assembler'ın yakalayamadığı, yalnız çalışma anında ortaya çıkan hatalardır.

Hata	Ne olur?	Önlem
RET'i unutmak	İşlemci ENDP'yi görmez; prosedürden sonraki kod ya da veri yürütülür.	Her prosedürün her yolu RET ile biter.
PUSH / POP dengesizliği	RET yanlış değeri IP'ye alır.	Her PUSH'un ters sırada bir POP'u olur.
Prosedüre düşmek	Ana kodun ortasındaki prosedür CALL edilmeden yürütülür; sonundaki RET programı bitirir.	Prosedürler ana programın RET'inden sonra yazılır ya da üzerinden JMP ile atlanır.
CALL yerine JMP	JMP dönüş adresi itmez; RET yığındaki ilgisiz bir değere döner.	Prosedüre yalnız CALL ile girilir.
Ad uyuşmazlığı	`p1 PROC` … `p2 ENDP` : assembler hatası.	Başta ve sonda aynı ad.
Erken RET	PUSH'tan sonra koşullu bir RET, saklanan değeri dönüş adresi sanır.	Bütün yollar tek çıkış etiketine gider.
LEA yerine MOV	`MOV SI, dizi` adresi değil dizinin ilk sözcüğünü yükler; prosedür ilgisiz bir adresten okur.	Adres için `LEA` ya da `MOV SI, OFFSET dizi`.
Gizlice bozulan DX	16 bitlik MUL ve DIV DX'i de yazar; çağıranın DX'i silinir.	Açıklamaya “Bozar: DX” ya da PUSH/POP DX.
Tür uyuşmazlığı	Yakın CALL + RETF ya da CALL FAR + RET: yığın 2 bayt kayar.	Tür PROC satırında; dönüş için hep RET.
Bayrağı bozmak	STC/CLC'den sonra ADD, CMP gibi bir komut CF'yi değiştirir.	Bayrağı kuran komut RET'ten önceki son bayrak değiştiren komut olur.
Sonuç yazmacını korumak	PUSH AX / POP AX, AX'teki sonucu siler.	Çıktı yazmaçları korunmaz.

Hata	Ne olur?	Önlem
Sonsuz özyineleme	Durma koşulu yoksa ya da CALL'dan sonra denetlenmiyorsa her çağrı 2 bayt yer, yığın taşar.	Taban durum CALL'dan önce denetlenir.

Tablo 8.2.29: Prosedürlerde sık yapılan hatalar.

Sık yapılan hata

Prosedür yerine JMP kullanmak. Aşağıdaki programda `CALL ekle` yerine `JMP ekle` yazılmıştır:

```
ORG 100h
MOV AX, 1
JMP ekle
MOV BX, 9      ; çalışır mı?
RET
ekle PROC
ADD AX, 5
RET
ekle ENDP
```

JMP yığına bir şey itmez. `ekle` 'nin RET'i yığının tepesindeki DOS'un 0000h'sini alır ve program orada, AX = 6 iken sessizce biter. `MOV BX, 9` hiç çalışmaz. Program çökmediği için hata fark edilmeyebilir. Yığında başka bir değer olsaydı RET rastgele bir adrese dönerdi.

8.2.9 Alıştırmalar ve çözümleri

Sunumdaki altı alıştırma burada ara adımlarıyla çözülmüştür.

1. 0120h adresindeki `CALL p` komutu 0100h'deki p'yi çağırıyor. Makine kodu nedir? Yığına ne itilir?

CALL 3 bayttır; sonraki komut 0123h'dedir. Uzaklık = 0100h − 0123h = −23h. 16 bitlik ikiye tümleyende −23h = FFDDh'dir. Düşük bayt önce yazılır: `E8 DD FF`. Yığına dönüş adresi olarak 0123h itilir. Sağlama: 0123h + FFDDh = 10100h, 16 bite indirgenince 0100h.

2. m2 örneğinde ikinci `CALL m2` 'den önce `MOV AL, 3` eklenseydi son AX ne olurdu?

Birinci çağrı AX = 1 × 2 = 0002h verir. Eklenen komut AL'yi 3 yapar; AH = 00h kalır. İkinci çağrı AX = 3 × 2 = 0006h verir. Bu kez ikinci çağrının birinci parametresi birinci çağrının sonucuna bağlı değildir. Eklenen komut 2 bayt olduğu için ikinci CALL'ın adresi ve dönüş adresi 2 bayt kayar, ama sonuç değişmez.

3. SP = FFFEh iken art arda 3 CALL yapılıyor (hiç RET yok). SP kaç olur?

Her yakın CALL yığına 2 bayt iter: FFFEh − 3 × 2 = FFFEh − 6 = FFF8h. Yığında üç dönüş adresi vardır. Çağrılar uzak olsaydı her biri 4 bayt iterdi ve SP = FFF2h olurdu.

4. CX'i bozan bir prosedürü LOOP içinde çağırmak neden hatalıdır? İki çözüm önerin.

LOOP, sayacı CX'te tutar. Prosedür CX'i değiştirirse dönüşte LOOP çağıranın sayacı yerine prosedürün bıraktığı değeri azaltır. Döngü yanlış sayıda döner; prosedür CX = 0 bırakıyorsa LOOP onu FFFFh yapar ve döngü neredeyse hiç bitmez. Birinci çözüm, prosedürün girişte `PUSH CX`, çıkışta `POP CX` yapmasıdır. İkinci çözüm, çağıranın CALL'ın çevresinde `PUSH CX` / `CALL` / `POP CX` yazmasıdır. CX prosedürün çıktısıysa yalnız ikinci çözüm uygulanabilir.

5. AL'deki küçük harfi büyük harfe çeviren `buyuk` prosedürünü yazın (girdi/çıktı AL).

```
buyuk PROC        ; Girdi/Çıktı: AL
cmp al, 'a'
jb son            ; 'a'dan küçük: değiştirme
cmp al, 'z'
ja son            ; 'z'den büyük: değiştirme
sub al, 20h       ; 'a' (61h) → 'A' (41h)
son: ret
buyuk ENDP
```

ASCII tablosunda küçük ve büyük harfler arasında sabit 20h fark vardır. Prosedür yalnız 'a'–'z' aralığındaki karakterleri değiştirir; JB ve JA işaretsiz karşılaştırmadır. Başka yazmaç kullanmadığı için PUSH/POP gerekmez. 'a', 'Z', '{' ve '`' karakterleriyle denendiğinde sonuç 'A', 'Z', '{' ve '`' olur; aralığın iki yanındaki karakterler değişmez.

6. print_me'yi, yazdığı karakter sayısını CX'te döndürecek şekilde değiştirin.

Prosedürün başına `MOV CX, 0`, döngüde `ADD SI, 1`'in yanına `INC CX` eklenir. CX çıktı olduğu için korunmaz. 'Hello World!' için dönüşte CX = 12 olur. INC CX bayrakları değiştirir, ama döngünün başındaki CMP her turda ZF'yi yeniden kurduğu için sonuç etkilenmez.

8.2.10 Özet

- **Prosedür**, belirli bir görevi yapan, CALL ile çağrılıp RET ile çağrının arkasına dönen kod parçasıdır. Kod bir kez yazılır, birçok yerden çağrılır.
- **PROC / ENDP** ve END direktiftir; makine kodu üretmez. CALL ve RET işlemci komutudur. Prosedürler ana programın RET'inden sonra yazılır; yoksa akış prosedüre düşer.
- **CALL** sonraki komutun adresini yığına iter (SP − 2) ve IP'yi hedefe çevirir. Yakın doğrudan CALL hedefi bağıl uzaklık olarak saklar: rel16 = hedef − (CALL adresi + 3).
- **RET**: IP ← [SP], SP + 2. .COM programında son RET, DOS'un ittiği 0000h'yi alır ve PSP'deki INT 20h programı bitirir.

- **Uzak CALL** önce CS'yi, sonra IP'yi iter (4 bayt); **RETF** önce IP'yi, sonra CS'yi çeker. Çağrı ile dönüşün türü uyuşmalıdır. **Dolaylı CALL** hedefi yazmaçtan ya da bellekten alır; atlama tablolarının temelidir.
- Çağrı yükü: CALL 19, RET 8, PUSH 11, POP 8 tur. İki yazmaç saklayan prosedürün yükü 65 turdur (5 MHz'de 13 µs). Çok kısa ve sık işler için makro daha uygundur.
- **Parametreler** yazmaçla, bellek değişkeniyle, adresle (SI/DI/BX) ya da yığınla geçirilir. Değerle geçirmede kopya, adresle geçirmede yer verilir. Sonuç yazmaçla, CF ile ya da bellekle döner.
- **Yazmaçların korunması**: prosedür kullandığı yazmaçları PUSH ile saklar ve ters sırayla POP eder; çıktı yazmaçları korunmaz. PUSH yapan prosedürün bütün yolları tek çıkış noktasında birleşir.
- **Yığın çerçevesi**: `PUSH BP` · `MOV BP, SP` · `SUB SP, n`. Parametreler [BP+4], [BP+6]…, yerel değişkenler [BP−2], [BP−4]… adresindedir. Çıkış: `MOV SP, BP` · `POP BP` · `RET n`.
- **Özyineleme**: prosedür kendini çağırır; her düzey yığında kendi dönüş adresini ve değerlerini tutar. Taban durum CALL'dan önce denetlenir. Yığın gereksinimi en derin çağrı yoluna göre hesaplanır ve kesmeler için pay bırakılır.
- Hataların çoğunun kökü, RET anında yığının tepesinde dönüş adresinin olmamasıdır: unutulan POP, erken RET, CALL yerine JMP, tür uyuşmazlığı.

8.2.11 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

prosedür *procedure (subroutine)* · prosedür çağrısı *procedure call* · dönüş adresi *return address* · direktif *directive* · etiket *label* · yığın *stack* · yığın işaretçisi (SP) *stack pointer* · taban işaretçisi (BP) *base pointer* · yığına itme / yığından alma (PUSH / POP) *push / pop* · LIFO (son giren ilk çıkar) *last in, first out* · parametre geçirme *parameter passing* · değerle / adresle parametre geçirme *pass by value / by reference* · yazmaçların korunması *register preservation* · yığın çerçevesi *stack frame* · yerel değişken *local variable* · özyineleme *recursion* · yığın taşması / alt taşması *stack overflow / underflow* · bağıl konum *offset* · kesim *segment* · bayrak *flag* · little-endian *little-endian*

8.2.12 Kendini Dene

1. 0150h adresindeki `CALL p`, 0180h'deki p'yi çağırıyor. Komutun baytları nedir, yığına hangi değer

itilir?

Cevap

Sonraki komut 0153h'dedir. Uzaklık 0180h − 0153h = 002Dh: `E8 2D 00`. Yığına 0153h itilir.

2. SP = FFFEh iken çağıran iki parametre itip yakın CALL yapıyor. Prosedür `PUSH BP`, `MOV BP, SP`, `SUB SP, 4` ile başlıyor. BP ve SP kaçtır? Birinci itilen parametre ve iki yerel değişken hangi adreslerdedir?

Cevap

İki parametre (4 bayt), CALL (2) ve PUSH BP (2): BP = FFFEh − 8 = FFF6h. SUB SP, 4'ten sonra SP = FFF2h. Birinci parametre [BP+6] = FFFCh'de, ikinci [BP+4] = FFFAh'dedir. Yereller [BP−2] = FFF4h ve [BP−4] = FFF2h'dedir. `MOV SP, BP`, `POP BP` ve `RET 4`'ten sonra SP = FFFEh olur.

3. AX = 1111h, BX = 2222h iken çağrılan bir prosedür `PUSH AX`, `PUSH BX`, `MOV AX, 0`, `MOV BX, 0`, `POP AX`, `POP BX`, `RET` yürütüyor. Dönüşte AX ve BX nedir? RET doğru adrese döner mi?

Cevap

Yığının tepesinde BX'in değeri (2222h) vardır; ilk POP onu AX'e alır. AX = 2222h, BX = 1111h olur: iki yazmaç yer değiştirmiştir. PUSH ve POP sayısı eşit olduğu için yığın dengelidir ve RET doğru döner. Hata yalnız yazmaç değerlerinde görülür.

4. Sonucunu CF ile döndüren bir prosedür `STC`'den sonra `OR AX, AX`, `POP CX` ve `RET` yürütüyor. Çağıran CF'yi kaç bulur?

Cevap

CF = 0. OR mantık komutudur ve CF ile OF'yi her zaman sıfırlar. STC'nin kurduğu değer kaybolur. POP ve RET bayraklara dokunmaz; bayrağı kuran komut RET'ten önceki son bayrak değiştiren komut olmalıdır.

5. 0700:0103 adresindeki `CALL FAR PTR uz`, 0700:0109'daki FAR prosedürü çağırıyor. Komut kaç bayttır? SP = FFFEh ise yığına ne yazılır? Prosedürün RET'i hangi kodla üretilir?

Cevap

Uzak doğrudan CALL 5 bayttır: `9A 09 01 00 07` (önce bağıl konum, sonra kesim). Dönüş adresi 0103h + 5 = 0108h'dir. Önce CS itilir: [FFFCh] = 0700h; sonra IP: [FFFAh] = 0108h. SP = FFFAh olur. FAR prosedürdeki RET, CB (RETF) olarak kodlanır ve dönüşte SP = FFFEh olur.

6. main, P'yi çağırıyor; P, AX, BX ve CX'i itip Q'yu çağırıyor; Q, `PUSH BP` ve `SUB SP, 4` yapıyor. Program SP = FFFEh ile başlıyorsa SP en çok kaça iner? Bir kesme için pay bırakılırsa en az kaça kadar yer ayrılmalıdır?

   Cevap

   2 (CALL P) + 6 (üç PUSH) + 2 (CALL Q) + 2 (PUSH BP) + 4 (yerel alan) = 16 bayt. SP en çok FFFEh − 10h = FFEEh'ye iner. Bir kesme 6 bayt daha iter: FFE8h. Kesme yordamının kendi saklamaları da hesaba eklenir.

7. Özyinelemeli `yaz` prosedürü AX = 1000 ile çağrılıyor. Ekranda ne görünür? SP en çok kaça iner?

   Cevap

   Ekranda 1000 görünür; sıfırlar da birer basamaktır. 1000 dört basamaklıdır, dört düzey: 4 × (2 + 2) = 16 bayt. SP en çok FFFEh − 16 = FFEEh'ye iner.

8. `a` değişkeni 0110h adresindedir ve 0005h değerini taşır. Bir programcı adresle geçirme için `LEA SI, a` yerine `MOV SI, a` yazıyor. SI kaç olur? Prosedür `[SI]`'dan okuduğunda nereye erişir?

   Cevap

   `MOV SI, a` değişkenin içeriğini yükler: SI = 0005h. Prosedür DS:0005h'den, yani .COM programında PSP'nin içinden okur. Okunan değer `a` ile ilgisizdir. Prosedür yazıyorsa PSP'yi bozar. Assembler bu hatayı yakalamaz, çünkü iki komut da geçerlidir.

9. Yerel değişkenli `kare_art` prosedürü (x·x + x) AX = 300 ile çağrılıyor. Dönüşte AX ve DX nedir? Sonuç doğru mudur?

   Cevap

   300 × 300 = 90.000 = 15F90h. 16 bitlik MUL, DX = 0001h, AX = 5F90h verir; CF = OF = 1 olur. ADD AX, [BP−2]: 5F90h + 012Ch = 60BCh. Dönüşte AX = 60BCh, DX = 0001h'dir. Doğru sonuç 90.300 = 160BCh'dir ve 16 bite sığmaz. Prosedür yalnız AX'e baktığı için sonuç yanlıştır; üstelik çağıranın DX'i bozulmuştur.

8.2.13 İleri okuma

- Intel, *The 8086 Family User's Manual* (1979): CALL, RET, PUSH ve POP komutlarının işlem tanımları, makine kodları ve saat turu sayıları. Tablo 8.2.8'deki değerler bu kaynaktandır.
- B. B. Brey, *The Intel Microprocessors* (8. baskı), “Procedures” kesimi: yakın ve uzak prosedürler, CALL ve RET'in biçimleri, yığınla parametre geçirme.

- K. R. Irvine, *Assembly Language for x86 Processors* (7. baskı), "Procedures" bölümü: yığın çerçevesi, yerel değişkenler, özyineleme ve çağırma kuralları ayrıntılı örneklerle.
- emu8086 belgeleri, *8086 Assembler Tutorial for Beginners*, Part 8 "Procedures": emu8086'da prosedür yazımı ve print_me örneği.

8.2.14 Kaynaklar

1. emu8086 belgeleri: *8086 Assembler Tutorial for Beginners* (Part 8: Procedures).
2. Intel, *The 8086 Family User's Manual*, 1979: CALL, RET ve yığın işlemleri.
3. B. B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson, 2009: Kesim 6.3 "Procedures".
4. K. R. Irvine, *Assembly Language for x86 Processors*, 7. baskı, Pearson, 2014: Bölüm 5 "Procedures".

KISIM III · BÖLÜM 9

Makrolar ve emu8086.inc Kütüphanesi

Makrolar, tekrar eden bir kod parçasına ad verir; prosedürlerden farklı olarak assembler tarafından kullanıldıkları her noktada metin olarak açılır. Bölümde MACRO ve ENDM ile parametreli makro tanımı, genişletme, parametre tuzakları, LOCAL ile etiket çakışmalarının önlenmesi ve makro ile prosedürün boyut ve hız açısından karşılaştırılması anlatılır; PUTC ve POWER makroları adım adım çalıştırılır. emu8086.inc kütüphanesi kısmında INCLUDE direktifi, PUTC, GOTOXY, PRINT ve PRINTN gibi makrolar ile sayı okuma ve yazma prosedürleri işlenir. Kütüphane makro ve prosedürleriyle yazılmış örnek programlarla bölüm tamamlanır.

Kod (8086): `github.com/sercankulcu/microprocessors-8086/Bolum09`

BU BÖLÜMÜN NOTLARI

9.1 Makrolar

9.2 emu8086.inc Kütüphanesi

KISIM III · BÖLÜM 9.1

Makrolar

Assembly programlarında aynı birkaç komut sık sık yan yana yazılır: bir karakteri ekrana basmak, birkaç yazmacı yığına saklamak, iki sayıyı toplayıp belleğe yazmak. Prosedür bu tekrarı çalışma anında, CALL ve RET ile çözer. Makro ise aynı işi derleme anında yapar: assembler makronun adını gördüğü her yere gövdesini yeniden yazar. İşlemci makroyu hiç görmez; yalnız ortaya çıkan komutları yürütür. Not önce makronun nasıl tanımlandığını ve açıldığını, parametrelerin neden metin gibi davrandığını anlatır. Ardından makro ile prosedür arasındaki boyut ve hız dengesini ve LOCAL direktifinin çözdüğü sorunu ele alır. PUTC ve POWER makroları üzerinden yazmaç koruma ve sınır durumları işlenir. Son kesimler tekrar bloklarını ve koşullu derlemeyi, yani assembler'ın kaynak üzerinde yaptığı daha ileri metin işlemlerini ele alır. Her örnek, açılmış kodun adım adım izlemesiyle yazmaç ve bellek düzeyinde gösterilir.

ÖĞRENME ÇIKTILARI

Makro tanımı
MACRO ve ENDM ile parametreli makro yazmak; makronun kullanıldığı satırdan önce tanımlanması gerektiğini açıklamak.

Genişletme
Bir makro çağrısının hangi komutlara açılacağını, parametreleri metin olarak yerine koyarak elle bulmak.

Makro mu, prosedür mü?
Kod boyutunu bayt bayt, ek süreyi saat turu olarak hesaplayıp verilen bir iş için makroyu ya da prosedürü gerekçesiyle seçmek.

LOCAL
Etiket içeren makrolarda *Duplicate declaration* hatasının nedenini açıklamak ve LOCAL ile her açılıma benzersiz etiket vermek.

Yan etkiler ve sınır durumları
Makronun bozduğu yazmaçları PUSH ve POP ile korumak; parametre tuzaklarını, işleç önceliği hatalarını ve LOOP'un CX = 0 tuzağını fark etmek.

İleri makro özellikleri
REPT, IRP ve IRPC ile derleme anında kod ve tablo üretmek; IF, IFB ve IFIDN ile koşullu derleme yapmak.

9.1.1 Makro kavramı

9.1.1.1 Assembler'ın yaptığı metin işi

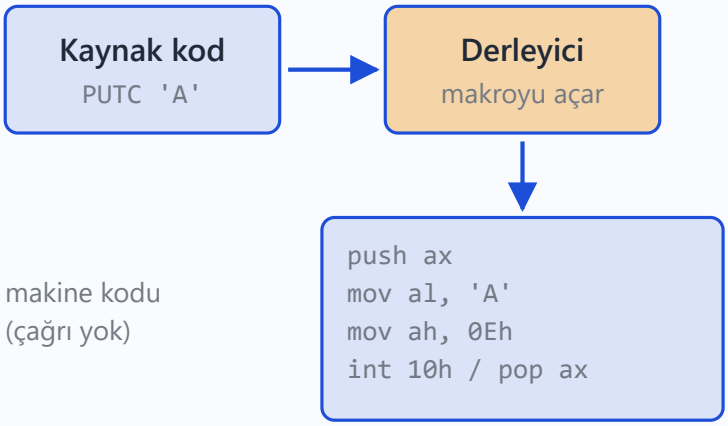
Bir assembly programı iki ayrı aşamadan geçer. Önce assembler (*assembler*) kaynak dosyayı okur ve makine koduna çevirir. Sonra işlemci bu makine kodunu bellekten getirip yürütür. Kaynakta yazılan her şey birinci aşamada bir karşılık bulur, ama her satır makine koduna dönüşmez. `ORG 100h`, `DB`, `EQU` gibi direktifler (*directive*) assembler'a bilgi verir; bunlardan bazıları bayt üretir, bazıları hiç üretmez. Makro da bu birinci aşamanın aracıdır.

Makro, tekrar eden bir kod parçasına ad verir. Bu yönüyle prosedüre benzer. Fark, adın nerede işlendiğindedir. Prosedürün adı makine kodunda bir adrese dönüşür; işlemci çalışma anında `CALL` ile o adrese gider ve `RET` ile geri döner. Makronun adı ise makine koduna hiç girmez. Assembler, makronun kullanıldığı her satırı silip yerine makronun gövdesini yazar. Bu işleme makro genişletme ya da açılım (*macro expansion*) denir. Genişletme bittikten sonra kaynakta artık makro yoktur; yalnız sıradan komutlar vardır.

Tanım 9.1.1

Makro (*macro*), `MACRO` ve `ENDM` direktifleri arasına yazılan, bir adı ve isteğe bağlı parametreleri olan kod kalıbıdır. Assembler, makronun adını bir komut gibi kullanan her satırı, gövdenin parametreleri yerine konmuş bir kopyasıyla değiştirir. Bu değiştirmeye **makro genişletme** (açılım) denir ve derleme anında, program çalışmadan önce yapılır.

Şekil 9.1.1 bu süreci özetler. Kaynakta tek satır olarak duran `PUTC 'A'`, assembler'dan geçtikten sonra beş komuta dönüşür. Makine kodunda bir çağrı, bir dönüş adresi ya da bir atlama yoktur. İşlemci açısından bu beş komut, programcının elle yazdığı beş komuttan farksızdır.



Şekil 9.1.1: Makro çağrısı derleme anında açılır. Kaynaktaki `PUTC 'A'` satırının yerine assembler gövdeyi yazar; makine kodunda çağrı yoktur.

Bu bakış açısından üç önemli sonuç çıkar. Birincisi, hiç kullanılmayan bir makro makine koduna tek bayt eklemez. Assembler tanımı okur, adını ve gövdesini bir tabloya yazar, ama gövdeden kod üretmez. Kod ancak makro kullanıldığında, kullanıldığı yerde üretilir. Bu yüzden yüzlerce makro içeren bir kütüphane dosyasını programa eklemek, programı büyütmez.

İkincisi, makro kullanıldığı satırdan önce tanımlanmış olmalıdır. Assembler kaynak dosyayı yukarıdan aşağıya okur. `PUTC 'A'` satırına geldiğinde PUTC'yi henüz tanımıyorsa bu satırı bilinmeyen bir komut sayar ve hata verir. Prosedürde böyle bir sıra kısıtı yoktur. Assembler iki geçişli çalıştığı için bir prosedürün adresini, prosedür dosyanın sonunda yazılmış olsa da ikinci geçişte bulur. Makro ise bir adres değil, bir metindir. Metni bilmeden yerine koyamaz.

Üçüncüsü, makronun her kullanımı programı büyütür. Aynı makro on yerde kullanılırsa gövde on kez yazılır. Prosedürün gövdesi ise bellekte bir kez durur. Bu fark, makro ile prosedür arasındaki seçimin temelidir.

9.1.1.2 Makro tanımlama

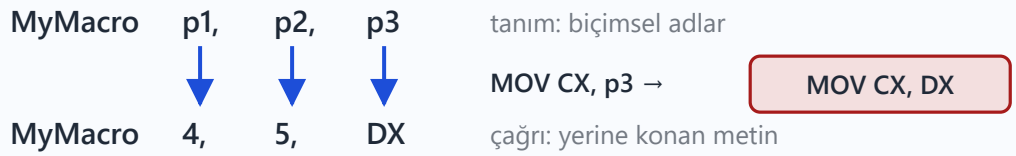
Bir makronun tanımı üç parçadan oluşur: başlık satırı, gövde ve bitiş satırı. Genel biçim şöyledir:

```
ad      MACRO  [parametre1, parametre2, ...]
        <komutlar>
        ENDM
```

`ad` makronun adıdır. Makro kullanılırken bu ad, bir komutun yerine yazılır: `MOV` yazılan yere `PUTC` yazılır. Adın ardından virgülle ayrılmış parametre adları gelir. Bunlar biçimsel adlardır; tanımda yalnız birer yer tutucudur. Çağrıda verilen değerler bu adların yerine metin olarak konur. Parametre hiç olmayabilir: yalnız bir kalıbı adlandıran makrolar da vardır.

Gövde, makro her kullanıldığında yeniden yazılacak satırlardır. Komut, direktif, başka bir makronun çağrısı, hatta veri tanımı olabilir. `ENDM` (*end macro*) gövdeyi bitirir. Prosedürdeki `ad ENDP` kalıbından farklı olarak `ENDM`'in önüne makronun adı yazılmaz.

Parametre adları gövdede yerine konacak metni işaretler. Assembler, gövdedeki her sözcüğü parametre adlarıyla karşılaştırır. Eşleşen sözcüğün yerine çağrıdaki ilgili metni yazar. Şekil 9.1.2 bu eşlemeyi `MyMacro 4, 5, DX` çağrısı için gösterir: birinci parametre p1 4'e, ikinci p2 5'e, üçüncü p3 ise DX'e eşlenir. Gövdedeki `MOV CX, p3` satırı böylece `MOV CX, DX` olur.



Şekil 9.1.2: Parametre eşlemesi. Tanımdaki biçimsel adlar sırayla çağrıdaki metinlerle eşlenir; `MOV CX, p3` satırı `MOV CX, DX` olur.

Burada kritik bir nokta vardır. Assembler parametrenin yazmaç mı, bellek mi, sabit mi olduğuyla ilgilenmez. Yalnız metni değiştirir. Ortaya çıkan satırın geçerli bir 8086 komutu olup olmadığına genişletmeden *sonra*, sıradan bir satır gibi bakılır. Bu yüzden makro tanımında hata olmasa da belirli bir çağrıda hata çıkabilir.

Sık yapılan hata

Makroyu programın sonuna, prosedürlerin yanına yazmak. Prosedür için doğru olan bu yerleşim makroda hataya yol açar: assembler çağrı satırına geldiğinde makroyu tanımadığı için satırı bilinmeyen komut sayar. Makro tanımları dosyanın başına ya da INCLUDE ile eklenen bir kütüphane dosyasına yazılır.

9.1.1.3 Genişletme adım adım

Genişletme en iyi somut bir örnekle anlaşılır. Üç parametreli MyMacro makrosu üç yazmaca birer değer yükler:

```
MyMacro  MACRO  p1, p2, p3
         MOV AX, p1
         MOV BX, p2
         MOV CX, p3
ENDM

ORG 100h
         MyMacro 1, 2, 3
         MyMacro 4, 5, DX
         RET
```

Assembler ilk dört satırı okuduğunda kod üretmez. MyMacro adını, üç parametre adını ve üç gövde satırını bir makro tablosuna kaydeder. `ORG 100h` konum sayacını 0100h'ye kurar. Ardından `MyMacro 1, 2, 3` satırı gelir. Assembler MyMacro'yu tablosunda bulur ve parametreleri eşler: p1 = 1, p2 = 2, p3 = 3. Sonra gövdeyi satır satır kopyalar:

1. `MOV AX, p1` satırında p1 sözcüğü 1 ile değişir: `MOV AX, 1`.
2. `MOV BX, p2` satırı `MOV BX, 2` olur.
3. `MOV CX, p3` satırı `MOV CX, 3` olur.

Bu üç satır, çağrı satırının yerini alır ve sıradan komutlar gibi makine koduna çevrilir. İkinci çağrı `MyMacro 4, 5, DX` için aynı işlem yeni bir eşlemeyle yinelenir: p1 = 4, p2 = 5, p3 = DX. Ortaya `MOV AX, 4`, `MOV BX, 5` ve `MOV CX, DX` çıkar. Son satırdaki DX bir sayı değil, bir yazmaç adıdır. Assembler bunu fark etmez, yalnız "DX" metnini yerine koyar. Sonuçta oluşan `MOV CX, DX` geçerli bir yazmaçtan yazmaca kopyadır.

emu8086 genişletilmiş kodu öykünücü penceresinde şöyle gösterir:

```
MOV AX, 00001h
MOV BX, 00002h
MOV CX, 00003h
MOV AX, 00004h
MOV BX, 00005h
MOV CX, DX
RET
```

Sabitler onaltılık olarak, beş haneyle yazılmıştır; bu yalnız bir gösterim alışkanlığıdır. Önemli olan, iki makro çağrısının altı komuta açılması ve makro adının listede hiç görünmemesidir. Öykünücüde tek adım çalıştırma da bu altı komut üzerinde ilerler. Programcı kaynakta iki satır yazmış, işlemci altı komut yürütmüştür.

Bu komutların makine kodu da hesaplanabilir. Bir 16 bitlik yazmaca anında değer yükleyen `MOV r16, imm16` komutu 3 bayttır: işlem kodu B8h + yazmaç numarası ve iki bayt veri. `MOV AX, 1` = `B8 01 00`, `MOV BX, 2` = `BB 02 00`, `MOV CX, 3` = `B9 03 00`. Yazmaçtan yazmaca `MOV CX, DX` ise 2 bayttır (`8B CA`). `RET` tek bayttır (`C3`). Programın toplam boyutu 5 × 3 + 2 + 1 = 18 bayt, yani 12h bayttır.

9.1.1.4 Kod izleme: açılmış kodu çalıştırmak

Açılmış kodun çalışması, sıradan bir programın çalışmasından farksızdır. Yine de bir ayrıntı izleme tablosunu anlamak için önemlidir. emu8086, bir .COM programını belleğe yüklediğinde CX yazmacına programın bayt cinsinden boyutunu yazar. Bu davranış DOS'un program yükleyicisinden gelir. Bu programda boyut 18 bayt olduğu için yükleme anında CX = 0012h'dir. Öteki genel amaçlı yazmaçlar sıfır, SP ise FFFEh'dir. SP'nin gösterdiği FFFEh adresinde 0000h sözcüğü durur; programın sonundaki RET bu sözcüğü alacaktır.

Tablo 9.1.1 açılmış kodun izlemesini verir. İlk üç adım birinci çağrının, sonraki üç adım ikinci çağrının ürettiği komutlardır. Komut sütununun altındaki küçük yazı, komutun hangi makronun açılımından geldiğini gösterir.

Adım	Adres	Komut	AX	BX	CX	DX	SP	Açıklama
1	0100h	`MOV AX, 1` MyMacro açılımı	0001h	0000h	0012h	0000h	FFFEh	1. açılım, p1 = 1: AX = 0001h
2	0103h	`MOV BX, 2` MyMacro açılımı	0001h	0002h	0012h	0000h	FFFEh	p2 = 2: BX = 0002h
3	0106h	`MOV CX, 3` MyMacro açılımı	0001h	0002h	0003h	0000h	FFFEh	p3 = 3: CX = 0003h; yükleme değeri 0012h silindi
4	0109h	`MOV AX, 4` MyMacro açılımı	0004h	0002h	0003h	0000h	FFFEh	2. açılım, p1 = 4
5	010Ch	`MOV BX, 5` MyMacro açılımı	0004h	0005h	0003h	0000h	FFFEh	p2 = 5
6	010Fh	`MOV CX, DX` MyMacro açılımı	0004h	0005h	0000h	0000h	FFFEh	p3 = DX: yazmaçtan yazmaca kopya, CX = DX = 0000h
7	0111h	`RET` ana program	0004h	0005h	0000h	0000h	0000h	yığının tepesindeki 0000h → IP; SP = 0000h
8	0000h	`INT 20h (PSP:0000h)` PSP	0004h	0005h	0000h	0000h	0000h	PSP:0000h'deki INT 20h programı bitirir

Tablo 9.1.1: İki MyMacro çağrısının açılmış kodunun izlemesi. Her satır, komut yürütüldükten sonraki durumdur; yükleme anında CX = 0012h (program boyutu).

Tabloda dikkat çeken birkaç nokta vardır. CX'in yükleme değeri 0012h, ilk iki adımda değişmeden durur ve üçüncü adımda 0003h ile silinir. İkinci çağrının son komutu CX'e DX'i kopyalar; DX başlangıçta sıfır olduğundan CX = 0000h olur. Adımların hiçbirinde SP değişmez. Makronun gövdesinde PUSH ya da POP olmadığı sürece makro yığına dokunmaz. Aynı iş bir prosedürle yapılsaydı her CALL, SP'yi 2 azaltır ve dönüş adresini yığına yazardı. Son RET ise makroya ait değildir; ana programın RET'idir. Yığının tepesindeki 0000h sözcüğünü IP'ye alır ve program kesimi önekindeki (PSP) `INT 20h` komutuna gider.

Bu programın süresi de saat turu (*clock cycle*) olarak hesaplanabilir. 8086'da `MOV r16, imm16` 4 tur, `MOV r16, r16` 2 tur sürer. Altı komut 5 × 4 + 2 = 22 tur tutar. Prosedürle yazılmış bir sürümde bu süreye her çağrı için CALL ve RET süreleri eklenirdi.

9.1.2 Parametreler

9.1.2.1 Parametre bir metindir: takas tuzağı

Yüksek düzeyli bir dilde fonksiyona verilen parametrenin değeri çağrı anında hesaplanır. Fonksiyona bu değer gider. Makroda böyle bir şey olmaz. Makro parametresi bir değer taşımaz, bir metin taşır. Assembler bu metni gövdedeki her geçtiği yere yazar. Metnin işaret ettiği yazmacın ya da değişkenin değeri ise ancak çalışma anında, o satır yürütüldüğünde okunur. Bu ayrım küçük görünür ama makrolardaki hataların önemli bir kısmı buradan çıkar.

Tanım 9.1.2

Makro parametresi, makro tanımında bir yer tutucu olarak yazılan biçimsel addır. Çağrıdaki karşılığı bir değer değil, bir metindir. Assembler bu metni gövdede parametrenin geçtiği her yere yazar. Metin bir yazmaç ya da değişken adıysa onun değeri, açılmış satır çalışma anında yürütüldüğünde okunur.

Bir programcının MyMacro'yu AX ile BX'i takas etmek için kullanmak istediğini düşünelim. Makronun ilk satırı AX'e p1'i, ikinci satırı BX'e p2'yi yazdığına göre `MyMacro BX, AX, 0` çağrısı "AX ← BX, BX ← AX" yapıyor gibi görünür. Programcı bu çağrıdan sonra AX = 2222h, BX = 1111h bekler. Açılımı yazınca durum değişir:

```
ORG 100h
MOV  AX, 1111h
MOV  BX, 2222h
MOV  AX, BX      ; MyMacro BX, AX, 0
MOV  BX, AX      ; p2 = AX (bozuk!)
MOV  CX, 0
RET
```

Adım	Adres	Komut	AX	BX	CX	Açıklama
1	0100h	`MOV AX, 1111h` ana program	1111h	0000h	0000h	AX = 1111h
2	0103h	`MOV BX, 2222h` ana program	1111h	2222h	0000h	BX = 2222h
3	0106h	`MOV AX, BX` MyMacro açılımı	2222h	2222h	0000h	p1 = BX: AX ← BX; 1111h burada kayboldu
4	0108h	`MOV BX, AX` MyMacro açılımı	2222h	2222h	0000h	p2 = AX: AX'in **şimdiki** değeri okunur, BX = 2222h
5	010Ah	`MOV CX, 0` MyMacro açılımı	2222h	2222h	0000h	p3 = 0
6	010Dh	`RET` ana program	2222h	2222h	0000h	sonuç AX = BX = 2222h

Adım	Adres	Komut	AX	BX	CX	Açıklama
7	0000h	`INT 20h (PSP:0000h)` PSP	2222h	2222h	0000h	program biter

Tablo 9.1.2: `MyMacro BX, AX, 0` ile takas denemesinin izlemesi.

Üçüncü adımda AX, BX'in değeriyle, yani 2222h ile dolar. AX'in eski değeri 1111h artık hiçbir yerde yoktur. Dördüncü adımdaki `MOV BX, AX`, programcının düşündüğü "eski AX"i değil, AX'in o anki değerini okur. Sonuç olarak iki yazmaç da 2222h olur. Bir yüksek düzeyli dilde parametreler çağrı anında hesaplandığı için bu hata oluşmazdı. Makroda ise p2 = AX, "çağrı anındaki AX'in değeri" demek değildir; yalnız "AX" yazısıdır.

Bu tuzağın genel kuralı şudur: makro gövdesinin değiştirdiği bir yazmaç, gövdede o değişiklikten sonra okunan bir parametreye verilmemelidir. MyMacro önce AX'i yazar, sonra p2'yi okur; p2'ye AX vermek bu yüzden yanlıştır. Gerçek bir takas için 8086'nın tek komutluk `XCHG AX, BX` komutu vardır. Bir makro takas yapacaksa ya XCHG kullanır ya da değerlerden birini önce yığına saklar.

> **Sık yapılan hata**
>
> Makro parametresini yüksek düzeyli dildeki değer parametresi gibi düşünmek. `TOPLA2 CX, AX` gibi bir çağrıda gövde önce AX'e CX'i yazıp sonra `ADD AX, AX` yaparsa, beklenen CX + AX değil 2 × CX bulunur. Makronun kendisi için kullandığı yazmaçlar belgelenir ve bu yazmaçlar parametre olarak verilmez.

9.1.2.2 Parametre türleri ve geçerlilik

Assembler parametreye bir tür vermediği için aynı makro çok farklı biçimlerde çağrılabilir. Hangi çağrının geçerli olduğuna makronun kendisi değil, açılımda oluşan komut karar verir. Aşağıdaki TOPLA makrosu bunu iyi gösterir:

```
TOPLA MACRO x, y, z
  MOV AX, x
  ADD AX, y
  MOV z, AX
ENDM
```

Makro x ile y'yi toplar ve sonucu z'ye yazar. Neden doğrudan `ADD z, x` gibi bir şey yazılmadığı sorulabilir. 8086'nın komut kümesi, iki bellek işlenenini tek komutta birleştirmez. `ADD`, `MOV`, `CMP` gibi komutların en çok bir işleneni bellekte olabilir. Bu kısıt donanımdan gelir: komutun ModR/M baytı yalnız bir bellek adresini tanımlayabilir. Bu yüzden makro, toplamayı AX üzerinden yapar. AX burada bir ara yazmaçtır ve makro onu bozar.

Çağrı	Açılım (sorunlu satır)	Durum
`TOPLA a, b, c`	`MOV AX, a` · `ADD AX, b` · `MOV c, AX`	geçerli; üç değişken

Çağrı	Açılım (sorunlu satır)	Durum
`TOPLA BX, 5, CX`	`MOV AX, BX` · `ADD AX, 5` · `MOV CX, AX`	geçerli; yazmaç, sabit, yazmaç
`TOPLA a, b, 7`	`MOV 7, AX`	hata: hedef sabit olamaz
`TOPLA k, b, c` (k DB)	`MOV AX, k`	hata: boyut uyuşmazlığı (16 bit ← 8 bit)
`TOPLA a, b`	`MOV , AX`	hata: eksik işlenen

Tablo 9.1.3: TOPLA makrosunun farklı çağrılarının açılımı ve geçerliliği.

Tablo 9.1.3 beş çağrıyı karşılaştırır. İlk iki çağrıda parametreler farklı türdedir ama açılım geçerli komutlar üretir. Üçüncü çağrıda z = 7 olduğu için son satır `MOV 7, AX` olur. Bir sabite yazılamaz; sabit, komutun içinde gömülü bir sayıdır, yazılabilecek bir bellek hücresi ya da yazmaç değildir. Dördüncü çağrıda k bir bayt değişkendir. `MOV AX, k` 16 bitlik bir yazmaca 8 bitlik bir bellek hücresini kopyalamaya çalışır. Assembler, değişkenin tanımından boyutunu bildiği için bu uyuşmazlığı yakalar. Beşinci çağrıda üçüncü parametre verilmemiştir. Assembler eksik parametreyi boş metinle değiştirir ve son satır `MOV , AX` gibi anlamsız bir satıra dönüşür.

Bu hataların hepsi makronun kullanıldığı satırda bildirilir. Oysa nedeni gövdedeki bir satırdır. Hata iletisini anlamanın yolu açılımı zihinde ya da kâğıt üzerinde yazmaktır. Aynı hata iletisi programın birkaç farklı satırında görünüyorsa neden çoğunlukla bu satırların hepsinde kullanılan makronun gövdesindedir.

MASM, eksik parametre sorununu tanım düzeyinde yakalamak için iki olanak sunar. Parametre adının ardına `:REQ` yazılırsa (ör. `z:REQ`) o parametre zorunlu olur ve verilmediğinde açık bir hata iletisi üretilir. `y:=<1>` biçimi ise parametreye bir varsayılan değer verir; çağrıda y yazılmazsa 1 kullanılır. Bu olanaklar MASM'a özgüdür; daha basit assembler'larda eksik parametre her zaman boş metin olarak kalır.

9.1.2.3 Kod izleme: TOPLA ile iki açılım

Aşağıdaki programda TOPLA makrosu iki farklı biçimde çağrılır. Birinci çağrıda üç parametre de değişkendir. İkinci çağrıda y yerine bir sabit verilir ve birinci çağrının sonucu girdi olarak kullanılır:

```
org 100h
TOPLA a, b, c       ; c = a + b
TOPLA c, 5, d       ; d = c + 5
ret
a dw 1200h
b dw 0345h
c dw ?
d dw ?
```

Açılmış kod altı komut ve bir RET'tir. Değişkenlerin adresleri, kodun boyutuna bağlıdır; bu yüzden önce bayt sayısı hesaplanmalıdır. `MOV AX, a` komutu AX'e özel kısa bir biçimle kodlanır: `A1` işlem kodu ve iki baytlık adres, toplam 3 bayt. `ADD AX, b` bellek işlenenli genel biçimi kullanır: `03 06` ve

iki baytlık adres, toplam 4 bayt. `MOV c, AX` yine kısa biçimdir (`A3` + adres), 3 bayt. Birinci açılım 3 + 4 + 3 = 10 bayttır.

İkinci açılımda ortadaki satır `ADD AX, 5`'tir. 8086'nın AX'e anında değer ekleyen kısa bir biçimi vardır: `05` işlem kodu ve iki bayt veri, toplam 3 bayt. Bu yüzden ikinci açılım 3 + 3 + 3 = 9 bayttır. Aynı makro, parametreye göre farklı uzunlukta kod üretmiştir. Kod 10 + 9 = 19 bayt, RET ile 20 bayttır. Program 0100h'de başladığına göre a, 0100h + 14h = 0114h adresindedir. b 0116h'de, c 0118h'de, d 011Ah'dedir.



Şekil 9.1.3: İkinci açılımın son komutundan sonra bellek ve AX. a, b, c, d sözcükleri 0114h'den başlayarak art arda durur; d'ye 154Ah yazılmıştır.

Adım	Adres	Komut	AX	a	b	c	d	Açıklama
1	0100h	`MOV AX, a` TOPLA açılımı	1200h	1200h	0345h	0000h	0000h	x = a: AX ← [0114h]
2	0103h	`ADD AX, b` TOPLA açılımı	1545h	1200h	0345h	0000h	0000h	y = b: 1200h + 0345h = 1545h
3	0107h	`MOV c, AX` TOPLA açılımı	1545h	1200h	0345h	1545h	0000h	z = c: [0118h] ← 1545h
4	010Ah	`MOV AX, c` TOPLA açılımı	1545h	1200h	0345h	1545h	0000h	2. açılım, x = c: AX ← [0118h]
5	010Dh	`ADD AX, 5` TOPLA açılımı	154Ah	1200h	0345h	1545h	0000h	y = 5 bir sabit: kısa AX biçimi `05 05 00`
6	0110h	`MOV d, AX` TOPLA açılımı	154Ah	1200h	0345h	1545h	154Ah	z = d: [011Ah] ← 154Ah
7	0113h	`ret` ana program	154Ah	1200h	0345h	1545h	154Ah	program biter

Tablo 9.1.4: `TOPLA a, b, c` ve `TOPLA c, 5, d` açılımlarının izlemesi (a = 0114h, b = 0116h, c = 0118h, d = 011Ah).

Tablo 9.1.4 izlemenin ayrıntısını verir. Birinci adımda AX, a'nın değeri 1200h ile dolar. İkinci adımda 0345h eklenir: 1200h + 0345h = 1545h. Üçüncü adımda bu toplam c'ye yazılır. İkinci açılım c'yi okur, 5 ekler ve sonucu d'ye yazar: 1545h + 5 = 154Ah. İki açılım aynı gövdeden üretilmiş olsa da birinde y bir bellek adresi, ötekinde bir sabittir. Assembler bu farkı yalnız metinden çıkarır ve uygun makine kodunu seçer.

Bellek erişimleri donanım açısından da incelenebilir. `MOV AX, a` komutu yürütülürken BIU, DS:0114h, yani fiziksel 07114h adresinden bir sözcük okur. a çift adreste olduğu için okuma tek bir yol döngüsünde (*bus cycle*) biter. Adres tek olsaydı 8086 sözcüğü iki ayrı bayt okumasıyla getirmek zorunda kalırdı. Makro bu ayrıntıyı değiştirmez. Açılımdaki her komut, elle yazılmış eşinin yaptığı bellek erişimlerini aynen yapar.

Bu örnekten çıkan bir başka ders, makronun AX'i bozmasıdır. Çağıran program, TOPLA'dan sonra AX'te eski bir değer bekliyorsa yanılır. Makronun hangi yazmaçları değiştirdiği, makronun belgesinde açıkça yazılmalıdır. Yazmaç koruma, PUTC makrosunda ayrıca ele alınır.

Örnek 9.1.1

`s1 dw 1234h`, `s2 dw 0F00Fh` için `TOPLA s1, s2, s3` çağrısından sonra s3 ve CF ne olur?

Açılım `MOV AX, s1`, `ADD AX, s2`, `MOV s3, AX` 'tir. 1234h + F00Fh = 10243h; bu 17 bitlik bir sonuçtur. AX'e alt 16 bit, 0243h sığar ve en yüksek bit CF'ye gider: CF = 1. MOV bayrakları değiştirmediği için CF makrodan sonra da 1 kalır. s3 = 0243h olur. Makro eldeyi hiçbir yere yazmadığı için işaretsiz toplamın doğru sonucu kaybolmuştur. Çağıran program bu durumu makrodan hemen sonra `JC` ile yakalayabilir.

9.1.2.4 İfadeler ve işleç önceliği

Parametre bir metin olduğuna göre çağrıda bir ifade de verilebilir: `ALAN 2+3, 4` gibi. Assembler bu ifadeyi önce hesaplamaz; olduğu gibi yerine koyar. İşleç önceliği, ifadenin yerine konduğu satırda yeniden uygulanır. Bu da beklenmedik sonuçlara yol açar:

```
ALAN MACRO en, boy
  MOV AX, en*boy
ENDM

FARK MACRO p, q
  MOV CX, p-q
ENDM
```

`ALAN 2+3, 4` çağrısında en = "2+3", boy = "4" olur. Açılım `MOV AX, 2+3*4` satırıdır. Assembler bu ifadeyi derleme anında hesaplar ve çarpma toplamadan önce yapıldığı için 2 + 12 = 14 bulur. Programcının beklediği (2 + 3) × 4 = 20 değeri ortaya çıkmaz. FARK makrosunda da aynı sorun vardır: `FARK 10, 4+1` çağrısı `MOV CX, 10-4+1` olur ve soldan sağa hesaplanınca CX = 7 bulunur, beklenen 5 değil.

Çağrı	Açılım	Makine kodu	Sonuç
`ALAN 2+3, 4`	`MOV AX, 2+3*4`	`B8 0E 00`	AX = 000Eh (14)
düzeltilmiş `ALAN 2+3, 4`	`MOV AX, (2+3)*(4)`	`B8 14 00`	AX = 0014h (20)

Çağrı	Açılım	Makine kodu	Sonuç
`FARK 10, 4+1`	`MOV CX, 10-4+1`	`B9 07 00`	CX = 0007h
düzeltilmiş `FARK 10, 4+1`	`MOV CX, (10)-(4+1)`	`B9 05 00`	CX = 0005h
`ALAN BX, 2`	`MOV AX, BX*2`	–	hata: * yalnız sabitlerle

Tablo 9.1.5: İfade parametrelerinin açılımı ve derleme anındaki değeri.

Çözüm, gövdede her parametreyi paranteze almaktır: `MOV AX, (en)*(boy)` ve `MOV CX, (p)-(q)`. Böylece hangi metin gelirse gelsin, önce kendi içinde hesaplanır. Tablo 9.1.5 iki biçimi makine kodlarıyla karşılaştırır. Dikkat edilirse makine kodunda çarpma ya da toplama komutu yoktur. İfade derleme anında hesaplanmış, sonucu `MOV` komutunun içine anında değer olarak gömülmüştür. İşlemci yalnız 000Eh ya da 0014h sayısını AX'e yükler.

Son satır bu noktanın sınırını gösterir. `ALAN BX, 2` çağrısı `MOV AX, BX*2` satırını üretir. BX'in değeri derleme anında bilinmez; yalnız çalışma anında bellidir. Assembler bir yazmacı çarpamaz, bu yüzden hata verir. Bir yazmacın değerini çarpmak için çalışma anında yürüyen bir komut gerekir: `MUL` ya da 2'nin kuvvetleri için `SHL`. Derleme anında hesaplanan ifade ile çalışma anında yürüyen komut arasındaki bu ayrım, makroları anlamanın anahtarıdır.

> **Sık yapılan hata**
>
> Gövdede parametreleri parantezsiz kullanmak: `en*boy` yerine `(en)*(boy)` yazılmalıdır. Makro yalnız tek bir sayıdan oluşan parametrelerle denendiğinde hata görünmez; ifade içeren ilk çağrıda sonuç sessizce yanlış çıkar. Assembler bir uyarı vermez, çünkü `2+3*4` geçerli bir ifadedir.

9.1.3 Makro ve prosedür

Makro ile prosedür aynı soruna iki farklı çözümdür: tekrar eden kodu bir ad altında toplamak. Biri işi derleme anında, öteki çalışma anında yapar. Bu fark kod boyutuna, hıza, yığın kullanımına ve etiketlerin davranışına yansır. Bu kesim önce iki yöntemi ayrı ayrı tanımlar, sonra bayt ve saat turu hesabıyla karşılaştırır.

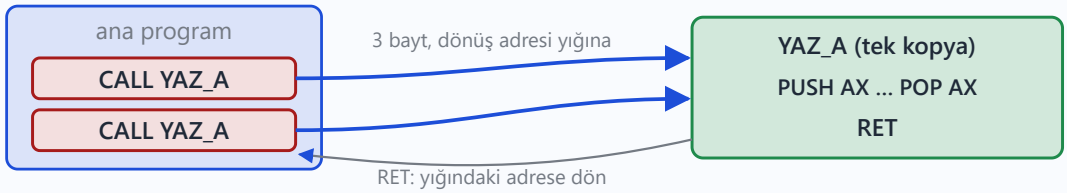
9.1.3.1 Prosedürün özellikleri

Prosedür (*procedure*), `PROC` ve `ENDP` direktifleri arasına yazılan, `CALL` ile çağrılıp `RET` ile dönen kod bloğudur. Gövdesi belleğin belirli bir adresinde tek kopya olarak durur. Aşağıdaki YAZ_A prosedürü ekrana 'A' yazar:

```
YAZ_A PROC
      PUSH AX
      MOV  AL, 'A'
      MOV  AH, 0Eh
      INT  10h
      POP  AX
      RET
YAZ_A ENDP

      CALL YAZ_A  ; 3 bayt
      CALL YAZ_A  ; 3 bayt
```

Yakın (*near*) bir `CALL` komutu yürütüldüğünde işlemci üç iş yapar. Önce SP'yi 2 azaltır. Sonra CALL'dan sonraki komutun adresini, yani dönüş adresini SS:SP'ye yazar. Son olarak IP'ye prosedürün adresini yükler. `RET` bunun tersini yapar: SS:SP'deki sözcüğü IP'ye alır ve SP'yi 2 artırır. Dönüş adresinin yığında saklanması sayesinde aynı prosedür programın farklı yerlerinden çağrılabilir; her seferinde doğru yere döner.



Şekil 9.1.4: Prosedür bellekte tek kopyadır. İki CALL aynı gövdeye gider; RET her seferinde yığındaki dönüş adresine, yani çağırana döner.

Yakın CALL komutu 3 bayttır: `E8` işlem kodu ve iki baytlık bağıl uzaklık. Bağıl uzaklık, prosedürün adresinden CALL'dan sonraki komutun adresi çıkarılarak bulunur. Aynı prosedür yüz kez çağrılsa da gövde bir kez yazılır; her çağrı yalnız 3 bayt ekler. Parametreler yazmaçlarla, bellekle ya da yığınla geçirilir. Prosedürün adı makine kodunda bir adrese dönüştüğü için prosedür dosyanın herhangi bir yerinde, çağrıdan sonra da tanımlanabilir.

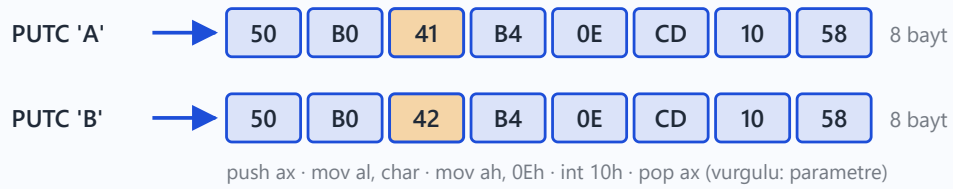
9.1.3.2 Makronun özellikleri

Aynı işi yapan PUTC makrosu ise şöyle yazılır:

```
PUTC MACRO char
     PUSH AX
     MOV  AL, char
     MOV  AH, 0Eh
     INT  10h
     POP  AX
ENDM

     PUTC 'A'  ; 8 bayt
     PUTC 'B'  ; 8 bayt
```

Makro yalnız adı yazılarak kullanılır; CALL yoktur. Parametre, makro adının ardından yazılır. Assembler her kullanımda gövdeyi yeniden yazdığı için iki PUTC, programda iki ayrı 8 baytlık kopya üretir (Şekil 9.1.5). Gövdenin baytları şunlardır: `PUSH AX` 1 bayt (`50`), `MOV AL, 'A'` 2 bayt (`B0 41`), `MOV AH, 0Eh` 2 bayt (`B4 0E`), `INT 10h` 2 bayt (`CD 10`), `POP AX` 1 bayt (`58`). Toplam 8 bayttır. Makroda RET olmadığına dikkat edilmelidir; makro, çağrılan değil, yerine yazılan bir koddur.



Şekil 9.1.5: Makro her kullanımda kodun içine açılır. İki PUTC kullanımı, art arda iki ayrı 8 baytlık kopya üretir; yığın kullanılmaz.

Etiketler konusunda da önemli bir fark vardır. Prosedürün gövdesi tek kopya olduğu için içindeki etiketler de bir kez tanımlanır. Makronun gövdesi ise her kullanımda yeniden yazılır. Gövdede bir etiket varsa ve makro iki kez kullanılırsa aynı etiket iki kez tanımlanmış olur ve assembler *Duplicate declaration* hatası verir. Bu sorunun çözümü olan LOCAL direktifi kendi kesiminde işlenir.

Ölçüt	Makro	Prosedür
Ne zaman işlenir?	derleme anında (metin değiştirme)	çalışma anında (CALL/RET)
Kullanım	`PUTC 'A'`	`CALL YAZ_A`
Kod boyutu	her kullanımda gövde kadar artar	gövde bir kez + her çağrı 3 bayt
Hız	CALL/RET ve yığın erişimi yok	her çağrıda CALL + RET süresi
Parametre	ad sonrası metin	yazmaç, bellek ya da yığın
Yığın	kullanmaz (gövde kullanmadıkça)	dönüş adresi için kullanır
Tanım yeri	kullanımdan önce	herhangi bir yerde
Bitiş	`ENDM`	`ad ENDP`

Ölçüt	Makro	Prosedür
Etiketler	LOCAL gerekir	sorun yok (tek kopya)

Tablo 9.1.6: Makro ile prosedürün karşılaştırması.

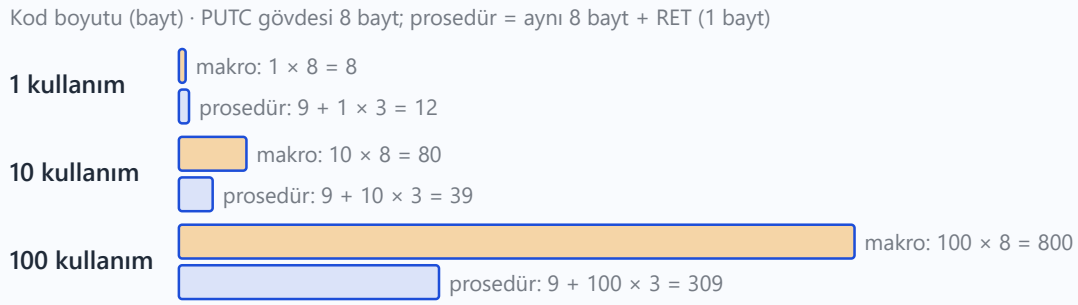
Tablo 9.1.6 iki yöntemi dokuz ölçütle yan yana koyar. Tablonun ilk satırı ötekilerin nedenidir. Makro derleme anında işlendiği için çalışma anına bir maliyet bırakmaz, ama her kullanımda yer kaplar. Prosedür çalışma anında işlendiği için yer kazandırır, ama her çağrıda zaman harcar.

9.1.3.3 Boyut hesabı ve başa baş noktası

Kod boyutunu kullanım sayısına göre hesaplamak için iki basit formül yeter. Gövdesi n bayt olan bir iş k yerde kullanılsın. Makroda gövde k kez yazılır: boyut k · n bayttır. Prosedürde gövde bir kez yazılır, sonuna 1 baytlık RET eklenir ve her kullanım 3 baytlık bir CALL'dır: boyut (n + 1) + 3k bayttır.

$$\text{makro} = k \cdot n \qquad \text{prosedür} = (n + 1) + 3k$$

PUTC'nin gövdesi n = 8 bayttır. Tek kullanımda makro 8 bayt, prosedür 8 + 1 + 3 = 12 bayt tutar; makro daha küçüktür. On kullanımda makro 80 bayta, prosedür 9 + 30 = 39 bayta çıkar. Yüz kullanımda fark büyür: makro 800 bayt, prosedür 9 + 300 = 309 bayttır. Şekil 9.1.6 bu üç durumu çubuklarla gösterir. Makronun çubuğu kullanım sayısıyla doğru orantılı büyür; prosedürün çubuğu 9 baytlık sabit bir parçanın üstüne her kullanımda yalnız 3 bayt ekler.



Şekil 9.1.6: PUTC gövdesi için kod boyutu: 1, 10 ve 100 kullanımda makro (8k bayt) ile prosedür (9 + 3k bayt). Fark kullanım sayısıyla büyür.

Makronun prosedürden büyük olmaya başladığı kullanım sayısına başa baş noktası denebilir. Prosedürün daha küçük olması için $k \cdot n > (n + 1) + 3k$ olmalıdır. Terimler toplanınca $k \cdot (n - 3) > n + 1$ bulunur. Bu eşitsizlikten iki sonuç çıkar.

- $n \le 3$ ise sol taraf sıfır ya da negatiftir ve eşitsizlik hiçbir k için sağlanmaz. Gövde bir CALL kadar kısaysa makro her zaman daha küçüktür. Bu durumda prosedür kod boyutunu azaltmaz, yalnız RET ve çağrı süresini ekler.

- n > 3 ise başa baş noktası k > (n + 1) / (n − 3)'tür. n = 8 için k > 9/5 = 1,8, yani iki kullanımdan itibaren prosedür daha küçüktür: 16 bayta karşı 15 bayt. n = 5 için k > 3, yani dört kullanımdan itibaren. n = 4 için k > 5; beş kullanımda ikisi de 20 bayttır, altı kullanımda prosedür 23 bayta karşı 24 bayt ile öne geçer.

Örnek 9.1.2

PUTC makrosu 50 kez kullanılıyor. Aynı iş prosedürle yapılsaydı kod boyutları ne olurdu?

Makro: 50 × 8 = 400 bayt. Prosedür: gövde ve RET bir kez (8 + 1 = 9 bayt), CALL'lar 50 × 3 = 150 bayt; toplam 159 bayt. Prosedür 241 bayt daha küçüktür. Sık yapılan bir hesap hatası, prosedür için yalnız CALL'ları (150 bayt) saymak ya da makro kopyasını RET ile 9 bayt saymaktır. Makroda RET yoktur.

Bu hesap neden önemlidir? Bir .COM programının kodu, verisi ve yığını tek bir 64 KB'lık kesime sığmak zorundadır. Gömülü bir sistemde program küçük bir ROM'a yazılır ve her bayt bir maliyettir. Gövdesi uzun bir makronun yüzlerce yerde kullanılması bu sınırları hızla zorlar.

9.1.3.4 Hız: CALL, RET ve önden getirme kuyruğu

Hız tarafında tablo tersine döner. Prosedürün her çağrısı, işin kendisine ek olarak bir CALL ve bir RET yürütür. Intel'in 8086 veri sayfasına göre yakın, doğrudan bir CALL 19 saat turu, yakın bir RET 8 saat turu sürer. Her çağrı böylece 27 tur ekler. 4,77 MHz'lik bir 8086'da bir tur yaklaşık 209,6 ns'dir; 27 tur yaklaşık 5,66 µs eder. 5 MHz'de bir tur 200 ns, 27 tur 5,4 µs'dir.

Bu sayının arkasında donanımda olan biten vardır. CALL komutu dönüş adresini yığına yazar; bu bir bellek yazma yol döngüsüdür. Bir yol döngüsü en az dört saat turu (T1–T4) sürer. RET de yığından bir sözcük okur; bu da bir okuma yol döngüsüdür. Daha önemlisi, her iki komut da programın akışını değiştirir. 8086'da yürütme birimi (EU) bir komutu yürütürken yol arayüz birimi (BIU) sonraki komut baytlarını önceden getirir. Bu baytlar altı baytlık önden getirme kuyruğunda (*prefetch queue*) bekler. Akış bir atlamayla değiştiğinde kuyruktaki baytlar işe yaramaz hâle gelir. BIU kuyruğu boşaltır ve yeni adresten getirmeye baştan başlar. EU, ilk komut baytı gelene kadar bekler. CALL ve RET'in uzun sürmesinin bir nedeni budur.

Makronun açılımında ise akış düz bir çizgidir. Komutlar bellekte art arda durur, BIU onları sırayla getirir ve kuyruk hiç boşaltılmaz. Yığına dönüş adresi yazılmaz, yığından okunmaz. PUTC'nin gövdesi kendi başına yaklaşık 78 tur sürer: PUSH 11, iki MOV 4'er, INT 51, POP 8 tur. Bu süreye BIOS'un kendi yordamı ve onun IRET'i eklenir. Prosedür sürümünde aynı işe her çağrıda 27 tur daha eklenir.

Komut	Bayt	Saat turu	Bellek/yığın erişimi
`PUSH AX`	1	11	1 yazma (yığın)
`MOV AL, 'A'`	2	4	yok
`MOV AH, 0Eh`	2	4	yok

Komut	Bayt	Saat turu	Bellek/yığın erişimi
`INT 10h`	2	51	3 yazma (FLAGS, CS, IP) + vektör okuma
`POP AX`	1	8	1 okuma (yığın)
makro kopyası toplam	**8**	**78**	
`CALL` (yakın, doğrudan)	3	19	1 yazma (dönüş adresi)
`RET` (yakın)	1	8	1 okuma (dönüş adresi)
prosedür çağrısı toplam	**3 (+ 9 bir kez)**	**105**	

Tablo 9.1.7: PUTC gövdesinin ve CALL/RET ekinin saat turu olarak süresi (8086 veri sayfası; BIOS yordamının kendi süresi hariç).

Tablo 9.1.7 iki sürümü bayt ve tur olarak karşılaştırır. Ek süre çağrı başına küçüktür ama birikir. Bir döngüde 10.000 kez çağrılan bir prosedür, 4,77 MHz'de 10.000 × 27 = 270.000 tur, yani yaklaşık 56,6 ms ek süre getirir. Kısa bir işte bu ek süre işin kendisinden bile uzun olabilir. Örneğin iki komutluk, toplam 6–8 tur süren bir iş prosedüre konursa, her çağrı süreyi dört beş katına çıkarır.

9.1.3.5 Seçim ölçütleri

Boyut ve hız birlikte tartılınca genel kural ortaya çıkar. Kısa, sık çalışan ve hız gerektiren kod makroya; uzun ve çok yerde kullanılan kod prosedüre uygundur. Beş tipik durum bu kuralı somutlaştırır:

1. **Ekrana tek karakter yazmak, 200 yerde.** Gövde kısadır (8 bayt) ama 200 kopya 1600 bayt eder. Prosedür 9 + 600 = 609 bayt tutar. Boyut önemliyse prosedür, hız önemliyse makro seçilir. Ekrana yazma zaten BIOS çağrısı içerdiği için 27 turluk ek çoğu zaman önemsizdir; burada prosedür daha akıllıca bir seçimdir.
2. **İki yazmacı takas eden tek satırlık iş.** `XCHG` zaten tek komuttur. Bir makro okunabilirliği artırabilir; prosedür ise tek komutluk işe 27 turluk ve 4 baytlık yük ekler. Makro seçilir.
3. **300 baytlık bir sıralama algoritması, 3 yerde.** Makro 900 bayt, prosedür yaklaşık 300 + 9 bayt tutar. Sıralamanın süresi yanında 27 tur önemsizdir. Açıkça prosedür.
4. **Sıkı bir döngüde her turda yapılan iki komutluk iş.** CALL/RET süresi işin kendisinden büyüktür. Döngü binlerce kez döndüğünde fark ölçülebilir hâle gelir. Makro.
5. **Parametreleri yazmaçlara koyup ortak bir kodu çağırmak.** İki yöntem birlikte kullanılır: makro parametreleri yazmaçlara yerleştirir ve prosedürü çağırır. Kullanım makro kadar rahat, boyut prosedür kadar küçük olur. emu8086.inc kütüphanesi de bu düzeni kullanır.

9.1.3.6 Kod izleme: makro hazırlar, prosedür yapar

Beşinci durumu küçük bir programla izleyelim. YAZ_RAKAM makrosu bir rakamı AL'ye koyar ve rakam_yaz prosedürünü çağırır. Prosedür rakamı ASCII koduna çevirip ekrana yazar:

```
YAZ_RAKAM MACRO d
    mov  al, d
    call rakam_yaz
ENDM

org 100h
    YAZ_RAKAM 4
    YAZ_RAKAM 7
    ret
rakam_yaz:
    add  al, '0'     ; rakam → ASCII
    mov  ah, 0Eh
    int  10h
    ret
```

Her makro kopyası yalnız 5 bayttır: 2 baytlık `MOV AL, imm8` ve 3 baytlık `CALL`. Birinci kopya 0100h–0104h'de, ikincisi 0105h–0109h'de durur. Ana programın RET'i 010Ah'dedir ve prosedür 010Bh'de başlar. Prosedürün gövdesi 2 + 2 + 2 + 1 = 7 bayttır. Makro on yerde kullanılsaydı kod 10 × 5 + 7 = 57 bayt tutardı. Gövdenin tamamı (`MOV AL`, `ADD AL`, `MOV AH`, `INT`) makroda olsaydı her kopya 8 bayt, on kopya 80 bayt olurdu.

Adım	Adres	Komut	AL	AH	SP	[SS:SP]	Sonraki IP	Ekran	Açıklama
1	0100h	`mov al, 4` YAZ_RAKAM açılımı	04h	00h	FFFEh	0000h	0102h	–	1. açılım: AL ← 4
2	0102h	`call rakam_yaz` YAZ_RAKAM açılımı	04h	00h	FFFCh	0105h	010Bh	–	dönüş adresi 0105h yığına; IP ← 010Bh
3	010Bh	`add al, '0'` rakam_yaz prosedürü	34h	00h	FFFCh	0105h	010Dh	–	4 + 30h = 34h ('4')
4	010Dh	`mov ah, 0Eh` rakam_yaz prosedürü	34h	0Eh	FFFCh	0105h	010Fh	–	AH = 0Eh: teletype
5	010Fh	`int 10h` rakam_yaz prosedürü	34h	0Eh	FFFCh	0105h	0111h	`4`	ekrana 4
6	0111h	`ret` rakam_yaz prosedürü	34h	0Eh	FFFEh	0000h	0105h	`4`	0105h yığından IP'ye
7	0105h	`mov al, 7` YAZ_RAKAM açılımı	07h	0Eh	FFFEh	0000h	0107h	`4`	2. açılım: AL ← 7
8	0107h	`call rakam_yaz` YAZ_RAKAM açılımı	07h	0Eh	FFFCh	010Ah	010Bh	`4`	dönüş adresi bu kez 010Ah
9	010Bh	`add al, '0'` rakam_yaz prosedürü	37h	0Eh	FFFCh	010Ah	010Dh	`4`	37h ('7')
10	010Dh	`mov ah, 0Eh` rakam_yaz prosedürü	37h	0Eh	FFFCh	010Ah	010Fh	`4`	
11	010Fh	`int 10h` rakam_yaz prosedürü	37h	0Eh	FFFCh	010Ah	0111h	`47`	ekranda 47
12	0111h	`ret` rakam_yaz prosedürü	37h	0Eh	FFFEh	0000h	010Ah	`47`	010Ah'ye dönüş

Adım	Adres	Komut	AL	AH	SP	[SS:SP]	Sonraki IP	Ekran	Açıklama
13	010Ah	`ret` ana program	37h	0Eh	0000h	–	0000h	`47`	ana programın RET'i

Tablo 9.1.8: YAZ_RAKAM makrosu ile rakam_yaz prosedürünün birlikte izlenmesi. [SS:SP] sütunu yığının tepesindeki sözcüğü gösterir.



Şekil 9.1.7: Birinci CALL'dan hemen sonra: dönüş adresi 0105h yığına yazılmış, SP = FFFCh olmuş, IP prosedürün başı 010Bh'ye gelmiştir.

Tablo 9.1.8 ve Şekil 9.1.7 birlikte okunmalıdır. İkinci adımda CALL yürütülür. Dönüş adresi, CALL'dan sonraki komutun adresi olan 0105h'dir. Bu sözcük SS:FFFCh'ye yazılır; SS = 0700h olduğundan fiziksel adres 07000h + FFFCh = 16FFCh'dir. SP = FFFCh olur ve IP 010Bh'ye atlar. Prosedür 4'e 30h ekleyip ASCII '4' (34h) elde eder ve INT 10h ile yazar. Altıncı adımdaki RET, yığının tepesindeki 0105h'yi IP'ye alır ve SP'yi FFFEh'ye geri getirir.

İkinci kopya aynı prosedürü çağırır ama dönüş adresi bu kez 010Ah'dir. Aynı tek gövde, iki farklı yerden çağrılmış ve her seferinde doğru yere dönmüştür. Ekranda 47 görünür. Son RET ana programındır ve yığının tepesindeki 0000h'yi alarak programı bitirir. Makro bu düzende parametreyi hazırlayan ince bir kılıf görevi görür; işin kendisi prosedürde, tek kopyadır.

9.1.3.7 Kod izleme: makronun içindeki RET

Prosedür alışkanlığıyla makronun sonuna RET yazmak, makrolarla ilgili en öğretici hatalardan biridir. Aşağıdaki YAZ makrosu bir karakter yazar ve RET ile biter:

```
YAZ MACRO c
    MOV AL, c
    MOV AH, 0Eh
    INT 10h
    RET             ; hatalı
ENDM

org 100h
    YAZ 'A'
    YAZ 'B'
    ret
```

Program derlenir; assembler açısından RET de sıradan bir komuttur. Açılmış kodda iki YAZ kopyası ve her birinin sonunda bir RET vardır. Programcı ekranda AB bekler.

Adım	Adres	Komut	AX	SP	Sonraki IP	Ekran	Açıklama
1	0100h	`MOV AL, 'A'` YAZ açılımı	0041h	FFFEh	0102h	–	1. açılım: AL ← 'A'
2	0102h	`MOV AH, 0Eh` YAZ açılımı	0E41h	FFFEh	0104h	–	AX = 0E41h
3	0104h	`INT 10h` YAZ açılımı	0E41h	FFFEh	0106h	`A`	ekrana A
4	0106h	`RET` YAZ açılımı	0E41h	0000h	0000h	`A`	makronun RET'i yığının tepesindeki 0000h'yi alır
5	0000h	`INT 20h` `(PSP:0000h)` PSP	0E41h	0000h	–	`A`	PSP'deki INT 20h programı bitirir; 0107h'den sonrası hiç çalışmaz

Tablo 9.1.9: Gövdesinde RET bulunan YAZ makrosunun iki kez kullanıldığı programın izlemesi.

Tablo 9.1.9 beklenenin olmadığını gösterir. Birinci kopya 'A'yı yazar. Dördüncü adımdaki RET, makronun RET'idir. Makroya CALL ile gelinmediği için yığında bir dönüş adresi yoktur. RET yine de yığının tepesinde ne bulursa onu IP'ye alır. Bu .COM programında tepede, DOS'un yerleştirdiği 0000h sözcüğü durur. IP = 0000h olur ve işlemci PSP'nin başındaki `INT 20h` komutuna gider. Program biter. YAZ 'B'nin kopyası ve ana programın RET'i hiç çalışmaz; ekranda yalnız A vardır.

Bu programın şanslı bir sonu vardır: yığının tepesinde tesadüfen programı bitiren bir adres durur. Makro bir prosedürün içinde ya da yığına bir şey itildikten sonra kullanılsaydı RET, yığındaki rastgele bir sözcüğe atlardı ve program kilitlenebilirdi. RET yalnız CALL ile girilen bir prosedürde anlamlıdır.

Sık yapılan hata

Makro gövdesinin sonuna RET yazmak. Makro CALL ile çağrılmaz; gövdedeki RET, makroyu kullanan kodun dönüş adresini (ya da yığında ne varsa onu) alır. Makro, ENDM'e gelindiğinde kendiliğinden "biter": açılımın son komutundan sonra, çağrı satırını izleyen komut yürütülür.

9.1.3.8 EQU, makro ve prosedür

Assembler'ın metin üzerinde çalışan bir başka aracı EQU direktifidir. `SATIR EQU 80` yazıldığında assembler, SATIR adını gördüğü her yere 80 yazar. Bu, tek satırlık ve parametresiz bir makroya benzer. Üç aracın karşılaştırması Tablo 9.1.10'da verilmiştir.

Ölçüt	EQU	Makro	Prosedür
Ne yerine konur?	tek bir değer ya da ifade	bir ya da daha çok satır	hiçbir şey; CALL ile gidilir
Parametre	yok	var (metin)	yazmaç, yığın ya da bellek
Ne zaman?	derleme	derleme	çalışma
Bellekte	yer kaplamaz	her kullanımda gövde kadar	tek kopya + her çağrıda 3 bayt
Örnek	`SATIR EQU 80`	`PUTC 'A'`	`CALL PRINT_NUM`

Tablo 9.1.10: EQU, makro ve prosedür.

EQU ile makronun ortak noktası, ikisinin de assembler'ın kaynak üzerinde yaptığı metin işlemleri olmasıdır. İşlemci ikisini de hiç görmez; yalnız ürettikleri komutları yürütür. Bir EQU sabiti tek başına bayt bile üretmez; ancak bir komutun içinde kullanıldığında o komutun anında değeri olur. Seçim kuralı basittir. Tek bir sayıya ad vermek için EQU, birkaç komutluk kısa kalıp için makro, uzun ya da yaygın bir iş için prosedür kullanılır.

9.1.4 Etiketler ve LOCAL

9.1.4.1 Aynı etiket iki kez

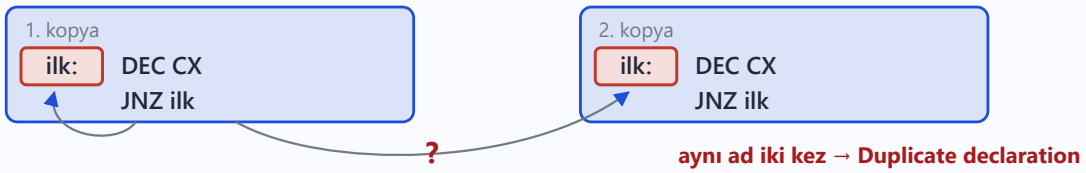
Döngü ya da koşullu atlama içeren her makronun gövdesinde bir etiket (*label*) bulunur. Etiket, assembler'ın bir adrese verdiği addır. Bir programda her etiket yalnız bir kez tanımlanabilir; aksi hâlde aynı ad iki farklı adresi gösterir ve atlamanın hedefi belirsizleşir. Makronun gövdesi her kullanımda yeniden yazıldığı için bu kural makrolarda kolayca çiğnenir. Basit bir bekleme döngüsü bunu gösterir:

```
BEKLE MACRO
ilk:  DEC CX
      JNZ ilk
ENDM

MOV CX, 5
BEKLE
MOV CX, 9
BEKLE
```

Makro bir kez kullanılsaydı sorun olmazdı. İki kullanımdan sonra assembler'ın gördüğü kod şudur:

```
MOV CX, 5
ilk:  DEC CX    ; 1. kopya
      JNZ ilk
MOV CX, 9
ilk:  DEC CX    ; 2. kopya !
      JNZ ilk
```



Şekil 9.1.8: İki açılımda aynı `ilk` etiketi. `JNZ ilk` iki adresten hangisine gideceğini bilemez; assembler *Duplicate declaration* hatası verir.

İkinci `ilk:` satırına gelindiğinde assembler bu adı sembol tablosunda zaten bulur. emu8086 *Duplicate declaration* (yinelenen bildirim) hatası verir ve program derlenmez. Hata iletisi ikinci BEKLE satırını gösterir; programcı o satıra baktığında yalnız makronun adını görür. Hatanın nedenini anlamak için yine açılımı düşünmek gerekir.

9.1.4.2 LOCAL direktifi

Sorunun çözümü, her açılımda etikete farklı bir ad vermektir. Bunu programcı elle yapamaz, çünkü gövde tek bir yerde yazılıdır. Assembler bu işi LOCAL direktifiyle üstlenir.

Tanım 9.1.3

LOCAL direktifi (*LOCAL directive*), makro gövdesinin ilk satırına yazılır ve ardından gelen adları makroya yerel kılar. Assembler her genişletmede bu adların yerine o ana kadar kullanılmamış, benzersiz bir ad koyar. MASM ve emu8086 bu adları `??0000`, `??0001`, `??0002` ... biçiminde, programdaki bütün açılımlar boyunca artan bir sayaçla üretir.

```
MyMacro2 MACRO
    LOCAL label1, label2
    CMP  AX, 2
    JE   label1
    CMP  AX, 3
    JE   label2
label1:
    INC  AX
label2:
    ADD  AX, 2
ENDM
```

MyMacro2 iki yerel etiket bildirir. İlk kullanımda label1 `??0000`, label2 `??0001` olur. İkinci kullanımda sayaç kaldığı yerden sürer: label1 `??0002`, label2 `??0003` olur. Sayaç makroya değil programa aittir; başka bir makronun yerel etiketleri de aynı diziden numara alır. `??` ile başlayan adlar bilerek seçilmiştir: programcının kendi etiketiyle çakışma olasılığı yoktur.

LOCAL'ın gövdenin ilk satırı olması gerekir; MACRO satırından hemen sonra gelmelidir. Assembler, yerel adları gövdenin geri kalanına uygulamadan önce bilmek ister. LOCAL yalnız etiketler için değil, makronun içinde tanımlanan değişken ve prosedür adları için de kullanılır. Makro bir dizgiyi kendi içinde tanımlıyorsa (PRINT makrosunda olduğu gibi), dizginin adı da yerel olmalıdır.

İki kullanımlı bir programın listesi bu adlandırmayı ve adresleri birlikte gösterir:

```
0100  3D 02 00   CMP AX, 2     ; 1. MyMacro2
0103  74 05      JE  ??0000
0105  3D 03 00   CMP AX, 3
0108  74 01      JE  ??0001
010A  40         ??0000: INC AX
010B  05 02 00   ??0001: ADD AX, 2
010E  3D 02 00   CMP AX, 2     ; 2. MyMacro2
0111  74 05      JE  ??0002
0113  3D 03 00   CMP AX, 3
0116  74 01      JE  ??0003
0118  40         ??0002: INC AX
0119  05 02 00   ??0003: ADD AX, 2
011C  C3         RET
```

Koşullu atlamaların makine kodu da öğreticidir. `JE ??0000` = `74 05`: 74h işlem kodu, 05h ise bağıl uzaklıktır. Atlama, JE'den sonraki komutun adresine (0105h) 5 eklenerek bulunur: 0105h + 5 = 010Ah. İkinci kopyadaki `JE ??0002` de `74 05`'tir: 0113h + 5 = 0118h. İki kopyanın makine kodu baytı baytına aynıdır, çünkü 8086'nın kısa atlamaları adresi değil uzaklığı saklar. Farklı olan, etiketlerin adları ve gösterdikleri mutlak adreslerdir.

9.1.4.3 Kod izleme: MyMacro2

MyMacro2'nin işi AX'in değerine göre değişir. AX = 2 ise label1'e atlar, INC ve ADD çalışır, AX 2 artı 1 artı 2 = 5 olur. AX = 3 ise label2'ye atlar, yalnız ADD çalışır, yine 5 olur. Başka her değerde iki atlama da gerçekleşmez ve akış label1'e düşer; AX 3 artar. `MOV AX, 2` ile başlayan programın izlemesi Tablo 9.1.11'dedir.

```
ORG 100h
MOV  AX, 2       ; MyMacro2 öncesi
CMP  AX, 2       ; MyMacro2 başlar
JE   ??0000
CMP  AX, 3
JE   ??0001
??0000:
INC  AX
??0001:
ADD  AX, 2       ; MyMacro2 biter
RET
```

Adım	Adres	Komut	AX	ZF	SF	CF	Sonraki IP	Açıklama
1	0100h	`MOV AX, 2` ana program	0002h	0	0	0	0103h	AX = 2
2	0103h	`CMP AX, 2` MyMacro2 açılımı	0002h	1	0	0	0106h	2 − 2 = 0: ZF = 1
3	0106h	`JE ??0000` MyMacro2 açılımı	0002h	1	0	0	010Dh	ZF = 1: ??0000'a (label1) atlanır
4	010Dh	`INC AX` MyMacro2 açılımı	0003h	0	0	0	010Eh	AX = 3
5	010Eh	`ADD AX, 2` MyMacro2 açılımı	0005h	0	0	0	0111h	AX = 5
6	0111h	`RET` ana program	0005h	0	0	0	0000h	program biter

Tablo 9.1.11: MyMacro2'nin AX = 2 ile izlenmesi; LOCAL etiketleri ??0000 (010Dh) ve ??0001 (010Eh) oldu.

İkinci adımda `CMP AX, 2`, AX'ten 2 çıkarır ama sonucu bir yere yazmaz; yalnız bayrakları kurar. 2 − 2 = 0 olduğundan ZF = 1 olur. Üçüncü adımdaki `JE` ZF'ye bakar ve ??0000'a, yani 010Dh'ye atlar. Sonraki IP sütunu bu atlamayı gösterir: 0108h yerine 010Dh. `CMP AX, 3` ve ikinci `JE` hiç yürütülmez. INC AX 3'ü, ADD AX, 2 ise 5'i verir. Programın başında AX = 3 olsaydı ilk JE atlamaz, ikinci JE label2'ye atlar ve sonuç yine 5 olurdu. AX = 7 olsaydı iki atlama da gerçekleşmez, INC ve ADD çalışır ve AX = 10 olurdu. AX = 0 ile başlayan bir program ise 3 bulurdu.

9.1.4.4 Kod izleme: BEKLE'nin iki kopyası

BEKLE makrosuna `LOCAL ilk` eklendiğinde hata ortadan kalkar:

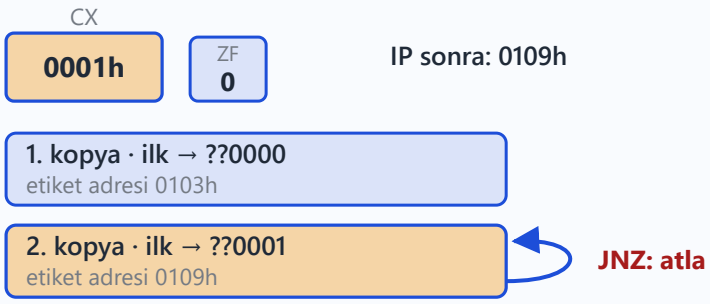
```
BEKLE MACRO
      LOCAL ilk
ilk:  DEC CX
      JNZ ilk
ENDM

org 100h
    mov cx, 3
    BEKLE
    mov cx, 2
    BEKLE
    ret
```

Birinci kopyada ilk etiketi `??0000` adını alır ve 0103h'dedir. İkinci kopyada `??0001` olur ve 0109h'dedir. Her kopya kendi etiketine döner.

Adım	Adres	Komut	CX	ZF	Sonraki IP	Açıklama
1	0100h	`mov cx, 3` ana program	0003h	0	0103h	1. kopyanın sayacı
2	0103h	`??0000: DEC CX` BEKLE açılımı	0002h	0	0104h	1. tur: CX = 2
3	0104h	`JNZ ??0000` BEKLE açılımı	0002h	0	0103h	ZF = 0: ??0000'a (0103h) dön
4	0103h	`??0000: DEC CX` BEKLE açılımı	0001h	0	0104h	2. tur
5	0104h	`JNZ ??0000` BEKLE açılımı	0001h	0	0103h	yine 0103h
6	0103h	`??0000: DEC CX` BEKLE açılımı	0000h	1	0104h	3. tur: CX = 0, ZF = 1
7	0104h	`JNZ ??0000` BEKLE açılımı	0000h	1	0106h	atlamaz; 1. kopya bitti
8	0106h	`mov cx, 2` ana program	0002h	1	0109h	2. kopyanın sayacı
9	0109h	`??0001: DEC CX` BEKLE açılımı	0001h	0	010Ah	1. tur: CX = 1
10	010Ah	`JNZ ??0001` BEKLE açılımı	0001h	0	0109h	??0001'e (0109h) dön, 0103h'ye değil
11	0109h	`??0001: DEC CX` BEKLE açılımı	0000h	1	010Ah	CX = 0, ZF = 1
12	010Ah	`JNZ ??0001` BEKLE açılımı	0000h	1	010Ch	atlamaz
13	010Ch	`ret` ana program	0000h	1	0000h	program biter; toplam 5 DEC

Tablo 9.1.12: LOCAL ilk etiketli BEKLE makrosunun iki kopyasının izlemesi.



Şekil 9.1.9: İkinci kopyanın ilk turu: `JNZ ??0001` 0109h'ye döner; birinci kopyanın 0103h'deki etiketine değil.

Tablo 9.1.12'de birinci kopya üç tur döner. DEC CX her turda CX'i bir azaltır ve sonuca göre ZF'yi kurar. CX sıfırdan farklı kaldıkça ZF = 0'dır ve JNZ 0103h'ye döner. Üçüncü turda CX = 0 olur, ZF = 1 olur ve JNZ atlamaz. İkinci kopya CX = 2 ile iki tur döner ve her seferinde 0109h'ye döner (Şekil 9.1.9). Toplam beş DEC yürütülür. DEC'in CF'ye dokunmadığı, yalnız ZF, SF, OF, AF ve PF'yi değiştirdiği de anımsanmalıdır.

Bekleme süresi saat turu olarak hesaplanabilir. `DEC r16` 2 tur sürer. `JNZ` atladığında 16, atlamadığında 4 tur sürer; atlamada önden getirme kuyruğu boşaltıldığı için fark büyüktür. CX = 3 ile birinci kopya 2 × (2 + 16) + (2 + 4) = 42 tur sürer. Böyle bir döngü, işlemci saatine bağlı kaba bir gecikme üretir.

9.1.4.5 Kod izleme: BUYUK ve işaretli karşılaştırma

Son örnek, LOCAL'ı işaretli karşılaştırmayla birleştirir. BUYUK makrosu p ile q'dan büyük olanı r'ye yazar:

```
BUYUK MACRO p, q, r
    LOCAL ok
    MOV AX, p
    CMP AX, q
    JGE ok
    MOV AX, q
ok: MOV r, AX
ENDM

org 100h
    BUYUK x, y, m1
    BUYUK y, 4, m2
    ret
x  dw 7
y  dw -3
m1 dw ?
m2 dw ?
```

Birinci çağrıda üç parametre de değişkendir. İkincisinde q bir sabittir; bu yüzden `CMP AX, 4` ve `MOV AX, 4` anında değer biçimini kullanır. y = −3, ikiye tümleyen gösterimde FFFDh olarak saklanır.

Adım	Adres	Komut	AX	SF	OF	CF	ZF	m1	m2	Açıklama
1	0100h	`MOV AX, x` BUYUK açılımı	0007h	0	0	0	0	0000h	0000h	p = x: AX = 7
2	0103h	`CMP AX, y` BUYUK açılımı	0007h	0	0	1	0	0000h	0000h	7 − (−3) = 10: SF = 0, OF = 0
3	0107h	`JGE ??0000` BUYUK açılımı	0007h	0	0	1	0	0000h	0000h	SF = OF: ??0000'a atla
4	010Ch	`??0000: MOV m1, AX` BUYUK açılımı	0007h	0	0	1	0	0007h	0000h	m1 = 7
5	010Fh	`MOV AX, y` BUYUK açılımı	FFFDh	0	0	1	0	0007h	0000h	p = y: AX = FFFDh (−3)
6	0112h	`CMP AX, 4` BUYUK açılımı	FFFDh	1	0	0	0	0007h	0000h	−3 − 4 = −7: SF = 1, OF = 0
7	0115h	`JGE ??0001` BUYUK açılımı	FFFDh	1	0	0	0	0007h	0000h	SF ≠ OF: atlamaz
8	0117h	`MOV AX, 4` BUYUK açılımı	0004h	1	0	0	0	0007h	0000h	q = 4 büyük olan
9	011Ah	`??0001: MOV m2, AX` BUYUK açılımı	0004h	1	0	0	0	0007h	0004h	m2 = 4
10	011Dh	`ret` ana program	0004h	1	0	0	0	0007h	0004h	program biter

Tablo 9.1.13: `BUYUK x, y, m1` ve `BUYUK y, 4, m2` açılımlarının izlemesi (x = 7, y = −3).

İkinci adımdaki `CMP AX, y`, 7 − (−3) işlemini yapar. İşaretli yorumla sonuç 10'dur: pozitiftir (SF = 0) ve taşma yoktur (OF = 0). `JGE` (büyük ya da eşitse atla) işaretli karşılaştırmanın sonucuna SF = OF koşuluyla bakar. Koşul sağlandığından ??0000'a atlanır ve m1 = 7 olur. İşaretsiz yorumla aynı çıkarma 0007h − FFFDh'dir; küçükten büyük çıkarıldığı için ödünç doğar ve CF = 1 olur. JGE CF'ye bakmadığı için bu bayrak burada önemsizdir.

İkinci açılımda AX = FFFDh (−3) ve `CMP AX, 4` −3 − 4 = −7 sonucunu verir: SF = 1, OF = 0. SF ≠ OF olduğu için JGE atlamaz. Akış `MOV AX, 4` ile sürer ve m2 = 4 olur. İkinci kopyanın etiketi ??0001'dir; birincisiyle çakışmaz.

Bu örnekte koşullu atlamanın seçimi önemlidir. JGE yerine işaretsiz karşılaştıran `JAE` kullanılsaydı ikinci karşılaştırma FFFDh ile 4'ü işaretsiz sayılar olarak ele alırdı. FFFDh = 65.533, 4'ten büyüktür; ödünç yoktur (CF = 0) ve JAE atlar. m2'ye 4 yerine FFFDh, yani −3 yazılırdı. Makro aynı kalsa da tek bir koşullu atlama komutu sonucun doğruluğunu belirler. İşaretli sayılarla JG, JGE, JL, JLE; işaretsiz sayılarla JA, JAE, JB, JBE kullanılır.

Sık yapılan hata

Etiket içeren bir makroyu bir kez deneyip doğru çalıştığını görünce LOCAL'ı unutmak. Hata ancak makro ikinci kez kullanıldığında ortaya çıkar; o zaman da hata iletisi gövdeyi değil kullanım satırını gösterir. Etiketi olan her makroya, kaç kez kullanılacağından bağımsız olarak

LOCAL yazılır.

9.1.5 PUTC ve yazmaçların korunması

9.1.5.1 PUTC makrosu

PUTC, ekrana tek bir karakter yazan kısa bir makrodur. emu8086.inc kütüphanesindeki PUTC makrosunun aynısıdır:

```
; karakter yazar, imleci ilerletir
PUTC  MACRO  char
      push   ax
      mov    al, char
      mov    ah, 0eh
      int    10h
      pop    ax
ENDM

ORG  100h
START:
      PUTC  'A'
      PUTC  'B'
      RET
```

Makronun asıl işini `INT 10h` yapar. AH = 0Eh, BIOS'un teletype (daktilo benzeri) çıkış hizmetidir. AL'deki karakter imlecin bulunduğu yere yazılır ve imleç bir sağa ilerler. Satır sonunda imleç alt satıra geçer; ekranın sonunda ekran bir satır kaydırılır. 0Dh (satır başı), 0Ah (alt satır), 07h (zil) ve 08h (geri silme) gibi denetim karakterleri yazılmaz, yorumlanır.

Hizmet AH ve AL'yi kullandığı için makro AX'i değiştirmek zorundadır. Makronun başındaki `push ax` ve sonundaki `pop ax` bu değişikliği çağırandan gizler. Makrodan çıkıldığında AX, makroya girilmeden önceki değerine döner. Makronun görünmez bir yan etkisi kalmaz. Bu ilke bütün kütüphane makroları için geçerlidir: makro kendi işi için yazmaç kullanabilir ama çıkarken onları eski hâline getirir.

Tanım 9.1.4

Yazmaçların korunması (*register preservation*), bir makronun ya da prosedürün değiştirdiği yazmaçları başta PUSH ile saklayıp sonda ters sırada POP ile geri yüklemesidir. Böylece çağıranın yazmaçlarındaki değerler kullanım boyunca bozulmaz; yalnız sonuç döndürmek için açıkça belirtilen yazmaçlar değişir.

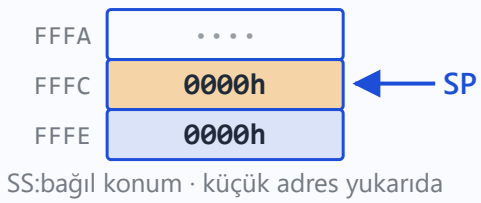
Programın sonundaki RET de önemlidir. İki PUTC kopyasından sonra bellekte programın kodu biter. RET olmasaydı işlemci son POP AX'ten sonraki baytları da komut sanıp yürütmeye devam ederdi. İşlemci kod ile veriyi ayırt etmez; IP'nin gösterdiği her baytı bir işlem kodu olarak çözer. Bellekte ne kaldıysa rastgele komutlar olarak çalışır ve sistem kilitlenebilir. Makro genişletme bu RET'i kendiliğinden eklemez.

9.1.5.2 Kod izleme: PUTC 'A' ve PUTC 'B'

Açılmış kod iki 8 baytlık kopya ve bir RET'tir; toplam 17 bayt. emu8086 yükleme anında CX'e program boyutunu yazdığı için izlemenin başında CX = 0011h'dir. Bu değer program boyunca değişmez, çünkü PUTC CX'e dokunmaz. AX ise başlangıçta 0000h'dir. Her kopyanın makine kodu aynıdır, yalnız karakter baytı farklıdır: `50 B0 41 B4 0E CD 10 58` ve `50 B0 42 B4 0E CD 10 58`.

Adım	Adres	Komut	AX	CX	SP	[SS:SP]	Ekran	Açıklama
1	0100h	`push ax` PUTC açılımı	0000h	0011h	FFFCh	0000h	–	çağıranın AX'i (0000h) yığına; SP = FFFCh
2	0101h	`mov al, 'A'` PUTC açılımı	0041h	0011h	FFFCh	0000h	–	AL ← 41h
3	0103h	`mov ah, 0eh` PUTC açılımı	0E41h	0011h	FFFCh	0000h	–	AX = 0E41h
4	0105h	`int 10h` PUTC açılımı	0E41h	0011h	FFFCh	0000h	`A`	ekrana A; imleç bir sağa
5	0107h	`pop ax` PUTC açılımı	0000h	0011h	FFFEh	0000h	`A`	AX eski değerine döndü; SP = FFFEh
6	0108h	`push ax` PUTC açılımı	0000h	0011h	FFFCh	0000h	`A`	2. kopya: AX yine saklanır
7	0109h	`mov al, 'B'` PUTC açılımı	0042h	0011h	FFFCh	0000h	`A`	AL ← 42h
8	010Bh	`mov ah, 0eh` PUTC açılımı	0E42h	0011h	FFFCh	0000h	`A`	AX = 0E42h
9	010Dh	`int 10h` PUTC açılımı	0E42h	0011h	FFFCh	0000h	`AB`	ekranda AB
10	010Fh	`pop ax` PUTC açılımı	0000h	0011h	FFFEh	0000h	`AB`	AX = 0000h: çağıranın AX'i bozulmadı
11	0110h	`RET` ana program	0000h	0011h	0000h	–	`AB`	program biter

Tablo 9.1.14: `PUTC 'A'` ve `PUTC 'B'` açılımlarının izlemesi; yükleme anında CX = 0011h (17 bayt).



Şekil 9.1.10: `push ax`'ten sonra yığın: çağıranın AX'i (0000h) FFFCh adresine saklanmış, SP = FFFCh olmuştur.

Tablo 9.1.14'te birinci adım AX'i yığına iter. SP 2 azalarak FFFCh olur ve SS:FFFCh'deki sözcüğe 0000h yazılır (Şekil 9.1.10). İkinci ve üçüncü adımlar AL'ye 'A' (41h), AH'ye 0Eh yükler; AX = 0E41h olur. Dördüncü adımda INT 10h ekrana A yazar ve imleci ilerletir. INT komutu kendi içinde FLAGS, CS ve IP'yi yığına iter, BIOS yordamına gider ve IRET ile döner; bu iç ayrıntı izlemede tek adım olarak görünür. Beşinci adımdaki POP AX yığındaki 0000h'yi AX'e alır ve SP'yi FFFEh'ye geri getirir. İkinci kopya aynı yolu 'B' ile izler ve ekranda AB oluşur. Son durumda AX = 0000h'dir: makro çağıranın AX'ini bozmamıştır.

Bu izlemede yığın her kopyada bir sözcük büyüyüp küçülür. PUSH ile POP aynı makronun içinde eşleştiği sürece yığın dengelidir: makroya girildiğindeki SP, makrodan çıkıldığında aynıdır. Bu denge bozulursa, örneğin gövdede bir koşullu atlama POP'u atlarsa, yığında fazladan bir sözcük kalır. Ana programın RET'i bu sözcüğü dönüş adresi sanır ve program yanlış bir yere gider.

9.1.5.3 Yazmaç saklama makroları

PUTC'deki `push ax` / `pop ax` kalıbı, büyük makrolarda dört yazmaca çıkar. Bu kalıp da bir makro olabilir:

```
SAKLA MACRO
  PUSH AX
  PUSH BX
  PUSH CX
  PUSH DX
ENDM

GERI_AL MACRO
  POP DX
  POP CX
  POP BX
  POP AX
ENDM
```

GERI_AL, yazmaçları SAKLA'nın ittiği sıranın tersiyle çeker. Yığın son giren ilk çıkar (*LIFO*) ilkesiyle çalışır: en son itilen DX, ilk çekilen olmalıdır. AX = 1111h, BX = 2222h, CX = 3333h, DX = 4444h iken SAKLA kullanılıp yazmaçlar sıfırlansın. GERI_AL'dan sonra dört yazmaç da eski değerine döner. Çekme sırası itme sırasıyla aynı yazılırsa (`POP AX`, `POP BX`, `POP CX`, `POP DX`) AX = 4444h, BX = 3333h, CX = 2222h, DX = 1111h olur: AX ile DX, BX ile CX yer değiştirir. Bu hata derleme anında yakalanmaz, çünkü her satır kendi başına geçerlidir.

Her SAKLA ve GERI_AL çifti 8 bayt ekler; her PUSH ve POP tek bayttır. Süre olarak 8086'da `PUSH r16` 11 tur, `POP r16` 8 tur sürer: 4 × 11 + 4 × 8 = 76 tur. Bu süre içinde sekiz yığın erişimi, yani sekiz yol döngüsü yapılır. Kısa bir makroda bu süre gövdenin kendisini aşabilir; bu yüzden makro yalnız gerçekten değiştirdiği yazmaçları saklamalıdır.

İki makro aynı gövdede birlikte kullanılmalıdır. Arada koşullu bir çıkış (örneğin bir hata durumunda gövdenin sonuna atlama) varsa bu çıkış da GERI_AL'dan geçmelidir; yoksa yığın dengesiz kalır. 80186 ve sonraki işlemcilerde tek komutluk `PUSHA` ve `POPA` bütün genel amaçlı yazmaçları saklar ve geri yükler. 8086'da bu komutlar yoktur; bu yüzden aynı iş makroyla yazılır.

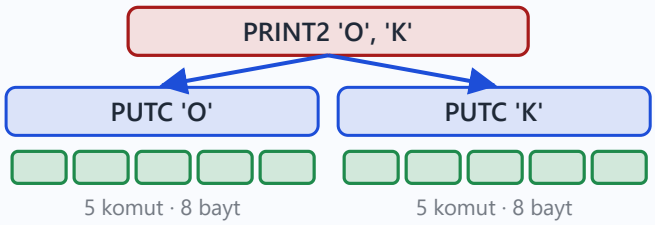
9.1.5.4 İç içe makrolar

Bir makronun gövdesi başka bir makronun çağrısını içerebilir. Aşağıdaki PRINT2 iki karakter yazmak için PUTC'yi iki kez kullanır:

```
PRINT2 MACRO c1, c2
       PUTC c1
       PUTC c2
ENDM

PRINT2 'O', 'K'
```

Assembler genişletmeyi iki düzeyde yapar. Önce dıştaki makroyu açar: `PRINT2 'O', 'K'` satırı `PUTC 'O'` ve `PUTC 'K'` satırlarına dönüşür. Bu iki satır da birer makro çağrısıdır; assembler onları da açar. Sonuçta tek satırlık çağrı on komuta ve 16 bayta dönüşür (Şekil 9.1.11). Ekranda OK görünür.



Şekil 9.1.11: İç içe açılım: PRINT2 iki PUTC çağrısına, her PUTC beş komuta açılır.

İç içe kullanımda tanım sırası yine önemlidir. PUTC, PRINT2'nin kullanıldığı satırdan önce tanımlanmış olmalıdır. Bir makronun kendini çağırması (özyineleme) ise bir sınır durumudur. Prosedürde özyinelemeyi yığın sınırlar ve bir koşul durdurur. Makroda durdurma koşulu çalışma anında değil, derleme anında olmalıdır; aksi hâlde genişletme hiç bitmez. Koşullu derleme direktifleri bu tür bir durdurmayı mümkün kılar.

9.1.6 POWER makrosu

9.1.6.1 Üs alma: MUL ve LOOP

POWER makrosu bir tabanın üssünü hesaplar: r = b^e. Yöntem en basit olanıdır. Geçici sonuç 1 ile başlatılır ve e kez tabanla çarpılır:

```
POWER macro b, e, r
    mov ax, 1        ; geçici sonuç = 1
    mov cx, e        ; üs → CX
    mov bx, b        ; taban → BX
powerloop:
    mul bx           ; DX:AX = AX * BX
    loop powerloop   ; e kez tekrarla
    mov r, ax        ; sonucu yaz
endm

ORG 100h
start:
    POWER base, exponent, result
    ret
base     dw 2     ; taban
exponent dw 3     ; üs
result   dw ?     ; sonuç
```

Makro iki önemli komut kullanır. `MUL BX` işaretsiz çarpmadır: AX'i BX ile çarpar ve 32 bitlik sonucu DX:AX yazmaç çiftine yazar. Yüksek 16 bit DX'e, düşük 16 bit AX'e gider. Bu yüzden MUL, işlenen olarak adı geçmese de DX'i her zaman değiştirir. Sonucun yüksek yarısı sıfır değilse CF ve OF 1 olur; sıfırsa ikisi de 0 olur. SF, ZF, AF ve PF ise MUL'dan sonra tanımsızdır.

`LOOP etiket`, CX'i 1 azaltır ve CX sıfırdan farklıysa etikete atlar. Bayrakları değiştirmez. CX burada döngü sayacıdır; makro ona üssü yükler. 8086'da LOOP atladığında 17, atlamadığında 5 tur sürer. 16 bitlik MUL ise işlenenlerin değerine göre 118 ile 133 tur arasında sürer; bu, komut kümesindeki en yavaş komutlardan biridir.

Açılımın boyutu da hesaplanabilir. `mov ax, 1` 3 bayt, `mov cx, exponent` ve `mov bx, base` bellek işlenenli genel biçimle 4'er bayt, `mul bx` 2 bayt, `loop` 2 bayt, `mov result, ax` AX'e özel kısa biçimle 3 bayttır. Toplam 18 bayttır; RET ile 19 bayt (13h) eder. Değişkenler kodun hemen arkasından başlar: base 0113h'de, exponent 0115h'de, result 0117h'dedir.

9.1.6.2 Kod izleme: POWER 2, 3

Açılmış kod şudur:

```
ORG 100h
mov ax, 1        ; POWER açılımı
mov cx, exponent
mov bx, base
powerloop:
mul bx
loop powerloop
mov result, ax
ret
base     dw 2    ; 0113h
exponent dw 3    ; 0115h
result   dw ?    ; 0117h
```

Adım	Adres	Komut	AX	BX	CX	DX	CF	result	Açıklama
1	0100h	`mov ax, 1` POWER açılımı	0001h	0000h	0000h	0000h	0	0000h	geçici sonuç 1
2	0103h	`mov cx, exponent` POWER açılımı	0001h	0000h	0003h	0000h	0	0000h	CX ← [0115h] = 3
3	0107h	`mov bx, base` POWER açılımı	0001h	0002h	0003h	0000h	0	0000h	BX ← [0113h] = 2
4	010Bh	`mul bx` POWER açılımı	0002h	0002h	0003h	0000h	0	0000h	1. tur: DX:AX = 1 × 2
5	010Dh	`loop powerloop` POWER açılımı	0002h	0002h	0002h	0000h	0	0000h	CX = 2 ≠ 0: powerloop
6	010Bh	`mul bx` POWER açılımı	0004h	0002h	0002h	0000h	0	0000h	2. tur: 2 × 2
7	010Dh	`loop powerloop` POWER açılımı	0004h	0002h	0001h	0000h	0	0000h	CX = 1
8	010Bh	`mul bx` POWER açılımı	0008h	0002h	0001h	0000h	0	0000h	3. tur: 4 × 2
9	010Dh	`loop powerloop` POWER açılımı	0008h	0002h	0000h	0000h	0	0000h	CX = 0: döngüden çık
10	010Fh	`mov result, ax` POWER açılımı	0008h	0002h	0000h	0000h	0	0008h	[0117h] ← 0008h
11	0112h	`ret` ana program	0008h	0002h	0000h	0000h	0	0008h	program biter

Tablo 9.1.15: `POWER base, exponent, result` açılımının izlemesi (base = 0113h, exponent = 0115h, result = 0117h).

Tablo 9.1.15 döngünün üç turunu gösterir. Birinci turda MUL, 1 × 2 = 2 hesaplar: DX:AX = 0000:0002h. Yüksek yarı sıfır olduğundan CF = 0'dır. LOOP, CX'i 3'ten 2'ye indirir ve powerloop'a döner. İkinci turda AX = 4, üçüncü turda AX = 8 olur. Üçüncü LOOP CX'i 0 yapar ve atlamaz. `mov result, ax` 0117h'deki sözcüğe 0008h yazar; $2^3 = 8$.

Tabloda DX'in her MUL'dan sonra 0000h olduğu görülür. Sonuç 16 bite sığdığı sürece bu böyledir. Sonuç 16 biti aşsaydı, örneğin 2^{16} hesaplansaydı, son MUL'dan sonra DX = 0001h, AX = 0000h ve CF = OF = 1 olurdu. Makro yalnız AX'i result'a yazdığı için result = 0 olurdu; doğru sonucun yüksek kısmı DX'te kalır ve kaybolur.

9.1.6.3 Dört tuzak

POWER makrosu küçük ama dört ayrı hata barındırır. Her biri makro yazarken sık görülen bir sorun türünü temsil eder.

1. Üs sıfır. Matematikte $b^0 = 1$'dir; makronun AX = 1 ile başlaması bu sonucu verecek gibi görünür. Ama MUL en az bir kez çalışır, çünkü döngünün koşulu sonda denetlenir. Daha kötüsü, LOOP önce CX'i azaltır, sonra sıfırla karşılaştırır. CX = 0 iken azaltma CX'i FFFFh yapar; bu sıfır değildir ve döngü sürer. Döngü ancak CX yeniden 0 olduğunda, yani 65.536 tur sonra biter. base = 2 için 2^{16}'dan sonra AX'in düşük 16 biti sıfırdır ve öyle kalır; result = 0000h olur. base = 1 için result = 1 çıkar. Bu tesadüfen doğrudur; 65.536 gereksiz MUL yine de çalışmıştır. Her MUL yaklaşık 120 tur sürdüğünden bu boşa giden zaman 4,77 MHz'de bir buçuk saniyeyi aşar.

2. Etiket. `powerloop` LOCAL ile bildirilmemiştir. Makro ikinci kez kullanıldığında etiket iki kez tanımlanır ve *Duplicate declaration* hatası alınır.

3. Yazmaç bozulması. Makro AX'e sonucu koyar; bu beklenen bir değişikliktir. Ama BX'e tabanı, CX'e sayacı yükler ve MUL yüzünden DX'i de değiştirir. Çağıranın bu üç yazmaçtaki değerleri kaybolur.

4. Taşma. Sonuç 65.535'i aşarsa yüksek kısım DX'te kalır ve yazılmaz. Makro, CF ile bu durumu çağırana bildirmez de; son MUL'un CF değeri makrodan sonra durur ama sonraki herhangi bir komut onu silebilir.

Düzeltilmiş sürüm ilk üç sorunu giderir:

```
POWER MACRO b, e, r
      LOCAL p_loop, p_end
      PUSH BX
      PUSH CX
      PUSH DX
      MOV  AX, 1
      MOV  CX, e
      MOV  BX, b
      JCXZ p_end   ; e = 0 → 1
p_loop: MUL BX
      LOOP p_loop
p_end: MOV r, AX
      POP  DX
      POP  CX
      POP  BX
ENDM
```

`JCXZ` (CX sıfırsa atla), LOOP'tan önce sayacı denetler. CX = 0 ise döngünün tamamını atlar ve AX'teki 1 doğrudan sonuca yazılır. LOCAL iki etiketi her açılımda benzersiz yapar. PUSH ve POP, BX, CX ve DX'i korur; makro artık yalnız AX'i değiştirir. Taşma sorunu ise bu sürümde de durur. Onu çözmek için ya sonuç 32 bit (DX:AX) olarak yazılmalı ya da taşma CF ile çağırana bildirilmelidir.

Sık yapılan hata

LOOP'lu bir döngüyü sayaç sıfır olabilecekken başlatmak. LOOP önce azaltır, sonra denetler; CX = 0 ile 65.536 kez döner. Sayacı bir değişkenden ya da parametreden alan her döngünün önüne `JCXZ` konur.

9.1.6.4 Kod izleme: düzeltilmiş POWER

Düzeltilmiş makronun `POWER taban, us, sonuc` açılımı, taban = 7 ve us = 2 için izlenir. Yazmaç korumanın etkisini görmek için çağıranın yazmaçlarında BX = 0AAAh, CX = 0BBBh, DX = 0CCCh olsun:

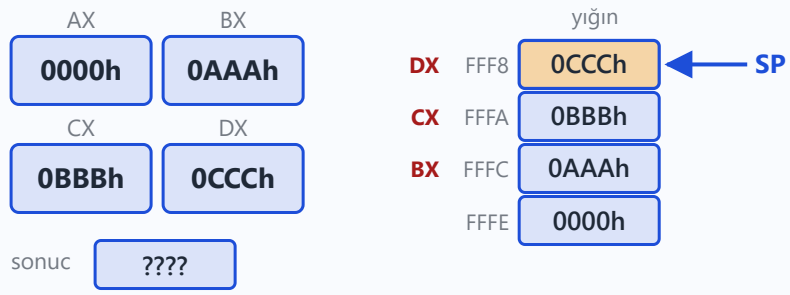
```
org 100h
push bx          ; POWER açılımı
push cx
push dx
mov  ax, 1
mov  cx, us
mov  bx, taban
jcxz ??0001      ; us = 0 → 1
??0000: mul bx
loop ??0000
??0001: mov sonuc, ax
pop  dx
pop  cx
pop  bx
ret
taban dw 7
us    dw 2
sonuc dw ?
```

Açılım 26 bayttır (011Ah'ye kadar); RET 011Ah'de, taban 011Bh'de, us 011Dh'de, sonuc 011Fh'dedir. LOCAL adları p_loop için ??0000, p_end için ??0001'dir.

Adım	Adres	Komut	AX	BX	CX	DX	SP	sonuc	Açıklama
1	0100h	`PUSH BX` POWER açılımı	0000h	0AAAh	0BBBh	0CCCh	FFFCh	0000h	BX saklandı
2	0101h	`PUSH CX` POWER açılımı	0000h	0AAAh	0BBBh	0CCCh	FFFAh	0000h	CX saklandı
3	0102h	`PUSH DX` POWER açılımı	0000h	0AAAh	0BBBh	0CCCh	FFF8h	0000h	DX saklandı; SP = FFF8h
4	0103h	`MOV AX, 1` POWER açılımı	0001h	0AAAh	0BBBh	0CCCh	FFF8h	0000h	geçici sonuç 1
5	0106h	`MOV CX, us` POWER açılımı	0001h	0AAAh	0002h	0CCCh	FFF8h	0000h	CX ← us = 2
6	010Ah	`MOV BX, taban` POWER açılımı	0001h	0007h	0002h	0CCCh	FFF8h	0000h	BX ← taban = 7

Adım	Adres	Komut	AX	BX	CX	DX	SP	sonuc	Açıklama
7	010Eh	`JCXZ ??0001` POWER açılımı	0001h	0007h	0002h	0CCCh	FFF8h	0000h	CX ≠ 0: atlama yok
8	0110h	`??0000: MUL BX` POWER açılımı	0007h	0007h	0002h	0000h	FFF8h	0000h	1 × 7; MUL DX'i de yazar
9	0112h	`LOOP ??0000` POWER açılımı	0007h	0007h	0001h	0000h	FFF8h	0000h	CX = 1
10	0110h	`??0000: MUL BX` POWER açılımı	0031h	0007h	0001h	0000h	FFF8h	0000h	7 × 7 = 49 = 0031h
11	0112h	`LOOP ??0000` POWER açılımı	0031h	0007h	0000h	0000h	FFF8h	0000h	CX = 0: döngü biter
12	0114h	`??0001: MOV sonuc, AX` POWER açılımı	0031h	0007h	0000h	0000h	FFF8h	0031h	[011Fh] ← 0031h
13	0117h	`POP DX` POWER açılımı	0031h	0007h	0000h	0CCCh	FFFAh	0031h	DX = 0CCCh
14	0118h	`POP CX` POWER açılımı	0031h	0007h	0BBBh	0CCCh	FFFCh	0031h	CX = 0BBBh
15	0119h	`POP BX` POWER açılımı	0031h	0AAAh	0BBBh	0CCCh	FFFEh	0031h	BX = 0AAAh; SP = FFFEh
16	011Ah	`ret` ana program	0031h	0AAAh	0BBBh	0CCCh	0000h	0031h	program biter

Tablo 9.1.16: Düzeltilmiş POWER makrosunun `POWER taban, us, sonuc` açılımı (taban = 7, us = 2); çağıranın yazmaçları BX = 0AAAh, CX = 0BBBh, DX = 0CCCh.



Şekil 9.1.12: Üç PUSH'tan sonra: çağıranın BX, CX ve DX değerleri yığında, SP = FFF8h.

Tablo 9.1.16'da ilk üç adım çağıranın yazmaçlarını yığına saklar (Şekil 9.1.12). Yığın aşağı doğru büyür: 0AAAh FFFCh'ye, 0BBBh FFFAh'ye, 0CCCh FFF8h'ye yazılır. Sonraki üç adımda yazmaçlar makronun kullanımına geçer: AX = 1, CX = 2, BX = 7. JCXZ, CX ≠ 0 olduğu için atlamaz. Birinci MUL 1 × 7 = 7 verir ve DX'i 0000h yapar. İkinci MUL 7 × 7 = 49 = 0031h verir. LOOP CX'i 0'a indirince döngü biter ve sonuc'a 0031h yazılır.

Son üç adım yığını ters sırada boşaltır. POP DX 0CCCh'yi, POP CX 0BBBh'yi, POP BX 0AAAh'yi geri getirir; SP yeniden FFFEh olur. Makrodan çıkıldığında BX, CX ve DX'in değerleri girişteki değerlerin aynısıdır. Yalnız AX değişmiştir ve AX zaten sonucu taşır. us = 0 ile çalıştırılsaydı JCXZ doğrudan ??0001'e atlar, sonuc = 1 olur ve tek bir MUL bile çalışmazdı.

9.1.7 Makro kütüphaneleri

9.1.7.1 INCLUDE ve emu8086.inc

Sık kullanılan makrolar her programa yeniden yazılmaz; ayrı bir dosyada toplanır. Bu dosyaya makro kütüphanesi (*library*) denir. `INCLUDE` direktifi, derleme anında kütüphane dosyasının metnini kaynağa, direktifin bulunduğu yere ekler. Assembler için bu, makroların programın başına elle yazılmasından farksızdır. Kullanılmayan makrolar kod üretmediği için büyük bir kütüphaneyi eklemek programı büyütmez; yalnız kullanılan makrolar açılır.

```
include 'emu8086.inc'
ORG 100h
    GOTOXY 10, 5
    PRINT 'Merhaba'
    PUTC  '!'
    RET
```

emu8086 ile gelen emu8086.inc dosyası bu düzenin hazır bir örneğidir. İçinde ekrana karakter ve metin yazan (PUTC, PRINT, PRINTN), imleci taşıyan (GOTOXY) ve gizleyen (CURSOROFF) makrolar bulunur. Sayı okuma ve yazdırma gibi uzun işler için kütüphane prosedür de sunar. Bu prosedürler `DEFINE_SCAN_NUM` gibi özel makrolarla programa bir kez eklenir ve CALL ile çağrılır. Böylece kütüphane, makro ile prosedürün birlikte kullanıldığı düzeni izler: kısa işler makro, uzun işler tek kopya prosedürdür. Kütüphanenin makroları ve prosedürleri ayrıntısıyla emu8086.inc kütüphanesini anlatan notta ele alınır.

9.1.7.2 Kod izleme: PRINT 'OK' açılımı

Kütüphanedeki PRINT makrosu, buraya kadarki teknikleri bir arada kullanır: yazmaç koruma, LOCAL etiketler ve makro içinde tanımlanan veri. `PRINT 'OK'` çağrısı şu koda açılır (LOCAL adları assembler'ın ürettiği biçimde):

```
org 100h
push ax          ; PRINT 'OK'
push si
jmp  ??0003      ; veriyi atla
??0001 db 'OK', 0
??0003: lea si, ??0001
??0000: mov al, cs:[si]
cmp  al, 0
jz   ??0002
inc  si
mov  ah, 0Eh
int  10h
jmp  ??0000
??0002: pop si
pop  ax
ret
```

Dizgi 'OK' ve sonundaki 0 baytı, makronun açıldığı yere, kodun içine gömülür: 0104h'den başlayarak 4Fh, 4Bh, 00h. İşlemci kod ile veriyi ayırt etmediği için bu üç baytı komut sanabilir. Bunu önlemek için makro önce `JMP` ile dizginin üzerinden atlar. Dizginin adı da LOCAL'dır (`??0001`); aynı programda ikinci bir PRINT kendi dizgisini başka bir adla tanımlar.

Dizgi kod kesiminde durduğu için onu okuyan komut CS: kesim önekini kullanır: `MOV AL, CS:[SI]`. Önek olmasaydı okuma varsayılan kesim olan DS'den yapılırdı. Bir .COM programında CS = DS olduğundan sonuç aynı çıkardı, ama makro DS'nin başka bir kesimi gösterdiği .EXE programlarında da çalışmak zorundadır. Önek, makine kodunda 2Eh baytıdır ve komutu bir bayt uzatır.

`LEA SI, ??0001` komutu emu8086'da `MOV SI, OFFSET ??0001` ile aynı 3 baytlık koda derlenir: `BE 04 01` . Gerçek MASM'da doğrudan adresli LEA 4 bayt (`8D 36` + adres) tutar ve açılım bir bayt uzun olurdu. Açılımın toplam boyutu 26 bayttır (0100h–0119h); bunun 3 baytı veridir. Genel olarak n karakterlik bir metin için PRINT açılımı 24 + n bayt tutar.

Adım	Adres	Komut	AL	SI	SP	ZF	Ekran	Açıklama
1	0100h	`push ax` PRINT 'OK' açılımı	00h	0000h	FFFCh	0	–	AX saklandı
2	0101h	`push si` PRINT 'OK' açılımı	00h	0000h	FFFAh	0	–	SI saklandı; SP = FFFAh
3	0102h	`jmp ??0003` PRINT 'OK' açılımı	00h	0000h	FFFAh	0	–	dizginin baytlarının üzerinden atla
4	0107h	`??0003: lea si, ??0001` PRINT 'OK' açılımı	00h	0104h	FFFAh	0	–	SI ← 0104h
5	010Ah	`??0000: mov al, cs:[si]` PRINT 'OK' açılımı	4Fh	0104h	FFFAh	0	–	AL ← CS:[0104h] = 'O'
6	010Dh	`cmp al, 0` PRINT 'OK' açılımı	4Fh	0104h	FFFAh	0	–	4Fh ≠ 0: ZF = 0
7	010Fh	`jz ??0002` PRINT 'OK' açılımı	4Fh	0104h	FFFAh	0	–	atlamaz

Adım	Adres	Komut	AL	SI	SP	ZF	Ekran	Açıklama
8	0111h	`inc si` PRINT 'OK' açılımı	4Fh	0105h	FFFAh	0	–	SI = 0105h
9	0112h	`mov ah, 0Eh` PRINT 'OK' açılımı	4Fh	0105h	FFFAh	0	–	
10	0114h	`int 10h` PRINT 'OK' açılımı	4Fh	0105h	FFFAh	0	`O`	ekrana O
11	0116h	`jmp ??0000` PRINT 'OK' açılımı	4Fh	0105h	FFFAh	0	`O`	döngü başına
12	010Ah	`??0000: mov al, cs:[si]` PRINT 'OK' açılımı	4Bh	0105h	FFFAh	0	`O`	AL = 'K'
15	0111h	`inc si` PRINT 'OK' açılımı	4Bh	0106h	FFFAh	0	`O`	SI = 0106h
17	0114h	`int 10h` PRINT 'OK' açılımı	4Bh	0106h	FFFAh	0	`OK`	ekranda OK
19	010Ah	`??0000: mov al, cs:[si]` PRINT 'OK' açılımı	00h	0106h	FFFAh	0	`OK`	AL = 00h
20	010Dh	`cmp al, 0` PRINT 'OK' açılımı	00h	0106h	FFFAh	1	`OK`	sonlandırıcı: ZF = 1
21	010Fh	`jz ??0002` PRINT 'OK' açılımı	00h	0106h	FFFAh	1	`OK`	çıkışa atla
22	0118h	`??0002: pop si` PRINT 'OK' açılımı	00h	0000h	FFFCh	1	`OK`	SI = 0000h
23	0119h	`pop ax` PRINT 'OK' açılımı	00h	0000h	FFFEh	1	`OK`	AX geri; SP = FFFEh
24	011Ah	`ret` ana program	00h	0000h	0000h	1	`OK`	program biter

Tablo 9.1.17: `PRINT 'OK'` açılımının izlemesi; dizgi 0104h'de, kodun içinde.



Şekil 9.1.13: Kodun içine gömülü dizgi: 0104h'den başlayarak 'O', 'K' ve sonlandırıcı 0. JMP bu baytların üzerinden atlar.

Tablo 9.1.17 döngünün üç turunu gösterir; tekrar eden MOV AH ve INT adımları kısaltılmıştır. Birinci turda AL = 4Fh ('O') okunur. `CMP AL, 0` ZF = 0 verir ve JZ atlamaz. SI bir ilerler, INT 10h 'O'yu yazar ve JMP döngü başına döner. İkinci tur 'K'yı yazar. Üçüncü turda AL = 00h okunur; CMP ZF = 1 verir ve JZ çıkışa (??0002) atlar. Sondaki 0 baytı, dizginin bittiğini gösteren sonlandırıcıdır. İki POP, SI'yı ve AX'i eski değerlerine döndürür; SP yeniden FFFEh olur. Ekrana OK yazılmıştır ve çağıranın yazmaçları bozulmamıştır.

Bu açılım makronun gücünü ve maliyetini birlikte gösterir. Kaynakta tek satır olan `PRINT 'OK'`, 26 bayt üretmiştir. Program boyunca yirmi farklı metin PRINT ile yazılırsa her biri kendi döngüsünü taşır. Aynı iş, dizgileri ayrı bir yerde tutan ve tek bir yazdırma prosedürü çağıran bir düzenle çok daha az yer tutar. Kütüphanenin PRINT_STRING gibi prosedürleri tam bu amaçla vardır.

9.1.8 Makro işleçleri

Makro genişletme metin işlemi olduğuna göre, metni parçalamak ve birleştirmek için özel işaretlere ihtiyaç vardır. MASM ve TASM bu amaçla beş makro işleci tanımlar. Bu işleçler yalnız makro tanımı ve çağrısı içinde anlam taşır.

İşleç	Görevi	Örnek ve açılım
`&`	parametreyi bitişik metne ekler	`SIFIRLA MACRO r` · `XOR r&X, r&X` → `SIFIRLA B`: `XOR BX, BX`
`< >`	virgül ya da boşluk içeren metni tek parametre yapar	`YAZ <'A', 'B'>` → tek parametre: `'A', 'B'`
`!`	sonraki karakteri sıradan karakter yapar	`<5 !> 3>` → metin: `5 > 3`
`%`	ifadeyi hesaplar, sonucu metin olarak koyar	`GOSTER %(4*8)` → parametre `32`
`;;`	makro açıklaması; açılıma kopyalanmaz	`;; yalnız tanımda görünür`

Tablo 9.1.18: Makro işleçleri.

`&` (birleştirme) işleci en sık kullanılanıdır. Assembler gövdede parametre adlarını bütün sözcükler olarak arar. `XOR rX, rX` yazılsaydı "rX" tek bir sözcük sayılır ve r parametresiyle eşleşmezdi; açılım `XOR rX, rX` olarak kalır ve bilinmeyen bir ad hatası verirdi. `r&X` yazımı, r'nin bir parametre adı olduğunu ve ardından gelen X'in ona yapıştırılacağını belirtir. `SIFIRLA B` çağrısında r = B olur ve açılım `XOR BX, BX` olur. Bir yazmacı kendisiyle XOR'lamak onu sıfırlar ve 2 baytlık, 3 turluk bir komuttur; `MOV BX, 0` ise 3 bayt tutar ve 4 tur sürer.

Açı ayraçları `< >`, virgül içeren bir metni tek parametre olarak geçirmeyi sağlar. Ayraçlar olmadan `YAZ 'A', 'B'` iki ayrı parametre sayılırdı. `!` işleci, kendisinden sonraki karakterin özel anlamını kaldırır; `<5 !> 3>` içindeki ilk `>` ayracı kapatmaz, metnin bir parçası olur. `%` işleci ise parametreyi metin olarak değil, değer olarak geçirir: `%(4*8)` önce 32'ye hesaplanır, makroya "32" metni gider. Bu, işleç önceliği tuzağına karşı paranteze alternatif bir korumadır.

Son işleç açıklamalarla ilgilidir. Tek `;` ile yazılan açıklama gövdenin bir parçasıdır ve her açılımda liste dosyasına yeniden yazılır. Çift `;;` ile yazılan açıklama yalnız tanımda kalır, açılıma kopyalanmaz. Yüz kez kullanılan bir makroda bu, liste dosyasını yüzlerce gereksiz satırdan kurtarır.

9.1.9 Tekrar blokları

9.1.9.1 REPT, IRP ve IRPC

Bazen bir kod parçasının, makro tanımlamaya gerek kalmadan, olduğu yerde birkaç kez yazılması istenir. Tekrar blokları bu işi yapar.

> **Tanım 9.1.5**
>
> **Tekrar bloğu** (*repeat block*), adı olmayan ve tanımlandığı yerde hemen açılan bir makrodur. `REPT n` gövdeyi n kez, `IRP p, <liste>` listedeki her öğe için bir kez (p o öğeyle değiştirilerek), `IRPC p, metin` metindeki her karakter için bir kez yazar. Blok `ENDM` ile biter. MASM 6'da IRP ve IRPC, FOR ve FORC adlarıyla da yazılır.

```
REPT 3
  SHL AX, 1
ENDM                ; → SHL AX, 1 · SHL AX, 1 · SHL AX, 1

IRP r, <AX,BX,CX>
  PUSH r
ENDM                ; → PUSH AX · PUSH BX · PUSH CX

IRPC c, OK
  PUTC '&c'
ENDM                ; → PUTC 'O' · PUTC 'K'
```

REPT örneği AX'i üç kez bir bit sola kaydırır, yani 8 ile çarpar: AX = 5 iken sonuç 40 (0028h) olur. IRP örneği üç yazmacı yığına iter; SAKLA makrosunun kısa bir yazımıdır. IRPC örneğinde c, sırayla O ve K karakterleri olur. Tırnak içinde c'nin parametre olarak tanınması için `&c` yazılır; tırnak içindeki metin aksi hâlde olduğu gibi kalırdı. Açılımdaki iki PUTC de birer makro olduğu için iç içe genişletme olur ve ekranda OK yazılır.

Tekrar bloğu bir döngü değildir. Döngü sayacına gerek kalmaz ama kod büyür: `REPT 100` gövdeyi yüz kez kopyalar. REPT örneğinin üç SHL'si 6 bayt tutar ve 3 × 2 = 6 turda çalışır. Aynı kaydırma `MOV CL, 3` ve `SHL AX, CL` ile 4 baytta yazılabilir. Ama 8086'da CL'li kaydırma her bit için 4 tur ekler ve iş 4 + (8 + 4 × 3) = 24 tur sürer. Kısa tekrarlarda REPT hem hızlı hem de makul boyuttadır; `REPT 100` ise 200 bayt tutar ve aynı iş sayaçlı bir LOOP ile tek kopyada yapılır.

Tekrar bloklarının en yaygın kullanımı derleme anında tablo üretmektir. Bunun için değeri değiştirilebilen bir assembler sembolü gerekir. `EQU` ile tanımlanan bir sabit yeniden tanımlanamaz. `=` ile tanımlanan sembol ise her atamada yeni bir değer alabilir. Aşağıdaki blok 0'dan 7'ye kadar sayıların karelerini üretir:

```
n = 0
kare LABEL BYTE
REPT 8
    DB n*n
    n = n + 1
ENDM
```

`kare LABEL BYTE`, o anki adrese bayt türünde bir ad verir ama yer ayırmaz. REPT gövdesi sekiz kez açılır. Her açılımda `DB n*n` o anki n'nin karesini bir bayt olarak yazar ve `n = n + 1` sembolü bir artırır. Sonuçta bellekte 0, 1, 4, 9, 16, 25, 36, 49 baytları art arda durur. n bir değişken değildir; bellekte yeri yoktur. Yalnız derleme anında, assembler'ın tablosunda yaşayan bir sayıdır. Aynı yöntemle maskeler, 2'nin kuvvetleri ya da önceden hesaplanmış sinüs değerleri gibi tablolar üretilir.

9.1.9.2 Kod izleme: REPT ile kare tablosu

Üretilen tablo, XLAT komutuyla bir sayının karesini çarpma yapmadan bulmak için kullanılabilir:

```
org 100h
lea  bx, kare
mov  al, 5
xlat            ; AL = kare[5]
mov  dl, al
mov  al, 7
xlat            ; AL = kare[7]
add  al, dl
ret
kare db 0, 1, 4, 9, 16, 25, 36, 49   ; REPT bloğunun ürettiği baytlar
```

`XLAT` (tablodan çevir), AL'yi BX'in gösterdiği tablonun indisi olarak kullanır: AL ← DS:[BX + AL]. Tablonun adresi önce BX'e yüklenmelidir. Kodun boyutu 3 + 2 + 1 + 2 + 2 + 1 + 2 + 1 = 14 bayttır; LEA emu8086'da 3 bayt tuttuğu için tablo 0100h + 0Eh = 010Eh'de başlar.

Adım	Adres	Komut	BX	AL	DL	Açıklama
1	0100h	`lea bx, kare` ana program	010Eh	00h	00h	BX ← 010Eh (kare'nin bağıl konumu)
2	0103h	`mov al, 5` ana program	010Eh	05h	00h	aranan sayı ve indis 5
3	0105h	`xlat` ana program	010Eh	19h	00h	AL ← [BX + AL] = [0113h] = 25
4	0106h	`mov dl, al` ana program	010Eh	19h	19h	ilk sonuç DL'de
5	0108h	`mov al, 7` ana program	010Eh	07h	19h	indis 7
6	010Ah	`xlat` ana program	010Eh	31h	19h	AL ← [0115h] = 49

Adım	Adres	Komut	BX	AL	DL	Açıklama
7	010Bh	`add al, dl` ana program	010Eh	4Ah	19h	25 + 49 = 74 = 4Ah
8	010Dh	`ret` ana program	010Eh	4Ah	19h	program biter

Tablo 9.1.19: REPT ile üretilen kare tablosundan XLAT ile okuma; tablo 010Eh'de başlar.



Şekil 9.1.14: Birinci XLAT: AL = 5 indisiyle kare + 5 = 0113h adresindeki 25 (19h) okunur.

Tablo 9.1.19'da birinci XLAT, 010Eh + 5 = 0113h adresindeki baytı okur: 25 = 19h (Şekil 9.1.14). Bu değer DL'ye saklanır. İkinci XLAT 010Eh + 7 = 0115h adresinden 49 = 31h okur. ADD AL, DL ikisini toplar: 25 + 49 = 74 = 4Ah, yani $5^2 + 7^2$. Program boyunca hiçbir çarpma komutu çalışmamıştır.

Hız farkı belirgindir. XLAT 11 turda biter. 8086'da 8 bitlik bir MUL ise 70–77 tur sürer. Tablonun bedeli bellektir: sekiz bayt. Bu, bellek ile zaman arasındaki klasik bir takastır ve REPT bu takası kaynakta tek bir satır gibi kısa tutar. Tablo derleme anında üretildiği için çalışma anında onu hesaplamak için de zaman harcanmaz.

9.1.10 Koşullu derleme

9.1.10.1 IF ailesi

Koşullu derleme, kaynak kodun bir bölümünün derleme anında programa alınıp alınmayacağını seçer. C dilindeki `#if` ve `#ifdef` önişlemci direktiflerinin assembly karşılığıdır.

Tanım 9.1.6

Koşullu derleme (*conditional assembly*), `IF`, `IFB`, `ELSE`, `ENDIF` gibi direktiflerle bir kaynak bölümünün derleme anında koda katılıp katılmayacağının seçilmesidir. Koşul derleme anında değerlendirilir; seçilmeyen dal makine koduna hiç girmez.

Direktif	Dal ne zaman alınır?	Tipik kullanım
`IF ifade`	ifade ≠ 0	deneme sürümü, seçenekler
`IFE ifade`	ifade = 0	IF'in tersi
`IFB <p>`	parametre boş	isteğe bağlı parametre
`IFNB <p>`	parametre dolu	isteğe bağlı parametre
`IFIDN <a>,<b>`	iki metin aynı	yazmaç adını denetleme
`IFDIF <a>,<b>`	iki metin farklı	yazmaç adını denetleme

Tablo 9.1.20: Koşullu derleme direktifleri.

Her koşullu blok `ENDIF` ile biter ve isteğe bağlı bir `ELSE` dalı taşıyabilir. Koşul, çalışma anındaki bir bayrağa ya da yazmaca bakmaz. Assembler'ın derleme anında bildiği şeylere bakar: sembollerin değerleri ve makro parametrelerinin metni. Bu yüzden koşullu derleme çalışma anında hiçbir komut, karşılaştırma ya da atlama üretmez.

İlk kullanım alanı programın farklı sürümlerini aynı kaynaktan üretmektir:

```
HATA_AYIKLA = 1
IF HATA_AYIKLA
  PUTC '*'          ; yalnız deneme sürümünde
ENDIF
```

HATA_AYIKLA = 1 iken PUTC açılır ve program 8 bayt büyür; ekrana önce bir yıldız yazılır. Sembol 0 yapıldığında bu satırlar hiç derlenmez ve programda iz bırakmaz. Deneme amaçlı ekran çıktıları ya da denetimler böylece kaynaktan silinmeden son sürümden çıkarılır.

İkinci kullanım alanı makroların içidir. `IFB` ve `IFNB` (boş, boş değil) isteğe bağlı parametreleri mümkün kılar. Aşağıdaki KAYDIR makrosu bir yazmacı sola kaydırır; kaydırma sayısı verilmezse 1 kabul edilir:

```
KAYDIR MACRO r, n
  IFB <n>           ;; n verilmedi mi?
    SHL r, 1
  ELSE
    MOV CL, n
    SHL r, CL
  ENDIF
ENDM
```

8086'da kaydırma sayısı ya 1'dir ya da CL'de verilir. `SHL AX, 3` gibi bir anında sayı 80186 ile gelmiştir. Makro bu kuralı kullanıcıdan gizler: n boşsa tek bitlik kaydırma, doluysa CL'li kaydırma üretir. Parametrenin açı ayraçları içinde yazılması (`<n>`), boş metnin de bir metin olarak karşılaştırılabilmesini sağlar.

`IFIDN` ve `IFDIF`, parametre tuzaklarına karşı bir güvenlik ağı kurar. KAYDIR CL'yi bozar; `KAYDIR CX, 2` çağrısı bu yüzden yanlış sonuç verir. `MOV CL, 2` CX'in düşük baytını ezer ve kaydırılan değer artık programcının CX'i değildir. Gövdeye `IFIDN <r>, <CX>` ile başlayan bir blok eklenip bu durumda bir hata iletisi üretilebilir. `EXITM` direktifi ise açılımı o noktada bitirir; kalan gövde satırları yazılmaz. Koşullu derlemeyle birlikte kullanıldığında özyinelemeli bir makronun durma koşulunu da sağlar.

9.1.10.2 Kod izleme: KAYDIR

```
org 100h
mov ax, 5
mov bx, 3
KAYDIR AX          ; → shl ax, 1
KAYDIR BX, 3       ; → mov cl, 3 · shl bx, cl
ret
```

Adım	Adres	Komut	AX	BX	CL	CF	Açıklama
1	0100h	`mov ax, 5` ana program	0005h	0000h	00h	0	AX = 5
2	0103h	`mov bx, 3` ana program	0005h	0003h	00h	0	BX = 3
3	0106h	`SHL AX, 1` KAYDIR açılımı	000Ah	0003h	00h	0	n boş, IFB dalı: AX = 5 × 2
4	0108h	`MOV CL, 3` KAYDIR açılımı	000Ah	0003h	03h	0	n = 3, ELSE dalı: sayı CL'ye
5	010Ah	`SHL BX, CL` KAYDIR açılımı	000Ah	0018h	03h	0	$3 \times 2^3 = 24 = 0018h$
6	010Ch	`ret` ana program	000Ah	0018h	03h	0	program biter

Tablo 9.1.21: `KAYDIR AX` ve `KAYDIR BX, 3` çağrılarının açılmış kodu.

Birinci çağrıda n boştur; assembler IFB dalını seçer ve tek bir `SHL AX, 1` yazar (2 bayt, `D1 E0`). AX = 5 × 2 = 000Ah olur. İkinci çağrıda n = 3'tür; ELSE dalı seçilir ve iki komut yazılır: `MOV CL, 3` (`B1 03`) ve `SHL BX, CL` (`D3 E3`), toplam 4 bayt. BX = $3 \times 2^3 = 24 = 0018h$ olur. Aynı makro iki farklı uzunlukta kod üretmiştir. Birinci açılım CL'ye dokunmaz; yalnız ikinci açılım CL'yi değiştirir.

İzlemede bir karşılaştırma ya da atlama yoktur. Hangi dalın yazılacağına assembler karar vermiş, işlemci yalnız sonucu yürütmüştür. Aynı seçim çalışma anında yapılsaydı her çağrıda bir CMP ve bir koşullu atlama gerekirdi; hem kod büyür hem de önden getirme kuyruğu boşaltılırdı.

9.1.11 Hata ayıklama ve sık yapılan hatalar

Makrolar kaynakta kodu kısaltır ama hata ayıklamayı zorlaştırır, çünkü yürüyen kod kaynakta yazılı değildir. Dört alışkanlık bu zorluğu azaltır.

Açılımı görmek. emu8086'nın öykünücü penceresindeki komut listesi makro adlarını değil, açılmış komutları gösterir; tek adım çalıştırma da bu komutlar üzerinde ilerler. Makro etiketleri LOCAL ile `??0000` gibi adlar aldığı için atlama hedefleri kaynaktaki adlarla görünmez. Hangi kopyanın çalıştığını bu numaralar gösterir.

Liste dosyası. MASM'da `.LISTMACROALL` açılımın bütün satırlarını liste dosyasına yazar. `.LISTMACRO` yalnız kod üreten satırları yazar, `.NOLISTMACRO` açılımı tamamen gizler. Liste dosyası her satırın adresini ve makine kodunu yan yana verdiği için bayt hesabı ve adres denetimi için en güvenilir kaynaktır.

Hata satırını doğru okumak. Gövdedeki bir hata, makronun kullanıldığı her satırda ayrı ayrı bildirilir. Aynı hata iletisi birkaç kez görülüyorsa neden çoğunlukla gövdededir.

Önce elle açmak. Yeni bir makro, gövdesi önce düz kod olarak yazılıp denenerek geliştirilir. Kod bir parametre değeriyle doğru çalıştıktan sonra parametreli makroya dönüştürülür. Sonra farklı türde parametrelerle (yazmaç, değişken, sabit, ifade) ayrı ayrı denenir.

Hata	Sonuç	Çözüm
kullanımdan sonra tanımlamak	çağrı bilinmeyen komut sayılır	tanımlar dosyanın başına ya da INCLUDE dosyasına
ENDM'i unutmak	dosyanın geri kalanı gövde sayılır; hata çok aşağıda görünür	her MACRO'nun ENDM'ini denetlemek
etiketi LOCAL yapmamak	ikinci kullanımda *Duplicate declaration*	`LOCAL` gövdenin ilk satırı
gövdede RET	RET çağıranın dönüş adresini ya da rastgele bir sözcüğü alır	makroda RET yazmamak
gövdenin bozduğu yazmacı parametre vermek	`TOPLA AX, BX, CX` gibi çağrılarda parametre okunmadan değişir	kullanılan yazmaçları belgelemek, IFIDN ile denetlemek
parantezsiz ifade	işleç önceliği değişir	`(en)*(boy)`
makroyu döngü sanmak	`REPT 1000` 1000 kopya üretir	çalışma anında tekrar için LOOP
uzun gövdeyi çok kullanmak	kod boyutu kullanım sayısıyla büyür	işi prosedüre taşımak, makroyu kılıf yapmak

Tablo 9.1.22: Sık yapılan makro hataları ve sonuçları.

Tablo 9.1.22 sık görülen sekiz hatayı toplar. Hepsinin ortak kökü aynıdır: makro bir metin kalıbıdır, çağrılan bir birim değildir. Bir makronun ne yapacağını anlamanın en güvenilir yolu, onu açıp ortaya çıkan komutları sıradan bir program gibi okumaktır.

9.1.12 Alıştırmalar ve çözümleri

Alıştırma 1. `MyMacro AX, BX, CX` çağrısının genişletilmiş hâli nedir? Hangi satır gereksizdir?

Çözüm. p1 = AX, p2 = BX, p3 = CX olur. Açılım `MOV AX, AX`, `MOV BX, BX`, `MOV CX, CX` satırlarıdır. Üçü de geçerli komutlardır ve 2'şer bayta derlenir (`8B C0`, `8B DB`, `8B C9`). Ama hiçbiri bir yazmacı değiştirmez; MOV bayrakları da etkilemez. Üç satırın üçü de gereksizdir: 6 bayt ve 6 tur boşa gider. AX =

1, BX = 2, CX = 3 ile başlayan bir program makrodan sonra da aynı değerleri taşır. Makro hatasız derlenir ama hiçbir iş yapmaz; assembler bu tür etkisiz açılımlar için uyarı vermez.

Alıştırma 2. İki sözcük değişkenini toplayıp üçüncüsüne yazan `TOPLA a, b, c` makrosunu yazın.

Çözüm.

```
TOPLA MACRO a, b, c
    MOV AX, a
    ADD AX, b
    MOV c, AX
ENDM
```

8086'da bellekten belleğe toplama olmadığı için AX ara yazmaçtır. Makro AX'i değiştirir; bu, makronun belgesinde belirtilmelidir. s1 = 1234h, s2 = F00Fh ile `TOPLA s1, s2, s3` çağrısı s3 = 0243h ve CF = 1 verir; işaretsiz toplamın 17. biti eldededir.

Alıştırma 3. PUTC makrosu 50 kez kullanılırsa, aynı işi yapan prosedür 50 kez çağrılırsa kod boyutları ne olur?

Çözüm. Makro: 50 × 8 = 400 bayt. Prosedür: gövde 8 bayt + RET 1 bayt = 9 bayt bir kez, artı 50 × 3 = 150 bayt CALL; toplam 159 bayt. Süre açısından prosedür 50 × 27 = 1350 tur ek getirir.

Alıştırma 4. İlk POWER makrosu `POWER base, exponent, result` ve ikinci bir çağrıyla iki kez kullanıldığında hangi hata alınır? Nasıl düzeltilir?

Çözüm. İkinci açılım `powerloop:` etiketini yeniden tanımlar; assembler *Duplicate declaration* (powerloop) hatası verir ve program derlenmez. Düzeltme, makronun ilk satırına `LOCAL powerloop` eklemektir. Böylece birinci açılımda etiket ??0000, ikincide ??0001 olur ve her LOOP kendi kopyasının etiketine döner.

Alıştırma 5. exponent = 0 iken ilk POWER makrosu kaç kez MUL çalıştırır?

Çözüm. 65.536 kez. MUL döngünün başında olduğu için önce bir kez çalışır. LOOP CX'i 0'dan FFFFh'ye indirir ve atlar. Döngü CX yeniden 0 olana kadar sürer: CX = 0 → FFFFh → … → 1 → 0. base = 2 ise result = 0000h, base = 1 ise result = 0001h çıkar. JCXZ ile sayaç önceden denetlenirse hiç MUL çalışmaz ve result = 1 olur.

Alıştırma 6. Bir makronun AX dışında hiçbir yazmacı bozmaması için hangi komutlar eklenmelidir?

Çözüm. Gövdenin başına, makronun değiştirdiği her yazmaç (AX hariç) için bir PUSH; sonuna aynı yazmaçlar için ters sırada POP eklenir. Düzeltilmiş POWER'da bu `PUSH BX`, `PUSH CX`, `PUSH DX` ve `POP DX`, `POP CX`, `POP BX`'tir. MUL gibi örtük olarak yazmaç değiştiren komutlar (MUL DX'i, LOOP CX'i, INT hizmetleri dönüş yazmaçlarını değiştirir) özellikle dikkate alınmalıdır. Bayrakların da korunması gerekiyorsa PUSHF ve POPF kullanılır.

9.1.13 Özet

- Makro, `ad MACRO p1, p2 … ENDM` ile tanımlanan bir kod kalıbıdır. Assembler, makronun kullanıldığı her satırın yerine gövdeyi yazar; buna genişletme denir. İşlemci makroyu hiç görmez.
- Makro, kullanıldığı satırdan önce tanımlanmalıdır. Kullanılmayan makro kod üretmez.
- Parametreler değer değil metindir. Değerleri, açılmış satır yürütüldüğünde okunur. Gövdenin değiştirdiği yazmaç parametre olarak verilmemelidir; ifade parametreleri gövdede paranteze alınmalıdır.
- Hangi çağrının geçerli olduğuna açılımda oluşan komut karar verir; hata kullanım satırında bildirilir ama nedeni gövdededir.
- Boyut: makro k · n bayt, prosedür (n + 1) + 3k bayt. Gövde 3 bayttan uzunsa prosedür, başa baş noktasından sonra daha küçüktür.
- Hız: her prosedür çağrısı CALL + RET ile 19 + 8 = 27 tur, iki yığın erişimi ve iki kuyruk boşaltma ekler. Makroda bu maliyet yoktur.
- Makroda RET yazılmaz; RET çağıranın dönüş adresini ya da yığında ne varsa onu alır.
- Etiketli makrolar LOCAL ile yazılır; assembler her açılımda ??0000, ??0001 … gibi benzersiz adlar üretir.
- Makronun değiştirdiği yazmaçlar PUSH ile saklanıp ters sırada POP ile geri yüklenir (PUTC, düzeltilmiş POWER).
- LOOP CX = 0 ile 65.536 kez döner; sayacı değişken olan döngülerin önüne JCXZ konur. MUL, DX'i de değiştirir.
- INCLUDE kütüphane metnini kaynağa ekler; emu8086.inc kısa işleri makro, uzun işleri prosedür olarak sunar.
- &, <>, !, % ve ;; makro işleçleri metni birleştirir, gruplar ve hesaplar.
- REPT, IRP ve IRPC adsız, yerinde açılan tekrar bloklarıdır; `=` sembolüyle derleme anında tablo üretirler.
- IF, IFE, IFB, IFNB, IFIDN, IFDIF koşulu derleme anında değerlendirir; seçilmeyen dal makine koduna girmez.

9.1.14 Kavram listesi

makro *macro* · makro genişletme *macro expansion* · LOCAL direktifi *LOCAL directive* · koşullu derleme *conditional assembly* · tekrar bloğu (REPT / IRP / IRPC) *repeat block* · direktif *directive* · assembler *assembler* · prosedür *procedure* · prosedür çağrısı *procedure call* · dönüş adresi *return address* · yazmaçların korunması *register preservation* · etiket *label* · kütüphane *library* · saat turu *clock cycle* · önden getirme kuyruğu *prefetch queue* · yol döngüsü *bus cycle* · liste çıktısı *listing file*

öykünücü *emulator*

9.1.15 Kendini Dene

1. TOPLA makrosu (`MOV AX, x` · `ADD AX, y` · `MOV z, AX`) AX = 5, BX = 7 iken `TOPLA BX, AX, CX` ile çağrılıyor. CX ne olur?

   Cevap

   Açılım `MOV AX, BX` · `ADD AX, AX` · `MOV CX, AX` 'tir. Birinci satır AX'i 7 yapar; y = AX metni artık 7'yi gösterir. ADD 7 + 7 = 14 verir: CX = **000Eh**, beklenen 12 değil.

2. `ALAN MACRO en, boy` / `MOV AX, en*boy` / `ENDM` makrosunun `ALAN 10-2, 3` çağrısı AX'e ne yükler? Parantezli sürüm ne yükler?

   Cevap

   Açılım `MOV AX, 10-2*3` 'tür; çarpma önce yapılır: 10 − 6 = **4**. Parantezli `(10-2)*(3)` = **24** (0018h) verir. İki durumda da çarpma derleme anında yapılır ve makine kodunda yalnız bir MOV vardır.

3. Gövdesi 4 bayt olan bir iş kaç kullanımdan itibaren prosedürle daha az yer tutar?

   Cevap

   Makro 4k, prosedür 5 + 3k bayt. 4k > 5 + 3k için k > 5. Beş kullanımda ikisi de 20 bayttır; **altı** kullanımdan itibaren prosedür küçüktür (24'e karşı 23 bayt).

4. Etiket içeren ama LOCAL kullanmayan bir makro programda yalnız bir kez kullanılırsa derleme hatası olur mu?

   Cevap

   Hayır. Tek açılımda etiket bir kez tanımlanır ve program derlenir. Hata, makro ikinci kez kullanıldığı anda *Duplicate declaration* olarak çıkar. Bu yüzden makro bir kez kullanılacak olsa da LOCAL yazılır.

5. MyMacro2 (AX = 2 ise label1'e, AX = 3 ise label2'ye atlar; label1: INC AX, label2: ADD AX, 2) AX = 0 ile çalıştırılırsa AX ne olur?

   Cevap

   İki JE de atlamaz ve akış label1'e düşer. INC AX 1'i, ADD AX, 2 ise **3**'ü verir.

6. KAYDIR makrosu (n doluysa `MOV CL, n` · `SHL r, CL`) CX = 5 iken `KAYDIR CX, 2` ile çağrılıyor. Programcı 5 × 4 = 20 bekliyor. CX ne olur?

   Cevap

   `MOV CL, 2`, CX'in düşük baytını ezer: CX = 0002h. `SHL CX, CL` 2'yi 2 bit kaydırır: CX = **0008h**. Gövdenin bozduğu yazmaç (CL, dolayısıyla CX) parametre olarak verilmiştir; IFIDN ile bu çağrı yakalanabilir.

7. `n = 1` · `tablo LABEL BYTE` · `REPT 4` · `DB n` · `n = n * 2` · `ENDM` bloğu belleğe hangi baytları yazar? Bu blok çalışma anında kaç komut yürütür?

   Cevap

   01h, 02h, 04h, 08h. Blok yalnız veri üretir; n derleme anında yaşayan bir semboldür. Çalışma anında **hiç** komut yürütülmez.

8. 5 MHz'lik bir 8086'da bir prosedür 1000 kez çağrılıyor. CALL ve RET'in getirdiği ek süre ne kadardır?

   Cevap

   1000 × (19 + 8) = 27.000 tur. Bir tur 200 ns olduğundan 27.000 × 200 ns = **5,4 ms**. Makro kullanılsaydı bu süre olmazdı ama kod 1000 kopya kadar büyürdü.

9. emu8086.inc'deki PRINT makrosu `PRINT 'Merhaba'` için kaç bayt üretir? Dizgi neden kodun içinde olduğu hâlde işlemci onu komut olarak yürütmez?

   Cevap

   'OK' için 26 bayt; her ek karakter bir bayt ekler. 7 karakterlik metin için 24 + 7 = **31 bayt**. Makro dizginin önüne bir JMP koyar ve işlemci dizginin baytlarının üzerinden atlar; dizgiye yalnız `MOV AL, CS:[SI]` ile veri olarak erişilir.

9.1.16 İleri okuma

- Microsoft, *MASM Programmer's Guide*: makro tanımı, LOCAL, makro işleçleri, tekrar blokları ve koşullu derleme direktiflerinin başvuru anlatımı.
- K. R. Irvine, *Assembly Language for x86 Processors*, Bölüm 10: makroları, koşullu derlemeyi ve tekrar bloklarını çok sayıda örnekle işler.
- emu8086 yardım belgeleri, *8086 Assembler Tutorial, Part 10 (Macros)*: emu8086'daki makro yazımı ve emu8086.inc kütüphanesi.

- B. B. Brey, *The Intel Microprocessors*: makro ve prosedür seçimini 8086 komut süreleri ve donanım yapısıyla birlikte ele alır.

9.1.17 Kaynaklar

1. emu8086 yardım belgeleri: *8086 Assembler Tutorial, Part 10 (Macros)* ve `emu8086.inc`.
2. Microsoft, *MASM Programmer's Guide* — MACRO, ENDM, LOCAL direktifleri.
3. K. R. Irvine, *Assembly Language for x86 Processors*, 7. baskı, Pearson, 2014 — Bölüm 10 (Makrolar).
4. B. B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson, 2009.
5. Intel, *8086 16-Bit HMOS Microprocessor* veri sayfası (komut süreleri).

KISIM III · BÖLÜM 9.2

emu8086.inc Kütüphanesi

Assembly ile yazılan hemen her program ekrana bir şey yazar, klavyeden bir şey okur ya da bir sayıyı onluk tabanda gösterir. Bu işlerin her biri birkaç BIOS ve DOS hizmetinin doğru sırayla çağrılmasını gerektirir. emu8086, sık kullanılan bu işleri emu8086.inc adlı hazır bir kütüphanede toplar. Not, kütüphanenin INCLUDE direktifiyle programa nasıl katıldığını ve programın boyutunu neyin belirlediğini anlatarak başlar. Ardından ekran makrolarını (PUTC, GOTOXY, PRINT, PRINTN, CURSOROFF, CURSORON) açılımlarıyla birlikte ele alır. Sonra DEFINE_ ve CALL ile kullanılan prosedürlere geçer: dizgi yazan PRINT_STRING ve PTHIS, klavyeden dizgi okuyan GET_STRING, ekranı temizleyen CLEAR_SCREEN, sayı okuyan SCAN_NUM ve sayı yazan PRINT_NUM ile PRINT_NUM_UNS. Her prosedürün mantığı sade bir eşdeğer kodla yazmaç yazmaç izlenir. Son kesimler bütün parçaları birleştiren örnek programlara ve kütüphane kullanılırken en sık yapılan hatalara ayrılmıştır.

ÖĞRENME ÇIKTILARI

INCLUDE direktifini açıklamak
Bir kütüphane dosyasının kaynağa derleme sırasında metin olarak eklendiğini, assembler'ın dosyayı hangi klasörlerde aradığını ve program boyutunu yalnız kullanılan makroların ve tanımlanan prosedürlerin belirlediğini açıklamak.

Ekran makrolarını kullanmak
PUTC, GOTOXY, PRINT, PRINTN, CURSOROFF ve CURSORON makrolarının hangi BIOS hizmetine ve hangi komut dizisine genişlediğini göstermek; imleç konumunu, satır sonunu ve ekran kaymasını önceden hesaplamak.

Prosedürleri doğru tanımlayıp çağırmak
DEFINE_ makrolarıyla prosedürleri programa eklemek, CALL ile çağırmak, her prosedürün girdi ve çıktı yazmaçlarını bilmek ve hangi yazmaçların korunduğunu söylemek.

Dizgi ve sayı prosedürlerini izlemek
PRINT_STRING, PTHIS, GET_STRING, SCAN_NUM, PRINT_NUM ve PRINT_NUM_UNS'nin mantığını yazmaç, yığın ve tampon düzeyinde adım adım izlemek; işaretli ve işaretsiz yorum farkını açıklamak.

Örnek programlar yazmak
Kütüphane makro ve prosedürleriyle sayı okuyan, hesaplayan, sonucu yazan ve ekranı düzenleyen programlar yazmak; sık yapılan hataları tanımak.

9.2.1 Kütüphane ve INCLUDE

9.2.1.1 Ortak işlevler kütüphanesi

Ekrana tek bir karakter yazmak bile 8086'da birkaç komut ister. Karakter AL'ye, hizmet numarası AH'ye konur ve BIOS'un ekran kesmesi `INT 10h` çağrılır. Bu sırada AX'in eski değeri bozulur; değer gerekiyorsa önce yığına itilip sonra geri alınmalıdır. Bir sayıyı onluk tabanda yazmak daha da uzundur: sayı 10'a bölünür, kalanlar basamak olur, basamaklar ASCII koduna çevrilir ve doğru sırayla yazılır. Klavyeden çok basamaklı bir sayı okumak da aynı işin tersidir. Bu kalıplar hemen her programda yeniden gerekir. Her seferinde yeniden yazılmaları hem zaman kaybıdır hem de hata kaynağıdır.

emu8086 bu sorunu bir kütüphaneyle çözer. Kurulumla gelen `emu8086.inc` dosyası ekran, klavye ve sayı işlemleri için hazır makrolar ve prosedürler içerir. Programcı bu dosyayı kaynağın başına ekler ve içindeki işlevleri adlarıyla kullanır. Kütüphane yeni bir yetenek getirmez: içindeki her şey BIOS ve DOS kesmeleriyle, yani programcının kendisinin de yazabileceği komutlarla kuruludur. Kazanç, sınanmış kodun yeniden kullanılmasıdır.

Tanım 9.2.1

Kütüphane (*library*), hazır makro ve prosedürleri içeren ve başka programlara eklenerek kullanılan kaynak dosyasıdır. emu8086'nın kütüphanesi `emu8086.inc` dosyasıdır; uzantısındaki *inc* ekleme (*include*) dosyası olduğunu gösterir.

Kütüphanenin bir sınırı da baştan bilinmelidir. emu8086.inc, emu8086 ortamına özgüdür. MASM ya da TASM gibi başka bir assembler bu dosyayı tanımaz; kütüphanenin makroları da o assembler'ların makro diliyle birebir uyumlu olmayabilir. İçerideki kod ise sıradan 8086 komutlarından oluşur. Bu yüzden kütüphanenin nasıl çalıştığını anlamak, başka ortamlarda aynı işi yapan kodu yazmayı da öğretir. Notun izleme kesimleri bu amaçla her prosedürün mantığını sade bir eşdeğer kodla gösterir.

9.2.1.2 INCLUDE direktifi

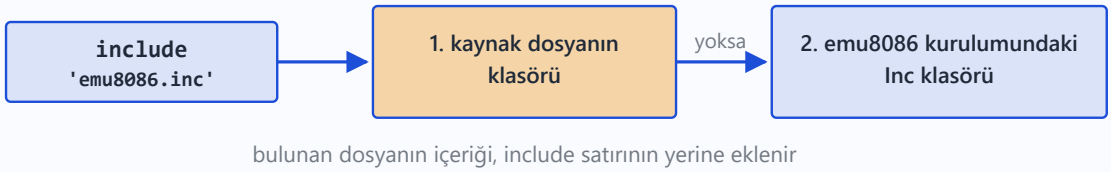
Başka bir dosyadaki tanımlar kaynağa `INCLUDE` direktifiyle alınır. emu8086.inc için satır şöyledir:

```
include 'emu8086.inc'
```

Tanım 9.2.2

INCLUDE, assembler'a belirtilen dosyanın metnini bu satırın yerine koymasını söyleyen direktiftir (*directive*). Direktif işlemci komutu üretmez; işi derleme sırasında biter. Çalışan programda "INCLUDE" diye bir komut yoktur.

Dosya adı tırnak içinde ya da tırnaksız yazılabilir: `include 'emu8086.inc'` ve `include emu8086.inc` aynı işi görür. Büyük-küçük harf farkı da önemsizdir. Assembler dosyayı iki yerde arar (Şekil 9.2.1). Önce derlenen kaynak dosyanın bulunduğu klasöre bakar. Dosya orada yoksa emu8086 kurulumundaki `Inc` klasörüne bakar. emu8086.inc kurulumla birlikte bu klasöre yerleştirildiği için çoğu programda ikinci adımda bulunur. Programcı kendi kütüphanesini yazıp kaynakla aynı klasöre koyarsa ilk adımda bulunur. Aynı adla bir dosya kaynak klasöründe de varsa, kaynak klasöründeki kazanır.



Şekil 9.2.1: INCLUDE satırı önce kaynak dosyanın klasöründe, bulunamazsa emu8086 kurulumundaki Inc klasöründe aranır. Bulunan dosyanın içeriği satırın yerine eklenir.

Direktifin yeri de önemlidir. Bir makro ancak tanımından sonra kullanılabilir; assembler kaynağı yukarıdan aşağı okur ve tanımını görmediği bir makro adını komut sanıp hata verir. Bu yüzden `include` satırı kaynağın en başına, `ORG 100h`'den önce yazılır. Satırın kendisi bayt üretmediği için başa konması programın adreslerini değiştirmez.

Sık yapılan hata

`include` satırını programın sonuna, `END`'den hemen önceye yazmak. Assembler `PUTC 'A'` gibi bir satıra geldiğinde PUTC'nin tanımını henüz görmemiştir ve bilinmeyen komut hatası verir. Makroların tanımı kullanımlarından önce gelmelidir.

9.2.1.3 INCLUDE sonrası kaynak

INCLUDE'un ne yaptığını görmenin en kolay yolu, assembler'ın gördüğü kaynağı düşünmektir. Assembler include satırını siler ve yerine emu8086.inc'in bütün metnini koyar. Ardından ortaya çıkan tek parça kaynağı derler. Sadeleştirilmiş görüntü şöyledir:

```
; include satırının yerine:
PUTC MACRO char     ; şablon: 0 bayt
  ...
ENDM
GOTOXY MACRO col, row
  ...
ENDM
DEFINE_SCAN_NUM MACRO
  ...               ; prosedür şablonu
ENDM
; programın kendisi:
ORG  100h
PUTC 'A'            ; burada 8 bayt
RET
```

Bu görüntüden üç sonuç çıkar. Birincisi, kütüphanenin büyük bölümü `MACRO … ENDM` bloklarından oluşur. Bir makro tanımı yalnız bir şablondur; assembler onu bir yere not eder ama karşılığında hiç bayt üretmez. Kullanılmayan bir makro programa tek bayt eklemez. İkincisi, kod makronun kullanıldığı yerde oluşur. Yukarıdaki programda `PUTC 'A'` satırı açılınca beş komutluk, 8 baytlık bir dizi üretir. Programın tamamı bu 8 bayt ile `RET` 'in 1 baytından oluşur: 9 bayt.

Üçüncü sonuç prosedürlerle ilgilidir. Kütüphanedeki prosedürler de doğrudan kod olarak değil, makroların içinde saklanır. Örneğin SCAN_NUM prosedürünün bütün kodu `DEFINE_SCAN_NUM` adlı bir makronun gövdesidir. Program `DEFINE_SCAN_NUM` satırını yazmazsa makro hiç açılmaz ve SCAN_NUM'ın kodu programa girmez. Böylece kütüphanenin tamamını eklemek programı büyütmez. Program boyutunu yalnız iki şey belirler: kullanılan makroların her kullanımı ve DEFINE_ ile tanımlanan prosedürler.

Kaynakta	Üretilen kod	Ne zaman?
`include 'emu8086.inc'`	yok	yalnız tanımlar okunur
makro kullanımı (`PUTC 'A'`)	makronun gövdesi, parametreleri yerleştirilmiş olarak	her kullanımda yeniden
`DEFINE_SCAN_NUM`	prosedürün bütün kodu	bir kez, satırın bulunduğu yerde
`CALL scan_num`	3 baytlık CALL komutu	her çağrıda

Tablo 9.2.1: emu8086.inc'ten programa ne zaman bayt girer?

Çalışan programın içinde “emu8086.inc” diye bir şey kalmaz. Bellekte yalnız açılmış makro komutları ve tanımlanmış prosedürler bulunur. emu8086'nın bellek penceresinde ya da sökücüsünde (*disassembler*) bakıldığında PUTC'nin yerinde PUSH, MOV, INT ve POP komutları görülür. Bu durum hata ayıklamada önemlidir: tek adım çalıştırırken bir makro satırı birden çok komut adımına karşılık gelir.

9.2.1.4 Kütüphanede neler var?

emu8086.inc'in içeriği iki gruba ayrılır (Tablo 9.2.2, Tablo 9.2.3). Makrolar doğrudan adlarıyla yazılır. Prosedürler önce bir DEFINE_ makrosuyla tanımlanır, sonra CALL ile çağrılır.

Makro	Görev	Arka planda
`PUTC char`	imlecin bulunduğu yere tek karakter yazar	INT 10h, AH = 0Eh
`GOTOXY col, row`	imleci verilen sütun ve satıra götürür	INT 10h, AH = 02h
`PRINT string`	dizgiyi yazar	0Eh ile karakter karakter
`PRINTN string`	dizgiyi yazar, sonra yeni satıra geçer	0Eh ile karakter karakter; sona 13, 10
`CURSOROFF`	imleci gizler	INT 10h, AH = 01h
`CURSORON`	imleci gösterir	INT 10h, AH = 01h

Tablo 9.2.2: Kütüphanedeki makrolar (doğrudan kullanılır).

Prosedür	Görev	Tanım satırı
`PRINT_STRING`	DS:SI'deki 0 ile biten dizgiyi yazar	`DEFINE_PRINT_STRING`
`PTHIS`	CALL'ın hemen ardındaki dizgiyi yazar	`DEFINE_PTHIS`
`GET_STRING`	klavyeden dizgi okur	`DEFINE_GET_STRING`
`CLEAR_SCREEN`	ekranı temizler	`DEFINE_CLEAR_SCREEN`
`SCAN_NUM`	klavyeden işaretli sayı okur, CX'e yazar	`DEFINE_SCAN_NUM`
`PRINT_NUM`	AX'i işaretli sayı olarak yazar	`DEFINE_PRINT_NUM` + `DEFINE_PRINT_NUM_UNS`
`PRINT_NUM_UNS`	AX'i işaretsiz sayı olarak yazar	`DEFINE_PRINT_NUM_UNS`

Tablo 9.2.3: Kütüphanedeki prosedürler (DEFINE_ ile tanımlanır, CALL ile çağrılır).

İki grubun farkı makro ile prosedür arasındaki temel farktır. Makro kullanıldığı her yere kopyalanır; kısa ve sık kullanılan işler için uygundur. Prosedürün kodu bellekte bir kez bulunur ve her kullanımda CALL ile oraya gidilir; uzun işler için uygundur. Kütüphane bu ayrımı mantıklı biçimde yapar. PUTC 8 bayttır ve bir CALL ile RET'in maliyeti işin kendisine yakın olurdu; bu yüzden makrodur. SCAN_NUM ise eksi işaretini, geri silmeyi ve taşmayı denetleyen yüzlerce baytlık bir koddur. Her kullanımda kopyalanması programı hızla büyütürdü; bu yüzden prosedürdür.

Prosedür çağrısının süresi de hesaplanabilir. 8086'da aynı kesim içindeki doğrudan bir CALL 19, yakın RET 8 saat turu sürer. 5 MHz'lik bir 8086'da bir tur 200 ns olduğundan çağırıp dönmenin bedeli 27 × 200 ns = 5,4 µs'dir. Bir karakter yazan `INT 10h` ise BIOS'un içinde yüzlerce komut çalıştırır. Dolayısıyla ekran ve klavye işlerinde CALL/RET'in süresi önemsizdir; asıl belirleyici, bellekte kaplanan yerdir.

9.2.2 Ekran makroları

9.2.2.1 Metin ekranı, imleç ve BIOS hizmetleri

emu8086'nın çıktı penceresi PC'nin standart metin ekranını taklit eder. Ekran 80 sütun ve 25 satırdan oluşur. Her hücrede bir karakter durur. Hücrelerin konumu (sütun, satır) çiftiyle verilir: sol üst köşe (0, 0), sağ alt köşe (79, 24)'tür. Sayma sıfırdan başlar; bu yüzden "10. sütun" soldan on birinci hücredir.

> **Tanım 9.2.3**
>
> **İmleç** (*cursor*), ekranda bir sonraki karakterin yazılacağı konumdur. BIOS imlecin sütununu ve satırını kendi veri alanında saklar. Karakter yazan hizmetler karakteri imlecin bulunduğu hücreye koyar ve imleci bir sağa ilerletir.

Ekranın arkasında bir bellek bölgesi vardır. Renkli metin modunda video kartının belleği işlemcinin adres uzayında B800h kesiminden başlar. Her hücre iki bayt tutar: karakterin ASCII kodu ve rengini belirleyen öznitelik baytı. Video kartı bu belleği saniyede onlarca kez okur ve ekrana çizer. Bir karakteri ekrana koymanın en doğrudan yolu bu belleğe `MOV` ile yazmaktır. İşlemci için bu sıradan bir bellek yazma yol döngüsüdür; adres yolunda B8xxxh gibi bir fiziksel adres, veri yolunda karakter kodu görünür.

Kütüphanenin makroları bu belleğe doğrudan yazmaz; BIOS'un ekran hizmetlerini kullanır. `INT 10h` komutu işlemciyi anakarttaki ROM'da bulunan BIOS yordamına götürür. Yordam, AH'deki hizmet numarasına bakarak ne yapacağına karar verir. İmleci ilerletmek, satır sonunda alt satıra geçmek ve son satırda ekranı kaydırmak gibi ayrıntıları BIOS üstlenir. Kütüphanenin kullandığı üç hizmet Tablo 9.2.4'tedir.

AH	Hizmet	Girdiler	Kullanan
01h	imleç şeklini ayarla	CH = başlangıç, CL = bitiş tarama satırı	CURSOROFF, CURSORON
02h	imleç konumunu ayarla	BH = sayfa, DH = satır, DL = sütun	GOTOXY
0Eh	karakter yaz ve imleci ilerlet (*teletype*)	AL = karakter	PUTC, PRINT, PRINTN

Tablo 9.2.4: Ekran makrolarının kullandığı INT 10h hizmetleri.

0Eh hizmetinin adı eski daktilo-telgraf makinelerinden gelir. Bu makineler gibi hizmet de karakteri yazar ve kâğıdı, yani imleci, ilerletir. Kontrol karakterlerini de makine gibi yorumlar: 13 imleci satır başına, 10 alt satıra götürür, 8 bir sola alır, 7 ses çıkarır. Bu özellik ekran makrolarının davranışının büyük bölümünü açıklar.

9.2.2.2 PUTC ve GOTOXY neye genişler?

PUTC tek bir parametre alır: yazılacak karakter. Parametre bir sabit (`PUTC 65`), bir karakter sabiti (`PUTC 'B'`) ya da 8 bitlik bir yazmaç veya değişken (`PUTC AL`) olabilir. Assembler makroyu açarken parametrenin yazıldığı yere bu metni koyar. `PUTC 'A'` şu beş komuta genişler:

```
PUSH AX        ; AX'i koru
MOV  AL, 'A'   ; parametre
MOV  AH, 0Eh   ; teletype
INT  10h
POP  AX        ; AX geri
```

Komutlar sırayla şunları yapar. `PUSH AX` SP'yi 2 azaltır ve AX'in o anki değerini yığına yazar. `MOV AL, 'A'` AL'ye 41h koyar; AH'ye dokunmaz. `MOV AH, 0Eh` hizmet numarasını AH'ye koyar; artık AX = 0E41h'dir. `INT 10h` bayrakları, CS'yi ve IP'yi yığına iter ve BIOS'a geçer. BIOS 'A'yı imlecin bulunduğu hücreye yazar, imleci bir sağa alır ve `IRET` ile döner. `POP AX` yığındaki eski değeri AX'e geri yükler ve SP'yi 2 artırır. Makrodan çıkıldığında AX, SP ve bayraklar girişteki değerlerine dönmüştür. Çağıran kod AX'in geçici olarak değiştiğini fark etmez.

Açılımın boyutu makine kodlarından hesaplanır: `PUSH AX` 1 bayt (50h), `MOV AL, 41h` 2 bayt (B0 41), `MOV AH, 0Eh` 2 bayt (B4 0E), `INT 10h` 2 bayt (CD 10), `POP AX` 1 bayt (58h). Toplam 8 bayttır.

GOTOXY iki parametre alır: önce sütun, sonra satır. `GOTOXY 10, 5` şöyle açılır:

```
PUSH AX
PUSH BX
PUSH DX
MOV  AH, 02h   ; imleç konumla
MOV  DH, 5     ; satır (row)
MOV  DL, 10    ; sütun (col)
MOV  BH, 0     ; sayfa 0
INT  10h
POP  DX
POP  BX
POP  AX
```

Makro AH, BH, DH ve DL'yi kullanır; bu yüzden AX, BX ve DX'i saklar. POP'lar PUSH'ların ters sırasıyla yapılır: en son itilen DX ilk geri alınır. Açılım 16 bayttır: üç PUSH ve üç POP 6 bayt, dört `MOV r8, sabit` 8 bayt, `INT 10h` 2 bayt. BH = 0, metin modunun ilk görüntü sayfasını seçer. Renkli metin kartının belleğine birden çok ekran sayfası sığar; emu8086 programları hep 0. sayfayı kullanır.

Parametre sırası en sık karışan noktadır. Makro matematikteki (x, y) sırasını kullanır: önce yatay konum (sütun), sonra dikey konum (satır). BIOS ise satırı DH'de, sütunu DL'de bekler. Makro bu dönüşümü kendisi yapar: ikinci parametreyi DH'ye, birinciyi DL'ye koyar.

Sık yapılan hata

`GOTOXY 5, 10` yazıp imlecin 5. satıra gitmesini beklemek. Makro 5'i sütun, 10'u satır olarak alır; imleç 10. satırın 5. sütununa gider. Benzer bir tuzak, konumu AH'de tutmaktır: `GOTOXY AL, AH` açılımında önce `MOV AH, 02h` çalışır, ardından `MOV DH, AH` satıra 2 yazar. Örneğin AL = 20, AH = 7 iken imleç 7. satıra değil 2. satıra, (20, 2) konumuna gider. Konum önce makronun kullanmadığı yazmaçlara alınmalıdır.

9.2.2.3 PRINT ve PRINTN

PRINT bir dizgiyi yazar. Parametresi kaynakta tırnak içinde verilen sabit bir metindir: `PRINT 'Hello World!'`. Makro bu metni koda gömer ve 0Eh hizmetiyle karakter karakter yazar. emu8086.inc'teki PRINT'in açılımı şu yapıdadır (yerel etiket adları sadeleştirildi):

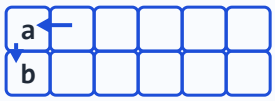
```
PUSH AX
PUSH SI
JMP  atla           ; dizginin üstünden atla
dizgi DB 'Hello World!', 0
atla:
LEA  SI, dizgi
sonraki:
MOV  AL, CS:[SI]    ; karakteri kod kesiminden oku
CMP  AL, 0
JZ   bitti
INC  SI
MOV  AH, 0Eh
INT  10h
JMP  sonraki
bitti:
POP  SI
POP  AX
```

Açılımın en ilginç yanı dizginin yeridir. Dizgi kodun ortasında durur ve işlemci onu komut sanmasın diye üstünden bir `JMP` ile atlanır. Makroya özgü etiketler `LOCAL` direktifiyle her açılımda benzersiz adlar alır; PRINT ikinci kez kullanıldığında etiket çakışması olmaz. Dizgi kodun içinde olduğu için kod kesimindedir ve karakterler `CS:[SI]` ile okunur. .COM programında CS = DS olduğu için önek fark yaratmaz; ama önek sayesinde makro DS'nin başka bir kesimi gösterdiği durumlarda da doğru çalışır.

Açılım SI'yi ve AX'i saklar. Boyutu dizginin uzunluğuna bağlıdır: n karakterlik bir dizgi için n + 1 bayt veri ve 23 bayt kod, toplam 24 + n bayt. `PRINT 'Hello World!'` 12 karakter için 36 bayt tutar. Döngü kodu PRINT'in her kullanımında yeniden kopyalanır.

PRINTN aynı işi yapar, yalnız dizginin sonuna 13 ve 10 baytlarını ekler. Makro dizgiyi `DB 'metin', 13, 10, 0` biçiminde tanımlar. 13 (CR, satır başı) imleci satırın 0. sütununa, 10 (LF, satır besleme) bir alt satıra götürür. İkisi birlikte yeni satıra geçişi sağlar (Şekil 9.2.2). `PRINT 'a'` ardından `PRINT 'b'`

ekranda *ab* bırakır. `PRINTN 'a'` ardından `PRINT 'b'` ise *a*'yı bir satıra, *b*'yi alt satırın başına yazar. PRINTN'in açılımı PRINT'inkinden 2 bayt uzundur.



Şekil 9.2.2: PRINTN 'a' ardından PRINT 'b': CR imleci satır başına, LF bir alt satıra götürür; b alt satırın başına yazılır.

PRINT'in parametresi derleme anında bilinen sabit bir metin olmalıdır. Makro metni koda gömdüğü için çalışma sırasında oluşan bir dizgiyi, örneğin klavyeden okunan bir adı yazamaz. `PRINT ad` yazıldığında assembler `DB ad, 0` satırını üretir. Bu satır ad değişkeninin adresini bayt olarak tanımlamaya çalışır; sonuç ya bir hata ya da anlamsız bir bayttır. Çalışma sırasında oluşan dizgiler PRINT_STRING ile yazılır.

9.2.2.4 Örnek: makrolarla ekran

emu8086'nın kütüphane belgesindeki ilk örnek üç makroyu birlikte kullanır:

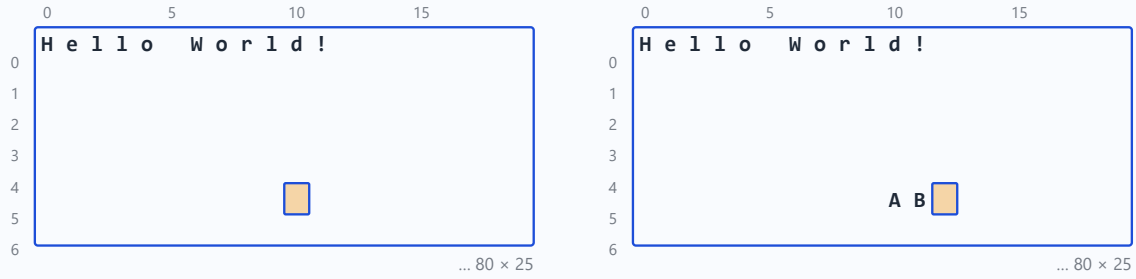
```
include emu8086.inc
ORG    100h
PRINT 'Hello World!'
GOTOXY 10, 5
PUTC 65           ; 65 - is an ASCII code for 'A'
PUTC 'B'
RET               ; return to operating system.
END               ; directive to stop the compiler.
```

İlk iki satır kod üretmez. `include` makro tanımlarını ekler, `ORG 100h` assembler'a programın 0100h bağıl konumundan başlayacağını söyler. .COM programlarında ilk 256 bayt (0000h–00FFh) DOS'un program kesimi önekine (PSP) ayrıldığı için kod 0100h'den başlar. Program başladığında ekran boştur ve imleç (0, 0)'dadır.

`PRINT 'Hello World!'` 12 karakteri 0. satırın 0–11. sütunlarına yazar. Her karakterden sonra BIOS imleci bir sağa aldığı için imleç (12, 0)'da kalır. `GOTOXY 10, 5` imleci 10. sütun, 5. satıra taşır; ekrana bir şey yazmaz. `PUTC 65` sayıyı karakter kodu olarak kullanır: 65 = 41h, ASCII'de 'A'dır. 'A' (10, 5) hücresine yazılır ve imleç (11, 5)'e geçer. `PUTC 'B'` aynı işi karakter sabitiyle yapar; assembler `'B'`'yi 42h'ye çevirir. 'B' (11, 5)'e yazılır, imleç (12, 5)'e geçer. `RET` programı bitirir. `END` direktifi yalnız assembler'a kaynağın bittiğini söyler; kod üretmez.

Satır	İmleç (sütun, satır)	Ekrana yazılan	Açıklama
`include emu8086.inc`	(0, 0)	—	makro tanımları eklenir, kod yok
`ORG 100h`	(0, 0)	—	.COM programı 0100h'den başlar
`PRINT 'Hello World!'`	**(12, 0)**	Hello World! (0–11. sütun, 0. satır)	12 karakter
`GOTOXY 10, 5`	**(10, 5)**	—	yalnız imleç taşınır
`PUTC 65`	**(11, 5)**	A, (10, 5) hücresine	65 = 41h = 'A'
`PUTC 'B'`	**(12, 5)**	B, (11, 5) hücresine	karakter sabiti
`RET`	(12, 5)	—	DOS'a dönüş

Tablo 9.2.5: Makrolarla ekran programının izlemesi (her satır çalıştıktan sonraki durum).



Şekil 9.2.3: GOTOXY 10, 5'ten sonra (solda) imleç (10, 5)'tedir; iki PUTC'den sonra (sağda) ekranın 5. satırında AB yazılıdır.

Programın makine kodu 0100h'den 0144h'ye kadar uzanır. 36 baytlık PRINT açılımı 0100h–0123h'yi, 16 baytlık GOTOXY açılımı 0124h–0133h'yi, iki PUTC açılımı 0134h–0143h'yi kaplar; `RET` 0144h'dedir. Kaynakta sekiz satır görünse de işlemci 35 farklı komut çalıştırır. PRINT'in döngüsü 12 kez döndüğü için yürütülen komut sayısı yüzü aşar.

9.2.2.5 CURSOROFF ve CURSORON

Metin ekranındaki imleç, video kartının yanıp söndürdüğü küçük bir çizgidir. Çizginin yüksekliği karakter hücresinin tarama satırlarıyla verilir. Renkli metin modunda bir karakter hücresi 8 ya da 16 tarama satırı yüksekliğindedir; imleç normalde hücrenin alt kısmındaki bir iki satırı kaplar. INT 10h'nin 01h hizmeti imlecin şeklini ayarlar: CH başlangıç, CL bitiş tarama satırıdır.

CH'nin 5. biti özel bir anlam taşır. Bu bit 1 yapılırsa imleç ekrana hiç çizilmez. Kütüphanedeki CURSOROFF, CH'ye 28h yazar: 28h = 0010 1000b'dir ve 5. biti 1'dir. CURSORON aynı hizmetle 5. biti 0 olan normal bir şekil yükler. İki makro da AX ile CX'i saklayıp geri yükler.

```
PRINTN 'Ad Soyad:'   ; yazıp alt satıra geç
CURSOROFF
GOTOXY 30, 12
PRINT 'Merhaba'
CURSORON
```

Bu parçada önce “Ad Soyad:” yazılır ve imleç alt satırın başına geçer. CURSOROFF imleci gizler. GOTOXY imleci (30, 12)'ye taşır ve “Merhaba” 30–36. sütunlara yazılır. Bu sırada yanıp sönen imleç ekranda dolaşmaz. CURSORON imleci yeniden görünür yapar; imleç (37, 12)'dedir. İmleci gizlemek yalnız görüntüyü değiştirir; imlecin konumu BIOS'ta tutulmaya ve yazı hizmetleri onu ilerletmeye devam eder. Oyun ve menü ekranlarında yanıp sönen imleç dikkat dağıttığı için gizlenir.

9.2.2.6 Kontrol karakterleri

ASCII tablosunun ilk 32 kodu yazdırılabilir bir şekle karşılık gelmez. Bu kodlara **kontrol karakteri** (*control character*) denir. 0Eh hizmeti bunların dördünü daktilo gibi yorumlar (Tablo 9.2.6). PUTC bu kodlarla kullanıldığında ekrana bir şekil yazmaz, imleci yönetir.

Kod	Ad	Etkisi
07h	BEL (zil)	hoparlörden kısa bir bip; imleç yerinden oynamaz
08h	BS (geri)	imleç bir sola gider; karakteri silmez; 0. sütunda etkisizdir
0Ah	LF (satır besleme)	imleç bir alt satıra iner, sütun değişmez; son satırda ekran bir satır yukarı kayar
0Dh	CR (satır başı)	imleç aynı satırın 0. sütununa döner; satırdaki yazı yerinde kalır

Tablo 9.2.6: PUTC ile verilen kontrol karakterleri.

Yeni satır bu yüzden iki karakterle yazılır: önce CR, sonra LF. PRINTN'in eklediği 13, 10 çifti bu iki hareketin birleşimidir. Sıra ters olsa da (10, 13) imleç aynı yere varır: önce bir alt satıra iner, sonra o satırın başına döner. DOS ve Windows metin dosyalarında satır sonu bu yüzden CR LF çiftidir.

Ekrandan bir karakteri silmek için BS tek başına yetmez; imleci sola alır ama karakter yerinde kalır. Silme üç adımla yapılır. `PUTC 8` imleci sola alır. `PUTC ' '` karakterin üstüne boşluk yazar ve imleci yine sağa geçirir. İkinci `PUTC 8` imleci boşalan hücreye geri getirir. Kütüphanenin geri silme tuşunu işleyen prosedürleri ekranı bu üçlüyle düzeltir.

9.2.2.7 CR ve LF ayrı ayrı: kod izleme

CR ile LF'nin ayrı ayrı ne yaptığını görmek için aşağıdaki program INT 10h'yi makrosuz kullanır. AH = 0Eh bir kez yüklenir; BIOS 0Eh hizmetinde AH'yi değiştirmediği için sonraki çağrılar için de geçerli kalır.

```
ORG  100h
MOV  AH, 0Eh     ; teletype
MOV  AL, 'a'
INT  10h
MOV  AL, 'b'
INT  10h
MOV  AL, 10      ; yalnız LF
INT  10h
MOV  AL, 'c'
INT  10h
MOV  AL, 13      ; yalnız CR
INT  10h
MOV  AL, 'd'
INT  10h
RET
```

Komutlar	AL	İmleç (sütun, satır)	Ekran
`MOV AH, 0Eh` / `MOV AL, 'a'` / `INT 10h`	61h	(1, 0)	a
`MOV AL, 'b'` / `INT 10h`	62h	(2, 0)	ab
`MOV AL, 10` / `INT 10h`	0Ah	**(2, 1)**	ab
`MOV AL, 'c'` / `INT 10h`	63h	(3, 1)	ab / ··c
`MOV AL, 13` / `INT 10h`	0Dh	**(0, 1)**	ab / ··c
`MOV AL, 'd'` / `INT 10h`	64h	(1, 1)	ab / d·c

Tablo 9.2.7: CR ve LF'nin ayrı ayrı etkisi.

Tablodaki “/” satır ayrımını, “·” boş hücreyi gösterir. Yalnız LF gönderildiğinde imleç alt satıra iner ama 2. sütunda kalır; 'c' alt satırın 2. sütununa yazılır ve satırlar merdiven gibi kayar. Yalnız CR gönderildiğinde imleç aynı satırın başına döner; satırdaki 'c' silinmez. 'd' 0. sütuna yazılır ve ekranda “ab”nin altında “d c” görünür (Şekil 9.2.4). Düzgün bir yeni satır için iki kod birlikte gerekir.



Şekil 9.2.4: Yalnız LF'den sonra imleç (2, 1)'dedir (solda); yalnız CR ve 'd'den sonra ekranda ab ve altında d c görünür (sağda).

INT 10h/0Eh AX dışında yazmaç değiştirmez; bu programda AH de çağrılar boyunca 0Eh kalır. Bu davranış BIOS'un belgelenmiş özelliğidir, ama başka hizmetler için genelleştirilmemelidir. Bazı INT 10h hizmetleri AX'i, bazıları BP'yi değiştirir. Kütüphane makrolarının AX'i her kullanımda saklaması bu yüzden güvenli bir alışkanlıktır.

9.2.2.8 Satır sonu ve ekran kayması

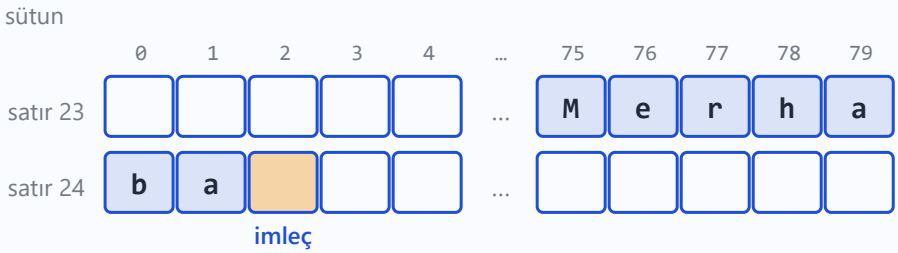
0Eh hizmeti 79. sütuna bir karakter yazdıktan sonra imleci kendiliğinden alt satırın başına geçirir. PRINT uzun bir dizgiyi bu sayede kesmeden sürdürür. İmleç son satırdayken (24) alt satıra geçmek gerekirse BIOS ekranın tamamını bir satır yukarı kaydırır. En üst satır kaybolur, en alta boş bir satır gelir ve imleç bu yeni satıra geçer. Kaydırma, video belleğindeki alt 24 satırın bir satır yukarı kopyalanmasıdır.

Örnek 9.2.1

`GOTOXY 75, 24` ardından `PRINT 'Merhaba'` çalışıyor. Yazının ekranda nerede kaldığını ve imlecin son konumunu bulalım.

1. İmleç (75, 24)'tedir. “M”, “e”, “r”, “h”, “a” sırasıyla 75, 76, 77, 78 ve 79. sütunlara yazılır.
2. 79. sütuna yazıldıktan sonra imleç alt satıra geçmelidir. İmleç son satırda olduğu için ekran bir satır yukarı kayar. “Merha” 23. satıra çıkar; 24. satır boşalır ve imleç (0, 24)'e gelir.
3. “b” ve “a” 24. satırın 0 ve 1. sütunlarına yazılır. İmleç (2, 24)'te kalır.

Sonuçta “Merha” 23. satırın 75–79. sütunlarında, “ba” 24. satırın başında, imleç (2, 24)'tedir (Şekil 9.2.5).



Şekil 9.2.5: Ekranın son iki satırı: “Merha” 23. satırın 75–79. sütunlarında, “ba” 24. satırın başında; imleç (2, 24)'te.

GOTOXY ve PUTC konumları denetlemez. `GOTOXY 85, 3` gibi bir çağrıda BIOS'un davranışı tanımlı değildir; imleç görünmeyen bir yere gidebilir ve sonraki yazılar beklenmedik yerlerde çıkar. Programcı konumları 0–79 (sütun) ve 0–24 (satır) aralığında tutmalıdır.

9.2.2.9 GOTOXY ve video belleği

Ekranda seçilen bir hücre üç farklı biçimde anlatılabilir: makro çağrısıyla (`GOTOXY 10, 5`), BIOS'a giden yazmaç değerleriyle (AH = 02h, DH = 5, DL = 10, BH = 0) ve hücrenin video belleğindeki adresiyle. Sonuncusu BIOS'un içinde ne olduğunu anlamak için yararlıdır.

Video belleğinde hücreler satır satır, soldan sağa dizilir. Her hücre iki bayt kapladığı için (sütun, satır) hücresinin bağıl konumu şöyle hesaplanır:

`bağıl konum = (satır × 80 + sütun) × 2`

Çift adresteki bayt karakterin ASCII kodu, tek adresteki bayt öznitelik baytıdır. `GOTOXY 10, 5` ve ardından `PUTC 'A'` için hesap şöyledir: (5 × 80 + 10) × 2 = 410 × 2 = 820 = 0334h. 'A' (41h) B800:0334h'ye, öznitelik baytı (07h, siyah üstüne açık gri) B800:0335h'ye yazılır. Fiziksel adres B8000h + 0334h = B8334h'dir. Ekranın sağ alt köşesi (79, 24) için bağıl konum (24 × 80 + 79) × 2 = 3998 = 0F9Eh olur. 80 × 25 hücrelik bir sayfa 4000 bayt tutar.

Bu adresleri bilmek, makroların yaptığı işi doğrudan bellek yazımıyla yapmayı da mümkün kılar. Örneğin `MOV AX, 0B800h`, `MOV ES, AX`, `MOV BYTE PTR ES:[0334h], 'A'` dizisi 'A'yı aynı hücreye yazar. Fark, imlecin ilerlememesi ve ekranın kaymamasıdır: doğrudan yazım BIOS'un üstlendiği bütün ayrıntıları programcıya bırakır. Kütüphane makroları daha yavaştır ama bu ayrıntıları düşünmeyi gerektirmez.

9.2.2.10 İki basamak yazdırma: kod izleme

PUTC'nin AX'i koruması küçük bir ayrıntı gibi görünür. Aşağıdaki örnek bu ayrıntının neden yararlı olduğunu gösterir. Program AL'deki 0–99 arası bir sayıyı iki onluk basamak olarak yazar. Basamakları ayırmak için `AAM` komutu kullanılır. Kodda iki PUTC'nin açılımı makro yerine açıkça yazılmıştır:

```
ORG  100h
MOV  AL, 58
AAM              ; AH = 5, AL = 8
ADD  AX, 3030h   ; '5', '8'
PUSH AX          ; PUTC AH açılımı
MOV  AL, AH
MOV  AH, 0Eh
INT  10h
POP  AX          ; AX geri: 3538h
PUSH AX          ; PUTC AL açılımı
MOV  AL, AL
MOV  AH, 0Eh
INT  10h
POP  AX
RET
```

Tanım 9.2.4

AAM (*ASCII adjust after multiply*), AL'yi 10'a böler: bölümü AH'ye, kalanı AL'ye yazar. 0–99 arası bir sayının onlar basamağı AH'de, birler basamağı AL'de kalır. SF, ZF ve PF AL'ye göre kurulur. 8086'da AAM 83 saat turu sürer.

`MOV AL, 58` AL'ye 58 = 3Ah koyar. `AAM` 58'i 10'a böler: AH = 5, AL = 8, yani AX = 0508h. Yalnız AL bölündüğü için sayı 0–99 aralığında olmalıdır; 100 ve üstünde bölüm de tek basamak olmaz. `ADD AX, 3030h` iki bayta birden 30h ekler. Bir basamağa 30h eklemek onu ASCII rakamına çevirir: 5 + 30h = 35h = '5', 8 + 30h = 38h = '8'. Toplamada iki bayt arasında elde oluşmaz, çünkü 08h + 30h = 38h 8 bite sığar.

İlk açılım onlar basamağını yazar. `PUSH AX` 3538h'yi yığına koyar, SP = FFFCh. `MOV AL, AH` AL'ye '5'i kopyalar ve `MOV AH, 0Eh` hizmet numarasını yükler; AX = 0E35h olur. Birler basamağı yazmaçtan silinmiştir. `INT 10h` ekrana 5 yazar. `POP AX` 3538h'yi geri getirir; '8' yeniden AL'dedir. İkinci açılım aynı kalıpla birler basamağını yazar. `MOV AL, AL` hiçbir şey değiştirmez ama zararsızdır; makro parametre AL olduğunda bu komutu üretir.

Komut(lar)	AH	AL	SP	[FFFCh]	Ekran
`MOV AL, 58`	00h	**3Ah**	FFFEh	—	
`AAM`	**05h**	**08h**	FFFEh	—	
`ADD AX, 3030h`	**35h**	**38h**	FFFEh	—	
`PUSH AX`	35h	38h	**FFFCh**	**3538h**	
`MOV AL, AH` / `MOV AH, 0Eh` / `INT 10h`	**0Eh**	**35h**	FFFCh	3538h	5
`POP AX`	**35h**	**38h**	**FFFEh**	—	5
ikinci açılım (5 komut)	35h	38h	FFFEh	—	58

Tablo 9.2.8: AAM ve PUTC ile iki basamak yazdırmanın izlemesi.

Programın sonunda ekranda 58 yazılıdır ve AX yine 3538h'dir (Tablo 9.2.8). PUTC AX'i koruduğu için iki basamak ayrı bir yazmaca kopyalanmadan art arda yazılabildi. Makro AX'i korumasaydı ilk PUTC '8'i silerdi ve programcı birler basamağını önceden başka bir yazmaca almak zorunda kalırdı. Kaynakta makrolarla yazıldığında aynı program yalnız `PUTC AH` ve `PUTC AL` satırlarını içerir.

9.2.3 Prosedürleri kullanmak

9.2.3.1 DEFINE_ ve CALL

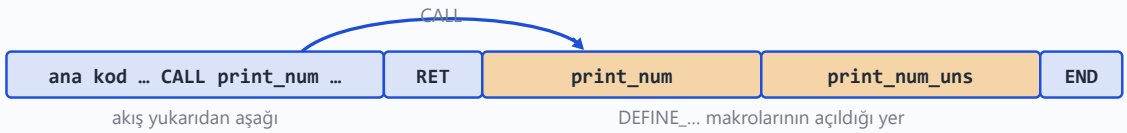
Kütüphanedeki bir prosedürü kullanmak iki adım ister. Önce prosedür bir kez tanımlanır: bunun için adı `DEFINE_` ile başlayan makro yazılır. Sonra programın gereken yerlerinde prosedür `CALL` ile çağrılır. Tipik bir iskelet şöyledir:

```
include 'emu8086.inc'
ORG 100h
    ...
    CALL print_num   ; kullan
    ...
RET
DEFINE_PRINT_NUM    ; tanımla
DEFINE_PRINT_NUM_UNS
END
```

Tanım 9.2.5

DEFINE_ makrosu, gövdesinde bir kütüphane prosedürünün bütün kodunu taşıyan makrodur. Makro açıldığı yere prosedürü `PROC … ENDP` bloğu olarak yerleştirir. Her prosedür programda en çok bir kez tanımlanır; iki kez tanımlanırsa aynı etiket iki kez oluşur ve assembler hata verir.

Makronun açıldığı yer, prosedürün bellekteki yeridir. Kaynakta DEFINE_ satırları genellikle ana programın sonundaki `RET` ile `END` direktifi arasına yazılır (Şekil 9.2.6). Bellekte önce ana kod, ardından RET, ardından prosedürlerin kodu gelir. İşlemci ana kodu yukarıdan aşağı çalıştırır, RET ile DOS'a döner ve prosedürlerin koduna yalnız CALL ile girer.



Şekil 9.2.6: Bellekte önce ana kod ve RET, ardından DEFINE_ makrolarıyla açılan prosedürler gelir; CALL prosedüre atlar.

Bu yerleşim alışılmış olandır ama tek güvenli yer değildir. emu8086.inc'teki her DEFINE_ makrosu bir koruma içerir: gövdenin başında prosedürün üstünden atlayan bir `JMP` vardır. DEFINE_SCAN_NUM'ın yapısı kabaca şöyledir:

```
JMP skip_proc_scan_num   ; akış gövdeye girmesin
SCAN_NUM PROC NEAR
    ...                  ; prosedürün kodu
    RET
SCAN_NUM ENDP
skip_proc_scan_num:
```

Bir DEFINE_ satırı yanlışlıkla `ORG 100h`'nin hemen altına yazılsa da program doğru çalışır. İşlemcinin ilk çalıştırdığı komut bu JMP'dir ve gövdenin sonrasına, ana programın başına atlar. Bedeli, programda fazladan bir JMP komutunun bulunmasıdır. Yine de DEFINE_ satırlarının RET ile END arasında toplanması programı okunur tutar ve gereksiz atlamaları önler.

Prosedür adları kaynakta küçük harfle yazılır: `CALL print_num`, `CALL scan_num`. emu8086 etiketlerde büyük-küçük harf ayırmaz; `CALL PRINT_NUM` da aynı etikete gider. Küçük harf, belgelerdeki yazımdır ve makro adlarıyla (`DEFINE_PRINT_NUM`) prosedür adını görsel olarak ayırır.

Bir prosedür başka bir prosedürü çağırıyorsa ikisi birlikte tanımlanmalıdır. PRINT_NUM sayının işaretini yazdıktan sonra mutlak değeri PRINT_NUM_UNS'ye yazdırır. Yalnız `DEFINE_PRINT_NUM` yazılırsa assembler PRINT_NUM'ın içindeki `CALL PRINT_NUM_UNS` satırında etiketi bulamaz ve tanımsız etiket hatası verir. Ters durum hata vermez: PRINT_NUM_UNS tek başına tanımlanabilir.

> **Sık yapılan hata**
>
> `CALL scan_num` yazıp `DEFINE_SCAN_NUM`'ı unutmak. Kütüphanenin eklenmiş olması prosedürün tanımlı olduğu anlamına gelmez; makro açılmadıkça scan_num etiketi yoktur. emu8086 derleme sırasında bu etiket için "undefined" hatası verir.

Makro ile prosedürün bellekteki maliyeti somut bir örnekle karşılaştırılabilir. Bir programda on farklı ileti yazılsın. PRINT ile her ileti kendi döngü kodunu taşır: her açılımda 23 bayt kod ve iletinin baytları bulunur. PRINT_STRING ile döngü bellekte bir kez bulunur; her ileti için yalnız 3 baytlık `LEA SI, ileti` ve 3 baytlık `CALL print_string` eklenir. Yedi karakterlik on iletiyle program PRINT kullanıldığında yaklaşık 310 bayt, PRINT_STRING kullanıldığında yaklaşık 160 bayt tutar. Kazancın karşılığı süredir: her çağrıda CALL ve RET fazladan 27 saat turu harcar.

9.2.3.2 Girdi ve çıktı yazmaçları

Bir prosedür parametrelerini yazmaçlarda alır ve sonucunu yazmaçta döndürür. Hangi yazmacın ne anlama geldiği prosedürün belgesinde yazar ve programcı buna uymak zorundadır (Tablo 9.2.9).

Prosedür	Girdi	Çıktı	Ayrıntı
PRINT_STRING	DS:SI = dizgi adresi	—	dizgi 0 ile bitmeli
PTHIS	dizgi CALL'ın hemen ardında	—	dönüş adresini kendisi ilerletir
GET_STRING	DS:DI = tampon, DX = boyut	tamponda 0 ile biten dizgi	en fazla DX − 1 karakter
CLEAR_SCREEN	—	—	imleç (0, 0)'a gider
SCAN_NUM	klavye	CX (işaretli)	CX'teki eski değer kaybolur
PRINT_NUM	AX (işaretli)	—	PRINT_NUM_UNS de tanımlı olmalı
PRINT_NUM_UNS	AX (işaretsiz)	—	0 … 65.535

Tablo 9.2.9: Kütüphane prosedürlerinin girdi ve çıktı yazmaçları.

Kütüphanenin prosedürleri kullandıkları öteki yazmaçları girişte yığına iter ve çıkışta geri alır. Örneğin GET_STRING AX, CX, DI ve DX'i, SCAN_NUM DX, AX ve SI'yi saklar. Çağıran program için değişen tek yazmaç çıktı yazmacıdır. Bayraklar ise korunmaz: prosedürün içindeki CMP ve aritmetik komutlar bayrakları değiştirir. Bir CALL'dan önce kurulan bayrağa CALL'dan sonra güvenilmemelidir.

Çıktı yazmacının bir sonucu da vardır. SCAN_NUM sonucu CX'e yazdığı için CX'teki eski değer kaybolur. CX'te bir döngü sayacı tutan program, SCAN_NUM'ı çağırmadan önce sayacı kendisi saklamalıdır:

```
MOV  CX, 3          ; üç sayı okunacak
oku: PUSH CX        ; sayacı sakla
CALL scan_num       ; CX = okunan sayı
MOV  AX, CX
ADD  BX, AX         ; toplama ekle
POP  CX             ; sayacı geri al
LOOP oku
```

PUSH CX olmasaydı LOOP, kullanıcının girdiği sayıyı sayaç olarak kullanırdı. Kullanıcı 0 girerse LOOP CX'i FFFFh'ye indirir ve döngü 65.535 kez daha döner.

9.2.3.3 PRINT_STRING ve PTHIS

Kütüphanede dizgi yazan iki prosedür vardır. İkisi de sonu 0 baytıyla işaretlenmiş dizgiyi yazar; farkları dizginin adresini nereden aldıklarıdır.

> **Tanım 9.2.6**
>
> **Sıfırla biten dizgi** (*null-terminated string*), karakterlerin ardından değeri 0 olan bir baytla biten dizgidir. Uzunluk ayrıca saklanmaz; dizgiyi okuyan kod 0 baytını görünce durur. C dilindeki dizgiler de bu biçimdedir.

PRINT_STRING dizginin adresini DS:SI'den alır. Kullanımı iki komuttur:

```
LEA  SI, msg
CALL print_string
...
msg DB 'Merhaba', 0
```

`LEA SI, msg`, msg'nin bağıl konumunu SI'ye yükler. emu8086 doğrudan adresli bu LEA'yı `MOV SI, OFFSET msg` ile aynı 3 baytlık koda (BE ll hh) derler; gerçek MASM'da aynı komut 4 bayttır. Prosedür AX ve SI'yi saklar, sonra SI'nin gösterdiği baytları sırayla okur. Her bayt 0 değilse 0Eh hizmetiyle yazılır ve SI bir artar. 0 okununca SI ve AX geri alınır ve prosedür döner. Dizgi programın veri alanında durur; aynı dizgi programın farklı yerlerinde yeniden yazdırılabilir.

PTHIS adını İngilizce *print this* (bunu yaz) sözünden alır. Dizginin adresini bir yazmaçtan değil, yığından alır. Dizgi CALL komutunun hemen ardına yazılır:

```
CALL pthis
DB 'Merhaba', 0
; buradan devam eder
```

PRINT makrosu her kullanımda döngü kodunu kopyalar; PRINT_STRING ve PTHIS ise tek kopya kodla istenen sayıda dizgi yazar. PTHIS'in PRINT_STRING'e üstünlüğü yazım kolaylığıdır: dizgiye ad vermek ve onu ayrı bir veri alanında tanımlamak gerekmez.

9.2.3.4 Dizgi yazdırma yolları

Kütüphaneyle birlikte bir dizgiyi ekrana yazmanın dört yolu vardır (Tablo 9.2.10). Dördüncüsü kütüphane dışındaki DOS hizmetidir ama karşılaştırma için tabloya alınmıştır.

Yöntem	Dizgi nerede?	Sonlandırıcı	Adres	Uygun durum
`PRINT 'metin'`	kodun içinde (makro açılımı)	0 (makro ekler)	makro kendisi alır (CS:SI)	bir kez yazılan sabit ileti
PRINT_STRING	veri alanında	0	DS:SI	birden çok yerde yazılan ya da çalışırken oluşan dizgi
PTHIS	CALL'ın hemen ardında	0	yığındaki dönüş adresi	çok sayıda kısa sabit ileti
INT 21h, AH = 09h	veri alanında	'$'	DS:DX	kütüphanesiz DOS programı

Tablo 9.2.10: Dizgi yazdırma yollarının karşılaştırması.

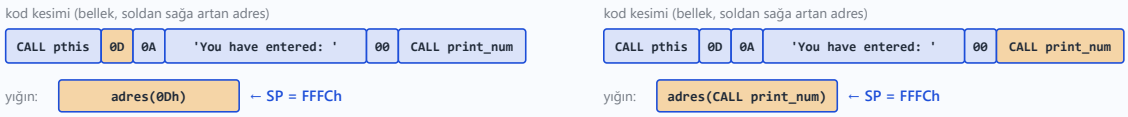
Sonlandırıcı kendisi yazdırılamaz. '$' ile biten bir dizgide '$' karakteri, 0 ile biten bir dizgide 0 baytı dizginin sonu sayılır. Fiyat yazan bir programda “100 $” iletisi INT 21h/09h ile yazılamaz; ileti '$'tan önce biter. Aynı ileti PRINT_STRING ile sorunsuz yazılır.

Tampon ile yöntemin uyumu da önemlidir. GET_STRING'in doldurduğu tampon 0 ile biter ve doğrudan PRINT_STRING ile yazılır. Aynı tampon INT 21h/09h ile yazılırsa DOS '$' arar. Tamponda '$' olmadığı için DOS tamponun ötesindeki baytları da ekrana döker ve ilk '$'a rastlayana kadar sürdürür. Ekranda anlamsız karakterler belirir.

9.2.3.5 PTHIS nasıl çalışır?

PTHIS, CALL komutunun yığına ne ittiğini bilmeye dayanır. CALL, kendisinden sonraki komutun adresini, yani **dönüş adresini** (*return address*) yığına iter ve prosedüre atlar. İşlemci “sonraki komutun” orada bir komut olup olmadığını denetlemez; yalnız CALL'ın uzunluğunu IP'ye ekler. CALL'ın hemen

ardında bir dizgi duruyorsa itilen adres dizginin ilk baytının adresidir. PTHIS bu adresi dizginin adresi olarak kullanır.



Şekil 9.2.7: CALL pthis'ten sonra yığında dizginin ilk baytının (0Dh) adresi durur (solda). PTHIS dizgiyi yazdıktan sonra yığındaki adresi 0 baytının bir ötesiyle, CALL print_num'un adresiyle değiştirir (sağda).

Prosedür dört iş yapar (Şekil 9.2.7). Önce dönüş adresini `POP SI` ile yığından SI'ye alır. Sonra SI'nin gösterdiği baytları 0'a kadar okur ve yazar; her okumadan sonra SI bir artar. 0 okunduğunda SI, 0 baytının bir ötesini gösterir: bu adres dizgiden sonraki ilk komuttur. Prosedür bu adresi `PUSH SI` ile yığına geri koyar. Son olarak `RET` yığındaki yeni adresi IP'ye alır. İşlemci dizgiyi atlayarak ondan sonraki komuttan devam eder; dizginin baytları hiçbir zaman komut sanılıp çalıştırılmaz.

Kütüphanedeki PTHIS çağıranın SI ve AX değerlerini de korur. AX yığına itilir. SI ise yığına itilemez, çünkü yığının üstünde dönüş adresi durur ve onun önce alınması gerekir. Prosedür bu yüzden SI'nin eski değerini kod kesimindeki 2 baytlık bir değişkende (`CS:temp1`) saklar ve sonunda oradan geri yükler. Dizginin baytları da kod kesiminde olduğu için `CS:[SI]` ile okunur.

Adreslerle izlemek düzeneği somutlaştırır. Sayı okuyup yazan örnek programda `CALL pthis` 010Bh'dedir ve 3 bayttır. Ardından 010Eh'den başlayan 21 baytlık dizgi gelir: 13, 10, “You have entered: ” ve 0. Dizgi 010Eh–0122h'yi kaplar; `CALL print_num` 0123h'dedir.

An	SP	Yığının üstü	SI	Açıklama
`CALL pthis` öncesi	FFFEh	0000h	msg1	IP = 010Bh
`CALL pthis` sonrası	**FFFCh**	**010Eh**	msg1	dönüş adresi = dizginin ilk baytı
`POP SI` sonrası	**FFFEh**	0000h	**010Eh**	eski SI CS:temp1'de
0 okununca	FFFEh	0000h	**0123h**	21 bayt okundu, 20 karakter yazıldı
`PUSH SI` sonrası	**FFFCh**	**0123h**	0123h	yeni dönüş adresi
`RET` sonrası	**FFFEh**	0000h	msg1	IP = 0123h: CALL print_num

Tablo 9.2.11: PTHIS çağrısında yığın ve SI (sayı oku ve yazdır programı).

Tablodaki “0000h”, DOS'un .COM programı başlarken yığına koyduğu sözcüktür (Tablo 9.2.11). Kütüphane sürümünde POP SI'den sonra AX de itildiği için yığının iç durumu tabloda gösterilenden bir sözcük daha derindir; tablo yalnız dönüş adresiyle ilgili adımları gösterir.

Aynı fikir PRINT makrosunda da vardır: PRINT de veriyi kodun içine gömer. Fark, PRINT'in verinin üstünden bir JMP ile atlaması, PTHIS'in ise RET'in döneceği adresi değiştirmesidir. Dönüş adresini değiştirmek güçlü ama tehlikeli bir tekniktir. Dizgi 0 ile bitmezse PTHIS dizgiden sonraki komut baytlarını da yazar ve 0 bulduğu rastgele bir yere döner; program kilitlenebilir.

9.2.3.6 GET_STRING ve CLEAR_SCREEN

GET_STRING kullanıcıdan bir satır metin okur. Okunan karakterler DS:DI'nin gösterdiği bellek alanına yazılır. Bu alana **tampon** (*buffer*) denir; programcı onu önceden `DB n DUP(0)` gibi bir tanımla ayırır. DX tamponun bayt cinsinden boyutudur. Prosedür dizginin sonuna 0 koyduğu için en fazla DX − 1 karakter alır. Kullanıcı Enter'a basınca okuma biter.

CLEAR_SCREEN ekranı temizler ve imleci sol üst köşeye, (0, 0)'a götürür. Girdi ve çıktı yazmacı yoktur. Prosedür önce INT 10h'nin 06h hizmetiyle (pencereyi yukarı kaydır) bütün ekranı boş satırlarla doldurur, sonra 02h hizmetiyle imleci (0, 0)'a taşır. Ekranın satır ve sütun sayısını BIOS veri alanından okur: 0040:0084h'deki bayt satır sayısının bir eksiğini, 0040:004Ah'deki sözcük sütun sayısını tutar. Bu yüzden DS'yi geçici olarak 0040h yapar; DS'yi ve kullandığı öteki yazmaçları yığında saklar.

```
LEA  DI, ad          ; tampon
MOV  DX, 20          ; boyut
CALL get_string
CALL clear_screen
LEA  SI, ad
CALL print_string
RET
ad DB 20 DUP(0)
```

Bu program en çok 19 karakterlik bir ad okur, ekranı temizler ve adı sol üst köşeye yazar. GET_STRING'in tamponu 0 ile bitirmesi sayesinde aynı tampon hiçbir dönüşüm yapılmadan PRINT_STRING'e verilir. Programın kullandığı üç prosedür için `DEFINE_GET_STRING`, `DEFINE_CLEAR_SCREEN` ve `DEFINE_PRINT_STRING` satırları RET'ten sonra eklenmelidir.

9.2.3.7 GET_STRING mantığı: kod izleme

Kütüphanedeki GET_STRING geri silmeyi ve dolu tamponu da işlediği için uzundur. Okuma döngüsünün özü aşağıdaki sade eşdeğerle izlenebilir. Tampon 4 bayttır ve kullanıcı “Ece” yazacaktır.

```
LEA  DI, ad      ; tampon adresi
MOV  DX, 4       ; tampon boyutu
oku: CMP DX, 1   ; 0 için 1 bayt kalsın
JBE  bitti
MOV  AH, 01h
INT  21h         ; AL = tuş (yankılı)
CMP  AL, 13      ; Enter mi?
JE   bitti
MOV  [DI], AL    ; tampona yaz
INC  DI
DEC  DX
JMP  oku
bitti: MOV BYTE PTR [DI], 0
RET
ad   DB 4 DUP(?)
```

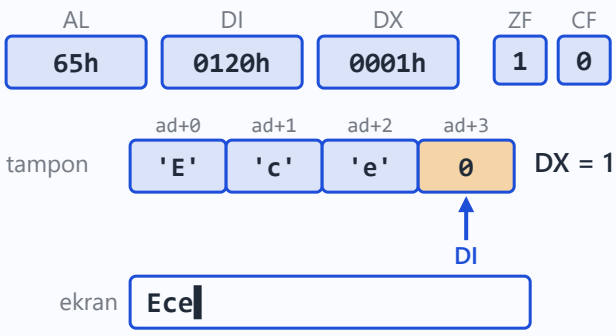
Program 0100h'den başlar. `LEA DI, ad` 3, `MOV DX, 4` 3, `CMP DX, 1` 3 bayttır; döngünün kalanı ve `MOV BYTE PTR [DI], 0` ile `RET` 011Ch'de biter. Tampon 011Dh'den başlar: ad+0 = 011Dh, ad+3 = 0120h.

Döngünün başındaki `CMP DX, 1` / `JBE bitti` çifti kalan yeri denetler. DX kalan bayt sayısıdır. DX ≤ 1 ise yalnız sondaki 0'a yer kalmıştır ve okuma biter. JBE işaretsiz karşılaştırma yapar; DX'in 0 olduğu durumda da atlar. INT 21h'nin 01h hizmeti bir tuş bekler, tuşun ASCII kodunu AL'ye yazar ve karakteri ekrana yankılar. Enter (13) değilse karakter tampona yazılır, DI bir ilerler ve kalan yer DX bir azalır.

Adım	AL	DI	DX	ZF	Tampon (ad+0…3)	Ekran
`LEA DI, ad` / `MOV DX, 4`	00h	**011Dh**	**0004h**	0	? ? ? ?	
`CMP DX, 1` / `JBE` (4 > 1, atlamaz)	00h	011Dh	0004h	0	? ? ? ?	
`INT 21h` ('E'), `CMP AL, 13`	**45h**	011Dh	0004h	0	? ? ? ?	E
`MOV [DI], AL` / `INC DI` / `DEC DX`	45h	**011Eh**	**0003h**	0	E ? ? ?	E
2. tur ('c')	**63h**	**011Fh**	**0002h**	0	E c ? ?	Ec
3. tur ('e')	**65h**	**0120h**	**0001h**	0	E c e ?	Ece
`CMP DX, 1` / `JBE bitti` (atlar)	65h	0120h	0001h	**1**	E c e ?	Ece
`MOV BYTE PTR [DI], 0`	65h	0120h	0001h	1	**E c e 0**	Ece

Tablo 9.2.12: GET_STRING okuma döngüsünün izlemesi (DX = 4, girdi “Ece”).

Dördüncü turun başında DX = 1'dir: yalnız sondaki 0 için yer kalmıştır (Tablo 9.2.12). `CMP DX, 1` eşitlik bulur ve ZF = 1 olur; JBE atlar. Kullanıcı daha fazla yazsa da okunmaz; en fazla DX − 1 = 3 karakter alınmıştır. Tampon 'E', 'c', 'e', 0 olur ve PRINT_STRING ile yazdırılabilir. Sade sürüm tampon dolunca kendiliğinden biter; kütüphanedeki sürüm ise Enter'ı bekler.



Şekil 9.2.8: Döngünün sonunda tampon: 'E', 'c', 'e' ve sondaki 0; DI ad+3'ü (0120h) gösterir.

9.2.3.8 GET_STRING ayrıntıları ve geri silme

Kütüphanedeki GET_STRING sade sürümden dört noktada ayrılır.

- **Yankı denetimi.** Tuş yankısız okunur: INT 16h'nin 00h hizmeti tuşu AL'ye (ASCII) ve AH'ye (tarama kodu) yazar ama ekrana bir şey basmaz. Prosedür yalnız tampona aldığı karakteri 0Eh hizmetiyle ekrana yazar. Böylece kabul edilmeyen tuşlar ekranda iz bırakmaz.
- **Dolu tampon.** DX − 1 karakter alındıktan sonra yeni harfler yok sayılır. Prosedür kendiliğinden bitmez; Enter ya da geri silme bekler. Kullanıcı yanlış yazdığı son harfi silip düzeltebilir.
- **Geri silme (8).** Tamponda harf varsa DI bir geri alınır ve ekrana 8, ' ', 8 yazılır. Tampon boşsa tuş yok sayılır; imleç girdinin başından geri gitmez ve ekrandaki istem silinmez.
- **Enter (13).** Enter tampona yazılmaz. Yerine 0 konur ve prosedür döner. DX ≤ 1 verilirse hiç okuma yapılmaz ve tampona dokunulmaz.

Klavyeden okumanın donanım tarafı da bilinmelidir. Bir tuşa basıldığında klavye denetleyicisi 8259 kesme denetleyicisinin IRQ1 hattını etkinleştirir. İşlemci o anki işini bırakıp BIOS'un INT 09h yordamını çalıştırır. Yordam tuşun tarama kodunu port 60h'den okur, ASCII karşılığını bulur ve ikisini 0040h kesimindeki klavye tamponuna yazar. INT 16h/00h bu tampondan okur; tampon boşsa tuş gelene kadar bekler. GET_STRING'in “beklemesi” bu döngünün içindedir.

GET_STRING döndüğünde imleç girilen metnin hemen sağındadır. Prosedür Enter'ı ekrana yazmaz; sonraki çıktı aynı satırda başlar. Alt satıra geçmek için çağrıdan sonra `PUTC 13` ve `PUTC 10` eklenir.

Geri silmenin mantığı yine sade bir eşdeğerle izlenir. Aşağıdaki kod tuşları yankısız okur, harfleri tampona ekler, geri silmede son harfi tampondan ve ekrandan siler, Enter'da tamponu 0 ile bitirir. Kullanıcı sırasıyla A, b, geri silme, c ve Enter tuşlarına basacaktır.

```
ORG  100h
LEA  DI, ad
oku: MOV AH, 0
INT  16h          ; AL = tuş (yankısız)
CMP  AL, 13       ; Enter mi?
JE   bitti
MOV  AH, 0Eh
CMP  AL, 8        ; geri silme mi?
JNE  ekle
CMP  DI, OFFSET ad
JE   oku          ; silinecek harf yok
DEC  DI
INT  10h          ; 8: imleç sola
MOV  AL, ' '
INT  10h          ; harfin yerine boşluk
MOV  AL, 8
INT  10h          ; yine sola
JMP  oku
ekle: MOV [DI], AL
INC  DI
INT  10h          ; yankı
JMP  oku
bitti: MOV BYTE PTR [DI], 0
RET
ad   DB 6 DUP(?)
```

Kodun iki ayrıntısı dikkat ister. `MOV AH, 0Eh` karşılaştırmalardan önce yapılır; böylece hem yankı hem de silme üçlüsü için AH hazırdır. MOV bayrakları değiştirmediği için ardından gelen CMP'nin sonucu etkilenmez. Geri silme dalındaki `INT 10h`, AL'de hâlâ 8 bulunduğu için imleci sola alır; tuşun kodu doğrudan kontrol karakteri olarak kullanılmıştır.

Olay	AL	DI	İmleç sütunu	Tampon	Ekran
`LEA DI, ad`	00h	ad+0	0	? ? ? ? ? ?	
INT 16h: 'A'; Enter ve geri silme değil	**41h**	ad+0	0	? ? ? ? ? ?	
`ekle` : yaz, ilerle, yankı	41h	**ad+1**	**1**	**A ? ? ? ? ?**	A
'b' için aynı yol	**62h**	**ad+2**	**2**	**A b ? ? ? ?**	Ab
INT 16h: geri silme; `CMP AL, 8` → ZF = 1	**08h**	ad+2	2	A b ? ? ? ?	Ab
`CMP DI, OFFSET ad` (eşit değil) / `DEC DI`	08h	**ad+1**	2	A b ? ? ? ?	Ab
8, ' ', 8 üçlüsü	08h	ad+1	**1**	A b ? ? ? ?	A
'c': b'nin baytına yazılır	**63h**	**ad+2**	**2**	**A c ? ? ? ?**	Ac
Enter: `JE bitti`, 0 yazılır	**0Dh**	ad+2	2	**A c 0 ? ? ?**	Ac

Tablo 9.2.13: Geri silmeyi işleyen okumanın izlemesi (A, b, geri silme, c, Enter).

`DEC DI` b'nin baytını bellekten silmez; o bayt yalnız dizginin dışında kalır ve sonraki harf onun üstüne yazılır (Tablo 9.2.13). Silme işi iki yerde birden yapılır: tamponda DI'yi geri almak, ekranda üçlüyü yazmak. İkisinden biri unutulursa ekran ile tampon birbirini tutmaz. Örneğin üçlü unutulursa ekranda “Abc”, tamponda “Ac” kalır. Kütüphane sürümü aynı girdiyle de tamponu 'A', 'c', 0 yapar; ekrana 'A', 'b', 8, ' ', 8 ve 'c' gönderir ve ekranda “Ac” görünür.

9.2.3.9 Tampon boyu ve taşma

GET_STRING tamponun gerçek boyunu bilmez; yalnız programcının DX'te verdiği sayıya güvenir. DX tamponun gerçek boyundan büyük verilirse prosedür tamponun ardındaki değişkenlerin üstüne yazar.

Örnek 9.2.2

Bellekte `ad DB 4 DUP(0)`, ardından `deger DW 0` ve `son DB 5` tanımlı. Program `MOV DX, 8` ile GET_STRING'i çağırıyor ve kullanıcı “Istanbul” yazıp Enter'a basıyor.

1. DX = 8 olduğu için prosedür en fazla 7 karakter alır: “Istanbu”. Sekizinci harf 'l' yok sayılır.
2. İlk dört karakter “Ista” tamponun dört baytına yazılır.
3. 'n' (6Eh) ve 'b' (62h) deger'in iki baytına yazılır. Sözcük little-endian okunduğu için deger = 626Eh olur.
4. 'u' (75h = 117) son'un üstüne yazılır; son = 117 olur.
5. Sondaki 0, son'dan sonraki bayta yazılır. O bayt da başka bir değişkene ya da koda ait olabilir.

Programda hiçbir hata iletisi çıkmaz. deger ve son sessizce bozulmuştur.

Bu hataya **tampon taşması** (*buffer overflow*) denir. Assembly'de dizi sınırlarını denetleyen bir mekanizma yoktur; her bellek yazımı yalnız adrese bakar. Güvenli kural basittir: DX her zaman tamponun DUP ile ayrılan boyuna eşit verilir. Bunu kendiliğinden sağlamanın bir yolu, tamponun hemen ardına bir etiket koyup `MOV DX, ad_son - ad` yazmaktır; böylece tampon büyütüldüğünde DX de kendiliğinden değişir.

Sık yapılan hata

DX'i “okunacak karakter sayısı” sanmak. DX tamponun boyudur ve sondaki 0'ı da içerir. 10 karakterlik ad okumak için tampon `DB 11 DUP(0)`, DX = 11 olmalıdır. `DB 10 DUP(0)` ve DX = 10 ile ancak 9 karakter alınır.

9.2.4 Sayı prosedürleri

9.2.4.1 SCAN_NUM, PRINT_NUM ve PRINT_NUM_UNS

İşlemci sayıları ikili tutar; kullanıcı ise sayıları onluk rakamlarla yazar ve okur. Klavyeden gelen “307”, üç ayrı ASCII kodudur: 33h, 30h ve 37h. Programın hesap yapabilmesi için bu kodların 0133h ikili sayısına çevrilmesi gerekir. Ekrana sayı yazarken iş tersine döner: 0133h'nin basamakları bulunur ve her biri ASCII koduna çevrilir. Kütüphanenin üç sayı prosedürü bu iki dönüşümü yapar (Tablo 9.2.14).

Prosedür	Ne yapar?	Girdi / çıktı	Tanım
SCAN_NUM	klavyeden çok basamaklı, işaretli bir sayı okur; Enter ile biter	çıktı: CX	`DEFINE_SCAN_NUM`
PRINT_NUM	AX'teki işaretli sayıyı onluk yazar	girdi: AX	`DEFINE_PRINT_NUM` + `DEFINE_PRINT_NUM_UNS`
PRINT_NUM_UNS	AX'teki işaretsiz sayıyı onluk yazar	girdi: AX	`DEFINE_PRINT_NUM_UNS`

Tablo 9.2.14: Sayı prosedürleri.

İki yazdırma prosedürünün farkı, AX'teki 16 bitin nasıl yorumlandığıdır. İşaretsiz yorumda 16 bit 0 … 65.535 aralığını gösterir. İşaretli yorumda ikiye tümleyen gösterimi kullanılır ve aralık −32.768 … 32.767'dir. En anlamlı bit 0 olan sayılarda iki yorum aynıdır. En anlamlı bit 1 olan sayılarda işaretli değer, işaretsiz değerden 65.536 çıkarılarak bulunur.

> **Tanım 9.2.7**
>
> **İkiye tümleyen** (*two's complement*) gösteriminde −x, x'in bütün bitleri ters çevrilip 1 eklenerek yazılır. 16 bitte bu, 65.536 − x'e eşittir. 8086'nın `NEG` komutu bir sayının ikiye tümleyenini alır.

Örneğin FFE7h'nin işaretsiz değeri 65.511'dir. En anlamlı biti 1 olduğu için işaretli değeri 65.511 − 65.536 = −25'tir. Tersinden bakıldığında 25 = 0019h; bitleri ters çevrilince FFE6h, 1 eklenince FFE7h elde edilir. Aynı FFE7h, PRINT_NUM ile “-25”, PRINT_NUM_UNS ile “65511” olarak yazılır. Bitler değişmez, yalnız yorum değişir.

9.2.4.2 SCAN_NUM mantığı: kod izleme

Onluk bir sayıyı soldan sağa okurken her yeni rakam önceki sonucu bir basamak sola kaydırır. “307” için 3 okunur; 0 gelince sonuç 3 × 10 + 0 = 30 olur; 7 gelince 30 × 10 + 7 = 307 olur. Kural kısaca *sonuç ← sonuç × 10 + rakam*'dır. Bu yöntem Horner yöntemi olarak da anılır ve sayının uzunluğunu önceden bilmeyi gerektirmez. SCAN_NUM'ın özü aşağıdaki sade eşdeğerdir; eksi işareti ve taşma denetimi çıkarılmıştır.

```
ORG  100h
XOR  CX, CX      ; sonuç = 0
oku: MOV AH, 01h
INT  21h         ; AL = tuş (yankılı)
CMP  AL, 13      ; Enter mi?
JE   bitti
SUB  AL, '0'     ; ASCII → rakam
CBW              ; AX = rakam
XCHG AX, CX      ; AX=sonuç, CX=rakam
MOV  BX, 10
MUL  BX          ; AX = sonuç × 10
ADD  CX, AX      ; CX = sonuç×10 + rakam
JMP  oku
bitti: RET
```

Kodun her satırı bir amaca hizmet eder. `XOR CX, CX` CX'i sıfırlar; bir yazmacı kendisiyle XOR'lamak `MOV CX, 0`'dan kısadır (2 bayta karşı 3 bayt) ve ZF'yi 1 yapar. `SUB AL, '0'` ASCII kodundan 30h çıkararak rakamın değerini bulur. `CBW` AL'yi işaretli olarak AX'e genişletir; 0–9 arası rakamlarda AH = 0 olur. `XCHG AX, CX` iki yazmacı takas eder: sonuç çarpma için AX'e geçer, rakam CX'te bekler. Takas gereklidir, çünkü `MUL` çarpılanı her zaman AX'te bekler. `MUL BX` DX:AX ← AX × BX işlemini yapar; sonuç 16 bite sığdığı sürece DX = 0 kalır. `ADD CX, AX` çarpımı rakama ekler ve yeni sonucu yine CX'e yazar.

Adım	AX	CX	BX	ZF	CX (onluk)	Ekran
`XOR CX, CX`	0000h	0000h	0000h	1	0	
`MOV AH, 01h` / `INT 21h` : '3'	**0133h**	0000h	0000h	1	0	3
`CMP` / `JE` / `SUB AL, '0'`	**0103h**	0000h	0000h	**0**	0	3
`CBW` / `XCHG AX, CX`	**0000h**	**0003h**	0000h	0	3	3
`MOV BX, 10` / `MUL BX` / `ADD CX, AX`	0000h	0003h	**000Ah**	0	3	3
'0' okunur, `SUB` , `CBW` , `XCHG`	**0003h**	**0000h**	000Ah	1	0	30
`MUL BX` / `ADD CX, AX`	**001Eh**	**001Eh**	000Ah	0	30	30
'7' okunur, `SUB` , `CBW`	**0007h**	001Eh	000Ah	0	30	307
`XCHG` / `MUL BX` / `ADD CX, AX`	**012Ch**	**0133h**	000Ah	0	307	307
Enter: AL = 0Dh, `CMP AL, 13`	**010Dh**	0133h	000Ah	**1**	307	307
`JE bitti` / `RET`	010Dh	0133h	000Ah	1	307	307

Tablo 9.2.15: SCAN_NUM mantığının izlemesi (girdi 307, ardından Enter).

Tablonun birkaç noktası açıklama ister (Tablo 9.2.15). INT 21h/01h AL'ye tuşu yazar ama AH'ye dokunmaz; AH, hizmet numarası olan 01h'de kalır ve AX = 0133h görünür. '0' okunduğunda SUB'ın sonucu 0'dır ve ZF = 1 olur. 30 × 10 = 300 = 012Ch'dir; 7 eklenince CX = 307 = 0133h olur. Enter okunduğunda CMP eşitlik bulur, ZF = 1 olur ve JE döngüden çıkar. Sonuç SCAN_NUM'daki gibi CX'tedir.

Çarpma bu döngünün en pahalı adımıdır. 8086'da `MUL r16` 118–133 saat turu sürer; 5 MHz'de yaklaşık 25 µs. Üç basamaklı bir sayı için üç çarpma yapılır. Bu süre kullanıcının tuşa basma hızı yanında önemsizdir. Yine de 10 ile çarpma kaydırmayla da yapılabilir. x × 10 = x × 8 + x × 2 olduğundan çarpım `SHL` ve `ADD` komutlarıyla çok daha az turda hesaplanır.

9.2.4.3 SCAN_NUM'ın denetimleri

Kütüphanedeki SCAN_NUM, sade eşdeğerin üzerine beş denetim ekler:

- **Eksi işareti.** '-' tuşu bir işaret bayrağı kurar. Enter'a basıldığında bayrak kuruluysa `NEG CX` uygulanır.
- **Rakam olmayan tuş.** Prosedür her tuşu önce ekrana yazar, sonra denetler. Tuş '0'–'9' aralığında değilse ekrandan silinir (8, ' ', 8) ve sayıya katılmaz.
- **Geri silme.** Son basamağı atar: CX ← CX / 10. Ekranda da son karakter silinir.
- **Taşma.** Sayı 16 bite sığmayacaksa son basamak kabul edilmez. Çarpmadan sonra DX ≠ 0 ise ya da toplamada CF = 1 olursa prosedür önceki değeri geri yükler ve son rakamı ekrandan siler.
- **Enter'ın yankısı.** Enter da ekrana yazılır. 13 kodu imleci satır başına döndürür ama alt satıra indirmez.

Son madde uygulamada en çok şaşırtan davranıştır. SCAN_NUM döndüğünde imleç girilen sayının bulunduğu satırın başındadır. Hemen ardından yazılan her şey girilen sayının ve istemin üstüne yazılır. Örnek programlarda SCAN_NUM'dan sonra `PUTC 13`, `PUTC 10` eklenmesi ya da bir sonraki iletinin 13, 10 ile başlaması bu yüzdendir.

Girdi	CX	PRINT_NUM	Açıklama
307	0133h	307	olağan durum
-25	FFE7h	-25	Enter'da NEG: 0019h → FFE7h
12a3	007Bh	123	'a' ekrandan silinir, sayıya katılmaz
40000	9C40h	-25536	16 bite sığar ama işaretli aralığı aşar
70000	1B58h	7000	son 0 kabul edilmez: 7000 × 10 = 70.000 > 65.535
-40000	63C0h	25536	9C40h'nin ikiye tümleyeni 63C0h

Tablo 9.2.16: SCAN_NUM'a verilen girdiler, CX'teki sonuç ve PRINT_NUM'ın yazdığı değer.

Tablodaki son üç satır prosedürün bir sınırını gösterir (Tablo 9.2.16). Basamaklar işaretsiz 16 bit olarak biriktirilir ve işaret en sonda uygulanır. Taşma denetimi yalnız 65.535 sınırına bakar. Bu yüzden 32.768 ile 65.535 arasındaki girdiler hata vermeden kabul edilir ama işaretli yorumda negatif görünür. Örneğin 40.000 = 9C40h'dir; işaretli değeri 40.000 − 65.536 = −25.536'dır. −40.000 girildiğinde 9C40h'nin ikiye tümleyeni 63C0h = 25.536 olur ve eksi işaretli bir girdi pozitif bir sonuç verir. Doğru sonuç için kullanıcının girdisi −32.768 … 32.767 aralığında kalmalıdır.

9.2.4.4 PRINT_NUM_UNS mantığı: kod izleme

Bir sayının onluk basamakları 10'a art arda bölünerek bulunur. Her bölmenin kalanı bir basamaktır, bölüm bir sonraki adımın sayısıdır. Sorun, basamakların sağdan sola, yani ters sırayla çıkmasıdır: 1205'in ilk kalanı 5, son kalanı 1'dir. Ekrana ise önce 1 yazılmalıdır. Yığın bu ters çevirme için doğal bir araçtır: kalanlar sırayla itilir, sonra ters sırayla çekilir.

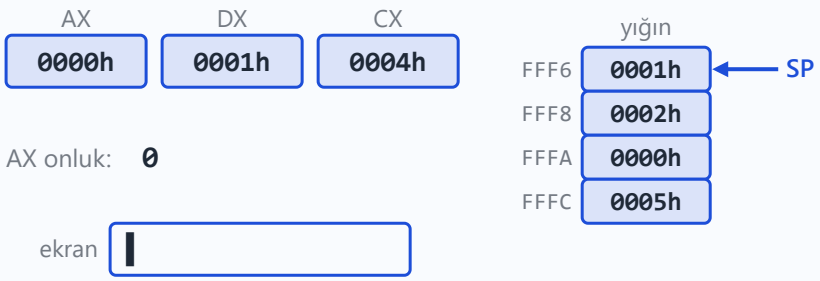
```
ORG  100h
MOV  AX, 1205    ; yazdırılacak sayı
MOV  BX, 10
XOR  CX, CX      ; basamak sayacı
bol: XOR DX, DX
DIV  BX          ; AX=bölüm, DX=kalan
PUSH DX          ; basamak yığına
INC  CX
OR   AX, AX      ; bölüm 0 mı?
JNZ  bol
yaz: POP AX      ; önce en anlamlı
ADD  AL, '0'
MOV  AH, 0Eh
INT  10h
LOOP yaz
RET
```

`DIV BX` 32 bitlik DX:AX'i BX'e böler: bölümü AX'e, kalanı DX'e yazar. Bölmeden önce `XOR DX, DX` zorunludur. DX'te bir önceki kalan durursa bölünen DX:AX çok büyük olur; bölüm 16 bite sığmazsa işlemci 0 tipi kesmeyi (bölme hatası) üretir. CX kaç basamak itildiğini sayar; ikinci döngüde LOOP aynı sayıda POP yapılmasını sağlar. `OR AX, AX` AX'i değiştirmez ama bayrakları kurar: bölüm 0 ise ZF = 1 olur ve döngü biter.

Adım	AX	DX	CX	SP	Yığın (üstten)	Ekran
başlangıç (`MOV`, `MOV`, `XOR`)	04B5h	0000h	0000h	FFFEh	—	
1. bölme: 1205 / 10	**0078h**	**0005h**	**0001h**	**FFFCh**	5	
2. bölme: 120 / 10	**000Ch**	**0000h**	**0002h**	**FFFAh**	0, 5	
3. bölme: 12 / 10	**0001h**	**0002h**	**0003h**	**FFF8h**	2, 0, 5	
4. bölme: 1 / 10	**0000h**	**0001h**	**0004h**	**FFF6h**	1, 2, 0, 5	
`OR AX, AX` → ZF = 1, JNZ atlamaz	0000h	0001h	0004h	FFF6h	1, 2, 0, 5	
1. yazma: POP, '0' ekle, INT 10h, LOOP	**0E31h**	0001h	**0003h**	**FFF8h**	2, 0, 5	1
2. yazma	**0E32h**	0001h	**0002h**	**FFFAh**	0, 5	12
3. yazma	**0E30h**	0001h	**0001h**	**FFFCh**	5	120
4. yazma, LOOP biter	**0E35h**	0001h	**0000h**	**FFFEh**	—	1205

Tablo 9.2.17: Yığınlı basamak ayırmanın izlemesi (AX = 1205 = 04B5h).

Dört bölmeden sonra yığının üstünde en anlamlı basamak, 1 durur (Tablo 9.2.17). Yazma döngüsü basamakları 1, 2, 0, 5 sırasıyla çeker ve yazar. Her POP'tan sonra `ADD AL, '0'` basamağı ASCII'ye çevirir ve `MOV AH, 0Eh` hizmet numarasını yükler; AX'in 0E31h, 0E32h gibi değerleri bundandır. Döngü bittiğinde SP yeniden FFFEh'dir: dört PUSH'a dört POP karşılık gelmiştir. Yığın tablosu DOS'un FFFEh'deki 0000h sözcüğünü göstermez.



Şekil 9.2.9: Bölmeler bittiğinde yığında FFF6h'den FFFCh'ye doğru 1, 2, 0 ve 5 basamakları durur; SP = FFF6h.

9.2.4.5 Kütüphanenin yöntemi: azalan bölen

Kütüphanedeki PRINT_NUM_UNS aynı çıktıyı başka bir yöntemle üretir. Sayıyı hep 10'a bölmek yerine basamak değerlerine böler: önce 10.000'e, sonra 1000'e, 100'e, 10'a ve 1'e. Her bölmenin bölümü bir basamaktır ve hesaplandığı anda yazılır. Kalan bir sonraki turun sayısı olur. Basamaklar soldan sağa çıktığı için yığına ve ters çevirmeye gerek kalmaz.

- Bölen BX 10.000 ile başlar ve her turdan sonra 10'a bölünür. BX 0 olunca yazdırma biter.
- Bölüm (0–9) bir basamaktır; `ADD AL, 30h` ile ASCII'ye çevrilip PUTC ile yazılır. 65.535 ÷ 10.000 = 6 olduğu için bölüm her zaman AL'ye sığar.
- CX bir bayrak olarak kullanılır. CX = 1 iken sayı bölenden küçükse tur atlanır; böylece baştaki sıfırlar yazılmaz. İlk basamak yazılınca CX = 0 olur ve aradaki sıfırlar artık yazılır.
- AX = 0 için ayrı bir dal vardır ve doğrudan '0' yazılır. Bu dal olmasaydı bütün turlar atlanır ve ekrana hiçbir şey yazılmazdı.

Bölen (BX)	Bayrak (CX)	Sayı (AX)	İşlem	Ekran
10.000	1	3007	sayı < bölen ve henüz basamak yok: atla	—
1000	1	3007	3007 ÷ 1000 = 3, kalan 7; CX = 0	3
100	0	7	7 ÷ 100 = 0, kalan 7	30
10	0	7	7 ÷ 10 = 0, kalan 7	300
1	0	7	7 ÷ 1 = 7, kalan 0	3007
0	0	0	bölen 0: bitti	3007

Tablo 9.2.18: Azalan bölenle 3007'nin yazdırılması (her turun başındaki durum).

3007 örneği bayrağın neden gerektiğini açıkça gösterir (Tablo 9.2.18). İlk turda 3007 < 10.000 olduğu için bölüm 0 olurdu; bu sıfır baştaki bir sıfırdır ve yazılmamalıdır. 100'e ve 10'a bölünen turlarda da bölüm 0'dır ama bu sıfırlar sayının içindedir ve yazılmalıdır. Aradaki fark yalnız “ilk basamak yazıldı mı?” sorusuyla ayırt edilir; CX bu bilgiyi tutar. 40.000 için ilk turda 40.000 ÷ 10.000 = 4 yazılır ve CX = 0 olur. Sonraki dört turda dört 0 yazılır; ekranda 40000 görünür.

Böleni küçültmek de bir bölmedir: kütüphane `BX ← BX / 10` işlemini DIV ile yapar. Bu sırada AX'teki sayı yığında saklanır. Prosedür AX, BX, CX ve DX'i girişte iter ve çıkışta geri alır; çağıran için hiçbir yazmaç değişmez.

9.2.4.6 PRINT_NUM ve eksi işareti: kod izleme

PRINT_NUM işaretli sayıyı yazmak için PRINT_NUM_UNS'yi kullanır. Sayı negatifse önce '-' yazılır, sonra sayının mutlak değeri işaretsiz olarak yazdırılır. Aşağıdaki kod kütüphanedeki PRINT_NUM'ın eşdeğeridir:

```
ORG  100h
MOV  AX, -25       ; AX = FFE7h
CALL print_num
RET
print_num PROC
PUSH AX
OR   AX, AX        ; SF = işaret biti
JNS  pozitif
NEG  AX            ; AX = 0019h
PUSH AX            ; PUTC '-' açılımı
MOV  AL, '-'
MOV  AH, 0Eh
INT  10h
POP  AX
pozitif: CALL print_num_uns
POP  AX            ; AX = FFE7h geri
RET
print_num ENDP
```

`MOV AX, -25` satırında assembler −25'in ikiye tümleyenini hesaplar ve AX'e FFE7h yükleyen komutu üretir. `CALL print_num` 0103h'dedir ve 3 bayttır; yığına dönüş adresi 0106h itilir. Prosedür önce AX'i saklar, çünkü NEG AX'i değiştirecektir. `OR AX, AX` sayıyı değiştirmeden bayrakları kurar: en anlamlı bit 1 olduğu için SF = 1 olur. `JNS` (işaret yoksa atla) atlamaz. `NEG AX` 0 − FFE7h = 0019h = 25 hesaplar; sonuç pozitif olduğu için SF = 0 olur. PUTC '-' açılımı eksi işaretini yazar. `CALL print_num_uns` 25'in basamaklarını yazar. Son olarak `POP AX` AX'e FFE7h'yi geri yükler.

Komut(lar)	AX	SP	SF	Yığın (FFFAh, FFFCh)	Ekran
`MOV AX, -25`	**FFE7h**	FFFEh	0	—, —	
`CALL print_num`	FFE7h	**FFFCh**	0	—, 0106h	

Komut(lar)	AX	SP	SF	Yığın (FFFAh, FFFCh)	Ekran
`PUSH AX` / `OR AX, AX`	FFE7h	**FFFAh**	**1**	FFE7h, 0106h	
`JNS` (atlamaz) / `NEG AX`	**0019h**	FFFAh	**0**	FFE7h, 0106h	
PUTC '-' açılımı (5 komut)	0019h	FFFAh	0	FFE7h, 0106h	-
`CALL print_num_uns` (dönüşten sonra)	0019h	FFFAh	0	FFE7h, 0106h	-25
`POP AX`	**FFE7h**	**FFFCh**	0	—, 0106h	-25
`RET` (print_num'dan)	FFE7h	**FFFEh**	0	—, —	-25

Tablo 9.2.19: PRINT_NUM eşdeğerinin izlemesi (AX = −25).

Sayı yazdırıldıktan sonra da AX'te durur (Tablo 9.2.19). Çağıran program aynı sayıyı başka bir işte kullanmaya devam edebilir. PRINT_NUM_UNS kullandığı bütün yazmaçları sakladığı için onun içindeki adımlar tabloda tek satıra indirgenmiştir. Kütüphanedeki PRINT_NUM ayrıca AX = 0 için doğrudan '0' yazan bir dal içerir. İşareti de `OR` yerine `CMP AX, 0` ile sınar; iki komut SF'yi aynı biçimde kurar.

Bu tasarımın bir sonucu dikkat çekicidir: −32.768 (8000h) için NEG yine 8000h üretir, çünkü +32 768 16 bitlik işaretli aralıkta yoktur. PRINT_NUM önce '-' yazar, sonra 8000h'yi PRINT_NUM_UNS'ye verir. PRINT_NUM_UNS 8000h'yi işaretsiz 32.768 olarak yazar ve ekranda doğru sonuç, −32768 görünür. Mutlak değerin işaretsiz yazılması bu uç durumu kendiliğinden çözer.

9.2.4.7 Basamak ayırma: iki yöntemin karşılaştırması

Ölçüt	Yığınlı yöntem (10'a böl, it, ters sırayla çek)	Azalan bölen (10.000, 1000, ...)
Bölme sayısı	basamak sayısı kadar	beşe kadar; ayrıca böleni küçültmek için birer bölme
Yığın kullanımı	basamak başına 2 bayt	yalnız saklanan yazmaçlar
Baştaki sıfırlar	kendiliğinden oluşmaz	bayrakla atlanır
Sabit genişlik (00042)	ek kod gerekir	bayrak kaldırılınca kendiliğinden
Başka taban	bölen 2, 8 ya da 16 yapılır; 16'da 9'dan büyük basamaklar 'A'–'F' olur	bölen dizisi tabana göre değişir (16'da 4096, 256, 16, 1)
Çıkış sırası	önce yığına, sonra ters sırayla ekrana	soldan sağa, hesaplandığı anda

Tablo 9.2.20: Yığınlı yöntem ile azalan bölen yönteminin karşılaştırması.

İki yöntem aynı çıktıyı verir: 1205 için ikisi de 1, 2, 0, 5 basamaklarını yazar (Tablo 9.2.20). Seçimi yığın kullanımı, sabit genişlik ve taban gereksinimi belirler. Süre açısından DIV'in maliyeti belirleyicidir. 8086'da `DIV r16` 144–162 saat turu sürer. 1205 için yığınlı yöntem 4 bölme yapar. Azalan bölen yöntemi 1000, 100, 10 ve 1'e dört bölme, böleni beş kez küçültmek için de beş bölme yapar; ilk turda bölme atlanır. Yığınlı yöntem bu yüzden yaklaşık iki kat hızlıdır. Kütüphanenin seçimi yığın kullanmama ve sabit genişliğe kolay uyarlanma yönünden yapılmıştır.

9.2.4.8 Aynı bitler, iki yorum

AX'e verilen bir değer için iki prosedürün ne yazacağı yan yana konduğunda yorum farkı açıkça görülür (Tablo 9.2.21).

AX	En anlamlı bit	PRINT_NUM	PRINT_NUM_UNS
0000h	0	0	0
7FFFh	0	32767	32767
8000h	1	-32768	32768
9C40h	1	-25536	40000
FFE7h	1	-25	65511
FFFFh	1	-1	65535

Tablo 9.2.21: Aynı 16 bit, iki prosedür.

Kural tektir: en anlamlı bit 0 ise iki prosedür aynı şeyi yazar; 1 ise PRINT_NUM işaretsiz değerin 65.536 eksiğini yazar. Program hangi yorumun doğru olduğunu bilmek zorundadır; işlemci bilmez. Toplama ve çıkarma komutları iki yorum için aynı bitleri üretir; yalnız bayrakların okunuşu değişir. İşaretsiz taşmayı CF, işaretli taşmayı OF gösterir. Bir sayaç, adres ya da uzunluk gibi negatif olamayan değerler PRINT_NUM_UNS ile, sıcaklık ya da fark gibi negatif olabilen değerler PRINT_NUM ile yazılır.

> **Sık yapılan hata**
>
> 32.767'yi aşabilen işaretsiz bir değeri PRINT_NUM ile yazmak. AX = 40.000 (9C40h) PRINT_NUM ile −25536 görünür. Değerin anlamı işaretsizse PRINT_NUM_UNS kullanılmalıdır.

9.2.5 Örnek programlar

9.2.5.1 Sayı oku ve yazdır

emu8086'nın kütüphane belgesindeki ikinci örnek dört prosedürü birlikte kullanır. Program kullanıcıdan bir sayı ister, sayıyı okur ve geri yazar:

```
include 'emu8086.inc'
ORG    100h
LEA    SI, msg1       ; ask for the number
CALL   print_string   ;
CALL   scan_num       ; get number in CX.
MOV    AX, CX         ; copy the number to AX.
CALL   pthis          ; print the following string:
DB  13, 10, 'You have entered: ', 0
CALL   print_num      ; print number in AX.
RET                   ; return to operating system.
msg1   DB  'Enter the number: ', 0
DEFINE_SCAN_NUM
DEFINE_PRINT_STRING
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS  ; required for print_num.
DEFINE_PTHIS
END                   ; directive to stop the compiler.
```

Programın akışı kısadır. `LEA SI, msg1` istem dizgisinin adresini SI'ye yükler ve PRINT_STRING onu yazar. SCAN_NUM sayıyı okuyup CX'e koyar. `MOV AX, CX` sayıyı PRINT_NUM'ın beklediği yazmaca taşır. PTHIS, CALL'ın ardındaki dizgiyi yazar: dizgi 13, 10 ile başladığı için önce alt satıra geçilir. PRINT_NUM sayıyı işaretli olarak yazar ve RET programı bitirir. msg1'den sonraki satırlar veri ve prosedür tanımlarıdır; akış onlara RET'ten sonra hiç ulaşmaz.

Ana programın bellekteki yerleşimi Tablo 9.2.22'dedir. Bütün CALL'lar 3 baytlık yakın çağrılardır (E8 ve 16 bitlik uzaklık). Dizginin kod alanının ortasında durduğuna dikkat edilmelidir: 010Eh–0122h arası komut değil veridir.

Adres	Bayt	Satır
0100h	3	`LEA SI, msg1` (BE 27 01)
0103h	3	`CALL print_string`
0106h	3	`CALL scan_num`
0109h	2	`MOV AX, CX` (8B C1)
010Bh	3	`CALL pthis`
010Eh	21	`DB 13, 10, 'You have entered: ', 0`
0123h	3	`CALL print_num`
0126h	1	`RET` (C3)
0127h	19	`msg1 DB 'Enter the number: ', 0`
013Ah	…	DEFINE_ makrolarıyla açılan prosedürler

Tablo 9.2.22: Sayı oku ve yazdır programının ana bölümünün yerleşimi.

Kullanıcının -25 yazıp Enter'a bastığını varsayalım. İzleme tablosunda CALL satırları için iki an gösterilir: prosedürün içi (yığında dönüş adresi varken) ve dönüş (Tablo 9.2.23).

Satır	SI	CX	AX	SP	Ekran
`LEA SI, msg1`	**0127h**	(boyut)	0000h	FFFEh	
`CALL print_string` (içinde)	0127h	(boyut)	0000h	**FFFCh**	Enter the number:␣
`CALL scan_num` (içinde)	0127h	…	0000h	FFFCh	Enter the number: -25
`CALL scan_num` (dönüş)	0127h	**FFE7h**	0000h	**FFFEh**	Enter the number: -25
`MOV AX, CX`	0127h	FFE7h	**FFE7h**	FFFEh	Enter the number: -25
`CALL pthis`	0127h	FFE7h	FFE7h	**FFFCh**	…
dizgi yazılır	0127h	FFE7h	FFE7h	FFFCh	… / You have entered:␣
`CALL print_num` (içinde)	0127h	FFE7h	FFE7h	FFFCh	… / You have entered: -25
`RET`	0127h	FFE7h	FFE7h	**0000h**	… / You have entered: -25

Tablo 9.2.23: Sayı oku ve yazdır programının izlemesi (girdi −25).

Tablonun birkaç ayrıntısı açıklama ister. SI program boyunca 0127h'de kalır, çünkü PRINT_STRING ve PTHIS SI'yi saklayıp geri yükler. AX de SCAN_NUM'dan sonra 0000h'dir; SCAN_NUM AX'i saklar. Değişen tek yazmaç çıktı yazmacı CX'tir: −25'in ikiye tümleyeni FFE7h. CX'in ilk değeri “(boyut)” olarak gösterilmiştir. emu8086 bir .COM programını yüklerken CX'e programın bayt cinsinden boyutunu koyar. Bu değer SCAN_NUM CX'i sıfırladığında kaybolur ve sonucu etkilemez. Son RET yığından DOS'un koyduğu 0000h'yi IP'ye alır; SP = FFFEh + 2 = 0000h olur. İşlemci PSP'nin başındaki `INT 20h` 'ye gider ve program biter.

Ekranın son durumu iki satırdır. Birinci satırda “Enter the number: -25”, ikincisinde “You have entered: -25” yazar. SCAN_NUM Enter'ı yalnız CR olarak yankıladığı için imleç birinci satırın başına dönmüştü. PTHIS dizgisinin başındaki 13 ve 10 imleci ikinci satırın başına götürür. Dizgi 13, 10 ile başlamasaydı “You have entered: ” birinci satırın üstüne yazılır ve istem silinirdi.

9.2.5.2 İki sayının toplamı

İkinci örnek iki sayı okur ve toplamlarını yazar. İşaretli taşma da denetlenir.

```
include 'emu8086.inc'
ORG  100h
CALL scan_num     ; 1. sayı → CX
MOV  BX, CX       ; sakla
PUTC 13
PUTC 10           ; alt satır
CALL scan_num     ; 2. sayı → CX
PUTC 13
PUTC 10
MOV  AX, BX
ADD  AX, CX       ; işaretli toplam
JO   tasma
CALL print_num
RET
tasma: PUTC '?'
RET
DEFINE_SCAN_NUM
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS
END
```

Programın üç ayrıntısı önemlidir. Birincisi, ikinci SCAN_NUM da sonucu CX'e yazar; birinci sayı kaybolmasın diye önce BX'e alınır. BX'in seçilmesi rastgele değildir: SCAN_NUM BX'i kullanmaz, PUTC yalnız AX'e dokunur. İkincisi, Enter yalnız satır başına döndürdüğü için her SCAN_NUM'dan sonra `PUTC 13` ve `PUTC 10` gelir; ikinci sayı ve sonuç ayrı satırlara yazılır. Üçüncüsü, `JO` işaretli taşmayı yakalar.

Tanım 9.2.8

Taşma bayrağı (OF, *overflow flag*), işaretli bir işlemin sonucu 16 bitlik işaretli aralığa (−32.768 … 32.767) sığmadığında 1 olur. Toplamada iki işlenen aynı işaretli, sonuç ise ters işaretliyse OF = 1'dir. `JO` komutu OF = 1 ise atlar.

Girdiler	BX	CX	AX = BX + CX	OF	Son satır
120, −45	0078h	FFD3h	004Bh	0	75
−7, −8	FFF9h	FFF8h	FFF1h	0	-15
30000, 5000	7530h	1388h	88B8h	**1**	?

Tablo 9.2.24: İki sayının toplamı programı farklı girdilerle.

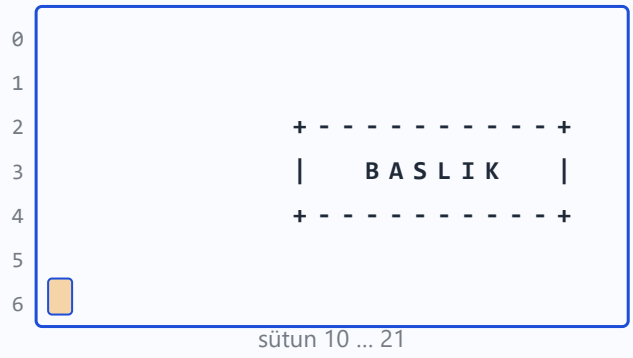
İlk iki satırda sonuç işaretli aralıktadır (Tablo 9.2.24). 120 + (−45) toplamasında CF = 1 olur, çünkü işaretsiz olarak 0078h + FFD3h 16 biti aşar. Bu elde işaretli hesapta anlamsızdır ve OF = 0 kalır. Üçüncü satırda iki pozitif sayının toplamı 35.000'dir ve 32.767'yi aşar. Sonuç 88B8h'nin en anlamlı biti 1'dir; iki pozitif sayının toplamı negatif görünür ve OF = 1 olur. JO atlar ve ekrana '?' yazılır. JO olmasaydı PRINT_NUM 88B8h'yi −30536 olarak yazardı.

9.2.5.3 Çerçeveli başlık

Üçüncü örnek ekranı temizler ve 10. sütundan başlayan çerçeveli bir başlık çizer. Kütüphane makroları programcının kendi prosedürüyle birlikte kullanılır.

```
include 'emu8086.inc'
ORG  100h
CALL clear_screen
GOTOXY 10, 2
CALL cizgi
GOTOXY 10, 3
PRINT '|  BASLIK  |'
GOTOXY 10, 4
CALL cizgi
GOTOXY 0, 6       ; imleç alta
RET
cizgi PROC        ; +----------+
PUTC '+'
MOV  CX, 10
tire: PUTC '-'
LOOP tire
PUTC '+'
RET
cizgi ENDP
DEFINE_CLEAR_SCREEN
END
```

cizgi prosedürü bir '+' yazar, on kez '-' yazar ve bir '+' ile biter: 12 karakterlik “+----------+”. Başlık satırındaki dizgi de 12 karakterdir; iki kenar çizgisi, iki boşluk, “BASLIK” ve iki boşluk. Bu yüzden üç satır 10–21. sütunlarda üst üste hizalanır (Şekil 9.2.10). Son `GOTOXY 0, 6` imleci çerçevenin altına alır; program bittiğinde emu8086'nın yazacağı bir şey başlığın üstüne binmez.



Şekil 9.2.10: Ekranın sol üst bölümü: 2–4. satırlarda, 10. sütundan başlayan çerçeveli BASLIK yazısı; imleç (0, 6)'da.

Programın iki noktası makro ile döngünün ilişkisini gösterir. Döngüdeki `PUTC '-'` kaynakta bir kez yazıldığı için bir kez açılır; 8 baytlık açılım 10 kez çalışır ama 10 kez kopyalanmaz. Makronun kopyalanması derleme anında, döngünün yinelenmesi çalışma anında olur. İkinci nokta yazmaçlarla ilgilidir. LOOP sayacı CX'tedir. PUTC yalnız AX'i kullanıp geri yüklediği için CX'e dokunmaz ve sayaç güvendedir. Döngüde CX'i bozan bir prosedür, örneğin SCAN_NUM çağrılsaydı sayacın PUSH CX / POP CX ile korunması gerekirdi. cizgi'nin tanımı da kütüphane prosedürleri gibi RET'ten sonradır.

9.2.6 Sık yapılan hatalar

Kütüphane kullanılırken yapılan hatalar çoğu zaman derleme sırasında fark edilmez; program çalışır ama yanlış sonuç verir. Aşağıdaki hatalar emu8086 ile çalışan öğrencilerin en sık karşılaştıklarıdır.

Sık yapılan hata

DEFINE_ satırını unutmak. `CALL scan_num` var ama `DEFINE_SCAN_NUM` yoksa assembler etiketi bulamaz ve “undefined” hatası verir. PRINT_NUM için `DEFINE_PRINT_NUM_UNS` da gerekir.

Sık yapılan hata

Dizgiyi 0 ile bitirmemek. PRINT_STRING ve PTHIS 0 görene kadar yazar. `msg DB 'Sayi: '` gibi 0'sız bir tanımda, dizgiden sonra gelen baytlar da ekrana dökülür. Bunlar çoğu zaman prosedür kodlarıdır ve ekranda anlamsız karakterler görünür. PTHIS'te ayrıca dönüş adresi yanlış yere kayar ve program kilitlenebilir.

Sık yapılan hata

Sonucu AX'te aramak. SCAN_NUM sonucu CX'e yazar. PRINT_NUM'dan önce `MOV AX, CX` unutulursa AX'teki eski değer yazılır. Genel kural: bir yazmacın korunduğu prosedürün belgesinde yazmıyorsa o yazmacın korunduğu varsayılmamalıdır.

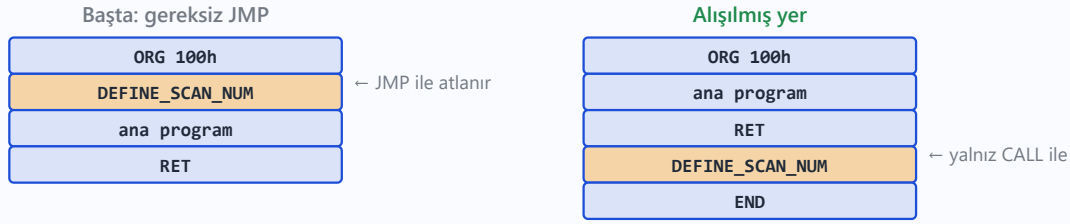
Sık yapılan hata

Enter'dan sonra alt satır beklemek. SCAN_NUM'dan sonra imleç girdinin bulunduğu satırın başındadır. Sonraki çıktı girilen sayının ve istemin üstüne yazılır. “Sayi: ” istemiyle 42 okunup hemen PRINT_NUM çağrılırsa ekranda “42yi: 42” kalır. Araya `PUTC 13`, `PUTC 10` eklenmelidir.

Sık yapılan hata

Tampon boyunu büyük vermek. DX, tamponun gerçek boyundan büyük verilirse GET_STRING tamponun ardındaki değişkenlerin üstüne yazar. Hata ancak o değişkenler kullanıldığında ortaya çıkar.

DEFINE_ satırlarının yeri ise bir hata kaynağı değildir. Her DEFINE_ makrosu prosedür gövdesinin üstünden atlayan bir JMP ile başladığı için satır programın başına konsa da işlemci gövdeye girmez (Şekil 9.2.11). Bedeli fazladan bir atlama komutudur. Alışılmış yer RET ile END arasıdır.



Şekil 9.2.11: DEFINE_SCAN_NUM ORG 100h'nin altına konsa da makronun ilk komutu JMP, prosedür gövdesini atlar (solda). Alışılmış yer RET ile END arasıdır; prosedüre yalnız CALL ile girilir (sağda).

9.2.7 Alıştırmalar ve çözümleri

Alıştırma 1. Ekranın ortasına (sütun 35, satır 12) “MERKEZ” yazan programı makrolarla yazınız.

```
include 'emu8086.inc'
ORG 100h
GOTOXY 35, 12
PRINT 'MERKEZ'
RET
```

“MERKEZ” 12. satırın 35–40. sütunlarına yazılır ve imleç (41, 12)'de kalır. Altı karakterlik bir metnin tam ortada durması için başlangıç sütunu (80 − 6) / 2 = 37 olmalıdır; 35 yaklaşık bir orta noktadır. Ortalamak için genel kural: başlangıç sütunu = (80 − uzunluk) / 2.

Alıştırma 2. Kullanıcıdan iki sayı okuyup toplamını yazdıran programı yazınız.

Çözüm, iki sayının toplamı örneğidir. Gereken parçalar: `CALL scan_num` → `MOV BX, CX` → `PUTC 13`, `PUTC 10` → `CALL scan_num` → `MOV AX, BX` → `ADD AX, CX` → `CALL print_num`; tanımlar `DEFINE_SCAN_NUM`, `DEFINE_PRINT_NUM` ve `DEFINE_PRINT_NUM_UNS`. PUTC 13 ve PUTC 10 unutulursa ikinci sayı birincinin üstüne yazılır ve ekranda yalnız son yazılanlar okunur.

Alıştırma 3. PRINT makrosu 10 kez kullanılırsa dizgiler ve döngü kodu kaç kez bulunur? PRINT_STRING ile karşılaştırınız.

PRINT ile 10 kopya oluşur: her açılım kendi dizgisini ve kendi 23 baytlık döngü kodunu taşır. PRINT_STRING ile döngü kodu bir kez bulunur; bellekte 10 dizgi ve her dizgi için bir `LEA SI` ile bir `CALL` (3 + 3 bayt) yer alır. Yedi karakterlik on iletiyle PRINT'li program yaklaşık 310, PRINT_STRING'li program yaklaşık 160 bayttır. İleti sayısı arttıkça fark büyür; buna karşılık her CALL/RET çifti 27 saat turu ekler.

Alıştırma 4. AX = 8000h iken PRINT_NUM ve PRINT_NUM_UNS ne yazar?

8000h'nin işaretsiz değeri 32.768'dir; PRINT_NUM_UNS “32768” yazar. En anlamlı biti 1 olduğu için işaretli değeri 32.768 − 65.536 = −32.768'dir; PRINT_NUM “-32768” yazar. Bu, 16 bitlik işaretli aralığın en küçük sayısıdır ve NEG onu değiştirmez; PRINT_NUM yine de doğru yazar, çünkü mutlak değeri işaretsiz yazdırır.

Alıştırma 5. `DEFINE_PTHIS`, `ORG 100h`'nin hemen altına yazılırsa ne olur?

Program yine doğru çalışır. Makronun ilk komutu `JMP skip_proc_pthis`'tir; işlemci PTHIS'in gövdesinin üstünden atlar ve ana programın ilk komutuna gelir. Bedeli fazladan bir JMP'dir ve ana programın adresleri gövdenin boyu kadar ileri kayar. Koruyucu JMP olmasaydı işlemci doğrudan PTHIS'in koduna girerdi: `POP SI` yığından DOS'un 0000h sözcüğünü alır, PSP'nin baytlarını dizgi diye yazar ve geçersiz bir adrese dönerdi.

Alıştırma 6. Kullanıcının adını GET_STRING ile alıp “Merhaba, ad!” yazdırınız.

```
include 'emu8086.inc'
ORG 100h
LEA  DI, ad
MOV  DX, 20
CALL get_string
PUTC 13
PUTC 10
PRINT 'Merhaba, '
LEA  SI, ad
CALL print_string
PUTC '!'
RET
ad DB 20 DUP(0)
DEFINE_GET_STRING
DEFINE_PRINT_STRING
END
```

Kullanıcı “Ece” yazıp Enter'a basarsa ekranın ilk satırında “Ece”, ikincisinde “Merhaba, Ece!” görünür. PUTC 13 ve PUTC 10 olmasaydı GET_STRING'den sonra imleç “Ece”nin hemen sağında kalır ve ekranda tek satırda “EceMerhaba, Ece!” görünürdü. Tampon 20 bayttır ve DX = 20; en çok 19 karakterlik bir ad alınır.

9.2.8 Özet

- **emu8086.inc**, ekran, klavye ve sayı işleri için hazır makrolar ve prosedürler içeren kütüphanedir. İçindeki her şey BIOS ve DOS kesmeleri üzerine kurulu sıradan 8086 kodudur.
- **INCLUDE** bir direktiftir: dosyanın metnini derleme sırasında satırın yerine koyar, kod üretmez. Dosya önce kaynak klasöründe, sonra emu8086'nın Inc klasöründe aranır. Satır kaynağın başına yazılır.
- Makro tanımı ve DEFINE_ yazılmamış prosedür programa bayt eklemez. Program boyutunu kullanılan makroların her kullanımı ve DEFINE_ ile tanımlanan prosedürler belirler.
- **Ekran makroları** INT 10h üzerine kuruludur: PUTC ve PRINT 0Eh, GOTOXY 02h, CURSOROFF ve CURSORON 01h hizmetini kullanır. Makrolar kullandıkları yazmaçları saklar; PUTC 8 bayt, GOTOXY 16 bayttır.
- **GOTOXY sütun, satır** sırasını kullanır; BIOS satırı DH'de, sütunu DL'de bekler. Sol üst (0, 0), sağ alt (79, 24)'tür. Hücrenin video belleğindeki adresi B800h:(satır × 80 + sütun) × 2'dir.
- **PRINT** dizgiyi koda gömer ve üstünden JMP ile atlar; her kullanımda döngü kodu kopyalanır. **PRINTN** sona 13, 10 ekler. Yeni satır için CR (13) ve LF (10) birlikte gerekir. 79. sütundan sonra imleç alt satıra geçer, son satırda ekran kayar.
- **Prosedürler** DEFINE_ ile bir kez tanımlanır, CALL ile çağrılır. Her DEFINE_ makrosu gövdenin üstünden atlayan bir JMP ile başlar; alışılmış yer RET ile END arasıdır. PRINT_NUM, PRINT_NUM_UNS'yi de gerektirir.
- Girdi ve çıktı yazmaçları: PRINT_STRING DS:SI, GET_STRING DS:DI ve DX, SCAN_NUM → CX, PRINT_NUM ve PRINT_NUM_UNS ← AX. Öteki yazmaçlar korunur, bayraklar korunmaz.
- **PTHIS**, CALL'ın yığına ittiği dönüş adresini dizginin adresi olarak kullanır; dizgiyi yazdıktan sonra adresi 0'ın ötesine taşır ve RET dizgiden sonraki komuta döner.
- **GET_STRING** en fazla DX − 1 karakter okur, tamponu 0 ile bitirir, geri silmeyi işler ve Enter'ı yazmaz. DX tamponun gerçek boyundan büyük verilirse tamponun ardındaki değişkenler ezilir.
- **SCAN_NUM** rakamları sonuç ← sonuç × 10 + rakam kuralıyla biriktirir; eksi işaretini en sonda NEG ile uygular, rakam olmayan tuşları ve 65.535'i aşan basamakları reddeder. Enter yalnız satır başına döndürür.
- **PRINT_NUM_UNS** basamakları azalan bölenle (10.000, 1000, …) soldan sağa yazar; baştaki sıfırları bir bayrakla atlar. Yığınlı yöntem aynı sonucu 10'a bölüp basamakları yığından ters sırayla çekerek verir. **PRINT_NUM** negatif sayıda '-' yazıp mutlak değeri işaretsiz yazdırır.
- Aynı 16 bit iki biçimde okunur: en anlamlı bit 1 ise işaretli değer = işaretsiz değer − 65.536 (FFE7h = −25 = 65.511).

9.2.9 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

kütüphane *library* · direktif *directive* · makro *macro* · makro genişletme *macro expansion* · prosedür *procedure* · prosedür çağrısı *procedure call* · dönüş adresi *return address* · parametre geçirme *parameter passing* · yazmaçların korunması *register preservation* · imleç *cursor* · dizgi *string* · kontrol karakteri *control character* · ASCII · BIOS · hizmet *service* · video belleği (metin modu) *video memory* · klavye tamponu *keyboard buffer* · işaretli / işaretsiz sayı *signed / unsigned number* · ikiye tümleyen *two's complement* · etiket *label*

9.2.10 Kendini Dene

1. `GOTOXY 3, 7` açılımında INT 10h çağrılmadan hemen önce AH, BH, DH ve DL nedir? Açılım kaç bayttır ve hangi yazmaçları saklar?

   Cevap

   AH = 02h, BH = 0, DH = 7 (satır), DL = 3 (sütun). Açılım 16 bayttır: üç PUSH ve üç POP 6 bayt, dört `MOV r8, sabit` 8 bayt, `INT 10h` 2 bayt. AX, BX ve DX yığında saklanıp ters sırayla geri alınır.

2. Renkli metin modunda (35, 12) hücresindeki karakter video belleğinin hangi bağıl konumunda durur? Fiziksel adresi nedir?

   Cevap

   (12 × 80 + 35) × 2 = 995 × 2 = 1990 = 07C6h. Karakter B800:07C6h'de, öznitelik baytı B800:07C7h'dedir. Fiziksel adres B8000h + 07C6h = B87C6h'dir.

3. Boş bir ekranda imleç (0, 0)'da iken `PRINTN 'Selam'` ve ardından `PUTC 'X'` çalışıyor. X hangi hücreye yazılır? İmleç sonunda nerededir?

   Cevap

   “Selam” 0. satırın 0–4. sütunlarına yazılır. PRINTN'in eklediği 13 imleci (0, 0)'a, 10 ise (0, 1)'e götürür. X (0, 1)'e yazılır ve imleç (1, 1)'e geçer.

4. Kullanıcı SCAN_NUM çalışırken sırasıyla 5, b, 0 tuşlarına ve Enter'a basıyor. Prosedür döndüğünde CX nedir? Ekranda ne kalır?

   Cevap

   'b' rakam olmadığı için ekrana yazılır, sonra 8, ' ', 8 ile silinir ve sayıya katılmaz. Sayı 5 × 10 + 0 = 50 olur: CX = 0032h. Ekranda “50” kalır; Enter imleci satırın başına döndürür.

5. Yığınlı basamak ayırma kodu (10'a böl, it, LOOP ile çek) AX = 0 için ne yazar? Kütüphanedeki azalan bölen yöntemi neden AX = 0 için ayrı bir dal ister?

   Cevap

   Yığınlı kod döngüye en az bir kez girer: 0 / 10 = 0, kalan 0 yığına itilir ve CX = 1 olur. OR AX, AX ZF'yi 1 yapar, yazma döngüsü bir kez döner ve “0” yazılır. Azalan bölen yönteminde bayrak CX = 1 iken sayı bölenden küçükse tur atlanır. AX = 0 her bölenden küçük olduğu için bütün turlar atlanır ve hiçbir şey yazılmaz; ayrı dal bu yüzden '0' yazar.

6. GET_STRING, `MOV DX, 1` ile çağrılıyor ve kullanıcı “abc” yazıp Enter'a basıyor. Tampona ne yazılır?

   Cevap

   Hiçbir şey. DX ≤ 1 iken tamponda yalnız sondaki 0'a yer vardır ve prosedür hiç okuma yapmadan döner; tampona 0 da yazılmaz, eski içerik olduğu gibi kalır. Basılan tuşlar klavye tamponunda bekler ve sonraki okuma hizmeti onları alır.

7. Bir .COM programında `CALL pthis` 0100h'de, hemen ardında `DB 'Tamam', 13, 10, 0` var. CALL yığına hangi adresi iter? PTHIS'in RET'i hangi adrese döner?

   Cevap

   CALL 3 bayttır; yığına 0103h itilir, bu dizginin ilk baytı 'T'nin adresidir. Dizgi 5 + 2 + 1 = 8 bayttır ve 0103h–010Ah'yi kaplar. PTHIS 0'ı 010Ah'de okuduğunda SI = 010Bh olur; bu değer yığına konur ve RET 010Bh'ye döner.

8. AX = C350h iken PRINT_NUM ve PRINT_NUM_UNS ne yazar?

   Cevap

   C350h = 50.000'dir; PRINT_NUM_UNS “50000” yazar. En anlamlı bit 1 olduğu için işaretli değer 50.000 − 65.536 = −15.536'dır; PRINT_NUM “-15536” yazar.

9.2.11 İleri okuma

- emu8086 kurulumundaki `Inc/emu8086.inc` dosyası: kütüphanedeki bütün makroların ve prosedürlerin kaynak kodu. SCAN_NUM'ın taşma denetimi ve PRINT_NUM_UNS'nin azalan bölen döngüsü burada satır satır okunabilir.
- emu8086 yardım belgeleri, “emu8086.inc — library of common functions” ve “Macros” sayfaları: kütüphanenin kullanım örnekleri ve makroların genel kuralları.

- B. B. Brey, *The Intel Microprocessors* (8. baskı): prosedürler, makrolar ve BIOS video hizmetleri; onluk sayıların ikiliye ve ikiliden onluğa dönüştürülmesi.
- R. Brown, “Ralf Brown's Interrupt List”: INT 10h, INT 16h ve INT 21h hizmetlerinin girdi ve çıktı yazmaçlarıyla ayrıntılı listesi.

9.2.12 Kaynaklar

1. emu8086 yardım belgeleri: “emu8086.inc — library of common functions” ve “Macros” sayfaları.
2. emu8086 kurulumundaki Inc/emu8086.inc dosyası (makro ve prosedürlerin kaynak kodu).
3. B. B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson, 2009 — prosedürler ve makrolar.
4. A. S. Tanenbaum, *Modern Operating Systems*, 3. baskı, Pearson, 2008.
5. Intel, *The 8086 Family User's Manual*, 1979 — CALL, RET, MUL, DIV, AAM komutlarının tanımları ve saat turu sayıları.

KISIM IV

Donanım ve Komut Kodlama

Bölüm 10–11

KISIM IV · BÖLÜM 10

8086 Donanımı ve Aygıt Kontrolü

8086, 40 pinli bir DIP paket içinde adres, veri, kontrol, durum ve kesme sinyallerini taşır. Bölümde adres ve veri hatlarının çoklanması, ALE ve adres kilitleri, BHE' ile bellek bankaları, T1–T4 durumlarından oluşan okuma ve yazma döngüleri, minimum ve maksimum mod ile 8284A ve 8288 yongaları anlatılır. Aygıt kontrolü kısmında IN ve OUT komutları, yoklama, INT 15h ile bekleme ve yazılımla gecikme işlenir. Trafik ışıkları, step motor, LED ekran, termometre, yazıcı ve robot gibi emu8086 aygıtlarının programlanmasıyla bölüm tamamlanır.

Kod (8086): `github.com/sercankulcu/microprocessors-8086/Bolum10`

BU BÖLÜMÜN NOTLARI

10.1 8086 Pinleri

10.2 Aygıt Kontrolü

KISIM IV · BÖLÜM 10.1

8086 Pinleri

Bir assembly komutunun yazmaçlar ve bayraklar üzerindeki etkisi programcının gördüğü yüzdür. Aynı komut, yonganın dışında da bir iz bırakır: pinlerde belirli bir sırayla adres, veri ve kontrol sinyalleri belirir. Bu not, 8086'yı bir devre elemanı olarak ele alır. 40 pinli kılıfın pinlerini gruplarına ayırır ve adres ile verinin aynı pinlerden nasıl geçtiğini anlatır. Çoklamayı çözen adres kilitlerini ve veri alıcı-vericilerini, çift ve tek bellek bankalarını, okuma ve yazma yol döngülerini saat turu düzeyinde inceler. Yavaş belleğin bekleme durumu istemesini, yolun başka bir yöneticiye devredilmesini, saat ve sıfırlama devresini, maksimum moddaki 8288 yol denetleyicisini ve kesme pinlerini tamamlar. Her konuda bir komutun hangi pinde hangi değeri ürettiği kod izlemeleriyle gösterilir.

ÖĞRENME ÇIKTILARI

Pinleri tanımak
8086'nın 40 pinini gruplarına ayırmak; her pinin yönünü, aktif düzeyini ve hangi T durumunda ne taşıdığını söylemek; minimum ve maksimum modda değişen sekiz pini eşleştirmek.

Çoklamayı çözmek
Adres ve verinin aynı pinlerden geçmesinin nedenini açıklamak; ALE, adres kilidi ve veri alıcı-vericisiyle ayrık adres ve veri yolunun nasıl kurulduğunu çizmek.

Bankaları seçmek
BHE' ve A0 ile çift ve tek bankanın seçimini açıklamak; bir komutun kaç veri yol döngüsü gerektirdiğini ve verinin hangi hatlardan geçtiğini bulmak.

Yol döngüsünü izlemek
Okuma ve yazma döngüsünü T1–T4 durumlarında sinyal sinyal izlemek; READY ile bekleme durumu eklenmesini açıklamak ve bir bellek için gereken bekleme durumu sayısını hesaplamak.

Durum ve kontrol pinlerini okumak
S2'–S0', S3–S6 ve QS1–QS0 kodlarını çözmek; RD', WR', M/IO', DT/R', DEN', HOLD/HLDA, RQ'/GT ve LOCK' sinyallerinin bir komut sırasında aldığı değerleri vermek.

Sistemi kurmak
8284A saat üretecinin, RESET sonrası başlangıcın, 8288 yol denetleyicisinin ve INTR, NMI, INTA' pinlerinin bir 8086 sistemindeki görevini açıklamak.

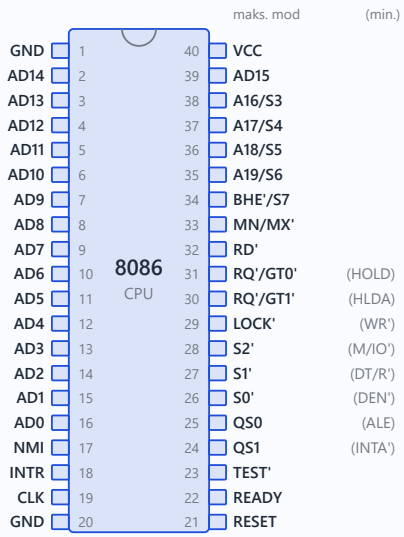
10.1.1 8086'nın kılıfı ve pinleri

Bir mikroişlemci, dış dünyayla yalnız pinleri üzerinden konuşur. Bellekten bir bayt okumak, bir porta değer yazmak, kesme isteği almak ya da saat sinyalini içeri almak için ayrı ayrı elektriksel bağlantılar gerekir. 8086'nın bu bağlantıları için ayrılan yer çok dardır: bütün işlemci 40 pinli bir kılıfa sığdırılmıştır. Bu kesim kılıfı, pinlerin adlandırılmasını ve pin gruplarını tanıtır. Ardından 40 pinin neden yetmediğini gösterir. Intel bu darlığı iki yolla aştı: aynı pini farklı zamanlarda farklı amaçla kullandı ve bir pin grubunun anlamını sistem türüne bağladı.

10.1.1.1 40 pinli DIP kılıf

Intel 8086, HMOS (*high-performance MOS*, yüksek başarımlı n kanallı MOS) teknolojisiyle üretilmiş 16 bitlik bir mikroişlemcidir. Yonga, **çift sıralı kılıf** (DIP, *dual in-line package*) denen dikdörtgen bir paketin içindedir. Paketin iki uzun kenarında yirmişer pin bulunur. Pinler sol üst köşeden başlayıp saat yönünün tersine numaralanır: sol kenarda yukarıdan aşağı 1–20, sağ kenarda aşağıdan yukarı 21–40. Paketin üst kenarındaki yarım daire biçimli çentik, 1 numaralı pinin yerini gösterir.

8086 tek bir +5 V besleme gerilimiyle çalışır. Besleme pin 40'a (V_{CC}), toprak pin 1 ile pin 20'ye (GND) bağlanır. Veri sayfası beslemeye ±%10 tolerans tanır: 4,5 V ile 5,5 V arasında yonga doğru çalışır. 5 MHz'lik 8086 en çok 340 mA akım çeker; bu, 5 V × 0,34 A = 1,7 W güç demektir. Bugünkü ölçülerle küçük görünen bu güç, 1978'de yonganın ısınması için ciddi bir değerdi.



Şekil 10.1.1: 8086'nın pin diyagramı. Sol kenarda 1–20, sağ kenarda 21–40 numaralı pinler bulunur. 24–31 numaralı pinlerin iki adı vardır: dışta maksimum moddaki, parantez içinde minimum moddaki ad.

Şekil 10.1.1 pinlerin kılıf üzerindeki yerini gösterir. İlk bakışta göze çarpan şey, birçok pinin bölü işaretiyle ayrılmış iki adı olmasıdır: A17/S4, BHE'/S7, RQ'/GT0' gibi. Bu çift adlar, pinin zaman içinde ya da sistem türüne göre iki farklı iş yaptığını anlatır.

Tanım 10.1.1

Mantık 1 düzeyindeyken (yaklaşık +5 V) görevini yapan sinyal **aktif yüksek** (*active high*) bir sinyaldir. Mantık 0 düzeyindeyken (yaklaşık 0 V) görevini yapan sinyal ise **aktif düşüktür** (*active low*). Veri sayfası aktif düşük sinyallerin adının üstüne çizgi çeker. Bu derste aynı anlam adın sonundaki kesme işaretiyle gösterilir: RD' = 0 "okuma yapılıyor" demektir.

Aktif düşük sinyaller TTL çağının bir alışkanlığıdır. TTL kapılarında bir hattı 0'a çekmek, 1'e çekmekten daha güçlü yapılabilir. Ayrıca bağlantısı kopmuş bir TTL girişi 1 gibi davranır; böylece kopuk bir "aktif düşük" hat kendiliğinden pasif kalır. Bu yüzden okuma, yazma, kesme kabul ve seçim sinyallerinin çoğu aktif düşüktür.

Bazı adlar iki durumu birden anlatır. M/IO' adı okunurken bölü işareti "ya da" gibi düşünülür: pin 1 ise birinci anlam (M, bellek), 0 ise ikinci anlam (IO', giriş/çıkış) geçerlidir. MN/MX', DT/R' ve S7 ile birlikte okunan BHE' de bu kalıptadır. Pinlerin yönü de üç türlüdür. **Giriş** (G) pini yalnız işlemciye bilgi getirir, **çıkış** (Ç) pini yalnız dışarı bilgi götürür. **Çift yönlü** (G/Ç) pin iki yönde de çalışır.

Sık yapılan hata

DT/R' sinyalini "DT/R tümüyle aktif düşük" diye okumak yanlıştır. Kesme işareti yalnız R'ye aittir: pin 1 iken DT (veri gönder), 0 iken R (veri al) anlamı geçerlidir. Aynı biçimde M/IO' sinyalinde M aktif yüksek, IO aktif düşüktür. Veri sayfasında üst çizgi yalnız adın ilgili parçasının üstündedir.

10.1.1.2 Yonganın içi

Plastik ya da seramik paketin ortasında birkaç milimetre karelik bir silikon parçası, yani **kalıp** (*die*) bulunur. Kalıbın kenarlarında **bağlantı pedleri** sıralanır. Her ped, saç telinden ince bir altın ya da alüminyum telle paketin bir pinine bağlanır. Paketin asıl işi bu küçük kalıbı korumak, ısısını dağıtmak ve tellerini baskılı devre kartına uygun aralıklı pinlere taşımaktır.



Şekil 10.1.2: 8086'nın kalıp fotoğrafı (Ken Shirriff, righto.com). Sol kenarda adres/veri pedleri, solda 16 bitlik yazmaçlar, ALU ve kuyruk yazmaçları, sağ altta mikrokod ROM'u görülür.

Şekil 10.1.2, işlevsel blokları işaretlenmiş bir 8086 kalıbını gösterir. Kalıpta yaklaşık 29.000 transistör vardır; üretim süreci 3 µm'dir. Sol kenardaki pedler AD pinlerine giden hatlardır. Yazmaçlar, ALU ve önden getirme kuyruğunun yazmaçları bu pedlerin yakınındadır, çünkü bu bloklar yolla en sık veri alışverişi yapan bölümlerdir. Sağ alttaki geniş düzenli alan mikrokod ROM'udur: karmaşık komutların adımlarını tutan iç bellek. Yol kontrol mantığı, pinlerdeki sinyallerin zamanlamasını üretir.

Kalıp fotoğrafı bir şeyi açıkça gösterir: işlemcinin içinde bellek yoktur. Yazmaçlar dışında bütün program ve veri dışarıdaki yongalardadır. İşlemcinin her bellek erişimi pinlerden geçmek zorundadır. Bu yüzden pinlerin sayısı ve zamanlaması, işlemcinin hızını doğrudan sınırlar.

10.1.1.3 Neden karmaşık pinler?

Bir mikroişlemci bellek ve G/Ç aygıtlarına adres yolu, veri yolu ve kontrol yolu üzerinden bağlanır. Her sinyale ayrı bir pin verilseydi 8086'nın ne kadar pine ihtiyacı olurdu? Kaba bir sayım şöyledir:

- Adres yolu 20 hat: A0–A19. 2^{20} = 1 MB bellek adreslenir.
- Veri yolu 16 hat: D0–D15.
- Kontrol, durum ve kesme sinyalleri yaklaşık 20 hat: okuma, yazma, bellek/G-Ç seçimi, hazır, kesme isteği, yol isteği, durum bilgisi.
- Güç ve saat 4 hat: V_{CC}, iki GND ve CLK.

Toplam 60'ı aşar. 1970'lerin sonunda 64 pinli paketler pahalıydı, kartta fazla yer kaplıyordu ve üretimde sorun çıkarıyordu. Intel, 8080 ve 8085'ten tanıdık olan 40 pinli kılıfta kalmayı seçti. Aradaki farkı kapatmanın tek yolu, bazı pinlere birden fazla iş yaptırmaktı.



Şekil 10.1.3: Ayrı pinlerle gereken sinyal sayısı 60 civarındadır; kılıfta 40 pin vardır. Fark çoklamayla kapatılır.

Tanım 10.1.2

Çoklama (*multiplexing*), aynı pinlerin farklı zamanlarda farklı bilgileri taşımasıdır. 8086'da zaman paylaşımlı çoklama kullanılır: AD0–AD15 pinleri bir yol döngüsünün ilk saat turunda adresin alt 16 bitini, sonraki turlarda veriyi taşır. A16–A19 pinleri ilk turda adresin üst 4 bitini, sonra durum bilgisini taşır.

8086'nın pin darlığının ikinci bir kaynağı da vardır. Kesim yazmaçlarıyla üretilen fiziksel adres 20 bittir, veri yolu ise 16 bittir. Adresin alt 16 biti veri hatlarıyla paylaşılabilir, ama fazladan 4 adres bitinin (A16–A19) eşleşecek veri hattı yoktur. Bu dört pin de boş kalmasın diye adres zamanı geçince durum bilgisi (S3–S6) taşır.

Çoklamanın iki bedeli vardır. Birincisi, sistemin geri kalanı ayrık ve kararlı bir adres yolu ister; bu yüzden işlemcinin dışına adresi yakalayıp tutan kilit devreleri eklenir. İkincisi, adres ve verinin aynı pinlerden sırayla geçmesi her bellek erişimini en az dört saat turuna yayar. Aynı teknolojiyle 60 pinli ayrık yollu bir işlemci daha az dış devreyle kurulabilirdi. Ama 40 pinli kılıf ucuzdu ve 8085 sistemlerinden gelen tasarımcılar çoklamalı yolu zaten tanıyordu.

10.1.1.4 Pin grupları

40 pin, görevlerine göre yedi gruba ayrılabilir. Tablo 10.1.1 bu grupları pin numaralarıyla verir.

Grup	Pinler	Açıklama
Güç	V_{CC} (40), GND (1, 20)	+5 V ± %10
Saat ve sıfırlama	CLK (19), RESET (21), READY (22)	üçü de 8284A saat üretecinden gelir

Grup	Pinler	Açıklama
Adres/veri	AD0–AD15 (2–16, 39)	T1'de adres, sonra veri
Adres/durum	A16/S3–A19/S6 (38–35), BHE'/S7 (34)	T1'de adres, sonra durum
Kesme	NMI (17), INTR (18), INTA' (24, minimum mod)	donanım kesmeleri
Ortak kontrol	RD' (32), TEST' (23), MN/MX' (33)	iki modda da aynı
Moda bağlı	24–31	minimum: INTA', ALE, DEN', DT/R', M/IO', WR', HLDA, HOLD; maksimum: QS1, QS0, S0', S1', S2', LOCK', RQ'/GT1', RQ'/GT0'

Tablo 10.1.1: 8086'nın pin grupları.

Adres/veri pinlerinin numaraları bir bakışta düzensiz görünür. AD15 pin 39'dadır; AD14'ten AD0'a kadar olan pinler ise 2'den 16'ya sıralanır. A19/S6 ile A16/S3 arasındaki dört pin, sağ kenarda 35–38 numaralarda ters sırayla durur. Bu yerleşim kalıptaki pedlerin sırasından gelir. Şekil 10.1.2'deki AD pedleri kalıbın sol kenarında art arda dizilir. En kısa tellerle paketin sol kenarındaki pinlere bağlanırlar.

Bir sinyalin yönü de grubuna göre değişir. AD0–AD15 çift yönlüdür: yazmada çıkış, okumada giriş olur. A16/S3–A19/S6 ve BHE'/S7 yalnız çıkıştır, çünkü bu pinleri hiçbir dış aygıt sürmez. CLK, RESET, READY, NMI, INTR, TEST' ve MN/MX' yalnız giriştir. RD', ALE, DEN', DT/R', M/IO', WR', INTA' ve HLDA çıkıştır; HOLD giriştir. Maksimum moddaki RQ'/GT0' ve RQ'/GT1' ise tek pinde iki yönlü bir el sıkışma yürütür.

Pin	Ad	Yön	Görev
1, 20	GND	—	toprak
2–16	AD14–AD0	G/Ç	adres (T1) / veri (T2–T4)
17	NMI	G	maskelenemeyen kesme, yükselen kenar
18	INTR	G	kesme isteği, aktif yüksek düzey
19	CLK	G	saat, %33 görev çevrimi
21	RESET	G	sıfırlama, aktif yüksek
22	READY	G	bellek/G-Ç hazır, aktif yüksek
23	TEST'	G	WAIT komutunun sınadığı giriş
24	INTA' / QS1	Ç	kesme kabul / kuyruk durumu
25	ALE / QS0	Ç	adres kilidi etkinleştirme / kuyruk durumu
26	DEN' / S0'	Ç	veri etkinleştirme / durum
27	DT/R' / S1'	Ç	veri gönder/al / durum
28	M/IO' / S2'	Ç	bellek ya da G/Ç / durum
29	WR' / LOCK'	Ç	yazma / yol kilidi
30	HLDA / RQ'/GT1'	Ç / G/Ç	yol devri onayı / yol isteği-bağışı

Pin	Ad	Yön	Görev
31	HOLD / RQ'/GT0'	G / G/Ç	yol devri isteği / yol isteği-bağışı
32	RD'	Ç	okuma
33	MN/MX'	G	mod seçimi
34	BHE'/S7	Ç	yüksek bayt etkin / yedek durum biti
35–38	A19/S6–A16/S3	Ç	adres (T1) / durum (T2–T4)
39	AD15	G/Ç	adres (T1) / veri (T2–T4)
40	V_{CC}	—	+5 V

Tablo 10.1.2: 8086'nın 40 pini: numara, ad, yön ve kısa görev (24–31 için minimum / maksimum mod).

Tablo 10.1.2 40 pinin tamamını tek listede toplar ve devre şeması okurken başvuru tablosu olarak kullanılabilir. Pin numaralarını ezberlemek gerekmez. Asıl önemli olan, her pinin hangi grupta olduğunu ve hangi anda ne taşıdığını bilmektir.

10.1.1.5 Minimum ve maksimum mod

Pin darlığına karşı ikinci çare, 24–31 numaralı sekiz pinin anlamını sistem türüne göre değiştirmektir. Küçük bir sistemde işlemci tektir ve okuma, yazma, adres kilidi gibi kontrol sinyallerini kendisi üretebilir. Büyük bir sistemde ise 8087 kayan nokta yardımcı işlemcisi ya da 8089 G/Ç işlemcisi gibi başka yongalar aynı yolu paylaşır. Bu durumda kontrol sinyallerini ayrı bir yonga, 8288 yol denetleyicisi üretir. 8086'nın boşalan pinleri de çok işlemcili düzen için gereken yeni sinyallere ayrılır.

Tanım 10.1.3

MN/MX' pini (pin 33) V_{CC}'ye bağlanırsa 8086 **minimum modda** çalışır. Bu, tek işlemcili küçük sistemdir; 24–31 numaralı pinler kontrol sinyallerini (INTA', ALE, DEN', DT/R', M/IO', WR', HLDA, HOLD) doğrudan üretir. MN/MX' toprağa bağlanırsa **maksimum modda** çalışır. Aynı pinler durum kodu, kuyruk durumu ve yol paylaşımı sinyallerine dönüşür: QS1, QS0, S0'–S2', LOCK', RQ'/GT1', RQ'/GT0'. Kontrol sinyallerini 8288 üretir.

Mod donanımla seçilir. MN/MX' pini kart üzerinde kalıcı olarak beslemeye ya da toprağa bağlanır ve işlemci çalışırken değişmez. Hiçbir komut modu değiştiremez; yazılım açısından iki mod arasında fark yoktur. Aynı program iki sistemde de aynı sonucu verir.

Pin	Minimum mod (MN/MX' = 1)	Maksimum mod (MN/MX' = 0)
24	INTA'	QS1
25	ALE	QS0
26	DEN'	S0'
27	DT/R'	S1'

Pin	Minimum mod (MN/MX' = 1)	Maksimum mod (MN/MX' = 0)
28	M/IO'	S2'
29	WR'	LOCK'
30	HLDA	RQ'/GT1'
31	HOLD	RQ'/GT0'

Tablo 10.1.3: Moda göre anlamı değişen sekiz pin.

Tablo 10.1.3'teki eşleşme rastgele değildir. Pin 28, minimum modda bellek ile G/Ç'yi ayıran M/IO' sinyalini, maksimum modda S2' durum bitini taşır; S2' de aynı ayrımı yapar (1 bellek, 0 G/Ç). Pin 27'deki DT/R' veri yönünü, S1' de okuma ile yazmayı ayırmaya yardım eder. Böylece maksimum moddaki 8288, minimum moddaki sinyallerin bilgisini kodlanmış biçimde alır ve onları yeniden üretir. Pin 30–31'deki HOLD/HLDA çiftinin maksimum mod karşılığı da yine yol devri yapan RQ'/GT pinleridir.

Örnek 10.1.1

Bir tasarımcı, 8087 yardımcı işlemcisi eklenecek bir kart hazırlıyor. MN/MX' nereye bağlanmalıdır ve minimum modda kullandığı hangi sinyalleri artık 8086'dan alamaz?

1. 8087, 8086'nın kuyruğunu QS1, QS0 pinlerinden izler ve yolu RQ'/GT pinleriyle ister. Bu pinler yalnız maksimum modda vardır; MN/MX' toprağa bağlanır.
2. Maksimum modda 24–31 numaralı pinler değiştiği için INTA', ALE, DEN', DT/R', M/IO', WR', HLDA ve HOLD sinyalleri 8086'dan alınamaz.
3. Bunların işini 8288 görür: S2'–S0' kodundan MRDC', MWTC', IORC', IOWC', INTA', ALE, DEN ve DT/R' sinyallerini üretir. HOLD/HLDA yerine RQ'/GT0' ve RQ'/GT1' kullanılır.

RD' pini (32) iki modda da 8086'dadır; ama maksimum modda bellek ve G/Ç okumaları genellikle 8288'in ürettiği MRDC' ve IORC' ile yapılır.

10.1.2 Adres ve veri pinleri

Bir bellek erişiminde işlemcinin söylemesi gereken iki şey vardır: hangi adres ve hangi veri. 8086 bu iki bilgiyi aynı pinlerden sırayla gönderir. Bu kesim önce adres/veri ve adres/durum pinlerinin zaman içindeki görevini, sonra çoklamanın dışarıda nasıl çözüldüğünü anlatır. Ardından 16 bitlik veri yolunun iki bayt bankasına nasıl bölündüğünü ve tek adresteki bir sözcüğün neden iki yol döngüsü sürdüğünü bir kod izlemesiyle gösterir.

10.1.2.1 AD0–AD15 ve A16/S3–A19/S6

AD0–AD15 pinleri adresin alt 16 bitini ve 16 bitlik veriyi taşır. Bir yol döngüsünün ilk saat turunda (T1) pinlerde adres bulunur; bu anda pinlere A0–A15 denir. Sonraki turlarda (T2, T3, T4 ve varsa bekleme turları) aynı pinler veri taşır; o zaman D0–D15 adını alırlar. Bayt erişiminde verinin yalnız

bir yarısı kullanılır: D0–D7 ya da D8–D15. Pinler çift yönlüdür. Yazmada işlemci veriyi dışarı sürer, okumada dışarıdaki aygıt veriyi içeri sürer. İşlemci gerektiğinde bu pinlerin sürücülerini tümüyle bırakabilir; buna **yüzer** ya da yüksek empedans (*float, tri-state*) durumu denir.

Tanım 10.1.4

Üç durumlu çıkış (*tri-state output*), 0 ve 1 düzeylerine ek olarak üçüncü bir durum alabilen çıkıştır: **yüksek empedans**. Bu durumda çıkış hattı ne 0'a ne 1'e sürer; elektriksel olarak hattan kopmuş gibidir ve hattı başka bir sürücü sürebilir. Bir hatta aynı anda yalnız bir sürücünün etkin olması gerekir.

A16/S3–A19/S6 pinleri adresin üst 4 bitini taşır. Bellek erişiminde T1 sırasında bu pinlerde A16–A19 bulunur. T2'den itibaren aynı pinler S3–S6 durum bitlerine geçer. G/Ç erişiminde adres yalnız 16 bittir; bu yüzden T1'de A16–A19 hatları 0 olur. S3 ve S4 erişimde hangi kesim yazmacının kullanıldığını, S5 kesme bayrağı IF'nin o anki değerini gösterir; S6 her zaman 0'dır. Durum bitleri kendi alt kesiminde ayrıntılı ele alınır.



Şekil 10.1.4: AD15–AD0 pinleri T1'de adresi, T2–T4'te veriyi taşır. A19/S6–A16/S3 pinleri T1'de adresin üst dört bitini, sonra durum bitlerini taşır.

BHE'/S7 (pin 34) bu gruba yakındır. T1 sırasında BHE' (*bus high enable*, yüksek yol etkin) olarak çalışır ve veri yolunun üst yarısının (D8–D15) bu döngüde kullanılacağını bildirir. T2–T4 boyunca S7 durum bitine döner; 8086'da S7 kullanılmayan yedek bir bittir. Şekil 10.1.4 bu zaman paylaşımını özetler: bir pinin anlamı, ona hangi saat turunda bakıldığına bağlıdır.

Sık yapılan hata

"Bellek, adresi döngü boyunca AD pinlerinden okur" düşüncesi yanlıştır. AD pinlerinde adres yalnız T1 boyunca durur. T2'de pinler veriye geçer ya da yüzer duruma alınır. Bellek yongası ise adresi T2–T4 boyunca da görmek zorundadır. Bu yüzden adres T1 sonunda dışarıdaki bir kilit devresine alınır. Bellek adresi o kilidin çıkışından okur.

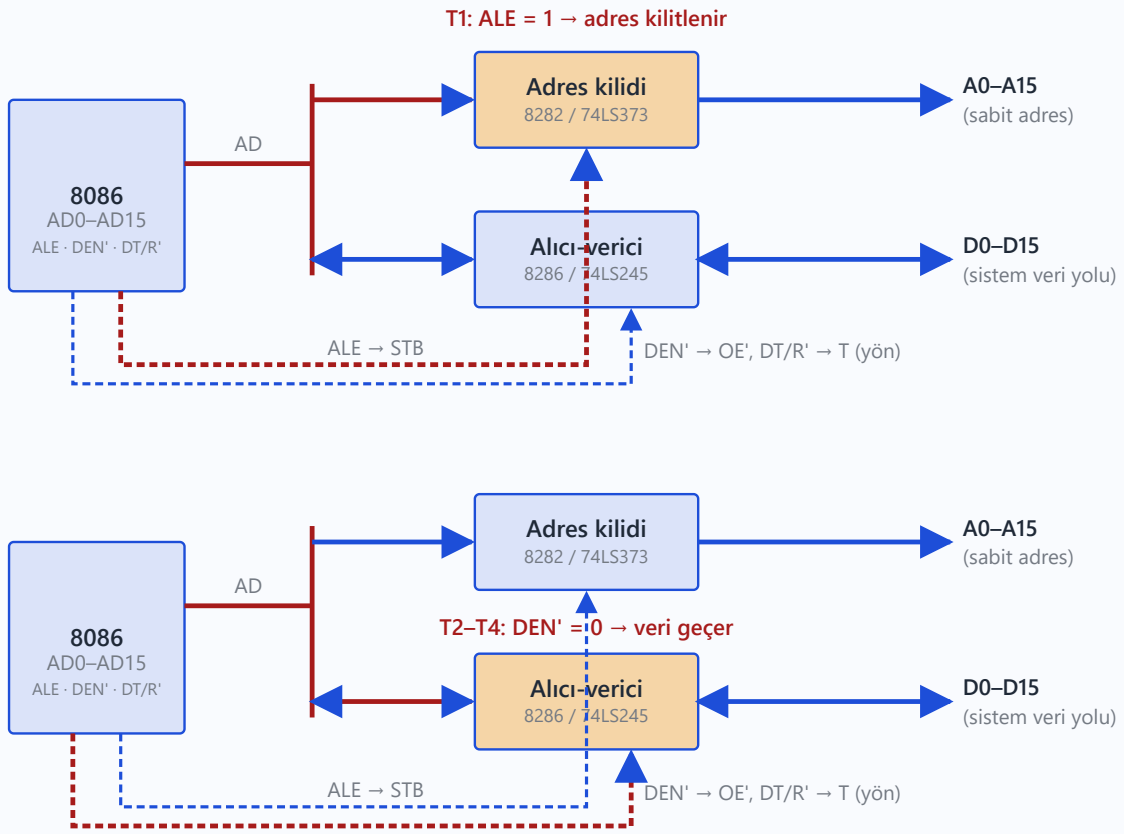
10.1.2.2 Çoklamayı çözmek: ALE ve adres kilidi

Sistemin geri kalanı, yani bellek yongaları, adres çözücüler ve G/Ç aygıtları, ayrık ve kararlı bir adres yolu ile ayrı bir veri yolu ister. 8086'nın çoklamalı yolundan bu iki yolu ayırmaya **çoklamayı çözme** (*demultiplexing*) denir. Bu iş için iki tür devre kullanılır: adresi yakalayıp tutan **adres kilidi** ve veri

hatlarını güçlendirip yönünü belirleyen **alıcı-verici**. Kilidin adresi ne zaman yakalayacağını işlemci **ALE** (*address latch enable*, adres kilidi etkinleştirme, pin 25) sinyaliyle bildirir.

Tanım 10.1.5

Kilit (*latch*), bir etkinleştirme girişi 1 iken girişindeki değeri çıkışına geçiren, etkinleştirme 0'a inince son değeri tutan bellek devresidir. Etkinleştirme 1 iken kilit "saydamdır"; düşen kenarda değeri yakalar. 8086 sistemlerinde 8 bitlik 8282 ya da 74LS373 kilitleri kullanılır; etkinleştirme girişine (STB ya da LE) ALE bağlanır.



Şekil 10.1.5: Çoklamanın çözülmesi. Üstte T1: 8086 adresi AD pinlerine koyar ve ALE'yi 1 yapar; ALE düşerken kilit adresi yakalar. Altta T2–T4: aynı pinler veri taşır; DEN' = 0 alıcı-vericiyi açar, DT/R' yönü seçer.

Şekil 10.1.5 bir döngüdeki iki evreyi yan yana gösterir. T1'de 8086 adresi AD pinlerine koyar ve ALE'yi kısa bir darbeyle 1 yapar. ALE 1 iken kilit saydamdır; pinlerdeki adres kilidin çıkışına geçer. ALE 0'a düştüğünde kilit o anki değeri yakalar ve döngünün sonuna kadar çıkışında tutar. T2'den sonra AD pinleri veri taşır, ama kilidin çıkışındaki adres değişmez. Bellek ve G/Ç aygıtları adresi kilidin çıkışında, veriyi alıcı-vericinin öteki ucunda görür.

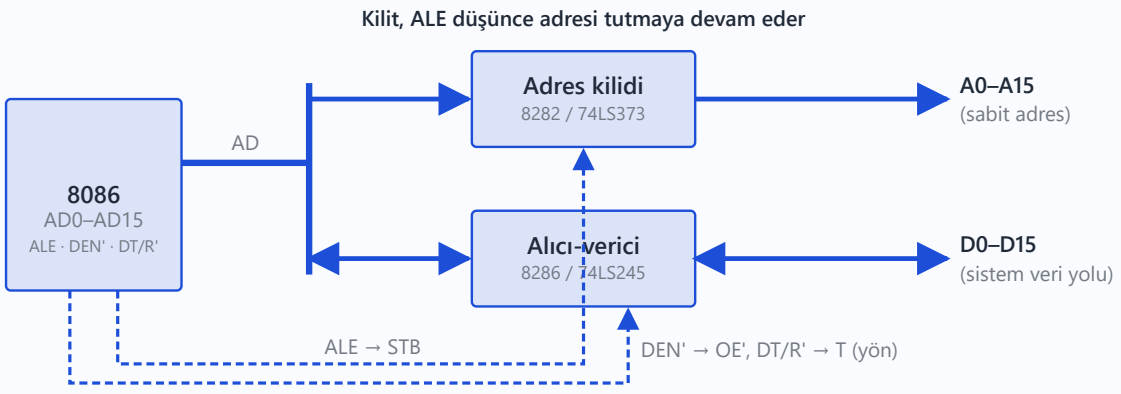
ALE'nin neden düşen kenarı kullandığı veri sayfasındaki zaman değerlerinden anlaşılır. 5 MHz'lik 8086'da ALE, T1'in başındaki saat kenarından en geç 80 ns sonra 1 olur (T_{CLLH}). Adres ise en geç 110 ns sonra geçerli hâle gelir (T_{CLAV}). Yani ALE yükselirken adres henüz kararlı olmayabilir. Buna karşılık veri sayfası iki şeyi garanti eder. Adres, ALE düşmeden en az $T_{AVAL} = T_{CLCH} - 60$ ns önce geçerlidir. ALE düştükten sonra da en az $T_{LLAX} = T_{CHCL} - 10$ ns daha pinlerde kalır. Saydam kilit bu yüzden doğru seçimdir: ALE 1 iken çıkış adresi izler, ALE düşerken kesinleşmiş adres yakalanır.

Örnek 10.1.2

5 MHz'lik 8086'da (T_{CLCL} = 200 ns) saatin düşük süresi en az T_{CLCH} = 118 ns, yüksek süresi en az T_{CHCL} = 69 ns'dir. Adres, ALE düşmeden en az ne kadar önce geçerlidir ve ALE düştükten sonra en az ne kadar pinlerde kalır?

1. $T_{AVAL} = T_{CLCH} - 60 = 118 - 60 = 58$ ns: kilidin girişindeki adres, yakalama anından en az 58 ns önce kararlıdır.
2. $T_{LLAX} = T_{CHCL} - 10 = 69 - 10 = 59$ ns: adres, yakalama anından sonra en az 59 ns daha pinlerde durur.
3. 74LS373'ün veri sayfasına göre girişteki veri, LE düşmeden en az 5 ns önce kurulmalı ve düştükten sonra en az 20 ns tutulmalıdır. 58 ns ≥ 5 ns ve 59 ns ≥ 20 ns olduğundan iki koşul da rahatça sağlanır.

Kilidin kurulma ve tutma süreleri, 8086'nın verdiği garantilerden küçük olduğu sürece yakalanan adres doğrudur.



Şekil 10.1.6: Üstte çoklamalı yol ve iki çözücü devre: adres kilidi (8282 / 74LS373) ve alıcı-verici (8286 / 74LS245). Altta döngünün geri kalanı: kilit, ALE düştükten sonra adresi tutmaya devam eder.

20 bitlik adres için 8 bitlik üç kilit gerekir. Biri A0–A7'yi, biri A8–A15'i yakalar. Üçüncüsü A16–A19 ile BHE' sinyalini tutar: bu beş sinyal de T1'den sonra durum bilgisine geçtiği için yakalanmak zorundadır. Üçüncü kilidin boş kalan girişleri kullanılmaz. Şekil 10.1.6 bu düzenin sadeleştirilmiş görünümünü verir.

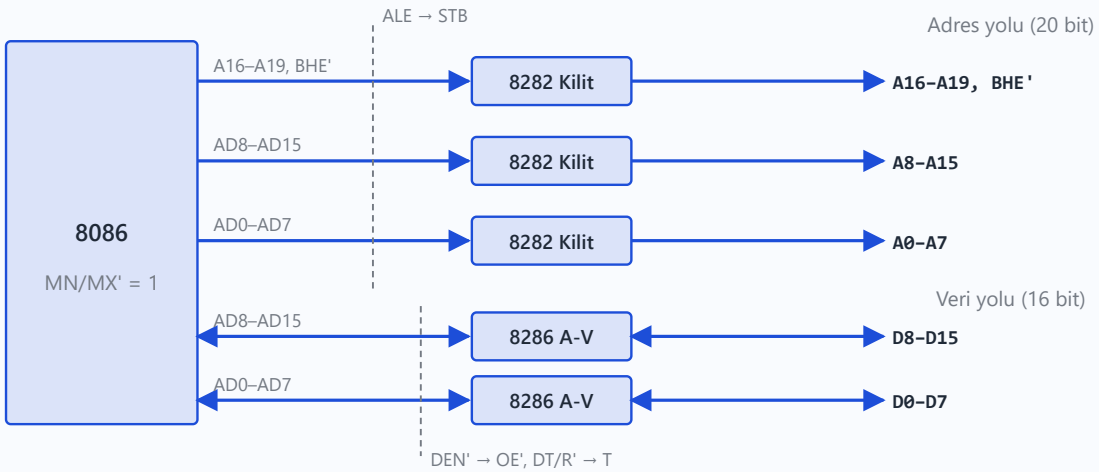
8282 ve 74LS373 aynı işi yapar: ikisi de 8 bitlik saydam kilittir ve çıkışları üç durumludur. 74LS373'ün iki denetim girişi vardır: LE (*latch enable*) ve OE' (*output enable*). LE'ye ALE bağlanır, OE' ise sürekli 0'da tutulur ki kilidin çıkışı adres yolunu hep sürsün. Yol başka bir yöneticiye devredildiğinde OE' 1 yapılarak kilidin çıkışları da yüzer duruma alınabilir. 8283, 8282'nin çıkışı ters çeviren sürümüdür.

Sık yapılan hata

Adres kilidi yerine yükselen kenarda yakalayan bir D flip-flop yongası (ör. 74LS374) kullanıp saat girişine doğrudan ALE bağlamak hatalıdır. Flip-flop adresi ALE'nin yükselen kenarında yakalar. Bu anda adres henüz geçerli olmayabilir: ALE en geç 80 ns'de yükselir, adres en geç 110 ns'de oturur. Flip-flop kullanılacaksa ALE'nin tersi saat olarak verilir; böylece yakalama ALE'nin düşen kenarına kayar.

10.1.2.3 Tamponlu ve kilitli sistem

Kilitler adres yolunu üretir. Veri yolu için ise **alıcı-verici** (*transceiver*) kullanılır. Alıcı-verici, iki yönlü bir tampondur: bir verici ve bir alıcının tek pakette birleşimidir. 8286 ve onun yerine sık kullanılan 74LS245 sekizer bitlik alıcı-vericilerdir; 16 bitlik veri yolu için iki tane gerekir. Bir alıcı-vericinin iki denetim girişi vardır. Yön girişi (8286'da T, 74LS245'te DIR) verinin hangi taraftan hangi tarafa akacağını seçer. Etkinleştirme girişi (OE' ya da G') 1 iken iki tarafı da yüzer duruma alır ve hatları birbirinden ayırır.



Şekil 10.1.7: Tamponlu ve kilitli minimum mod sistemi: üç 8282 kilidi 20 bitlik adres yolunu ve BHE' sinyalini, iki 8286 alıcı-vericisi 16 bitlik veri yolunu üretir. Kilitleri ALE, alıcı-vericileri DEN' ve DT/R' denetler.

Şekil 10.1.7 tam bir minimum mod yol arayüzünü gösterir. Alıcı-vericinin yön girişine 8086'nın DT/R' çıkışı, etkinleştirme girişine DEN' çıkışı bağlanır. DT/R' = 1 iken veri işlemciden belleğe, 0 iken bellekten işlemciye akar. DEN' yalnız veri aktarılırken 0 olur; T1'de 1 kalır ki AD pinlerindeki adres veri yo-

luna sızmasın. Bu iki sinyal, kontrol pinlerinin anlatıldığı alt kesimde zamanlamalarıyla birlikte ele alınır.

Tamponların ikinci görevi **sürme gücü** sağlamaktır. 8086'nın bir çıkış pini düşük düzeyde en çok 2,5 mA akım çekebilir ve veri sayfası çıkış yükünü 20–100 pF ile sınırlar. Bir 74LS girişinin düşük düzeyde verdiği akım 0,4 mA civarındadır. Bu yüzden 8086'nın tek bir pini ancak birkaç LS girişini güvenle sürebilir. Onlarca bellek yongasının bağlandığı büyük bir kartta her yol hattı birçok girişe gider ve uzun izlerin kapasitesi de eklenir. Kilitler ve alıcı-vericiler bu yükü işlemcinin yerine taşır. Bu nedenle büyük sistemlerde adres, veri ve kontrol yollarının hepsi tamponlanır.

Örnek 10.1.3

8086'nın bir AD pini düşük düzeyde 2,5 mA çekebiliyor. Her biri düşük düzeyde 0,4 mA veren 74LS girişlerinden en çok kaç tanesini doğrudan sürebilir? Bir 74LS245 çıkışı düşük düzeyde 24 mA çekebiliyorsa kaç giriş sürer?

1. 8086 pini: 2,5 / 0,4 = 6,25 → en çok 6 giriş. Hat kapasitesi ve gürültü payı için pratikte daha azı kullanılır.
2. 74LS245 çıkışı: 24 / 0,4 = 60 giriş.

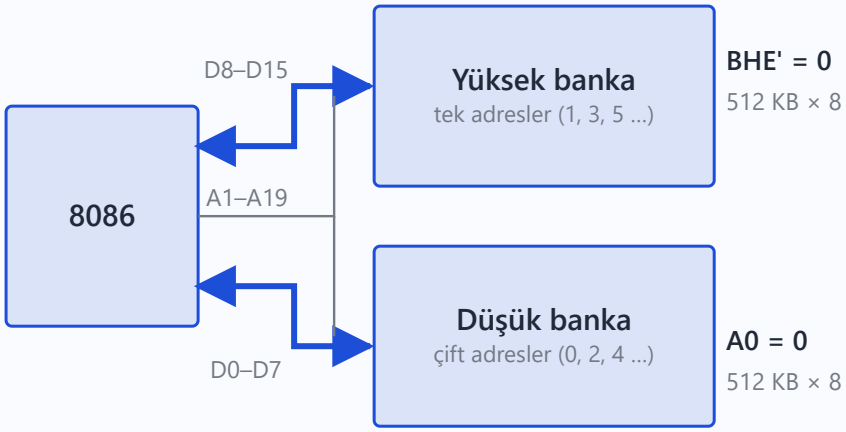
Alıcı-verici, akım sürme kapasitesini yaklaşık on katına çıkarır. Bellek kartı büyüdükçe bu fark belirleyici olur.

10.1.2.4 BHE' ve A0: bellek bankaları

8086'nın veri yolu 16 bittir, ama bellek bayt düzeyinde adreslenir. Her adres bir baytı gösterir. İşlemcinin hem tek bir baytı hem de iki ardışık bayttan oluşan bir sözcüğü tek döngüde aktarabilmesi için 1 MB'lık bellek fiziksel olarak iki yarıya bölünür. Her yarıya **bellek bankası** denir.

Tanım 10.1.6

Düşük banka (çift banka) çift adresli baytları tutar ve veri yolunun D0–D7 hatlarına bağlanır. **Yüksek banka** (tek banka) tek adresli baytları tutar ve D8–D15 hatlarına bağlanır. Her banka 512 KB'tır ve A1–A19 adres hatlarıyla paralel adreslenir. Düşük bankayı A0 = 0, yüksek bankayı BHE' = 0 seçer.



Şekil 10.1.8: Çift ve tek bellek bankaları. Düşük banka D7–D0'a bağlanır ve A0 = 0 ile seçilir; yüksek banka D15–D8'e bağlanır ve BHE' = 0 ile seçilir. İki banka da A19–A1 ile adreslenir.

Şekil 10.1.8'de iki bankanın aynı adres hatlarını paylaştığı görülür. Bankalara A0 değil A1–A19 gider; çünkü bir banka içindeki ardışık baytlar 2'şer artan adreslerdedir. Örneğin düşük bankanın 0., 1., 2. baytları 00000h, 00002h, 00004h adreslerine, yüksek bankanınkiler 00001h, 00003h, 00005h adreslerine karşılık gelir. A19–A1 aynı olan iki adres, yani 2k ve 2k + 1, iki bankada aynı satırdadır ve tek döngüde birlikte okunabilir.

BHE'	A0	Aktarım	Kullanılan hatlar
0	0	tüm sözcük (çift adres)	D15–D0
0	1	yüksek bayt (tek adres)	D15–D8
1	0	düşük bayt (çift adres)	D7–D0
1	1	aktarım yok	—

Tablo 10.1.4: BHE' ve A0 ile banka seçimi.

Tablo 10.1.4 dört durumu verir. Çift adresli bir bayt yazılırken yalnız düşük bankanın yazma izni açılmalıdır; yüksek bankadaki komşu bayt bozulmamalıdır. Bu yüzden bankaların seçimi bellek yazma sinyaliyle birleştirilir: düşük bankanın yazma girişi WR' ve A0'dan, yüksek bankanınki WR' ve BHE' sinyallerinden türetilir. Okumada ise iki bankanın birlikte okunmasının bir zararı yoktur; işlemci yalnız gereken yarıyı alır. Yine de çoğu tasarımda banka seçimi okumada da uygulanır.

Tek adresten başlayan bir sözcüğün düşük baytı yüksek bankada, yüksek baytı ise bir sonraki satırın düşük bankasındadır. İki bayt farklı satırlarda olduğu için tek döngüde birlikte okunamaz. 8086 bunu kendiliğinden iki yol döngüsüne böler: önce tek adresteki baytı D8–D15 üzerinden, sonra bir sonraki çift adresteki baytı D0–D7 üzerinden aktarır. Bu bölme programcıya görünmez; komut doğru sonucu verir, yalnız daha uzun sürer. Veri sayfası bu cezanın komut getirmede değil, yalnız sözcük işlenenlerinde olduğunu belirtir: komutlar kuyruğa her zaman çift adresten başlayan sözcükler olarak getirilir.

Sık yapılan hata

"Tek adresteki bir bayt da iki döngü sürer" düşüncesi yanlıştır. Bayt erişimi hangi adreste olursa olsun tek döngüdür: çift adresteki bayt D0–D7'den, tek adresteki bayt D8–D15'ten geçer. İki döngü yalnız tek adresten başlayan sözcük erişiminde gerekir. Verileri çift adrese hizalamak için assembler'ın `EVEN` direktifi kullanılır.

10.1.2.5 Kod izleme: hizalı ve hizasız erişim

Aşağıdaki program aynı sözcüğü bir kez çift, bir kez tek adrese yazar; ardından bir bayt ve bir sözcük okur. Her veri erişiminde kaç yol döngüsü yapıldığını, BHE' ile A0'ın değerini ve verinin hangi hatlardan geçtiğini izlemek, banka düzenini somutlaştırır. Bellekte 20100h–20102h adresleri başlangıçta bilinmiyor (??) kabul edilir.

```
mov ax, 2000h
mov ds, ax
mov ax, 1234h
mov [0100h], ax    ; çift adres
mov ax, 0ABCDh
mov [0101h], ax    ; tek adres
mov al, [0100h]    ; bayt
mov bx, [0101h]    ; tek adres
```

İlk iki komut DS'ye 2000h yükler; veri kesimi fiziksel 20000h adresinden başlar. Bu iki komut yalnız yazmaçları değiştirir. Yolda yalnız komut baytlarını getiren döngüler görülür, veri için döngü yoktur. `mov ax, 1234h` AX'e 1234h koyar: AH = 12h, AL = 34h.

`mov [0100h], ax` fiziksel adresi 20000h + 0100h = 20100h olarak hesaplar. Adres çifttir. 8086 T1'de BHE' = 0 ve A0 = 0 verir; iki banka birden seçilir. Sözcük tek döngüde yazılır: AL'deki 34h D7–D0 üzerinden çift adrese (20100h), AH'deki 12h D15–D8 üzerinden tek adrese (20101h) gider. Little-endian düzen, banka düzeniyle doğal olarak uyuşur: düşük bayt düşük adrese ve düşük bankaya yazılır.

`mov ax, 0ABCDh` AX'i ABCDh yapar. Sayının başındaki 0, assembler'ın ABCDh'yi bir etiket adı sanmaması içindir. `mov [0101h], ax` ise tek adres 20101h'ye yazar. 8086 bu erişimi iki döngüye böler. Birinci döngüde adres 20101h'dir: BHE' = 0, A0 = 1 ile yalnız yüksek banka seçilir ve AL'deki CDh D15–D8 üzerinden 20101h'ye yazılır. İkinci döngüde adres 20102h'dir: BHE' = 1, A0 = 0 ile yalnız düşük banka seçilir ve AH'deki ABh D7–D0 üzerinden 20102h'ye yazılır. Bu yazma, 20101h'deki eski 12h'nin üzerine CDh yazar.

`mov al, [0100h]` çift adresten bir bayt okur: BHE' = 1, A0 = 0, veri D7–D0'dan gelir. AL = 34h olur; AH değişmez, AX = AB34h. Son komut `mov bx, [0101h]` yine hizasız bir sözcük okumasıdır. Birinci döngü 20101h'deki CDh'yi D15–D8 üzerinden getirir. İşlemci bu baytı sözcüğün düşük yarısına, BL'ye yerleştirir. İkinci döngü 20102h'deki ABh'yi D7–D0 üzerinden getirir; bu bayt BH'ye gider. Sonuç BX = ABCDh olur.

Komut	Döngü	FA	BHE'	A0	D15–D8	D7–D0	RD'/WR'	Sonuç
`mov [0100h],ax`	1/1 yazma	20100h	0	0	12h	34h	1/0	[20100h] = 34h, [20101h] = 12h
`mov [0101h],ax`	1/2 yazma	20101h	0	1	CDh	—	1/0	[20101h] = CDh
	2/2 yazma	20102h	1	0	—	ABh	1/0	[20102h] = ABh
`mov al, [0100h]`	1/1 okuma	20100h	1	0	—	34h	0/1	AX = AB34h
`mov bx, [0101h]`	1/2 okuma	20101h	0	1	CDh	—	0/1	BL = CDh
	2/2 okuma	20102h	1	0	—	ABh	0/1	BX = ABCDh

Tablo 10.1.5: Hizalı ve hizasız erişim programının veri yol döngüleri.

Tablo 10.1.5 dört veri komutunun toplam 6 veri yol döngüsü yaptığını gösterir: 1 + 2 + 1 + 2. Programın sonunda AX = AB34h, BX = ABCDh olur; bellekte 20100h = 34h, 20101h = CDh, 20102h = ABh bulunur. Program emu8086'da çalıştırıldığında yazmaç ve bellek değerleri aynı çıkar. Emülatör bankaları ve döngüleri göstermez; döngü sayısı ve BHE'/A0 değerleri yalnız fiziksel adresin tek ya da çift olmasından çıkar.

Komut kodlarının uzunlukları da bilgi verir. `mov [0100h], ax` A3 00 01 olarak 3 bayttır; AX için kısa akümülatör biçimi kullanılır. `mov bx, [0101h]` ise 8B 1E 01 01 olarak 4 bayttır. Bu baytlar, veri döngülerinin yanında ayrıca komut getirme döngüleriyle kuyruğa alınır.

Örnek 10.1.4

5 MHz'lik bir 8086, bekleme durumu olmayan bir bellekte 1000 sözcüklük bir diziyi okuyor. Dizi çift adreste başlarsa ve tek adreste başlarsa veri döngüleri ne kadar sürer?

1. Bir yol döngüsü 4 × 200 ns = 800 ns'dir.
2. Çift adreste başlayan dizide her sözcük tek döngüdür: 1000 × 800 ns = 800 µs.
3. Tek adreste başlayan dizide her sözcük iki döngüdür: 2000 × 800 ns = 1600 µs.

Yalnız hizalama farkı veri erişim süresini iki katına çıkarır. Yığının çift adreste tutulması da bu yüzden önemlidir: kesme ve yordam çağrılarındaki her PUSH ve POP bir sözcük erişimidir.

10.1.3 Yol döngüleri

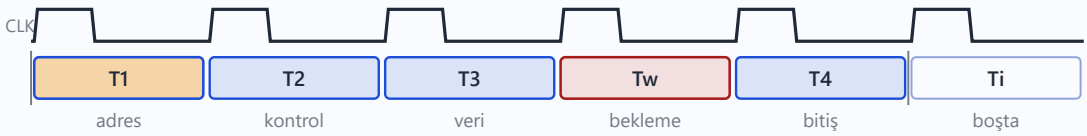
Adres, veri ve kontrol sinyallerinin pinlerde hangi sırayla belirdiğini anlamak için zamanı saat turlarına bölmek gerekir. Bu kesim bir yol döngüsünün T durumlarını tanımlar, ardından yazma ve okuma döngülerini tur tur izler. Okuma döngüsündeki bütün kontrol sinyallerini veri sayfasındaki zaman değerleriyle birlikte verir. Son olarak yavaş bir belleğin READY sinyaliyle bekleme durumu istemesini ve bir bellek için kaç bekleme durumu gerektiğinin hesabını anlatır.

10.1.3.1 Yol döngüsü ve T durumları

Tanım 10.1.7

İşlemcinin bellek ya da G/Ç ile her veri alışverişi bir **yol döngüsüdür** (*bus cycle*). 8086'da bir yol döngüsü en az dört saat turu sürer. Bu turlara **T durumları** denir: T1, T2, T3, T4. Yavaş bir aygıt için T3 ile T4 arasına **bekleme durumları** (Tw, *wait state*) eklenebilir. İşlemcinin yolu hiç kullanmadığı turlar **boşta durumlarıdır** (Ti, *idle state*).

Her T durumu saatin bir düşen kenarında başlar ve bir sonraki düşen kenarda biter. Pinlerin anlamı T durumuna göre değişir. T1'de AD pinlerinde adres, A16–A19'da adresin üst bitleri bulunur ve ALE darbesi verilir. T2 çoğunlukla hazırlıktır: okumada yolun yönü değiştirilir, kontrol sinyalleri (RD', WR') etkinleşir. Veri aktarımı T3 ve T4'te yapılır. T4, döngünün kapanışıdır; kontrol sinyalleri pasife döner ve bir sonraki döngü için yol hazırlanır.



Şekil 10.1.9: Yol döngüsünün zaman çizelgesi: T1 adres, T2 kontrol, T3 veri, gerekirse Tw bekleme, T4 bitiş. Yol kullanılmadığında işlemci boşta (Ti) turlarındadır.

Saat frekansı bilinince bir yol döngüsünün süresi doğrudan hesaplanır. 5 MHz'lik saatte bir tur T = 1 / 5 MHz = 200 ns'dir; bekleme durumu olmayan bir yol döngüsü 4 × 200 = 800 ns sürer. Bir saniyede en çok 1 / 800 ns = 1,25 milyon yol döngüsü yapılabilir. Her döngü en çok 2 bayt taşıdığı için yolun en yüksek bant genişliği yaklaşık 2,5 MB/s'dir.

Örnek 10.1.5

8086 (5 MHz), 8086-2 (8 MHz) ve 8086-1 (10 MHz) için bekleme durumsuz yol döngüsü süresini ve yolun en yüksek bant genişliğini bulalım.

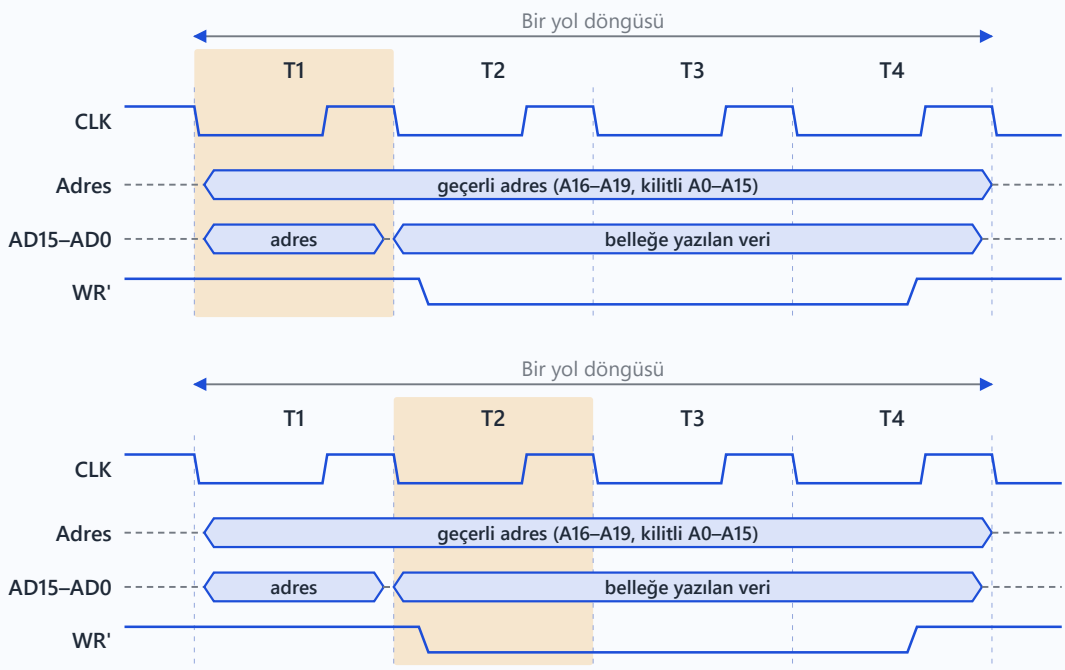
1. 5 MHz: T = 200 ns, döngü 800 ns, $1{,}25 \times 10^6$ döngü/s × 2 bayt = 2,5 MB/s.
2. 8 MHz: T = 125 ns, döngü 500 ns, 2×10^6 döngü/s × 2 bayt = 4 MB/s.
3. 10 MHz: T = 100 ns, döngü 400 ns, $2{,}5 \times 10^6$ döngü/s × 2 bayt = 5 MB/s.

Bu değerler üst sınırdır. Gerçekte yolun bir kısmı komut getirmeye gider, hizasız sözcükler iki döngü sürer ve yavaş bellekler bekleme durumu ekler.

Boşta durumları işlemcinin yolu kullanmadığı anlarda ortaya çıkar. Örneğin önden getirme kuyruğu doluyken ve yürütülen komut bellek istemiyorsa BIU'nun yapacak işi yoktur; yol Ti turlarında bekler. Veri sayfası bu turların iç işler için kullanıldığını belirtir. Ti turlarında kontrol sinyalleri pasif kalır ve yol, HOLD isteğiyle başka bir yöneticiye kolayca devredilebilir.

10.1.3.2 Yazma yol döngüsü

Yazma döngüsü dört T durumundan oluşur. Şekil 10.1.10'da bir bellek yazmasının sadeleştirilmiş zaman çizelgesi görülür.



Şekil 10.1.10: Yazma yol döngüsü. Üstte T1: hedef adres AD ve A pinlerinde, ALE ile kilitlere alınır. Altta T2: AD pinleri yazılacak veriyi taşır ve WR' = 0 olur.

1. **T1:** İşlemci yazılacak hedef adresi AD0–AD15 ve A16–A19 pinlerine koyar. ALE darbesi adresi dış kilitlere alır. M/IO' bir önceki döngünün T4'ünde geçerli hâle gelmiştir; bellek yazmasında 1, G/Ç yazmasında 0'dır.
2. **T2:** AD pinleri adresten veriye geçer: işlemci yazılacak veriyi hemen sürer. WR' sinyali T2'nin başında 0 olur. A16–A19 pinleri durum bitlerine (S3–S6) geçer.

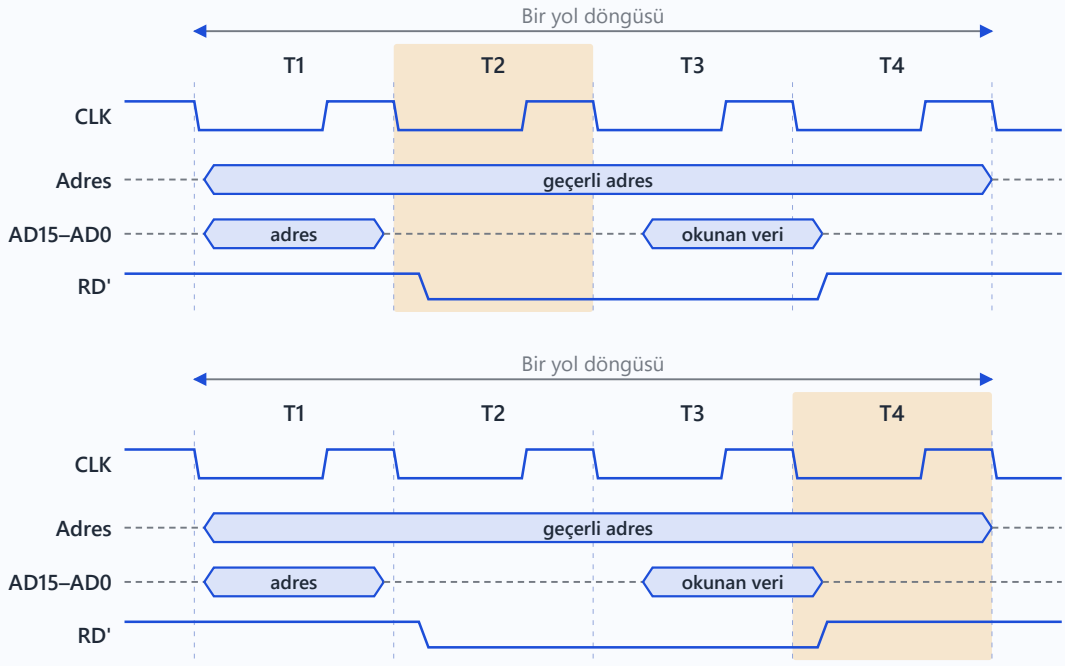
3. **T3:** Veri ve WR' kararlı kalır. Bellek veriyi yakalamaya hazırlanır. Aygıt hazır değilse READY'yi 0 tutar; bu durumda T3'ten sonra Tw eklenir.
4. **T4:** WR' 1'e döner. Belleklerin çoğu veriyi WR' sinyalinin yükselen kenarında yazar. Veri, T4'ün ortasına kadar pinlerde kalır; böylece yükselen kenarda bellek kararlı veriyi görür.

Yazma döngüsünde adres T1'de, veri T2'den T4'e kadar aynı pinlerdedir ve arada boşluk yoktur. Bunun nedeni verinin kaynağının işlemcinin kendisi olmasıdır: adresi süren sürücüler, bir sonraki turda veriyi sürmeye devam eder. Yolun yönü değişmez.

Veri sayfası WR' darbesinin genişliğini $T_{WLWH} \geq 2T_{CLCL} - 60$ olarak verir. 5 MHz'de bu $2 \times 200 - 60 = 340$ ns eder. Bellek yongasının istediği en kısa yazma darbesi bundan kısa olmalıdır; değilse bekleme durumu eklenerek darbe uzatılır. Her Tw, WR' sinyalinin 0 kaldığı süreyi bir tur daha uzatır.

10.1.3.3 Okuma yol döngüsü

Okuma döngüsü de dört T durumundan oluşur. Önemli fark, AD pinlerinde adres ile veri arasında bir boşluk bulunmasıdır. Şekil 10.1.11 bu boşluğu gösterir.



Şekil 10.1.11: Okuma yol döngüsü. Üstte T3: bellek istenen veriyi veri yoluna koyar. Altta T4: 8086 veriyi T4'ün başında örnekler, RD' 1'e döner ve bellek yolu bırakır. Adres ile okunan veri arasında T2'de AD pinleri yüzer.

1. **T1:** Yazmadaki gibi okunacak adres pinlere konur ve ALE ile kilitlenir.
2. **T2:** 8086, AD pinlerinin çıkış sürücülerini üç durumlu yapar; pinler yüzer. Bu andan sonra bellek yolu sürebilir. RD' sinyali 0 olur.
3. **T3:** Bellek ya da G/Ç aygıtı RD' = 0'ı görünce çıkışlarını açar ve istenen veriyi veri yoluna koyar.

4. **T4:** T4'ün başında, yani T3'ü bitiren düşen saat kenarında 8086 veriyi pinlerden okur. RD' 1'e döner ve bellek çıkışlarını kapatarak yolu bırakır.

Tanım 10.1.8

Yol dönüşü (*bus turnaround*), çift yönlü bir hatta sürücünün değiştiği andır. Okuma döngüsünde AD hatlarını T1'de işlemci, T3–T4'te bellek sürer. T2'deki yüzer durum bu dönüş içindir: işlemci adresi sürmeyi bırakmadan bellek veriyi sürmeye başlarsa iki çıkış aynı hatta çakışır.

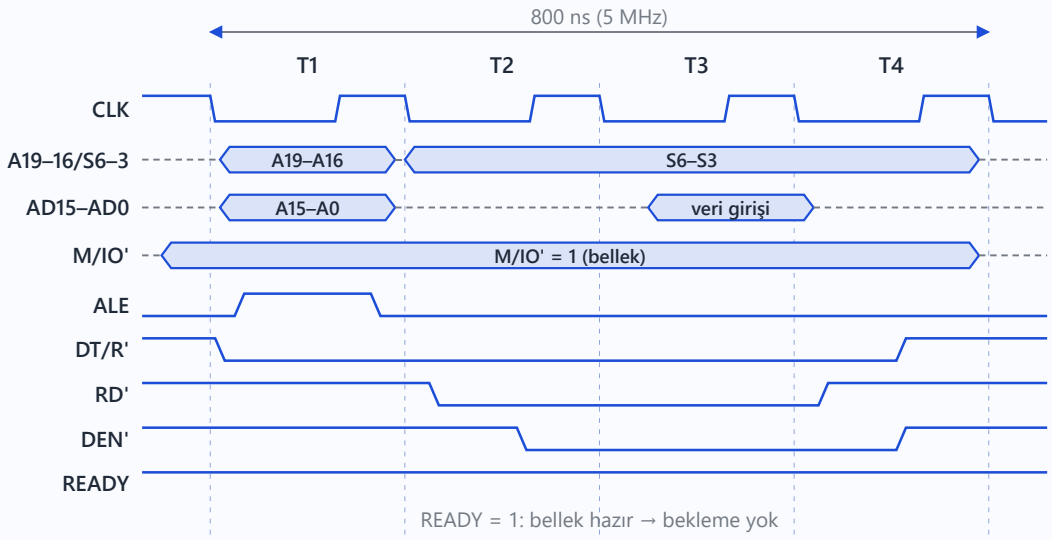
Çakışma sadece yanlış veri demek değildir. Bir çıkış hattı 1'e, öteki 0'a sürüyorsa iki sürücü arasında kısa devreye yakın bir akım akar; buna **yol çekişmesi** (*bus contention*) denir. Sürekli çekişme yongaları ısıtır ve zamanla bozar. Bu yüzden veri sayfası iki ayrı garanti verir. AD pinleri, T2'nin başındaki saat kenarından en geç T_{CLAZ} = 80 ns sonra yüzer. RD' sinyali de yerel yol yüzene kadar T2'de 1 kalır. Bellek de çıkışını ancak RD' = 0 olduktan sonra açar. Böylece işlemcinin bıraktığı hattı bellek güvenle alır.

Sık yapılan hata

"Okumada da yazmadaki gibi veri T2'de hazırdır" varsayımı yanlıştır. Okumada veri dış aygıttan gelir ve aygıt ancak RD' = 0'ı gördükten sonra çıkışını açar. RD' T2 içinde düşer, aygıtın erişim gecikmesi de eklenir; veri en erken T3 içinde yola çıkar ve T4'ün başında örneklenir. Yazmada ise veriyi işlemci sürdüğü için T2'nin başından beri pinlerdedir.

10.1.3.4 Okuma için yol zamanlaması

Sadeleştirilmiş çizelgeler yalnız adres, veri ve okuma sinyalini gösterir. Minimum modda bir bellek okumasında altı kontrol sinyali daha rol oynar: M/IO', ALE, DT/R', RD', DEN' ve READY. Şekil 10.1.12 hepsini aynı saat ekseninde verir.



Şekil 10.1.12: Minimum modda bir bellek okuma döngüsünde bütün sinyaller (5 MHz, 800 ns). M/IO' döngü boyunca 1, ALE T1'de darbe, DT/R' döngü boyunca 0, RD' ve DEN' T2–T4 arasında 0'dır; READY = 1 olduğu için bekleme durumu yoktur.

Sinyalleri turlara göre okuyalım:

- **T1:** M/IO' = 1 (bellek) önceki döngünün T4'ünden beri geçerlidir ve döngü boyunca kalır. Adres AD ve A pinlerindedir. ALE darbesi adresi kilitler. DT/R' = 0'dır: alıcı-verici “alma” yönüne ayarlanır. DEN' = 1'dir: alıcı-verici henüz kapalıdır.
- **T2:** AD pinleri yüzer; A19–A16 pinlerinde S6–S3 durum bitleri belirir. RD' = 0 olur. Ardından DEN' = 0 olur ve alıcı-verici açılır. Veri sayfasına göre okumada DEN', T2'nin ortasından T4'ün ortasına kadar aktiftir; yazmada ise T2'nin başından T4'ün ortasına kadar.
- **T3:** Bellek veriyi veri yoluna koyar; veri alıcı-vericiden geçerek AD pinlerine ulaşır. İşlemci T3 sırasında READY'yi örnekler. READY = 1 ise döngü T4'e geçer.
- **T4:** T4'ün başında veri örneklenir. RD', DEN' ve DT/R' pasife döner. M/IO' bir sonraki döngünün türüne göre T4 içinde değişebilir.

Simge	Anlamı	8086 (5 MHz)	8086-2 (8 MHz)	8086-1 (10 MHz)
T_{CLCL}	saat periyodu	200–500 ns	125–500 ns	100–500 ns
T_{CLCH}	saatin düşük süresi (en az)	118 ns	68 ns	53 ns
T_{CHCL}	saatin yüksek süresi (en az)	69 ns	44 ns	39 ns
T_{CLAV}	saatten adresin geçerli olmasına (en çok)	110 ns	60 ns	50 ns
T_{CLAZ}	saatten adresin yüzmesine (en çok)	80 ns	50 ns	40 ns
T_{CLRL}	saatten RD' sinyalinin düşmesine (en çok)	165 ns	100 ns	70 ns
T_{DVCL}	verinin saat kenarından önce kurulması (en az)	30 ns	20 ns	5 ns
T_{RYHCH}	READY'nin saat kenarından önce kurulması (en az)	118 ns	68 ns	53 ns
T_{RLRH}	RD' darbe genişliği (en az)	$2T_{CLCL} - 75$	$2T_{CLCL} - 50$	$2T_{CLCL} - 40$

Simge	Anlamı	8086 (5 MHz)	8086-2 (8 MHz)	8086-1 (10 MHz)
T_{WLWH}	WR' darbe genişliği (en az)	$2T_{CLCL} - 60$	$2T_{CLCL} - 40$	$2T_{CLCL} - 35$

Tablo 10.1.6: 8086'nın bazı zaman değerleri (veri sayfası, minimum mod). T_{CLCL} saat periyodudur.

Tablo 10.1.6 veri sayfasının AC karakteristikleri tablosundan seçilmiş değerleri verir. Simgelerin adlandırma kuralı okunmalarını kolaylaştırır: ilk iki harf başlangıç olayını, son iki harf bitiş olayını anlatır. CL "saat düşer", CH "saat yükselir", AV "adres geçerli", RL "RD' düşer", DV "veri geçerli" demektir. T_{CLAV}, saatin düşen kenarından adresin geçerli olmasına kadar geçen en uzun süredir.

Saat periyodunun üst sınırı da dikkat çeker: T_{CLCL} en çok 500 ns olabilir, yani saat 2 MHz'in altına inemez. 8086 dinamik bir tasarımdır; iç düğümlerindeki yükler belirli bir süreden fazla beklerse kaybolur. Bu yüzden saat durdurulamaz ya da çok yavaşlatılamaz.

10.1.3.5 READY ve bekleme durumu

Bir bellek yongasının **erişim süresi**, adresi aldığı andan veriyi çıkışına koyduğu ana kadar geçen süredir. Erişim süresi uzun olan yavaş bir bellek, verisini T3'ün sonuna kadar hazırlayamaz. 8086 bu durumu READY girişiyle çözer.

Tanım 10.1.9

READY (pin 22), adreslenen bellek ya da G/Ç aygıtının veri aktarımını tamamlayacağını bildiren aktif yüksek giriştir. İşlemci READY'yi T3 sırasında örnekler. READY = 0 ise T4'e geçmez, bir **bekleme durumu** (Tw) ekler ve READY'yi Tw içinde yeniden örnekler. READY 1 olana kadar Tw eklemeyi sürdürür.





Şekil 10.1.13: Bekleme durumlu okuma döngüsü. Üstte T3: READY = 0 görülür, işlemci T4 yerine bir Tw ekler ve bütün sinyaller olduğu gibi kalır. Altta T4: Tw'de READY = 1 örneklenmiştir; veri okunur. Döngü 5 × 200 = 1000 ns sürer.

Şekil 10.1.13 tek bekleme durumlu bir okumayı gösterir. Yavaş bellek seçildiğinde adres çözücü READY'yi hemen 0'a çeker. T3'te işlemci READY = 0'ı görür ve T4'e geçmez; bir Tw ekler. Tw sırasında RD', DEN', DT/R' ve adres oldukları gibi kalır. Bellek bu ek turda verisini hazırlar ve READY'yi 1'e çıkarır. Tw içinde READY = 1 örneklenince sıradaki tur T4 olur ve veri okunur. Döngü dört yerine beş tur, yani 5 × 200 = 1000 ns sürer; her bekleme durumu döngüye 200 ns ekler.

READY'nin zamanlaması sıkıdır. 5 MHz'de READY, T3'teki yükselen saat kenarından en az T_{RYHCH} = 118 ns önce kararlı olmalıdır. Saatin düşük süresi de yaklaşık 133 ns olduğundan READY pratikte T3'ün başında hazır olmalıdır. Ayrıca 8086'nın READY girişi içeride eş zamanlanmaz: kurulma ve tutma süreleri sağlanmazsa işlemci kararsız davranabilir. Bu yüzden bellek ve G/Ç aygıtlarının hazır sinyali doğrudan 8086'ya değil, 8284A saat üretecinin RDY girişine bağlanır. 8284A, RDY'yi saatle eş zamanlar ve 8086'ya kurallara uygun bir READY verir. Veri sayfası RDY'nin T2, T3 ve Tw sonlarına doğru örneklendiğini belirtir.

Sık yapılan hata

READY'yi aktif düşük bir "bekle" sinyali gibi düşünmek kafa karıştırır. READY aktif yüksektir: 1 "hazırım, devam et", 0 "hazır değilim, bekle" anlamına gelir. Hiçbir yavaş aygıtın bağlı olmadığı bir sistemde READY sürekli 1 tutulur. Bağlantısı unutulmuş ve 0'da kalmış bir READY, işlemciyi ilk bellek erişiminde sonsuz bekleme durumuna sokar.

10.1.3.6 Kaç bekleme durumu gerekir?

Bir bellek için gereken bekleme durumu sayısı, işlemcinin belleğe tanıdığı süre ile belleğin erişim süresi karşılaştırılarak bulunur. Okumada adres, T1'in başındaki saat kenarından en geç T_{CLAV} sonra geçerli olur. Veri ise T3'ün sonunda, T4'e geçen saat kenarında örneklenir ve bu kenardan en az T_{DVCL} önce hazır olmalıdır. Bu iki an arasında üç tur vardır. Aradaki sürenin bir kısmını da adres kilidi, adres çözücü ve veri alıcı-vericisinin gecikmeleri yer; bunların toplamına t_{tampon} diyelim.

Belleğe kalan süre şöyle yazılır:

$$t_{izin} = 3 \cdot T - T_{CLAV} - T_{DVCL} - t_{tampon}$$

Her Tw bu pencereyi bir T uzatır. n bekleme durumuyla izin verilen süre $t_{izin} + n \cdot T$ olur. Belleğin erişim süresi $t_{erişim}$ bu süreyi aşmamalıdır:

$$t_{erişim} \le t_{izin} + n \cdot T \;\Rightarrow\; n = \lceil (t_{erişim} - t_{izin}) / T \rceil \quad \text{(sonuç negatifse } n = 0\text{)}$$

Örnek 10.1.6

5 MHz'lik bir 8086 sisteminde T = 200 ns, veri sayfasından $T_{CLAV} \le 110$ ns ve $T_{DVCL} \ge 30$ ns alınıyor; kilit, çözücü ve alıcı-verici gecikmelerinin toplamı $t_{tampon} = 40$ ns kabul ediliyor. Erişim süresi 700 ns olan bir bellek için kaç bekleme durumu gerekir?

1. $t_{izin} = 3 \times 200 - 110 - 30 - 40 = 600 - 180 = 420$ ns.
2. $n = \lceil (700 - 420) / 200 \rceil = \lceil 280 / 200 \rceil = \lceil 1{,}4 \rceil = 2$.
3. Sınama: 1 Tw ile pencere 420 + 200 = 620 ns olur; 620 < 700, yetmez. 2 Tw ile 420 + 400 = 820 ns olur; 820 ≥ 700, yeter.
4. Döngü süresi (4 + 2) × 200 = 1200 ns.

Erişim süresi 400 ns olan bir bellek için 400 ≤ 420 olduğundan bekleme durumu gerekmez; döngü 800 ns'de kalır.

Hesapta tavan işlevi ($\lceil\ \rceil$) kullanılır, çünkü bekleme durumu tam tur olarak eklenir; 1,4 tur eklemek mümkün değildir. Bir bellek pencereye 1 ns bile sığmıyorsa bir tam tur daha eklenir. Bu yüzden erişim süresi pencereyi biraz aşan bir bellek, sistemi bir tur yavaşlatır.

Bu hesap yalnız “adresten veriye” koşulunu inceler. Okumada ikinci bir koşul daha vardır. Bellek çıkışını ancak RD' = 0 olduktan sonra açar. RD' sinyalinin düşmesinden verinin çıkışa gelmesine kadar geçen **çıkış etkinleştirme süresi** de pencereye sığmalıdır. RD', T2'nin başından en geç $T_{CLRL} = 165$ ns sonra düşer. Veri, T2'nin başından 2T = 400 ns sonraki kenardan T_{DVCL} önce hazır olmalıdır. Bu ikinci pencere 400 − 165 − 30 = 205 ns eder; alıcı-verici gecikmesi de buradan düşülür. Belleğin çıkış etkinleştirme süresi bu değerden kısa olmalıdır. Her Tw bu pencereyi de bir T uzatır.

Örnek 10.1.7

8 MHz'lik bir 8086-2 sistemi aynı 700 ns'lik belleği kullanıyor. Veri sayfası $T_{CLAV} \le 60$ ns, $T_{DVCL} \ge 20$ ns veriyor; $t_{tampon} = 40$ ns. Kaç Tw gerekir ve bekleme durumları hızlı saatin kazancını ne kadar geri alır?

1. T = 125 ns; $t_{izin} = 3 \times 125 - 60 - 20 - 40 = 375 - 120 = 255$ ns.
2. $n = \lceil (700 - 255) / 125 \rceil = \lceil 445 / 125 \rceil = \lceil 3{,}56 \rceil = 4$.
3. Sınama: 3 Tw ile 255 + 375 = 630 ns, yetmez; 4 Tw ile 255 + 500 = 755 ns, yeter.

4. Döngü (4 + 4) × 125 = 1000 ns.

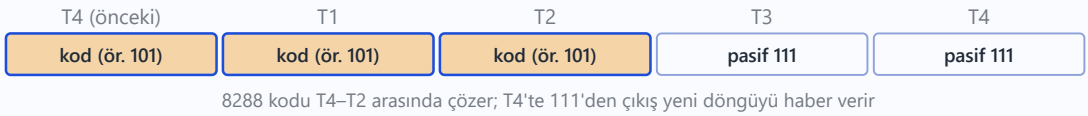
5 MHz'lik sistemde aynı bellek 1200 ns'lik döngü veriyordu. Saat 1,6 kat hızlansa da bellek döngüsü yalnız 1,2 kat kısalır (1200 ns'den 1000 ns'ye). Yavaş bellek, hızlı işlemcinin kazancının büyük kısmını bekleme durumlarında harcatır.

10.1.4 Durum pinleri

8086, her yol döngüsünde ne yaptığını dışarıya kodlanmış bir biçimde de bildirir. Bu bilgiyi taşıyan pinlere **durum pinleri** denir. Maksimum modda S2', S1', S0' pinleri döngünün türünü verir; 8288 yol denetleyicisi bu kodu çözerek okuma ve yazma komutlarını üretir. İki modda da A16–A19 ve BHE' pinleri, adres zamanından sonra S3–S7 durum bitlerini taşır. Maksimum modda QS1 ve QS0 ise önden getirme kuyruğunda olanları anlatır. Bu kesim üç grubu sırayla ele alır ve bir programın her komutunda hangi kodların göründüğünü izler.

10.1.4.1 S2', S1', S0'

S2', S1' ve S0' maksimum modda 28, 27 ve 26 numaralı pinlerdir. Üç bit sekiz farklı kod verir; her kod bir yol döngüsü türünü anlatır. Kodlar önceki döngünün T4'ünde, T1'de ve T2'de geçerlidir. T3'te ya da READY = 1 iken bir Tw'de pinler pasif koda (1, 1, 1) döner.



Şekil 10.1.14: S2' S1' S0' durum kodu önceki döngünün T4'ünde, T1'de ve T2'de geçerlidir; T3'te pasif koda (111) döner. Pasif koddan başka bir koda geçiş yeni bir döngünün başladığını haber verir.

Durum pinlerinin zamanlaması 8288'in işini kolaylaştırır. Pinler döngüler arasında pasif koddadır. T4'te pasif koddan herhangi bir koda geçiş, 8288'e “yeni bir yol döngüsü başlıyor” der. 8288 bu kodu bir sonraki T1'de çözer, ALE'yi üretir ve uygun komut sinyalini hazırlar. T3'te pinlerin pasife dönmesi de döngünün sonuna yaklaşıldığını bildirir.

S2'	S1'	S0'	Döngü türü	8288 komutu
0	0	0	kesme kabul	INTA'
0	0	1	G/Ç portu oku	IORC'
0	1	0	G/Ç portuna yaz	IOWC', AIOWC'
0	1	1	durdurma (HLT)	yok
1	0	0	kod erişimi (komut getirme)	MRDC'
1	0	1	bellek oku	MRDC'

S2'	S1'	S0'	Döngü türü	8288 komutu
1	1	0	belleğe yaz	MWTC', AMWC'
1	1	1	pasif (döngü yok)	yok

Tablo 10.1.7: S2' S1' S0' kodları ve 8288'in ürettiği komut sinyalleri.

Tablo 10.1.7 bir kurala göre okunabilir. S2' = 1 ise döngü bellekle ilgilidir; S2' = 0 ise G/Ç ya da özel bir döngüdür. S1' ve S0' ikilisi 01 iken okuma, 10 iken yazma anlamına gelir. Bu kural minimum mod sinyalleriyle örtüşür: S2' minimum moddaki M/IO' ile aynı pini (28), S1' ise DT/R' ile aynı pini (27) paylaşır. 00 ve 11 ikilileri özel durumlardır: bellekte 00 komut getirmeyi, G/Ç tarafında 00 kesme kabulünü, 11 ise durdurmayı anlatır.

Durum pinleri aktif düşüktür. Bu yüzden bütün bitlerin 1 olduğu 111 kodu “hiçbir döngü yok” anlamına gelir. Bir hat kopar ya da yüzerse 1'e çekilir ve 8288 boşuna komut üretmez.

> **Sık yapılan hata**
>
> 100 (kod erişimi) ile 101 (bellek oku) kodlarının farklı komut ürettiğini sanmak yanlıştır. 8288 ikisinde de MRDC' üretir; bellek için ikisi aynı okumadır. Ayrım, yolu izleyen başka yongalar içindir. Örneğin bir hata ayıklama donanımı yalnız komut getirmelerini kaydedebilir. Maksimum modda sistem, kod ve veri için ayrı bellekler de kurabilir.

10.1.4.2 S3–S6 ve BHE'/S7

A16/S3, A17/S4, A18/S5 ve A19/S6 pinleri T1'de adresin üst dört bitini taşır, T2–T4 boyunca durum bitlerine geçer. Bu bitler iki modda da vardır.

S4 (A17)	S3 (A16)	Kullanılan kesim
0	0	ES (ek kesim)
0	1	SS (yığın)
1	0	CS ya da kesim yok (G/Ç, kesme vektörü)
1	1	DS (veri)

Tablo 10.1.8: S4 ve S3 bitlerinin gösterdiği kesim.

Tablo 10.1.8, S4 ve S3 bitlerinin o döngüde fiziksel adresi hesaplamak için hangi kesim yazmacının kullanıldığını nasıl gösterdiğini verir. Komut getirme her zaman CS ile yapılır ve 10 kodunu verir. G/Ç erişiminde ve kesme vektörü okumada adres kesim yazmacından gelmez; bunlar da 10 kodunu kullanır. Kesim öneki kodu değiştirir: `MOV AL, ES:[BX]` komutunun okuması 00 (ES) verir.

Diğer durum bitleri şunlardır:

- **S5 (A18):** IF kesme bayrağının o anki değeridir. Her saat turunun başında güncellenir. Dışarıdaki bir izleme devresi, işlemcinin kesmelere açık olup olmadığını bu bitten görür.
- **S6 (A19):** 8086'da her zaman 0'dır; yolu 8086'nın kullandığını gösterir.
- **S7 (BHE'):** T2–T4'te BHE' pininde görülen yedek durum bitidir; 8086'da bir anlamı yoktur.

S3 ve S4 bitleri ilginç bir sistem olanağı sunar. Bu bitler ALE ile kilitlenmez, ama T2'den itibaren pinlerde durur ve bir dış devre onları adres hatlarına ek olarak kullanabilir. Böylece her kesim için ayrı bir 1 MB bellek bağlanabilir: kod, veri, yığın ve ek kesim ayrı bellek bankalarına gider. Toplam adres alanı kuramsal olarak 4 MB'a çıkar. Bu düzen pratikte yaygınlaşmadı, ama durum bitlerinin işlevini iyi gösterir.

Örnek 10.1.8

Aşağıdaki erişimlerin veri döngülerinde S4 S3 kaçtır?

1. `MOV AX, [SI+2]` : taban yazmacı yok, indis SI; varsayılan kesim DS → 11.
2. `MOV AX, [BP+SI]` : adreste BP var; varsayılan kesim SS → 01.
3. `STOSB` : dizgi hedefi her zaman ES:DI → 00.
4. `CALL` ile dönüş adresinin yığına yazılması: SS → 01.
5. `OUT DX, AL` : G/Ç, kesim yok → 10.

Kesim kuralı komutun yazımından çıkar. Durum bitleri bu kuralın donanımdaki yansımasıdır.

10.1.4.3 Kod izleme: komutlar ve durum kodları

Aşağıdaki program, çeşitli yol döngüsü türleri üreten yedi komuttan oluşur. Maksimum modda çalışan bir 8086'nın her veri döngüsünde S2' S1' S0' ve S4 S3 pinlerinde ne görüldüğünü ve 8288'in hangi komutu ürettiğini izleyelim. Bellekte DS:0200h adresinde 1234h sözcüğü, 60h numaralı portta da 5Ah değeri bulunduğu varsayılır. Program .COM biçimindedir; CS = DS = SS = 0700h ve başlangıçta SP = FFFEh, AX = 0000h'dir.

```
org 100h
mov bx, 0200h
mov ax, [bx]      ; bellek oku
mov [bx+2], ax    ; belleğe yaz
in al, 60h        ; G/Ç oku
out 61h, al       ; G/Ç yaz
push ax           ; yığına yaz
hlt               ; dur
```

`mov bx, 0200h` yalnız yazmaçla çalışır. Yolda bu komut için yalnız komut getirme döngüleri görülür: S2' S1' S0' = 100, S4 S3 = 10 (CS). 8288 bu kod için MRDC' üretir; bellek, CS:IP'nin gösterdiği komut baytlarını verir.

`mov ax, [bx]` bir bellek okumasıdır: kod 101. Adres DS:0200h'dir; DS = 0700h olduğundan fiziksel adres 07000h + 0200h = 07200h olur. Adres çift olduğu için sözcük tek döngüde gelir. Veri kesimi kullanıldığı için S4 S3 = 11'dir. 8288 MRDC' üretir ve AX = 1234h olur.



Şekil 10.1.15: Solda `mov ax, [bx]` : bellek okuma, S2' S1' S0' = 101, S4 S3 = 11 (DS), 8288 MRDC' üretir. Sağda `mov [bx+2], ax` : belleğe yazma, 110, kesim yine DS, 8288 MWTC' ve AMWC' üretir.

`mov [bx+2], ax` belleğe yazar: kod 110. Etkin adres 0202h, fiziksel adres 07202h'dir; yine çift adres ve tek döngü. 8288 MWTC' (bellek yazma komutu) üretir. Ayrıca bir saat önce başlayan AMWC' (ileri bellek yazma) çıkışını da verir. Kesim DS olduğundan S4 S3 = 11 kalır. 07202h'ye 34h, 07203h'ye 12h yazılır.

`in al, 60h` bir G/Ç okumasıdır: kod 001 ve 8288 IORC' üretir. G/Ç adresi kesim yazmacından gelmediği için S4 S3 = 10'dur; A19–A16 hatları T1'de 0'dır. Port 5Ah'yi verir ve AL = 5Ah, AX = 125Ah olur. Port numarası 60h çift olduğu için bayt D7–D0 üzerinden gelir.



Şekil 10.1.16: Solda `out 61h, al` : G/Ç yazma, 010, 8288 IOWC' üretir; S2' = 0 olduğundan bellek bu döngüye karışmaz. Sağda `push ax` : yine bellek yazma (110), ama yığın kesimi kullanılır: S4 S3 = 01 (SS).

`out 61h, al` G/Ç yazmasıdır: kod 010 ve 8288 IOWC' ile AIOWC' üretir. S2' = 0 olduğundan bellek denetim mantığı bu döngüye hiç karışmaz. Port 61h tek adres olduğundan AL'deki 5Ah D15–D8 üzerinden çıkar ve BHE' = 0 olur.

`push ax` da bir bellek yazmasıdır (110), ama adres yığın kesiminden gelir. PUSH önce SP'yi 2 azaltır: SP = FFFCh. Sözcük SS:FFFCh'ye, yani fiziksel 07000h + FFFCh = 16FFCh adresine yazılır. S4 S3 = 01'dir (SS). Durum bitleri sayesinde bir sistem yığını ayrı bir belleğe koyabilir.

`hlt` işlemciyi durdurur. Maksimum modda 8086 S2' S1' S0' = 011 kodunu verir; 8288 bu kod için komut üretmez, yalnız bir ALE darbesi verir. Minimum modda ise işlemci durdurma durumuna girdiğini denetim sinyali olmayan tek bir ALE darbesiyle bildirir. İşlemci bu durumdan ancak bir kesme ya da RESET ile çıkar.

Komut	S2' S1' S0'	Döngü	S4 S3	8288 komutu	FA	AX	SP
`mov bx,0200h`	100	kod getirme	10 (CS)	MRDC'	CS:0100h	0000h	FFFEh
`mov ax,[bx]`	101	bellek oku	11 (DS)	MRDC'	07200h	1234h	FFFEh
`mov [bx+2],ax`	110	belleğe yaz	11 (DS)	MWTC', AMWC'	07202h	1234h	FFFEh
`in al,60h`	001	G/Ç oku	10	IORC'	port 0060h	125Ah	FFFEh
`out 61h,al`	010	G/Ç yaz	10	IOWC', AIOWC'	port 0061h	125Ah	FFFEh
`push ax`	110	belleğe yaz	01 (SS)	MWTC', AMWC'	16FFCh	125Ah	FFFCh
`hlt`	011	durdurma	—	yok	—	125Ah	FFFCh

Tablo 10.1.9: Durum kodları programının izlemesi (yalnız veri döngüleri; her komutun ayrıca 100 kodlu getirme döngüleri vardır).

Tablo 10.1.9'daki yazmaç ve bellek değerleri program emu8086'da çalıştırıldığında da aynı çıkar. Son durumda AX = 125Ah ve SP = FFFCh olur; 07202h'deki sözcük 1234h, yığına itilen sözcük 125Ah'dir. Durum kodları ve 8288 komutları ise emülatörde görünmez; onlar Tablo 10.1.7 ve Tablo 10.1.8'deki kurallardan çıkar. Döngü aralarında pinler 111 (pasif) koduna döner.

10.1.4.4 Durum kodunu çözmek

Beş durum pininin değerinden döngünün türü ve kullanılan kesim üç adımda bulunur. Önce S2'ye bakılır: 1 ise bellek, 0 ise G/Ç ya da özel döngü. Sonra S1' S0' ikilisi okunur: 01 okuma, 10 yazma, 00 ve 11 özel durum. Son olarak S4 S3 kesimi verir.

Örnek 10.1.9

Bir mantık çözümleyicisi T2 sırasında S2' S1' S0' = 1 0 1 ve S4 S3 = 1 1 gösteriyor. Döngü nedir ve hangi komut bu döngüyü üretmiş olabilir?

1. S2' = 1 → bellek döngüsü.
2. S1' S0' = 01 → okuma. 101 kodu “bellek oku”dur; 8288 MRDC' üretir.
3. S4 S3 = 11 → kesim DS.

DS'den bellek okuyan herhangi bir komut bu döngüyü üretir: `MOV AL, [BX]`, `ADD AX, [SI]`, `LODSB` gibi. Kod 100 olsaydı döngü bir komut getirme olurdu ve S4 S3 = 10 (CS) görülürdü.

10.1.4.5 QS1, QS0: kuyruk durumu

8086'nın önden getirme kuyruğu yonganın içindedir. Kuyruğa giren baytlar yolda görünür, ama hangi baytın ne zaman yürütülmeye başladığı dışarıdan bilinemez. Maksimum modda 24 ve 25 numaralı pinler bu bilgiyi dışarı verir.

QS1	QS0	Anlamı
0	0	işlem yok: kuyruk bu turda değişmedi
0	1	kuyruktan işlem kodunun ilk baytı alındı
1	0	kuyruk boşaltıldı (dallanma)
1	1	kuyruktan sonraki bayt alındı

Tablo 10.1.10: QS1 ve QS0 kuyruk durum kodları.

Kuyruk durumu, kuyruk işleminin yapıldığı saat turunu izleyen turda geçerlidir. Tablo 10.1.10'daki kodların başlıca kullanıcısı 8087 yardımcı işlemcisidir. 8087, 8086 ile aynı yolu dinler ve getirilen komut baytlarını kendi kuyruğuna kopyalar. QS kodlarıyla 8086'nın hangi baytı yürütmeye aldığını izler. Kuyruktan çıkan baytın bir ESC (kaçış) komutu olduğunu görünce o komutu kendisi yürütür. 01 kodu yeni bir komutun başladığını, 11 kodu aynı komutun sonraki baytlarını bildirir. 10 kodu bir dallanmadan sonra kuyruğun boşaltıldığını gösterir; 8087 de o anda kendi kopyasını temizler.

10.1.5 Kontrol pinleri

Adres ve veri pinleri “nereye” ve “ne” sorularını yanıtlar. Kontrol pinleri ise “ne zaman” ve “nasıl” sorularını yanıtlar. Aktarım okuma mı yazma mı, bellekle mi G/Ç ile mi? Alıcı-verici hangi yönde açılacak, yol başka bir yöneticiye verilecek mi? Bu kesim minimum moddaki okuma/yazma ve alıcı-verici sinyalleriyle başlar, iki komutun yol döngülerini sinyal sinyal izler. Ardından yolun paylaşılmasını sağlayan HOLD/HLDA, RQ'/GT ve LOCK' pinlerini, LOCK önekli bir semafor programıyla ve TEST' girişiyle tamamlar.

10.1.5.1 RD', WR' ve M/IO'

RD' (pin 32) aktif düşük bir çıkıştır ve işlemcinin bellekten ya da G/Ç'den okuma yaptığını bildirir. Okuma döngüsünün T2, T3 ve Tw turlarında 0'dır. Bu pin iki modda da vardır. **WR'** (pin 29, minimum mod) yazma darbesidir; yazma döngüsünün T2, T3 ve Tw turlarında 0'dır. Veri sayfası iki sinyal arasındaki ince bir farkı vurgular: WR' T2'nin başında düşer, RD' ise yolun yüzmesine zaman tanımak için T2'nin içine biraz gecikmeli düşer.

M/IO' (pin 28, minimum mod) bellek erişimini G/Ç erişiminden ayırır: 1 bellek, 0 G/Ç demektir. Sinyal, döngüden önceki T4'te geçerli olur ve döngünün son T4'üne kadar değişmez. Böylece adres çözücüler, adres daha kilitlenmeden hangi adres uzayının seçildiğini bilir. Bellek yongalarının seçim mantığına M/IO' = 1, G/Ç aygıtlarınınkine M/IO' = 0 koşulu eklenir. 8088'de bu sinyal ters kutupludur ve IO/M' adını taşır: 1 G/Ç, 0 bellek demektir. 8085 ile uyum için böyle seçilmiştir.

M/IO'	RD'	WR'	İşlem	Örnek komut
1	0	1	bellek okuma	`MOV AL, [BX]`
1	1	0	bellek yazma	`MOV [BX], AL`
0	0	1	G/Ç okuma	`IN AL, 60h`
0	1	0	G/Ç yazma	`OUT 43h, AL`

Tablo 10.1.11: M/IO', RD' ve WR' ile dört temel işlem (minimum mod).

Tablo 10.1.11'de RD' ile WR' sinyallerinin hiçbir satırda birlikte 0 olmadığı görülür; işlemci aynı anda hem okuyup hem yazamaz. Bu üç sinyalden dört ayrı komut sinyali türetilebilir. Örneğin bellek okuma komutu bir VEYA kapısıyla elde edilir: MEMR' = RD' + (M/IO')'. Çıkış yalnız RD' = 0 ve M/IO' = 1 iken 0 olur. Maksimum moddaki 8288 de aynı dört sinyali (MRDC', MWTC', IORC', IOWC') durum kodundan üretir.

Örnek 10.1.10

Minimum modda MEMR', MEMW', IOR' ve IOW' sinyallerini RD', WR' ve M/IO' sinyallerinden kapılarla üretelim. Dört sinyal de aktif düşüktür.

1. MEMR' yalnız RD' = 0 ve M/IO' = 1 iken 0 olmalı: MEMR' = RD' + (M/IO')'. M/IO' bir DEĞİL kapısından geçirilip RD' ile VEYA'lanır.
2. MEMW' = WR' + (M/IO')': WR' = 0 ve M/IO' = 1 iken 0.
3. IOR' = RD' + M/IO': RD' = 0 ve M/IO' = 0 iken 0.
4. IOW' = WR' + M/IO': WR' = 0 ve M/IO' = 0 iken 0.

Sınama: `OUT 43h, AL` döngüsünde M/IO' = 0, RD' = 1, WR' = 0'dır. MEMR' = 1 + 1 = 1, MEMW' = 0 + 1 = 1, IOR' = 1 + 0 = 1, IOW' = 0 + 0 = 0. Yalnız IOW' etkindir. Bir 74LS32 ve bir 74LS04 bu çözücü için yeter.

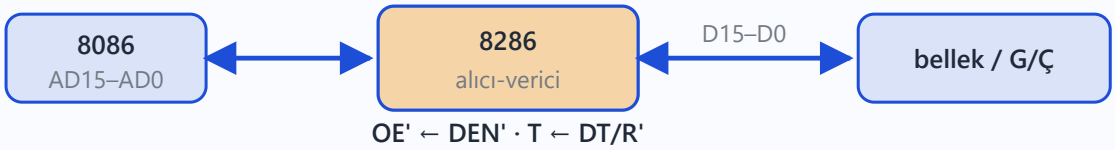
10.1.5.2 DT/R' ve DEN'

Veri yolunda 8286 ya da 74LS245 alıcı-vericileri kullanan bir sistemde işlemcinin iki sinyal daha üretmesi gerekir: alıcı-vericinin yönü ve ne zaman açılacağı.

DT/R' (*data transmit / receive*, veri gönder/al, pin 27) alıcı-vericinin yön girişine bağlanır. 1 iken veri işlemciden dışarı akar (yazma), 0 iken dışarıdan işlemciye akar (okuma). Zamanlaması M/IO' ile aynıdır: döngüden önceki T4'te geçerli olur ve döngü boyunca kalır. Maksimum moddaki S1' ile mantıksal olarak eşdeğerdir.

DEN' (*data enable*, veri etkinleştirme, pin 26) alıcı-vericinin çıkış etkinleştirme (OE') girişine bağlanır. Bellek ve G/Ç erişimlerinde ve kesme kabul döngülerinde aktif düşüktür. Okuma ve kesme kabulde T2'nin ortasından T4'ün ortasına, yazmada T2'nin başından T4'ün ortasına kadar 0 olur. T1'de

mutlaka 1'dir: bu sırada AD pinlerinde adres vardır ve alıcı-verici açık olsaydı adres veri yoluna sızardı.



Şekil 10.1.17: 8286 alıcı-vericisi: DEN' çıkışı etkinleştirir (OE'), DT/R' veri yönünü (T) seçer.

Şekil 10.1.17 bağlantıyı özetler. DEN' = 0 iken alıcı-verici açıktır ve yönü DT/R' seçer. Okumada DT/R' = 0 ve veri bellekten işlemciye, yazmada DT/R' = 1 ve veri işlemciden belleğe akar. DEN' = 1 iken iki taraf da yüzer; işlemcinin AD pinleri ile sistem veri yolu birbirinden ayrılır.

Sık yapılan hata

Okumada alıcı-vericiyi RD' ile aynı anda, hatta önce açmak çekişme yaratır. DEN' okumada T2'nin ortasında, yani işlemci AD pinlerini bıraktıktan sonra düşer. Alıcı-verici daha erken açılsaydı, işlemcinin hâlâ sürdüğü adres ile alıcı-vericinin sürmeye başladığı veri aynı hatlarda çakışırdı. Yazmada böyle bir tehlike yoktur; bu yüzden DEN' yazmada T2'nin başında düşer.

10.1.5.3 Komuttan yol döngülerine: OUT ve IN

Bir komutun yürütülmesi, birden çok yol döngüsünden oluşur: önce komut baytları getirilir, sonra komutun kendi veri döngüsü yapılır. Aşağıdaki iki komut bu zinciri minimum mod sinyalleriyle izlemeye elverişlidir. CS = 01A7h, AL = 12h kabul edilir; 9Ah numaralı port okunduğunda 3Ch değerini verir.

```
; CS = 01A7h, AL = 12h
; FA      bayt   komut
01A88h  E6 59  out 59h, al
01A8Ah  E4 9A  in  al, 9Ah
```

Komutların bağıl konumu IP = 0018h ve 001Ah'dir. Fiziksel adresler 01A70h + 0018h = 01A88h ve 01A70h + 001Ah = 01A8Ah eder. `out 59h, al` komutu E6 59 olarak kodlanır: E6 “sabit numaralı porta AL'yi yaz” işlem kodu, 59h port numarasıdır. `in al, 9Ah` ise E4 9A'dır. İzlemeyi sade tutmak için döngülerin sırayla yapıldığı varsayılır.

1. **Döngü 1, T1 (kod getirme):** Adres 01A88h çifttir: BHE' = 0, A0 = 0 ve tek döngüde bir sözcük getirilir. M/IO' = 1, ALE = 1, DT/R' = 0 (okuma), DEN' = 1.
2. **Döngü 1, T2–T4:** RD' = 0, DEN' = 0. Bellek sözcüğü verir: çift adresteki E6h (işlem kodu) D7–D0'dan, tek adresteki 59h (port) D15–D8'den gelir. İki bayt kuyruğa girer.

3. **Döngü 2, T1 (G/Ç yazma):** M/IO' = 0. Adres yalnız A7–A0'da anlamlıdır (59h); A15–A8 ve A19–A16 sıfırdır. Port numarası tektir: BHE' = 0, A0 = 1. Yazma olduğu için DT/R' = 1.
4. **Döngü 2, T2–T4:** WR' = 0, DEN' = 0. AL'deki 12h D15–D8 üzerinden çıkar; tek adres yüksek bankaya karşılık gelir. Çıkış portu veriyi genellikle WR' sinyalinin yükselen kenarında kilitler.
5. **Döngü 3, T1 (kod getirme):** Sonraki komut 01A8Ah'dedir (çift): yine tek döngüde bir sözcük getirilir. M/IO' = 1, DT/R' = 0.
6. **Döngü 3, T2–T4:** E4h (IN AL, imm8) D7–D0'dan, 9Ah (port) D15–D8'den okunur.
7. **Döngü 4, T1 (G/Ç okuma):** Port adresi 009Ah çifttir: BHE' = 1, A0 = 0 ile yalnız düşük banka seçilir. M/IO' = 0, DT/R' = 0.
8. **Döngü 4, T2–T4:** RD' = 0, DEN' = 0. Port 3Ch değerini D7–D0'a koyar; değer T4'ün başında AL'ye alınır: AL = 3Ch.

Döngü	Tür	A19–A0	BHE'	A0	D15–D8	D7–D0	M/IO'	RD'	WR'	DT/R'	AL
1	kod getirme	01A88h	0	0	59h	E6h	1	0	1	0	12h
2	G/Ç yazma	00059h	0	1	12h	—	0	1	0	1	12h
3	kod getirme	01A8Ah	0	0	9Ah	E4h	1	0	1	0	12h
4	G/Ç okuma	0009Ah	1	0	—	3Ch	0	0	1	0	3Ch

Tablo 10.1.12: OUT ve IN komutlarının yol döngüleri (minimum mod, bekleme durumu yok).

Tablo 10.1.12 dört döngünün T2–T4 değerlerini toplar; T1'de RD' ve WR' 1, DEN' 1 ve ALE 1'dir. Yol etkinliği 4 döngü × 4 T = 16 saat turudur. 5 MHz'de bu 3,2 µs eder. Program emu8086'da çalıştırıldığında makine kodları (E6 59, E4 9A) ve AL'nin son değeri 3Ch aynı çıkar; emülatör pinleri göstermez.

Bu izleme sadeleştirilmiş sıralı bir modeldir. Gerçekte BIU, kuyrukta en az iki baytlık yer açıldıkça kodu önden getirir. Döngü 3, döngü 2'den önce yapılmış olabilir; kuyruk boşsa da bazı getirmeler boşta turlarına kayabilir. Döngü türleri ve pin değerleri değişmez, yalnız sıraları değişebilir.

Sık yapılan hata

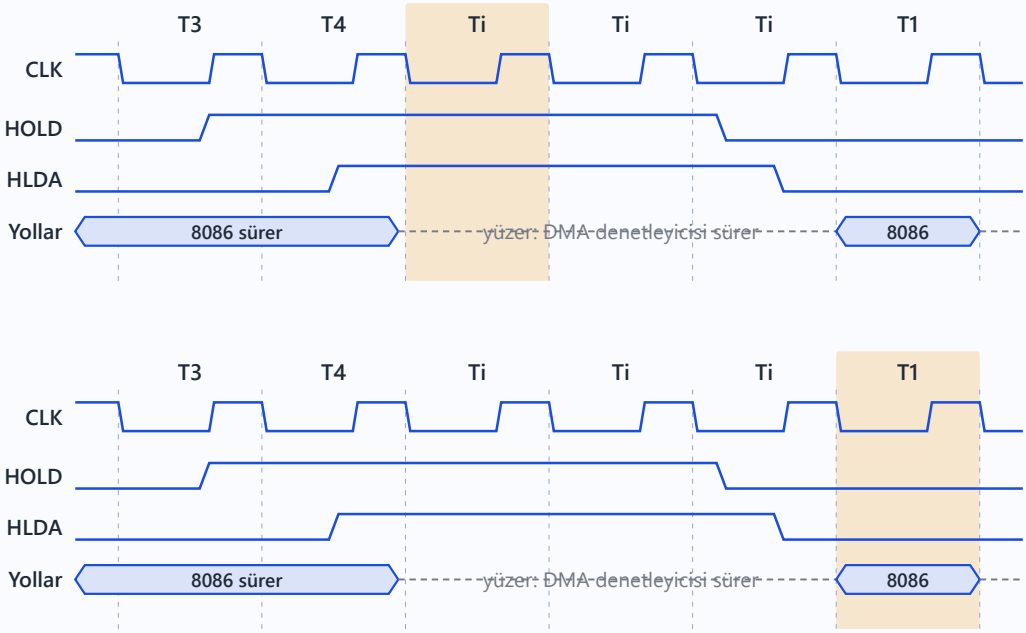
8 bitlik bir çevre birimini yalnız D7–D0'a bağlayıp ona tek numaralı bir port adresi vermek çalışmaz. Tek adresli bir port baytı D15–D8 üzerinden aktarılır; D7–D0'a bağlı aygıt bu veriyi hiç görmez. Veri sayfası bu yüzden alt yarıya bağlı 8 bitlik bir aygıtın bütün yazmaçlarına çift adres verilmesini ister. Aygıt üst yarıya bağlanmışsa yazmaçlarına tek adres verilir.

10.1.5.4 HOLD ve HLDA

Yolu kullanan tek aygıt işlemci değildir. Bir **doğrudan bellek erişimi** (DMA, *direct memory access*) denetleyicisi, örneğin 8237, bir disk denetleyicisinden gelen veriyi işlemciye uğratmadan doğrudan belleğe yazabilir. Bunun için yolu bir süre işlemciden devralması gerekir. Minimum modda bu devir HOLD ve HLDA pinleriyle yapılır.

Tanım 10.1.10

HOLD (pin 31), başka bir yol yöneticisinin yerel yolu istediğini bildiren aktif yüksek giriştir. İşlemci içinde bulunduğu yol döngüsünü bitirir. Bir T4 ya da Ti turunun ortasında **HLDA** (*hold acknowledge*, pin 30) çıkışını 1 yaparak onay verir. Onayla birlikte adres, veri ve kontrol pinlerini yüzer duruma alır. HOLD 0'a inince HLDA'yı düşürür ve yolu geri alır.



Şekil 10.1.18: HOLD/HLDA ile yolun devredilmesi. Üstte: işlemci döngüsünü bitirir ve T4'ün ortasında HLDA = 1 ile yolu bırakır; pinler yüzer. Altta: HOLD = 0 olunca işlemci HLDA'yı düşürür ve yolu geri alarak yeni bir döngüye başlar.

Şekil 10.1.18 devrin adımlarını gösterir. DMA denetleyicisi HOLD'u 1 yapar. İşlemci o anki yol döngüsünü yarıda kesmez; döngü biter ve bir sonraki T4 ya da Ti'nin ortasında HLDA = 1 olur. Aynı anda AD0–AD15, A16–A19, BHE', RD', WR', M/IO', DT/R' ve DEN' yüzer duruma geçer. ALE ise hiçbir zaman yüzmez; bu sırada 0'da kalır. DMA denetleyicisi artık yolu kendisi sürer.

Yol devredilmişken işlemci tümüyle durmaz. EU, kuyrukta bekleyen komutları yürütmeye devam edebilir. Yalnız yol gerektiren bir işe, yani bir bellek erişimine ya da kuyruğun boşalıp yeni getirme gerekmesine gelince bekler. Çarpma gibi uzun süren yazmaç komutları DMA ile örtüşebilir.

Veri sayfası HOLD isteğinin hangi döngülerde hemen kabul edilmeyeceğini de sayar. İstek T2'den sonra gelirse döngü bittikten sonraki ilk uygun anda kabul edilir. Tek adresteki bir sözcüğün ilk yarısı aktarılıyorsa ikinci yarısı da bitmeden yol bırakılmaz. Kesme kabul dizisinin iki döngüsü arasına girilmez ve LOCK önekli bir komut yürütülürken yol bırakılmaz. HOLD girişi içeride eş zamanlanmaz; kurulma süresi sağlanamıyorsa dışarıda bir flip-flopla saate eş zamanlanmalıdır.

Örnek 10.1.11

5 MHz'lik bir minimum mod sisteminde DMA denetleyicisi, bir bellek okuma döngüsünün T1'inde HOLD = 1 yapıyor. Yol en erken ne zaman devredilir?

1. İşlemci başlamış döngüyü bitirir: T2, T3 ve T4 turları sürer.
2. HLDA, T4'ün ortasında 1 olur. T1'in başından bu ana kadar 3,5 tur geçer: 3,5 × 200 = 700 ns.
3. Döngü tek adresteki bir sözcüğün ilk yarısı olsaydı, ikinci yarının döngüsü de beklenir ve gecikme 4 tur daha uzardı.

DMA'nın yanıt süresi bu yüzden birkaç yüz nanosaniye ile birkaç mikrosaniye arasında değişir; LOCK'lu uzun komutlar bu süreyi uzatır.

10.1.5.5 RQ'/GT0', RQ'/GT1' ve LOCK'

Maksimum modda yol devri HOLD/HLDA yerine **RQ'/GT0'** (pin 31) ve **RQ'/GT1'** (pin 30) pinleriyle yapılır. Adları istek (*request*) ve bağış (*grant*) sözcüklerinden gelir. Bu pinleri 8087 ya da 8089 gibi öteki yerel yol yöneticileri kullanır. Her pin çift yönlüdür ve tek bir hat üzerinde üç darbelik bir el sıkışma yürütür:

1. Yolu isteyen yönetici hatta bir saat turu genişliğinde 0 darbesi verir: istek (RQ).
2. 8086, bir T4 ya da T1 turunda aynı hatta bir tur genişliğinde darbe verir: bağış (GT). Yerel yolu yüzer duruma bırakır ve bir sonraki saat turunda yol devri durumuna girer.
3. Yönetici işini bitirince hatta üçüncü bir darbe verir: bırakma. 8086 bir sonraki turda yolu geri alır.

İki yönetici arasındaki her devirden sonra bir boş saat turu bulunmalıdır. RQ'/GT0' pininin önceliği RQ'/GT1' pininden yüksektir: iki istek aynı anda gelirse önce GT0 bağlı yönetici yolu alır. Pinlerin içinde çekme dirençleri vardır; kullanılmazlarsa boş bırakılabilirler. Tek hatta iki yönlü iletişim pin tasarrufu sağlar: minimum moddaki HOLD ve HLDA iki pin tutarken burada iki ayrı yönetici iki pinle desteklenir.

Tanım 10.1.11

LOCK' (pin 29, maksimum mod), öteki yol yöneticilerinin sistem yolunu almamasını isteyen aktif düşük çıkıştır. `LOCK` önekiyle yazılan bir komut yürütülürken aktiftir: önek çözüldükten sonraki saat turunda 0 olur ve önekli komutun son yol döngüsünün sonunda 1'e döner. Kesme kabul dizisinde de birinci döngünün T2'sinden ikinci döngünün T2'sine kadar kendiliğinden aktiftir.

LOCK' aktifken RQ'/GT pininden gelen bir istek kaydedilir, ama ancak kilit bittikten sonra kabul edilir. Böylece önekli komutun bütün yol döngüleri art arda, araya başka bir yönetici girmeden yapılır. Çok işlemcili sistemlerde LOCK' sinyali, ortak yolu yöneten hakem devresine (ör. 8289 yol hakemi) bağlanır ve sistem yolunun da kilit süresince bırakılmamasını sağlar.

10.1.5.6 Kod izleme: LOCK XCHG ile semafor

İki işlemci ortak bir bellek bölgesini kullanıyorsa, aynı anda yalnız birinin o bölgeye girmesi gerekir. Bunun için ortak bellekte bir bayrak, yani bir **semafor** tutulur: 0 “boş”, 1 “dolu” anlamına gelir. Bölgeye girmek isteyen işlemci bayrağı okur; 0 ise 1 yapar ve girer. Okuma ile yazmanın arasına öteki işlemci girmemelidir. `LOCK` önekli `XCHG` bunu sağlar.

```
org 100h
mov bx, offset bayrak
bekle: mov al, 1
lock xchg al, [bx]   ; oku + yaz
cmp al, 0             ; boş muydu?
jne bekle            ; doluysa dene
; ... kritik bölge ...
mov byte ptr [bx], 0  ; bırak
ret
bayrak db 0
```

Program .COM biçimindedir ve 100h'den başlar. Komut uzunlukları şöyledir: `mov bx, offset bayrak` 3 bayt (BB 10 01), `mov al, 1` 2 bayt, `lock xchg al, [bx]` 3 bayt (F0 86 07), `cmp al, 0` 2 bayt, `jne bekle` 2 bayt, `mov byte ptr [bx], 0` 3 bayt ve `ret` 1 bayt. Toplam 16 bayt kod 100h–10Fh arasını doldurur; `bayrak` değişkeni 0110h'dedir. BX = 0110h olur.

`mov al, 1` AL'ye bayrağa yazılacak “dolu” değerini koyar. `lock xchg al, [bx]` iki yol döngüsü yapar: önce bayrağı okur, sonra AL'deki 1'i bayrağa yazar ve okunan eski değeri AL'ye koyar. F0 baytı LOCK önekidir. Önek çözülünce LOCK' = 0 olur ve iki döngü boyunca öteki yöneticinin RQ'/GT isteği kabul edilmez. Okuma ile yazma arasına başka işlemci giremez.



Şekil 10.1.19: LOCK' ile semafor. Solda: `lock xchg` başladı, LOCK' = 0; birinci yol döngüsü bayrağı okur (0) ve öteki yöneticinin isteği bu sürede kabul edilmez. Sağda: ikinci döngü AL'deki 1'i bayrağa yazar; AL eski değeri (0) alır.

XCHG bitince LOCK' 1'e döner ve yol yeniden paylaşılabilir. `cmp al, 0` eski değeri sınar: eski değer 0 olduğu için ZF = 1 olur. `jne bekle` ZF = 1 olduğundan atlamaz; bayrak boştu ve artık bu işlemcinindir. Öteki işlemci aynı kodu çalıştırsaydı XCHG ona 1 döndürürdü, ZF = 0 olur ve döngüde beklemeye devam ederdi. Kritik bölge bitince `mov byte ptr [bx], 0` bayrağa 0 yazarak semaforu serbest bırakır.

Komut	Yol döngüleri	LOCK'	AL	bayrak [0110h]	ZF
`mov bx, offset bayrak`	—	1	00h	0	0
`mov al, 1`	—	1	01h	0	0
`lock xchg al, [bx]` (okuma)	bellek oku	0	01h	0	0
`lock xchg al, [bx]` (yazma)	belleğe yaz	0	00h	1	0
`cmp al, 0`	—	1	00h	1	1
`jne bekle`	— (atlamaz)	1	00h	1	1
`mov byte ptr [bx], 0`	belleğe yaz	1	00h	0	1

Tablo 10.1.13: Semafor programının izlemesi (bayrak başlangıçta 0).

Tablo 10.1.13'teki değerler emu8086'da çalıştırıldığında da aynı çıkar. XCHG'den sonra AL = 00h ve 0110h'deki bayrak 1'dir. CMP'den sonra ZF = 1, son komuttan sonra bayrak yeniden 0 olur. Emülatörde öteki işlemci yoktur ve LOCK' pini görünmez; kilidin etkisi veri sayfasındaki kurala göre izlenmiştir. CMP ayrıca PF = 1 yapar: sonuç 00h'dir ve bir bitlerinin sayısı (0) çifttir.

> **Sık yapılan hata**
>
> Semaforu `MOV AL, [BX]` , `CMP AL, 0` ve `MOV [BX], 1` ile ayrı komutlarla kurmak iki işlemcili bir sistemde çalışmaz. Okuma ile yazma arasında öteki işlemci yolu alıp aynı 0'ı okuyabilir. İkisi de bayrağı boş görür ve ikisi de kritik bölgeye girer. LOCK öneki olmadan XCHG bile güvenli değildir: XCHG'nin okuma ve yazma döngüleri arasında RQ'/GT ile yol devredilebilir.

10.1.5.7 TEST'

TEST' (pin 23), `WAIT` komutunun sınadığı aktif düşük bir giriştir. İşlemci `WAIT` 'i yürütürken TEST' = 0 ise hemen sonraki komuta geçer. TEST' = 1 ise boşta bekler ve pini yeniden sınar; TEST' 0 olana kadar `WAIT` 'i tekrar tekrar yürütür. Bu bekleyiş yol döngüsü harcamaz. Giriş her saat turunun yükselen kenarında içeride eş zamanlanır; veri sayfasına göre en az 5 saat turu aktif kalmalıdır.

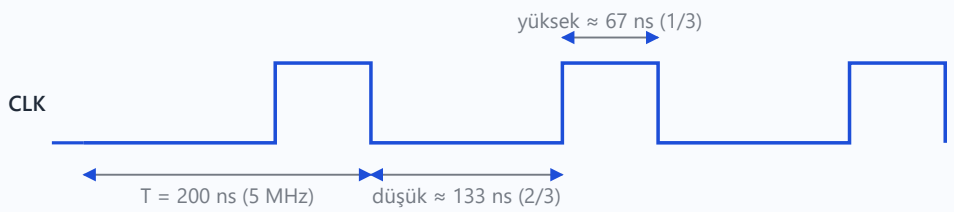
TEST' girişinin tipik kullanıcısı 8087'dir. 8087'nin BUSY çıkışı 8086'nın TEST' girişine bağlanır. 8087 bir kayan nokta işlemi yürütürken BUSY = 1'dir. 8086, sonucu okumadan önce `FWAIT` (WAIT ile aynı komut) yürütür ve 8087'nin işini bitirmesini bekler. Kesmeler açıksa bekleme sırasında bir kesme gelebilir; işlemci kesmeye hizmet eder ve dönüşte `WAIT` 'i yeniden yürütür.

10.1.6 Saat, sıfırlama ve besleme

Bir işlemcinin çalışabilmesi için üç şey gerekir: kararlı bir besleme, düzenli bir saat ve iyi tanımlanmış bir başlangıç anı. 8086'da saat ve sıfırlama sinyallerini 8284A saat üreteci hazırlar. Bu kesim CLK pinini, RESET pinini ve sıfırlamadan sonra işlemcinin ilk komutu nereden getirdiğini, ardından 8284A'nın iç işlevlerini anlatır.

10.1.6.1 CLK

CLK (pin 19), işlemcinin ve yol denetiminin temel zamanlamasını sağlayan saat girişidir. 8086'nın saati simetrik değildir: **görev çevrimi** (*duty cycle*) %33'tür. Saat bir periyodun üçte birinde 1, üçte ikisinde 0'dır. Veri sayfasına göre bu asimetri iç zamanlamayı en iyi duruma getirmek için seçilmiştir. İşlemcinin içindeki bazı işlemler saatin uzun düşük evresine, bazıları kısa yüksek evresine yerleştirilmiştir.



Şekil 10.1.20: 8284A'dan gelen %33 görev çevrimli saat (5 MHz): periyot 200 ns, yüksek evre yaklaşık 67 ns, düşük evre yaklaşık 133 ns.

8086 üç hız sınıfında üretildi: 8086 5 MHz'de, 8086-2 8 MHz'de ve 8086-1 10 MHz'de çalışır. Tablo 10.1.6'daki T_{CLCH} ve T_{CHCL} değerleri bu asimetriyi sınırlar. 5 MHz'de düşük evre en az 118 ns, yüksek evre en az 69 ns olmalıdır. Kusursuz bir %33 saatte bu değerler 133 ve 67 ns'dir; 118 ns'lik alt sınır biraz pay bırakır. Yüksek evrenin 69 ns alt sınırı ise 67 ns'nin biraz üstündedir. Bu yüzden saat üretecinin görev çevrimi tam üçte bir değil, ona çok yakın tutulur. Böyle bir saati sıradan bir osilatörle üretmek zordur; 8284A bu iş için tasarlanmıştır.

10.1.6.2 RESET, V_{CC} ve GND

Tanım 10.1.12

RESET (pin 21), işlemcinin o anki faaliyetini hemen sonlandıran aktif yüksek giriştir. Sinyal en az dört saat turu 1 kalmalıdır; güç açılışında RESET'in 0'a inmesi, besleme oturduktan en az 50 µs sonra olmalıdır. RESET 1 kaldıkça işlemci uyur. RESET 0'a indiğinde yaklaşık 10 saat turu süren bir iç başlatma dizisi çalışır, ardından işlemci FFFF0h adresindeki komuttan yürütmeye başlar.

RESET'in yükselen kenarında işlemci bütün yol işlemlerini keser. Üç durumlu bütün çıkışlar yüzer; ALE ve HLDA 0'a sürülür. Yazmaçların bir kısmı sabit değerlere getirilir, geri kalanlar belirsiz kalır. Bu sabit değerler, sistem tasarımcısının ilk komutu nereye koyacağını belirler.

Öğe	Değer
CS	FFFFh
IP	0000h
DS, SS, ES	0000h
FLAGS	temizlenir (IF = 0: kesmeler kapalı, TF = 0)
Önden getirme kuyruğu	boş
İlk komut adresi	FFFFh × 10h + 0000h = FFFF0h

Tablo 10.1.14: RESET'ten sonra işlemcinin durumu.

Tablo 10.1.14'teki ilk komut adresi FFFF0h, 1 MB'lık adres alanının son 16 baytının başıdır. Bu adreste kalıcı bir bellek, yani ROM (BIOS) bulunmalıdır; RAM'in içeriği açılışta rastgeledir. 16 bayt bir başlatma programı için çok azdır. Bu yüzden FFFF0h'ye genellikle ROM'daki asıl başlangıç koduna giden bir uzak JMP konur. IF = 0 olduğu için başlangıç kodu kesilmeden çalışır; kesme vektör tablosu henüz kurulmamıştır ve kesme gelmesi çökme demek olurdu.

V_{CC} (pin 40) +5 V DC beslemedir. GND için iki pin vardır: 1 ve 20. İki toprak pini, çıkışlar aynı anda anahtarlanırken çekilen yüksek akım darbelerini paylaştırır. Tek bir toprak pininin ince bağlantı telinden ani bir akım geçerse telin endüktansı toprak düzeyinde sıçramalar yaratır. Buna **toprak sıçraması** (*ground bounce*) denir. İki pin bu gürültüyü azaltır. Kartta besleme pinlerinin yanına 0,1 µF'lık bir dekuplaj kondansatörü konur.

> **Sık yapılan hata**
>
> Sıfırlamadan sonra işlemcinin 00000h adresinden başladığını düşünmek yanlıştır. 00000h–003FFh arası kesme vektör tablosudur ve RAM'dedir. 8086, CS = FFFFh ve IP = 0000h ile başlar; ilk fiziksel adres FFFF0h'dir. Aynı adres F000h:FFF0h gibi başka bir kesim:bağıl konum çiftiyle de yazılabilir, ama 8086'nın yüklediği değerler FFFFh ve 0000h'dir.

10.1.6.3 RESET sonrası ilk komut: adım adım

RESET'ten ilk BIOS koduna kadar işlemcinin yaptıkları sekiz adımda izlenebilir. Örnek, IBM PC'nin BIOS'unda kullanılan ilk komutu alır: FFFF0h'de beş baytlık bir uzak atlama, `EA 5B E0 00 F0`.

```
; RESET = 1 (≥ 4 saat turu)
; CS ← FFFFh, IP ← 0000h
; DS, SS, ES ← 0000h
; FLAGS ← 0000h ; IF = 0
; kuyruk boşaltılır
; RESET 1 → 0
; getir FFFF0h   ; EA 5B E0 00 F0
jmp far F000h:E05Bh
; getir FE05Bh   ; BIOS
```

1. RESET pini en az dört saat turu 1 tutulur. İşlemci o anki işini hemen bırakır; yazmaçların içeriği belirsizdir.
2. Donanım CS = FFFFh, IP = 0000h yükler. Bu değerleri programcı seçmez; işlemcinin içine yerleşiktir.
3. Öteki kesim yazmaçları DS, SS ve ES 0000h olur.
4. Bayraklar temizlenir: IF = 0 olduğu için kesmeler kapalıdır.
5. Kuyruk boşaltılır. RESET 0'a inince BIU ilk getirmeyi başlatır.
6. İlk adres FFFFh × 10h + 0000h = FFFF0h'dir. Burada ROM'daki beş baytlık uzak atlama durur: EA 5B E0 00 F0.
7. EA uzak JMP'nin işlem kodudur. Ardından önce bağıl konum (E05Bh), sonra kesim (F000h) gelir; ikisi de little-endian yazılmıştır: 5B E0 → E05Bh, 00 F0 → F000h. JMP yürütülünce CS:IP = F000h:E05Bh olur ve kuyruk yeniden temizlenir.
8. Yeni fiziksel adres F0000h + E05Bh = FE05Bh'dir. IBM PC'de BIOS'un açılış kodu bu adrestedir; 16 bayta sığmayan başlangıç programı buradan sürer.



Şekil 10.1.21: RESET sonrası ilk komut. Solda: ilk adres FFFF0h, 1 MB'ın son 16 baytı; ROM'da EA 5B E0 00 F0 durur. Sağda: uzak JMP sonrası CS:IP = F000h:E05Bh, yeni adres FE05Bh (BIOS kodu).

Adım	CS	IP	DS/SS/ES	FLAGS	Adres yolu
RESET = 1	????	????	????	????	yüzer
CS, IP yüklenir	FFFFh	0000h	????	????	—
kesimler sıfırlanır	FFFFh	0000h	0000h	????	—
bayraklar temizlenir	FFFFh	0000h	0000h	0000h	—
ilk getirme	FFFFh	0000h	0000h	0000h	FFFF0h

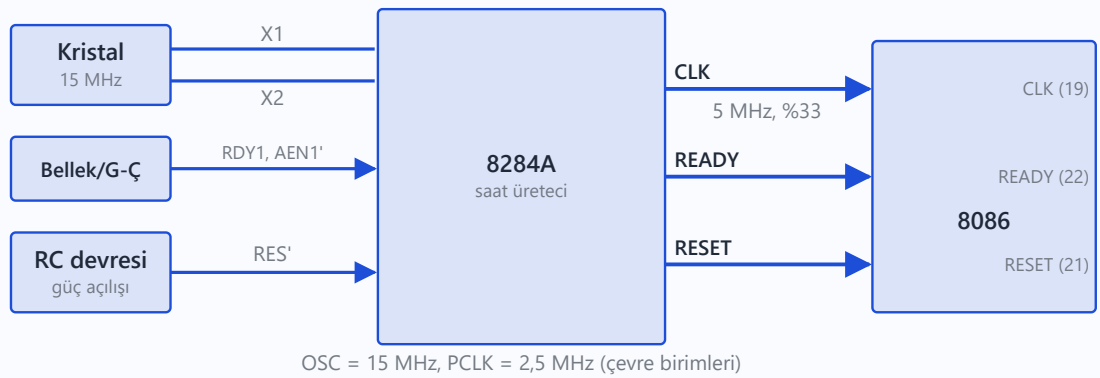
Adım	CS	IP	DS/SS/ES	FLAGS	Adres yolu
`jmp far F000h:E05Bh`	F000h	E05Bh	0000h	0000h	—
BIOS kodu getirilir	F000h	E05Bh	0000h	0000h	FE05Bh

Tablo 10.1.15: RESET'ten BIOS koduna yazmaçlar.

Tablo 10.1.15 adımları yazmaçlar üzerinden özetler. İlk getirme FFFF0h'den başlar; adres çifttir ve BIU kuyruğu sözcük sözcük doldurur: FFFF0h, FFFF2h, FFFF4h. Uzak atlamadan sonra kuyruk temizlenir ve getirme FE05Bh'den sürer. Bu adres tek olduğu için BIU ilk getirmede tek bir bayt alır, sonra çift adreslerden sözcük getirmeye devam eder.

10.1.6.4 8284A saat üreteci

8284A, 8086 ve 8088 için tasarlanmış 18 pinli bir saat üretecidir. Üç işi vardır: %33 görev çevrimli CLK sinyalini üretmek, READY'yi saate eş zamanlamak ve RESET'i saate eş zamanlamak.



Şekil 10.1.22: 8284A saat üretecinin bağlantısı: X1–X2'ye 15 MHz'lik kristal, RDY1 ve AEN1' girişlerine bellek/G-Ç hazır sinyali, RES' girişine güç açılışı RC devresi bağlanır. Çıkışlar CLK, READY ve RESET 8086'ya gider; OSC ve PCLK çevre birimleri içindir.

Saat üretimi bir kristalle başlar. X1 ve X2 pinlerine bağlanan kristal, 8284A'nın içindeki osilatörü çalıştırır. 8284A bu frekansı 3'e böler: 15 MHz'lik kristal 5 MHz'lik CLK verir. Üçe bölme %33 görev çevrimini de kendiliğinden üretir: üç kristal periyodunun birinde çıkış 1, ikisinde 0'dır. Osilatör frekansı OSC çıkışından da verilir; PCLK çıkışı ise CLK'nin yarısı frekansında, %50 görev çevrimli bir çevre birimi saatidir. 15 MHz'lik kristalle OSC = 15 MHz, PCLK = 2,5 MHz olur. F/C' girişi 1 yapılırsa saat kristal yerine EFI girişindeki dış bir kaynaktan alınır.

Örnek 10.1.12

IBM PC'de 8284A'ya 14,31818 MHz'lik bir kristal bağlanmıştır. CLK, saat periyodu ve bekleme durumsuz yol döngüsü süresi nedir? PCLK kaç MHz'tir?

1. CLK = 14,31818 / 3 = 4,77273 MHz.

2. T = 1 / 4,77273 MHz ≈ 209,5 ns.
3. Yol döngüsü 4 × 209,5 ≈ 838 ns.
4. PCLK = CLK / 2 ≈ 2,386 MHz.

14,31818 MHz, renkli televizyonun renk alt taşıyıcısının (3,579545 MHz) dört katıdır. IBM bu kristali hem işlemci saati hem de görüntü kartı için kullanarak maliyeti düşürdü. 8086 sistemlerinde de aynı hesap geçerlidir.

8284A'nın ikinci işi READY'dir. Bellek ve G/Ç aygıtlarının hazır sinyalleri RDY1 ya da RDY2 girişlerine bağlanır; AEN1' ve AEN2' girişleri hangi RDY girişinin geçerli olduğunu seçer. Bu düzen, iki ayrı yol yöneticisinin hazır sinyallerini tek bir 8284A ile işlemeye yarar. ASYNC' girişi eş zamanlamanın bir mi iki aşamalı mı olacağını belirler. 8284A, RDY'yi saat kenarlarında örnekler ve 8086'ya, kurulma ve tutma koşullarını sağlayan bir READY sinyali verir. 8086'nın READY girişi içeride eş zamanlanmadığı için bu adım zorunludur.

Üçüncü iş RESET'tir. Güç açılışında besleme 0 V'tan 5 V'a birkaç milisaniyede yükselir. RES' girişine bağlanan bir RC devresi, kondansatör dolarken bu girişte yavaş yükselen bir gerilim oluşturur. RES' girişinde bir Schmitt tetikleyici vardır: gerilim belirli bir eşiği geçince çıkış temiz bir kenarla değişir ve yavaş yükselen gerilim gürültülü bir sinyale dönüşmez. 8284A bu sinyali saate eş zamanlar ve aktif yüksek RESET olarak 8086'ya verir. Kartta bir “reset” düğmesi de kondansatörü boşaltarak aynı devreyi tetikler.

Örnek 10.1.13

RES' girişine R = 10 kΩ ve C = 10 µF'lık bir RC devresi bağlanmıştır. RES' girişinin Schmitt eşiği yaklaşık 1,7 V kabul edilsin (V_{CC} = 5 V). Güç açıldıktan sonra RESET ne kadar süre aktif kalır?

1. Kondansatör gerilimi $v(t) = 5 \cdot (1 - e^{-t/RC})$ ile yükselir; $RC = 10.000 \times 10 \times 10^{-6} = 0{,}1$ s.
2. $v(t) = 1{,}7$ V için $1 - e^{-t/RC} = 0{,}34 \rightarrow e^{-t/RC} = 0{,}66 \rightarrow t = -0{,}1 \cdot \ln 0{,}66 \approx 0{,}0416$ s.
3. RESET yaklaşık 42 ms aktif kalır. Bu süre, veri sayfasının istediği 50 µs'den çok uzundur ve beslemenin oturmasına da zaman bırakır.

Eşik değeri 8284A'nın veri sayfasından alınmalıdır; burada hesap yöntemini göstermek için yaklaşık bir değer kullanıldı.

10.1.7 Maksimum mod

Minimum mod, tek işlemcili küçük sistemler için yeterlidir. Sisteme 8087 yardımcı işlemcisi, 8089 G/Ç işlemcisi ya da ikinci bir 8086 eklenince kontrol sinyallerini ayrı bir yonga üretmelidir. 8086'nın da yol paylaşımını destekleyen pinlere ihtiyacı olur. Bu kesim moda göre değişen pinleri bir kez daha toplar, sonra 8288 yol denetleyicisini ve tam bir maksimum mod sistemini anlatır.

10.1.7.1 MN/MX' ve moda göre pinler

Minimum modda (MN/MX' = 1) 24–31 numaralı pinler şu sinyalleri taşır:

- **ALE** (25): adres kilidi etkinleştirme. Her yol döngüsünün T1'inde aktif yüksek bir darbedir ve 8282/8283 kilitlerini yükler. Veri sayfası ALE'nin hiçbir zaman yüzmediğini özellikle belirtir: yol devredildiğinde bile ALE sürülmeye devam eder.
- **INTA'** (24): kesme kabul.
- **DEN', DT/R', M/IO', WR'** (26–29): alıcı-verici denetimi ve bellek/G-Ç ayrımı.
- **HOLD, HLDA** (31, 30): yol devri (DMA).

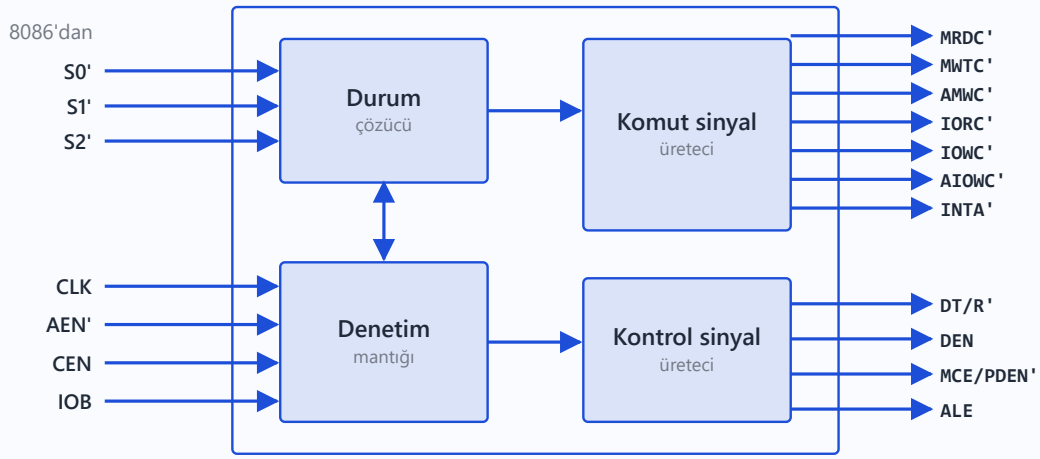
Maksimum modda (MN/MX' = 0) aynı pinler şu sinyallere dönüşür:

- **S2', S1', S0'** (28–26): döngü türü; 8288 bu koddan kontrol sinyallerini üretir.
- **QS1, QS0** (24, 25): kuyruk durumu; 8087 için.
- **LOCK'** (29): yolu kilitler.
- **RQ'/GT0', RQ'/GT1'** (31, 30): yol isteği ve bağışı.

ALE'nin yüzmemesinin bir nedeni vardır. Yol devredildiğinde başka bir yönetici adresi kendi kilitleriyle ya da doğrudan sürer. 8086'nın ALE'si yüzseydi, çekme dirençlerinin belirsiz düzeyi kilitleri yanlışlıkla saydam yapabilirdi. ALE 0'da sürülünce 8086'nın kilitleri kapalı kalır ve yol devri sırasında adres bozulmaz.

10.1.7.2 8288 yol denetleyicisi

> **Tanım 10.1.13**
>
> **8288 yol denetleyicisi** (*bus controller*), maksimum modda 8086'nın S2'–S0' durum kodunu çözen 20 pinli yongadır. Bu koddan bellek ve G/Ç komut sinyallerini (MRDC', MWTC', AMWC', IORC', IOWC', AIOWC', INTA') ve yol denetim sinyallerini (ALE, DEN, DT/R', MCE/PDEN') üretir.



Şekil 10.1.23: 8288 yol denetleyicisinin iç yapısı: durum çözücü S2'–S0' kodunu çözer, denetim mantığı CLK, AEN', CEN ve IOB girişleriyle zamanlamayı kurar; komut ve kontrol sinyali üreteçleri çıkışları verir.

8288'in girişleri 8086'nın durum pinleri ve 8284A'nın saatidir. İçerideki durum çözücü Tablo 10.1.7'deki kodu çözer. Denetim mantığı komutların zamanlamasını saatle kurar. Çıkışlar iki gruptur:

- **Komut sinyalleri:** MRDC' (bellek oku), MWTC' (bellek yaz), IORC' (G/Ç oku), IOWC' (G/Ç yaz) ve INTA' (kesme kabul). "A" ile başlayan AMWC' ve AIOWC' "ileri" yazma sinyalleridir: normal yazma sinyalinden bir saat turu önce başlarlar. Yavaş bir bellek ya da G/Ç aygıtı yazma işlemine erken başlayabilsin diye vardır.
- **Kontrol sinyalleri:** ALE, DEN ve DT/R' artık 8288'den gelir. DEN burada aktif yüksektir; alıcı-vericiye bağlanırken bir tersleyici gerekebilir. MCE/PDEN' çıkışı, kesme denetleyicilerinin ardışık bağlanması ya da G/Ç yolunun ayrı tamponlanması için kullanılır.

8288'in denetim girişleri de vardır. AEN' (*address enable*) 1 iken komut çıkışları yüzer; çok işlemcili sistemde yol hakemi, yolu başka bir işlemciye verdiğinde bu girişi kullanır. CEN (*command enable*) 0 iken komut çıkışları pasif kalır. IOB (*I/O bus mode*) 1 iken 8288 G/Ç komutlarını yerel bir G/Ç yolu için üretir. Pinleri boşalan 8086 ise bu sinyallerin yerine QS1, QS0, LOCK' ve RQ'/GT pinlerini kullanır.

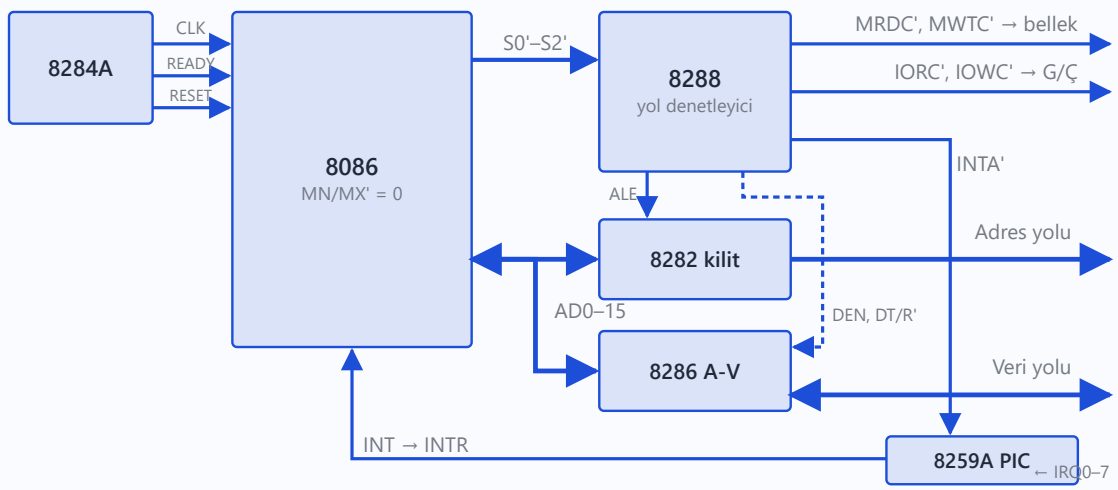
Örnek 10.1.14

Maksimum modda `MOV [BX], AX` komutunun veri döngüsünde 8288 hangi sinyalleri üretir? BX çift bir adres gösteriyor.

1. 8086 önceki döngünün T4'ünde S2' S1' S0' = 110 (belleğe yaz) verir.
2. 8288 T1'de ALE darbesini üretir; adres kilitlere alınır. DT/R' = 1 (gönder) olur.
3. AMWC' T2'nin başında, MWTC' bir saat turu sonra 0 olur. DEN = 1 alıcı-vericiyi açar.
4. T3'te durum pinleri pasife (111) döner; 8288 komutları T4'te sonlandırır.

Aynı döngü minimum modda WR' = 0, M/IO' = 1, DT/R' = 1 ve DEN' = 0 ile yapılırdı; sinyallerin işlevi aynı, üreten yonga farklıdır.

10.1.7.3 Maksimum mod 8086 sistemi



Şekil 10.1.24: Maksimum mod 8086 sistemi: 8284A saati, 8288 komut ve kontrol sinyallerini, 8282 kilitleri adres yolunu, 8286 alıcı-vericileri veri yolunu üretir; 8259A kesme denetleyicisi sekiz donanım kesmesini (IRQ0–IRQ7) INTR'ye bağlar.

Şekil 10.1.24 tam bir maksimum mod sistemini gösterir. 8284A, CLK'yi hem 8086'ya hem 8288'e verir; ikisi aynı saatle çalışmalıdır. 8086'nın S2'–S0' pinleri 8288'e gider. 8288 ALE ile 8282 kilitlerini, DEN ve DT/R' ile 8286 alıcı-vericilerini denetler; komut sinyalleriyle de bellek ve G/Ç'yi sürer. 8259A programlanabilir kesme denetleyicisi, sekiz donanım kesme isteğini öncelik sırasına koyup INTR girişine bağlar ve INTA' darbelerine tip numarasıyla yanıt verir. Bu sistemde 8087 eklenecekse QS1, QS0 ve RQ'/GT0' pinlerine bağlanır; BUSY çıkışı da TEST' girişine gider.

Minimum ve maksimum mod sistemleri yazılım açısından tamamen aynıdır. Bir program, hangi modda çalıştığını anlayamaz. Fark yalnız donanım tasarımındadır: yonga sayısı, yolun paylaşılabilmesi ve ek işlemcilerin desteklenmesi.

10.1.8 Kesme pinleri

Donanım kesmelerinin yazılım yanı, yani vektör tablosu, IRET ve kesme hizmet yordamları, kesme mekanizmasıyla birlikte ele alınır. Bu kesim aynı konuyu pinler düzeyinde inceler: kesme isteğinin işlemciye hangi pinden, hangi zamanlamayla girdiği ve işlemcinin tip numarasını kesme denetleyicisinden hangi yol döngüleriyle okuduğu.

10.1.8.1 INTR, NMI ve INTA'

INTR (pin 18), maskelenebilir kesme isteği girişidir. Aktif yüksektir ve **düzey tetiklemelidir**: istek, hat 1'de tutulduğu sürece geçerlidir. İşlemci INTR'yi her komutun son saat turunda örnekler; dizgi komutlarında her tek aktarımın sonunda da örnekler. IF = 1 ise isteği kabul eder ve kesme kabul döngülerine girer. IF = 0 ise istek bekler; CLI komutu bu yüzden INTR'yi kapatır. Giriş her saat turunun yükselen kenarında içeride eş zamanlanır.

NMI (pin 17), maskelenemeyen kesme girişidir. **Kenar tetiklemelidir**: 0'dan 1'e geçiş isteği başlatır. Yükselen kenar yonganın içinde kilitlenir ve o anki komutun sonunda hizmet edilir. NMI her zaman tip 2 kesmeye yol açar; tip numarası dışarıdan okunmaz. Yazılımla, yani IF ile maskelenemez ve INTR'den yüksek önceliklidir. Veri sayfası NMI'nın en az iki saat turu 1 kalmasını ister. Tipik kullanımı güç kesintisi uyarısı ve bellek eşlik hatasıdır.

INTA' (pin 24, minimum mod), kesme kabul döngülerinin okuma darbesidir. Aktif düşük bir çıkıştır ve her kesme kabul döngüsünün T2, T3 ve Tw turlarında 0'dır. Maksimum modda INTA' sinyalini 8288 üretir.

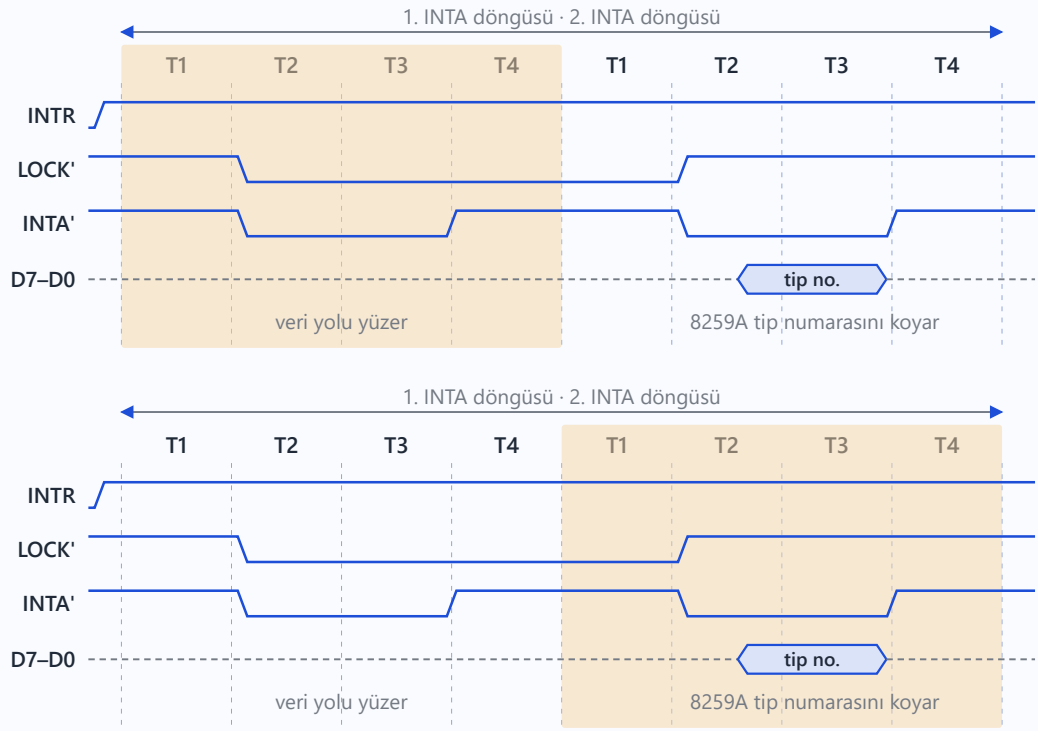
Özellik	INTR (18)	NMI (17)
Tetikleme	düzey, aktif yüksek	yükselen kenar
Maskeleme	IF = 0 ile (CLI)	maskelenemez
Örnekleme	komutun son saat turunda	kenar içeride kilitlenir, komut sonunda hizmet
Tip numarası	ikinci INTA' döngüsünde D7–D0'dan okunur	her zaman 2
Kabul döngüsü	iki INTA' döngüsü	yok
Kullanım	8259A üzerinden çevre birimleri	güç kesintisi, eşlik hatası

Tablo 10.1.16: INTR ve NMI pinlerinin karşılaştırması.

> **Sık yapılan hata**
>
> INTR'yi kenar tetiklemeli sanmak iki tür hataya yol açar. Kısa bir darbe, işlemci onu örneklemeden önce biterse kaybolur; INTR, kabul edilene kadar 1'de tutulmalıdır. Öte yandan hizmet yordamı bitince istek hâlâ 1'deyse işlemci aynı kesmeye yeniden girer. Bu yüzden istek kaynağı, kabulü görünce isteğini düşürmelidir; 8259A bunu INTA' darbeleriyle yapar. Kenar tetiklemeli olan NMI'dır.

10.1.8.2 Donanım kesmesi: INTA döngüleri

INTR = 1 ve IF = 1 iken o anki komut bitince işlemci arka arkaya iki **kesme kabul döngüsü** (*interrupt acknowledge cycle*) çalıştırır. Kesme kabul döngüsü bir okuma döngüsüne benzer; fark, RD' yerine INTA' sinyalinin etkinleşmesi ve adres yolunun yüzmesidir.



Şekil 10.1.25: İki INTA' döngüsü. Üstte birinci döngü: INTA' darbesi kesme denetleyicisine isteğin kabul edildiğini bildirir; veri yolu yüzer kalır. Altta ikinci döngü: 8259A tip numarasını D7–D0'a koyar ve işlemci okur.

1. **Birinci INTA' döngüsü:** İşlemci INTA' sinyalini T2–T3 boyunca 0 yapar. Bu darbe, kesme denetleyicisine (8259A) isteğin kabul edildiğini bildirir. 8259A, bekleyen istekler arasında en yüksek öncelikliyi seçer ve tip numarasını hazırlar. Bu döngüde veri yolu yüzer kalır; hiçbir veri okunmaz. Veri sayfasına göre BHE' bu döngünün T1'inde 0'dır.
2. **İkinci INTA' döngüsü:** İkinci INTA' darbesinde 8259A tip numarasını D7–D0 hatlarına koyar. Örneğin IRQ0 için PC'de bu sayı 08h'dir. İşlemci baytı T4'ün başında okur.
3. **Vektör:** İşlemci tip numarasını 4 ile çarpar ve kesme vektör tablosundaki adresi bulur: 08h × 4 = 00020h. Bu adresten yeni IP'yi, 00022h'den yeni CS'yi okur. FLAGS, CS ve IP'yi yığına iter, IF ve TF'yi temizler ve hizmet yordamına atlar.

Maksimum modda iki döngü arasına başka bir yol yöneticisinin girmemesi için LOCK' kendiliğinden aktiftir. Veri sayfasına göre LOCK', birinci döngünün T2'sinden ikinci döngünün T2'sine kadar 0'dır. Yol devri isteği de ikinci döngünün sonuna kadar kabul edilmez. Araya bir DMA döngüsü girseydi 8259A'nın iki darbelik el sıkışması bozulabilirdi.

Örnek 10.1.15

5 MHz'lik bir minimum mod sisteminde 8259A, IRQ1 (klavye) isteğini tip 09h ile bildiriyor. İki INTA' döngüsü ve vektör okuması kaç yol döngüsü ve ne kadar süre alır? Hizmet yordamının adresi hangi fiziksel adreslerden okunur?

1. İki kesme kabul döngüsü: 2 × 800 ns = 1,6 µs. İkinci döngüde D7–D0'da 09h okunur.

2. Vektör adresi 09h × 4 = 00024h. IP 00024h'den, CS 00026h'dan okunur; iki adres de çift olduğu için iki sözcük okuması 2 döngüdür.
3. FLAGS, CS ve IP'nin yığına itilmesi 3 sözcük yazmasıdır; yığın çift adresteyse 3 döngü.
4. Toplam en az 2 + 2 + 3 = 7 yol döngüsü: 7 × 800 ns = 5,6 µs. Buna iç işlem turları ve hizmet yordamının ilk komutlarının getirilmesi eklenir.

Yığın tek adreste olsaydı her itme iki döngü sürer ve süre 3 döngü uzardı. Veri sayfasının yığını çift adreste tutma önerisi bu yüzdendir.

10.1.9 Alıştırmalar ve çözümleri

Aşağıdaki sekiz alıştırma pinler, yol döngüleri ve saat üzerine bilgiyi bir arada kullanır. Her birinin çözümü ara adımlarıyla verilir.

Örnek 10.1.16

8086 8 MHz'de çalışıyor. Bekleme durumu olmayan bir yol döngüsü kaç ns sürer? İki Tw eklenirse?

1. T = 1 / 8 MHz = 125 ns.
2. Bekleme durumsuz döngü: 4 × 125 = 500 ns.
3. İki Tw ile: (4 + 2) × 125 = 6 × 125 = 750 ns.

Örnek 10.1.17

Minimum modda bir çevre birimine `OUT` ile yazılırken M/IO', RD', WR' ve DT/R' pinleri veri döngüsünün T2–T4 turlarında hangi değerleri alır?

1. Hedef G/Ç'dir: M/IO' = 0.
2. İşlem yazmadır: RD' = 1, WR' = 0.
3. Veri işlemciden dışarı akar: DT/R' = 1.

DEN' de T2–T4 boyunca 0'dır; T1'de 1'dir.

Örnek 10.1.18

`MOV [2003h], AX` kaç veri yol döngüsü gerektirir? Her döngüde BHE' ve A0 ne olur? (DS = 0000h.)

1. Fiziksel adres 02003h tektir; sözcük iki döngüye bölünür.

2. 1. döngü: adres 02003h, BHE' = 0, A0 = 1; AL D15–D8 üzerinden yüksek bankaya yazılır.
3. 2. döngü: adres 02004h, BHE' = 1, A0 = 0; AH D7–D0 üzerinden düşük bankaya yazılır.

AX = BEEFh ile emu8086'da çalıştırıldığında 02003h'e EFh, 02004h'e BEh yazılır.

Örnek 10.1.19

S2' S1' S0' = 1 0 0 ile 1 0 1 arasındaki fark nedir? 8288 her ikisi için hangi komutu üretir?

1. 100: kod (komut) getirme. 101: veri okuma.
2. İkisi de bellekten okumadır; 8288 ikisinde de MRDC' üretir.

Fark bellek için değil, 8087 ya da bir hata ayıklama donanımı gibi yolu izleyen yongalar içindir; onlar komut baytlarını veri okumalarından ayırmak zorundadır.

Örnek 10.1.20

8259A, IRQ0 isteğini tip 08h ile bildiriyor. İki INTA' darbesinde veri yolunda ne görülür? Maksimum modda LOCK' bu sırada ne durumdadır?

1. 1. darbe: veri yolu yüzer; yalnız kabul bildirilir.
2. 2. darbe: 8259A 08h'yi D7–D0'a koyar; işlemci okur.
3. LOCK', 1. döngünün T2'sinden 2. döngünün T2'sine kadar 0 (aktif) kalır.

İşlemci daha sonra vektörü 08h × 4 = 00020h adresinden okur.

Örnek 10.1.21

Kristal 24 MHz olursa 8284A hangi CLK frekansını üretir? Bu, hangi 8086 sürümü için uygundur?

1. CLK = 24 / 3 = 8 MHz, T = 125 ns.
2. 8 MHz 8086-2'nin hızıdır. 5 MHz'lik 8086 bu saatte çalıştırılamaz: T_{CLCL} en az 200 ns olmalıdır.

Örnek 10.1.22

T = 125 ns olan bir sistemde $T_{CLAV} + T_{DVCL} + t_{tampon}$ = 120 ns kabul ediliyor. Erişim süresi 450 ns olan bir bellek için kaç Tw gerekir?

1. $t_{izin} = 3 \times 125 - 120 = 375 - 120 = 255$ ns.

2. n = ⌈(450 − 255) / 125⌉ = ⌈195 / 125⌉ = ⌈1,56⌉ = 2.
3. Sınama: 1 Tw ile 380 ns, yetmez; 2 Tw ile 505 ns, yeter.
4. Döngü 6 × 125 = 750 ns.

Örnek 10.1.23

`OUT 9Bh, AL` komutunun G/Ç yazma döngüsünde A7–A0, BHE', A0 ve verinin geçtiği hatlar nedir?

1. Port numarası adres hatlarına çıkar: A7–A0 = 9Bh, A19–A8 = 0. M/IO' = 0.
2. 9Bh tektir (1001 1011b, en düşük bit 1): BHE' = 0, A0 = 1.
3. Tek adresli bayt yüksek bankaya gider: AL D15–D8 üzerinden çıkar.

Bu port D7–D0'a bağlı 8 bitlik bir aygıtta olsaydı veriyi hiç görmezdi.

10.1.10 Özet

- **Kılıf:** 8086, 40 pinli DIP kılıfta, tek +5 V beslemeyle (pin 40; toprak 1 ve 20) çalışan 16 bitlik HMOS işlemcidir. Adın sonundaki kesme işareti aktif düşük sinyali gösterir. Pinler yedi gruba ayrılır: güç, saat/sıfırlama, adres/veri, adres/durum, kesme, ortak kontrol ve moda bağlı pinler.
- **Çoklama:** 60'ı aşan sinyal 40 pine zaman paylaşımlı çoklamayla sığar. AD0–AD15 T1'de adres, sonra veri; A16–A19 T1'de adres, sonra S3–S6 taşır. ALE darbesinin düşen kenarında 8282/74LS373 kilitleri adresi yakalar; 20 bit ve BHE' için üç kilit gerekir. 8286/74LS245 alıcı-vericileri veri yolunu ayırır ve sürme gücü sağlar.
- **Bankalar:** 1 MB bellek iki 512 KB'lık bankaya bölünür: çift adresler D7–D0'da (A0 = 0), tek adresler D15–D8'de (BHE' = 0). Çift adresteki sözcük tek döngü, tek adresteki sözcük iki döngü sürer; bayt erişimi her zaman tek döngüdür.
- **Yol döngüsü:** en az dört T durumundan oluşur: T1 adres, T2 yön/kontrol, T3 veri, T4 bitiş. 5 MHz'de 800 ns. Yazmada veri T2'den itibaren pinlerdedir; okumada T2'de AD pinleri yüzer (yol dönüşü) ve veri T4'ün başında örneklenir.
- **READY:** T3 sırasında örneklenir; 0 ise T3 ile T4 arasına Tw eklenir. Bekleme durumu sayısı n = $\lceil (t_{erişim} - t_{izin}) / T \rceil$ ile bulunur; $t_{izin} = 3T - T_{CLAV} - T_{DVCL} - t_{tampon}$. READY, 8284A'nın RDY girişinden eş zamanlanarak verilir.
- **Durum pinleri:** S2' S1' S0' (maksimum mod) döngü türünü kodlar: 000 kesme kabul, 001 G/Ç oku, 010 G/Ç yaz, 011 HLT, 100 kod getirme, 101 bellek oku, 110 bellek yaz, 111 pasif. S4 S3 kesimi verir (00 ES, 01 SS, 10 CS/yok, 11 DS); S5 = IF, S6 = 0. QS1 QS0 kuyruk durumunu 8087'ye bildirir.

- **Kontrol pinleri:** RD' ve WR' okuma/yazma darbeleridir, M/IO' bellek ile G/Ç'yi ayırır. DT/R' alıcı-vericinin yönünü, DEN' açılmasını denetler; DEN' T1'de 1'dir. HOLD/HLDA (minimum) ve RQ'/GT (maksimum) yolu başka bir yöneticiye devreder. LOCK' önekli komut boyunca yolu kilitler; LOCK XCHG ile bölünmez semafor kurulur. TEST', WAIT komutunun beklediği giriştir.
- **Saat ve sıfırlama:** 8284A kristali 3'e böler (15 MHz → 5 MHz), %33 görev çevrimli CLK üretir, READY ve RESET'i eş zamanlar. RESET en az 4 saat turu 1 tutulur; ardından CS = FFFFh, IP = 0000h, DS = SS = ES = 0000h, bayraklar temiz (IF = 0) ve ilk komut FFFF0h'den getirilir.
- **Maksimum mod:** MN/MX' = 0 iken 24–31 numaralı pinler QS1, QS0, S0'–S2', LOCK' ve RQ'/GT pinlerine dönüşür; 8288 yol denetleyicisi durum kodundan MRDC', MWTC', AMWC', IORC', IOWC', AIOWC', INTA', ALE, DEN ve DT/R' üretir.
- **Kesme pinleri:** INTR düzey tetiklemelidir, IF ile maskelenir ve iki INTA' döngüsüyle kabul edilir; ikinci döngüde tip numarası D7–D0'dan okunur. NMI yükselen kenarla tetiklenir, maskelenemez ve her zaman tip 2'dir.

10.1.11 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

çift sıralı kılıf (DIP) *dual in-line package* · aktif düşük / aktif yüksek *active low / active high* · çoklama *multiplexing* · kilit *latch* · tampon *buffer (bus transceiver)* · üç durumlu çıkış, yüksek empedans *tri-state output, high impedance* · bellek bankası *memory bank* · hizalı / hizasız erişim *aligned / unaligned access* · yol döngüsü *bus cycle* · T durumu *T state* · bekleme durumu (Tw) *wait state* · boşta durumu (Ti) *idle state* · yol dönüşü *bus turnaround* · erişim süresi *access time* · durum kodu *status code* · minimum / maksimum mod *minimum / maximum mode* · yol denetleyicisi (8288) *bus controller* · saat üreteci (8284A) *clock generator* · görev çevrimi *duty cycle* · sıfırlama (RESET) *reset* · doğrudan bellek erişimi (DMA) *direct memory access* · yol yöneticisi *bus master* · semafor *semaphore* · kesme onayı (INTA) *interrupt acknowledge* · maskelenemeyen kesme (NMI) *non-maskable interrupt* · programlanabilir kesme denetleyicisi (8259A) *programmable interrupt controller*

10.1.12 Kendini Dene

1. 8086-2 (8 MHz) için veri sayfası $T_{CLAV} \le 60$ ns ve $T_{DVCL} \ge 20$ ns veriyor; $t_{tampon} = 40$ ns. Erişim süre-

si 300 ns olan bir bellek için kaç bekleme durumu gerekir ve döngü kaç ns sürer?

Cevap

T = 125 ns; t_{izin} = 375 − 60 − 20 − 40 = 255 ns. n = ⌈(300 − 255) / 125⌉ = ⌈0,36⌉ = 1. Sınama: 0 Tw ile 255 ns yetmez, 1 Tw ile 380 ns yeter. Döngü 5 × 125 = 625 ns.

2. DS = 1000h iken `MOV [1FFFh], AX` çalışıyor. Hangi fiziksel adreslere kaç döngüde yazılır? Her döngüde BHE', A0 ve kullanılan hatlar nedir?

Cevap

FA = 10000h + 1FFFh = 11FFFh, tek adres; iki döngü. 1. döngü: 11FFFh, BHE' = 0, A0 = 1, AL D15–D8 üzerinden. 2. döngü: 12000h, BHE' = 1, A0 = 0, AH D7–D0 üzerinden.

3. Maksimum modda şu döngülerde S2' S1' S0' ve S4 S3 kaçtır: `POP BX`'in okuması, `IN AL, DX`'in okuması, `MOV ES:[DI], AL`'nin yazması?

Cevap

`POP BX` : bellek oku 101, kesim SS → 01. `IN AL, DX` : G/Ç oku 001, kesim yok → 10. `MOV ES:[DI], AL` : belleğe yaz 110, kesim ES → 00.

4. Minimum modda bir komut getirme döngüsünün T2–T4 turlarında M/IO', RD', WR', DT/R' ve DEN' hangi değerleri alır? Bu değerler `MOV AL, [BX]`'in veri döngüsünden farklı mıdır?

Cevap

M/IO' = 1, RD' = 0, WR' = 1, DT/R' = 0, DEN' = 0. Değerler bir bellek okumasıyla aynıdır; kontrol pinlerinden komut getirme ile veri okuma ayırt edilemez. Ayrımı T2–T4'teki S4 S3 bitleri (getirmede 10 = CS, `[BX]` okumasında 11 = DS) ve maksimum modda durum kodu (100 / 101) yapar.

5. Adres kilidi olarak yükselen kenarda yakalayan bir D flip-flop yongası kullanılıp saat girişine doğrudan ALE bağlanırsa ne olur? 5 MHz'lik 8086'nın zaman değerleriyle açıklayın.

Cevap

Flip-flop adresi ALE'nin yükselen kenarında yakalar. ALE, T1'in başından en geç T_{CLLH} = 80 ns sonra yükselir; adres ise en geç T_{CLAV} = 110 ns sonra geçerli olur. Yakalama anında adres henüz oturmamış olabilir ve yanlış adres kilitlenir. Çözüm ALE'yi tersleyip yakalamayı düşen kenara kaydırmak ya da saydam kilit kullanmaktır.

6. 8284A'ya 30 MHz'lik bir kristal bağlanıyor. CLK, PCLK ve OSC frekansları nedir? Bekleme durum-

suz yol döngüsü kaç ns sürer ve saatin yüksek evresi yaklaşık kaç ns'dir?

Cevap

CLK = 30 / 3 = 10 MHz (8086-1), PCLK = 5 MHz, OSC = 30 MHz. T = 100 ns; yol döngüsü 400 ns. Yüksek evre yaklaşık T / 3 ≈ 33 ns, düşük evre yaklaşık 67 ns.

7. RESET'ten sonra FFFF0h adresinde `EA 00 E0 00 F0` baytları bulunuyor. Bu komut ne yapar ve işlemci sonraki komutu hangi fiziksel adresten getirir? O anda IF kaçtır?

Cevap

EA uzak JMP'dir. Bağıl konum 00 E0 → E000h, kesim 00 F0 → F000h. CS:IP = F000h:E000h, fiziksel adres F0000h + E000h = FE000h. Sıfırlama bayrakları temizlediği için IF = 0'dır; kesmeler kapalıdır.

8. Bir semafor `MOV AL, [BX]` / `CMP AL, 0` / `JNE bekle` / `MOV BYTE PTR [BX], 1` dizisiyle kuruluyor. İki işlemcili bir sistemde neden güvenli değildir? Kaç bayt ve hangi önekle düzeltilir?

Cevap

Okuma ile yazma ayrı komutlardadır; arada öteki işlemci yolu alıp aynı 0'ı okuyabilir ve ikisi birden kritik bölgeye girer. Düzeltme: `MOV AL, 1` ve `LOCK XCHG AL, [BX]`. XCHG okuma ve yazmayı tek komutta yapar; LOCK öneki (F0h, 1 bayt) iki döngü arasında yolun devredilmesini engeller.

9. Minimum modda bir DMA denetleyicisi, bir okuma döngüsünün T3 turunda HOLD = 1 yapıyor. İşlemci bu döngünün T4'ünde yolu bırakır mı?

Cevap

Bırakmaz. Veri sayfasına göre yolun o döngünün T4'ünde bırakılması için isteğin en geç T2'de gelmiş olması gerekir. T3'te gelen istek, bir sonraki uygun T4 ya da boşta turunda kabul edilir; HLDA o turun ortasında 1 olur.

10.1.13 İleri okuma

- Intel, *8086 16-Bit HMOS Microprocessor* veri sayfası (231455): bu notun sayısal değerleri pin açıklamalarından, zamanlama çizelgelerinden ve AC karakteristikleri tablosundan alınmıştır.
- B. B. Brey, *The Intel Microprocessors* (8. baskı), Bölüm 9: 8086/8088'in pinleri, saat üreteci, yolun tamponlanması, yol zamanlaması, READY ile bekleme durumları ve minimum/maksimum mod, şemalarla anlatılır.

- D. V. Hall, *Microprocessors and Interfacing* (2. baskı): 8086 sisteminin adres çözümü, bellek ve G/Ç arayüzü ve 8288 ile maksimum mod tasarımı örnek kartlar üzerinden işlenir.
- K. Shirriff'in righto.com'daki 8086 yazıları: kalıp fotoğrafları üzerinden yol arayüzü, kuyruk ve mikrokodun transistör düzeyindeki yapısı incelenir.

10.1.14 Kaynaklar

1. Intel, *8086 16-Bit HMOS Microprocessor*, veri sayfası (231455).
2. Intel, *8284A Clock Generator and Driver*, *8288 Bus Controller*, *8282/8283 Octal Latch* veri sayfaları.
3. Texas Instruments, *SN74LS373 Octal Transparent D-Type Latches* ve *SN74LS245 Octal Bus Transceivers* veri sayfaları.
4. Barry B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson, 2009 (Bölüm 9: 8086/8088 donanım özellikleri).
5. Douglas V. Hall, *Microprocessors and Interfacing: Programming and Hardware*, 2. baskı, McGraw-Hill.
6. Mehmet Bodur, *Mikroişlemcilere Giriş: Assembler ile Yazılım ve Arayüz*, EMO, 2014.
7. Ken Shirriff, 8086 kalıp fotoğrafları ve tersine mühendislik yazıları, righto.com.

KISIM IV · BÖLÜM 10.2

Aygıt Kontrolü

Bir mikroişlemcinin dış dünyayla bağı, çevre birimlerine yazdığı ve onlardan okuduğu baytlardır. Trafik lambası yakmak, bir motoru bir adım döndürmek ya da bir sıcaklık değerini okumak, işlemcinin gözünden bakıldığında aynı iştir: belirli bir porta bir sayı yazmak ya da o porttan bir sayı okumak. Bu not, 8086'nın bu işi yapan iki komutunu, `IN` ve `OUT`'u, G/Ç adres uzayını ve port numarasının komuta nasıl yazıldığını anlatır. Ardından aygıtın hazır olmasını beklemenin üç yolu (yoklama, kesme, doğrudan bellek erişimi) karşılaştırılır. Gerçek donanım tarafında 8255 programlanabilir çevre arayüzü, unipolar step motor sürücüsü ve 7 bölütlü gösterge işlenir. emu8086'nın yedi sanal aygıtını süren programlar ise satır satır ve yazmaç düzeyinde izlenir: trafik ışıkları, step motor, LED ekran, termometre, yazıcı, robot ve test aygıtı.

ÖĞRENME ÇIKTILARI

Port tabanlı G/Ç
`IN` ve `OUT` komutlarının dört biçimini, port numarasının sabit ya da DX ile verilmesini ve verinin yalnız AL ya da AX üzerinden geçmesi kuralını açıklamak; port eşlemeli ve bellek eşlemeli G/Ç'yi karşılaştırmak.

Bit düzeyinde denetim
Bir porttaki her bitin bir lambaya, bir mıknatısa ya da bir durum bilgisine karşılık geldiğini çözümlemek; istenen davranışı veren bayt ya da sözcüğü hesaplamak.

Yoklama
Aygıtın hazır ya da meşgul bitini bekleyen döngüler yazmak; yoklamanın bedelini kesme ve doğrudan bellek erişimiyle karşılaştırmak.

Zamanlama
INT 15h / AH = 86h için CX:DX bekleme süresini hesaplamak; LOOP ile yazılımla gecikmenin saat turu sayısını ve süresini bulmak.

Gerçek arayüz yongaları
8255 PPI'nin kontrol sözcüğünü ve bit kur/sil sözcüğünü oluşturmak; unipolar step motoru ULN2003 üzerinden, 7 bölütlü göstergeyi çoklamalı taramayla sürmek.

emu8086 aygıtları
Trafik ışıkları, step motor, LED ekran, termometre, yazıcı, robot ve test aygıtını süren programları okumak, adım adım izlemek ve değiştirmek.

Gömülü sistem bakışı
Programın ROM'dan RAM'e yüklenip sonsuz bir döngüde çalışma ilkesini açıklamak.

10.2.1 G/Ç temelleri

10.2.1.1 Çevre birimi, port ve gömülü sistem

Bir mikroişlemci tek başına yalnız yazmaçlarındaki ve belleğindeki sayılarla uğraşır. Bir lambanın yanması, bir motorun dönmesi ya da bir tuşa basılması, işlemcinin dışındaki bir devrenin işidir. İşlemciye bağlanan ve onunla veri alışverişi yapan bu devrelere **çevre birimi** (*peripheral*) denir. Klavye, ekran kartı, yazıcı, zamanlayıcı yongası, analog-sayısal çevirici ve bir motor sürücüsü birer çevre birimidir.

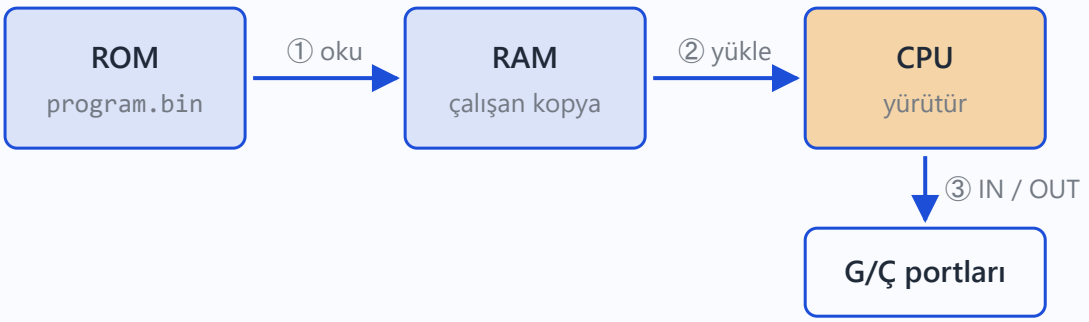
Çevre biriminin işlemciye görünen yüzü bir ya da birkaç yazmaçtır. Bir yazmaç, aygıta gidecek komutu ya da veriyi tutar. Bir başkası aygıttan gelen veriyi, bir üçüncüsü aygıtın durumunu gösterir: hazır mı, meşgul mü, hata var mı. Bu yazmaçların her biri bir adresle seçilir. İşlemcinin bu adresler üzerinden okuyup yazdığı kapıya **G/Ç portu** (*I/O port*) denir. G/Ç, giriş/çıkış sözünün kısaltmasıdır.

Tanım 10.2.1

G/Ç portu, bir çevre birimindeki yazmacın işlemciye görünen adresidir. 8086'da porta `IN` komutuyla okunur, `OUT` komutuyla yazılır. Bir porta yazılan değerin anlamını yalnız o porta bağlı donanım belirler: aynı `OUT` komutu bir aygıtta lambayı yakar, başka bir aygıtta motoru döndürür.

Port numaraları işlemciye değil, sisteme aittir. 8086 yalnız bir adres üretir; o adreste hangi aygıtın duracağını anakarttaki **adres çözücü** belirler. Adres çözücü, adres yolundaki bitlere bakıp yalnız kendi adres aralığında bir aygıtı etkinleştiren bir kapı devresidir. Bu yüzden temel port numaraları sistemden sisteme değişebilir. Aynı aygıt başka bir adres çözücüyle başka bir porta da taşınabilir. IBM PC'de klavye denetleyicisi 60h'de, kesme denetleyicisi 20h–21h'de durur; emu8086'nın sanal trafik lambaları ise port 4'tedir. Bu numaralar bir donanım tasarımının sonucudur, 8086'nın kuralı değildir.

Çevre birimlerini süren programların en tipik yeri **gömülü sistemlerdir**. Çamaşır makinesi, yazıcı, modem ve trafik lambası denetleyicisi bir mikroişlemci ya da mikrodenetleyici çevresinde kurulur. Bu cihazlarda işletim sistemi ya da disk çoğu zaman yoktur. Program bir geliştirme bilgisayarında derlenir ve düz bir ikili dosyaya (.bin) dönüştürülür. Bu dosya bir **ROM** (salt okunur bellek) yongasına yazılır. Cihaz açılınca işlemci programı ROM'dan okur, gerekirse RAM'e kopyalar ve yürütmeye başlar. Şekil 10.2.1 bu üç adımı gösterir.



Şekil 10.2.1: Gömülü sistemde programın yolu. (1) Açılışta işlemci ROM'daki `program.bin`'i okur, (2) çalışan kopyayı RAM'e yükler, (3) program aygıtlarla G/Ç portları üzerinden, `IN` ve `OUT` komutlarıyla konuşur.

Gömülü bir programın kendine özgü bir biçimi vardır. Masaüstü programı işini bitirince işletim sistemine döner. Gömülü programın döneceği bir yer yoktur. Program bu yüzden bir hazırlık kısmıyla başlar, ardından hiç bitmeyen bir döngüye girer. Döngü her turda aygıtların durumunu okur, karar verir ve aygıtlara yeni değerler yazar. Trafik ışıkları, step motor ve termostat programları tam olarak bu biçimdedir.

emu8086 bu dünyayı yazılımla taklit eder. 8086 öykünücüsüne bağlı yedi sanal aygıt, belirli port numaralarında bekler (Tablo 10.2.1). Aygıtlar *virtual devices* menüsünden açılır. Programın başına `#start=Traffic_Lights.exe#` gibi bir yönerge yazılırsa öykünücü aygıt penceresini program başlarken kendisi açar. Bu satır 8086 komutu değildir; yalnız öykünücüye verilmiş bir talimattır ve derlenen koda bayt eklemez.

Aygıt	Port(lar)	Yön	Veri
Trafik lambaları (*traffic lights*)	4	OUT	sözcük: 12 lamba biti
Step motor (*stepper motor*)	7	OUT / IN	bit 0–2 mıknatıslar, bit 7 hazır
Robot	9, 10, 11	OUT / IN	komut, veri, durum
Test aygıtı (*simple io test*)	110, 112	IN / OUT	bayt (110), sözcük (112)
Termometre ve ısıtıcı	125, 127	IN / OUT	sıcaklık, ısıtıcı aç/kapa
Yazıcı (*printer*)	130	OUT	karakter (INT 21h/05h ile)
LED ekran (*LED display*)	199	OUT	sözcük: işaretli sayı

Tablo 10.2.1: emu8086'ya bağlı sanal aygıtlar ve portları.

Tablodaki bütün port numaraları 255'ten küçüktür. Bu ayrıntı küçük görünür ama komutun yazılışını belirler: 0–255 aralığındaki bir port numarası doğrudan komutun içine yazılabilir.

10.2.1.2 IN ve OUT komutları

8086'da porta erişen iki komut vardır. `IN` bir porttan okur ve okunan değeri biriktiriciye (AL ya da AX) yazar. `OUT` biriktiricideki değeri bir porta yazar. Port numarası iki yolla verilir: 0–255 arasındaysa komutun içinde 8 bitlik bir sabit olarak, aksi hâlde DX yazmacında. Tablo 10.2.2 bütün biçimleri listeler.

Biçim	Anlamı	Port aralığı	Makine kodu	Saat turu
`IN AL, port`	porttan 1 bayt oku	0–255 (sabit)	`E4 pp`	10
`IN AX, port`	porttan 1 sözcük oku	0–255 (sabit)	`E5 pp`	10
`IN AL, DX` / `IN AX, DX`	port numarası DX'te	0–65.535	`EC` / `ED`	8
`OUT port, AL` / `OUT port, AX`	porta yaz	0–255 (sabit)	`E6 pp` / `E7 pp`	10
`OUT DX, AL` / `OUT DX, AX`	port numarası DX'te	0–65.535	`EE` / `EF`	8

Tablo 10.2.2: IN ve OUT komutlarının biçimleri, makine kodları ve 8086 süreleri.

Makine kodları biçimin mantığını açık eder. Sabit portlu biçim iki bayttır: ilk bayt işlem kodu, ikinci bayt port numarasıdır (pp). Örneğin `IN AL, 7` komutu `E4 07`, `OUT 4, AX` komutu `E7 04`, `IN AX, 112` komutu `E5 70` olarak derlenir (112 = 70h). Port numarası için yalnız bir bayt ayrıldığından 255'ten büyük bir sayı bu biçime sığmaz. DX'li biçim ise tek bayttır; port numarası komutta değil yazmaçtadır. `OUT DX, AL` komutu yalnız `EE`'dir. İşlem kodunun en düşük biti verinin genişliğini seçer: 0 ise bayt (AL), 1 ise sözcük (AX). E4/E5, E6/E7, EC/ED ve EE/EF çiftleri bu yüzden birbirinden 1 farklıdır.



Şekil 10.2.2: IN ve OUT'ta veri yolu. Veri yalnız AL ya da AX üzerinden geçer; port numarası komuttaki sabit (n) ya da DX'tir.

İki komutun da en katı kuralı biriktiricidir. Veri yalnız AL (bayt) ya da AX (sözcük) üzerinden geçer. `OUT 4, BX` ya da `IN CL, 7` yazılamaz; böyle bir işlem kodu 8086'da yoktur. BX'teki bir değeri porta göndermek için önce `MOV AX, BX` yazılır. İşlenenlerin sırası da sabittir: `OUT`'ta önce port, sonra AL/AX gelir; `IN`'de tersi. Sıra `MOV`'daki gibi "hedef, kaynak" düzenine uyar. `OUT`'un hedefi port, `IN`'in hedefi biriktiricidir.

Bu komutlar bayraklara dokunmaz. `IN AL, 125` okunan sıcaklık eksi bile olsa SF'yi değiştirmez; değeri sınamak için ardından `CMP` ya da `TEST` gerekir. Okunan değer yalnız biriktiriciye gider, başka hiçbir yazmaç değişmez.

Sözcük biçimi iki ardışık portu birlikte kullanır. `OUT 4, AX` komutu AL'yi port 4'e, AH'yi port 5'e yazar. `IN AX, 112` ise AL'yi port 112'den, AH'yi port 113'ten alır. Bellekteki sözcüklerde olduğu gibi düşük bayt düşük adrestedir (*little-endian*). 8086'nın 16 bitlik veri yolu, çift adresli bir sözcüğü tek yol döngüsünde aktarır. Tek adresli bir sözcük iki ayrı bayt aktarımı gerektirir ve 4 saat turu daha uzun sürer. Aygıt yazmaçlarının çift adreslere yerleştirilmesinin bir nedeni budur.

Saat turları ilginç bir ayrıntı taşır. Intel'in kılavuzuna göre DX'li biçim 8 turla, sabit portlu biçimden (10 tur) daha hızlıdır; üstelik bir bayt daha kısadır. Sık erişilen bir port için DX bir kez yüklenip döngü boyunca kullanılırsa her erişimde iki tur kazanılır.

> **Sık yapılan hata**
>
> `OUT 7, BL`, `IN BX, DX` ya da `OUT AL, 7` yazmak. İlk ikisi biriktirici kuralını, sonuncusu işlenen sırasını çiğner. Assembler üçünü de reddeder. Doğrusu `MOV AL, BL` · `OUT 7, AL`, `IN AX, DX` · `MOV BX, AX` ve `OUT 7, AL`'dir.

10.2.1.3 Port numarası: sabit ya da DX ile

8086'nın G/Ç adres uzayı 16 bittir: 0000h'den FFFFh'ye kadar 65.536 bayt portu. Bunların ilk 256'sına komutun içindeki sabitle ulaşılır. Geri kalanlar için port numarası önce DX'e yüklenir. Örneğin 300 numaralı porta bir bayt yazmak şöyle yapılır:

```
mov dx, 300     ; 300 = 012Ch > 255
mov al, 5Ah
out dx, al      ; port 012Ch ← 5Ah
```

İlk komut DX'e 012Ch yükler (`BA 2C 01`). İkinci komut gönderilecek veriyi AL'ye koyar. Üçüncü komut tek baytlık `EE`'dir; işlemci G/Ç yol döngüsünde adres yoluna DX'i, veri yoluna AL'yi çıkarır. `OUT 300, AL` yazılsaydı assembler bunu reddederdi, çünkü 300 bir bayta sığmaz.

DX'li biçim yalnız büyük port numaraları için değildir. Port numarası program çalışırken hesaplanacaksa da DX gerekir. Örneğin aynı kartın dört yazmacı 310h, 311h, 312h ve 313h ise taban adres DX'e yüklenir ve `INC DX` ile sıradaki yazmaca geçilir. Sabit portlu biçimde port numarası derleme anında kesinleşir, değiştirilemez.

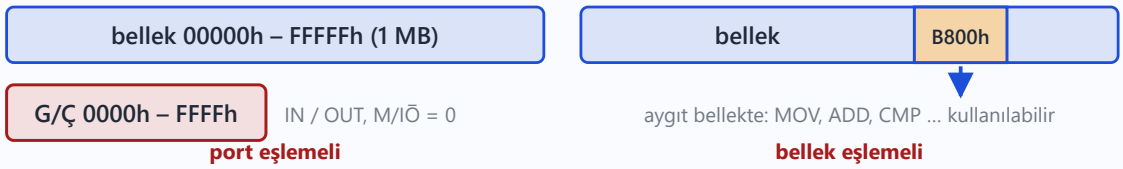
emu8086'nın sanal aygıtlarının hepsi 255'in altında olduğundan bu aygıtları süren programlar sabit portlu biçimi kullanır: `OUT 4, AX`, `IN AL, 7`, `OUT 199, AX`. 8255 gibi gerçek kartlar ise PC'de çoğu zaman 300h–31Fh aralığına yerleşir ve DX'le sürülür.

10.2.1.4 Port eşlemeli ve bellek eşlemeli G/Ç

Bir aygıt yazmacını işlemciye bağlamanın iki yolu vardır. Birincisinde aygıtlar bellekten ayrı bir adres uzayında durur ve yalnız özel G/Ç komutlarıyla erişilir. Buna **port eşlemeli G/Ç** (*port-mapped* ya da *isolated I/O*) denir. İkincisinde aygıt yazmaçları bellek adreslerinin bir bölümüne yerleştirilir ve sıradan bellek komutlarıyla okunup yazılır. Buna **bellek eşlemeli G/Ç** (*memory-mapped I/O*) denir.

Tanım 10.2.2

Port eşlemeli G/Ç'de aygıtlar 64 K'lık ayrı bir G/Ç adres uzayındadır; erişim yalnız `IN` ve `OUT` ile olur ve bellek adresi harcanmaz. **Bellek eşlemeli G/Ç**'de aygıt yazmaçları 1 MB'lık bellek uzayının içindedir; `MOV`, `ADD`, `CMP`, `AND` gibi bellek işlenenli her komut aygıta erişebilir.



Şekil 10.2.3: 8086'nın iki adres uzayı. Solda 1 MB'lık bellek (00000h–FFFFFh); B800h kesimindeki metin ekranı belleği bellek eşlemeli bir aygıttır. Sağda 64 K'lık G/Ç uzayı (0000h–FFFFh); ona yalnız IN ve OUT ulaşır.

İşlemci iki uzayı aynı adres ve veri yollarıyla kullanır. Hangisine erişildiğini bir denetim sinyali bildirir: 8086'da M/IO̅ ucu bellek döngüsünde 1, G/Ç döngüsünde 0'dır. `IN` ve `OUT` bu ucu 0 yapan tek komutlardır. Bu sinyalin, okuma ve yazma uçlarının ve yol döngüsünün saat turu düzeyindeki ayrıntısı 8086 pinlerinin konusudur. Programcı açısından sonuç basittir: port 4 ile bellek adresi 0000:0004 birbirinden tamamen bağımsız iki yerdir.

Bellek eşlemeli G/Ç'nin en tanıdık örneği PC'nin metin ekranıdır. 80×25 metin modunda ekranın her hücresi B800:0000'dan başlayan bellekte iki bayt tutar: karakter kodu ve öznitelik (renk) baytı. Ekran kartı oraya yazılan baytı hemen görüntüye çevirir. Aşağıdaki program ekranın sol üst köşesine mavi zemin üzerine sarı bir “A”, yanına bir “B” yazar:

```
org 100h
mov ax, 0B800h
mov es, ax              ; ES = metin ekranı kesimi
mov di, 0               ; satır 0, sütun 0
mov al, 'A'
mov ah, 1Eh             ; mavi zemin, sarı yazı
mov es:[di], ax         ; karakter + öznitelik
mov byte ptr es:[di+2], 'B'
ret
```

Programda tek bir G/Ç komutu yoktur, ama ekran değişir. Kesim yazmacına sabit doğrudan yüklenemediği için B800h önce AX'e, oradan ES'ye aktarılır. `mov es:[di], ax` tek komutla iki bayt yazar: AL'deki 41h B800:0000'a, AH'deki 1Eh B800:0001'e gider. Fiziksel adres B800h × 16 + 0 = B8000h'dir. Son komut yalnız karakter baytını yazar; ikinci hücrenin özniteliği eski değerinde (07h, siyah zemin üzerine açık gri) kalır. Tablo 10.2.3 programın izlemesidir.

Adım	Komut	AX	ES	DI	Belleğe yazılan	Açıklama
1	`mov ax, 0B800h`	B800h	0700h	0000h		kesim değeri önce genel yazmaca
2	`mov es, ax`	B800h	B800h	0000h		ES = B800h: metin ekranı belleği
3	`mov di, 0`	B800h	B800h	0000h		ilk hücre: satır 0, sütun 0
4	`mov al, 'A'`	B841h	B800h	0000h		AL = 41h ('A')
5	`mov ah, 1Eh`	1E41h	B800h	0000h		AH = 1Eh: mavi zemin, sarı yazı
6	`mov es:[di], ax`	1E41h	B800h	0000h	B800:0000 ← 41h, B800:0001 ← 1Eh	tek MOV: karakter ve öznitelik; ekranda renkli A
7	`mov byte ptr es:[di+2], 'B'`	1E41h	B800h	0000h	B800:0002 ← 42h	ikinci hücrenin karakter baytı; öznitelik (07h) korunur
8	`ret`	1E41h	B800h	0000h		program biter

Tablo 10.2.3: Bellek eşlemeli G/Ç: metin ekranına MOV ile yazmak.

İki yöntemin artısı ve eksisi birbirinin aynası gibidir (Tablo 10.2.4). Bellek eşlemeli aygıtta bütün bellek komutları ve bütün adresleme modları kullanılabilir. Örneğin `INC BYTE PTR ES:[DI]` aygıt yazmacını tek komutla bir artırır, `TEST` doğrudan aygıtın durum bitini sınar. Buna karşılık aygıt bellek adreslerini harcar ve o adreslere bellek yerleştirilemez. Port eşlemeli aygıt bellek uzayına dokunmaz ama yalnız iki komutla, yalnız AL/AX üzerinden erişilir. Okunan değer üzerinde işlem yapmak için önce biriktiriciye alınması gerekir.

Özellik	Port eşlemeli	Bellek eşlemeli
Adres uzayı	ayrı, 64 K (0000h–FFFFh)	bellekle ortak, 1 MB'ın bir bölümü
Komutlar	yalnız IN, OUT	MOV, ADD, CMP, TEST, AND …
Veri yazmacı	yalnız AL / AX	herhangi bir yazmaç ya da sabit
Adresleme	sabit (0–255) ya da DX	bütün bellek adresleme modları
Bellek kaybı	yok	aygıtın kapladığı adresler
8086 sinyali	M/$\overline{IO}$ = 0	M/$\overline{IO}$ = 1
Örnek	emu8086 aygıtları, 8255, 8259A	B800h metin ekranı, A000h grafik belleği

Tablo 10.2.4: Port eşlemeli ve bellek eşlemeli G/Ç'nin karşılaştırması.

8086 ailesi iki yöntemi de destekler ve PC ikisini birlikte kullanır. Denetim yazmaçları az sayıda ve küçük olan aygıtlar (klavye denetleyicisi, zamanlayıcı, kesme denetleyicisi) porttadır. Büyük veri alanı isteyen aygıtlar (ekran belleği) bellektedir. Ayrı G/Ç komutu olmayan işlemcilerde (ör. ARM) bütün aygıtlar bellek eşlemelidir.

Sık yapılan hata

Port numarasını bellek adresiyle karıştırmak. `MOV AL, [4]` komutu port 4'ü değil, DS:0004 bellek baytını okur; trafik lambalarına hiçbir şey olmaz. Port eşlemeli bir aygıta yalnız `IN` ve `OUT` ulaşır.

10.2.1.5 Yoklama, kesme ve DMA

Aygıtlar işlemciden çok daha yavaştır. Bir step motor yeni konumuna milisaniyeler içinde gelir; aynı sürede 8086 binlerce komut yürütür. Yazıcı bir karakteri basarken, robot bir adım atarken işlemcinin yeni komutu göndermek için beklemesi gerekir. Aygıtın hazır olduğunu öğrenmenin üç yolu vardır.

En basit yol **yoklamadır** (*polling*). İşlemci aygıtın durum yazmacını bir döngü içinde tekrar tekrar okur. “Hazır” biti 1 olunca döngüden çıkar ve veriyi gönderir ya da alır. emu8086'nın step motoru buna iyi bir örnektir: port 7'den okunan baytın bit 7'si motor hazır olunca 1'dir.

```
bekle: in   al, 7          ; durum oku
       test al, 10000000b  ; bit 7: hazır mı?
       jz   bekle          ; 0 ise yeniden oku
```

`TEST`, AL ile maskenin mantıksal VE'sini hesaplar ama sonucu bir yere yazmaz; yalnız bayrakları kurar. Bit 7 sıfırsa sonuç sıfırdır ve ZF = 1 olur; `JZ` döngünün başına döner. Bit 7 birse ZF = 0'dır ve döngü biter. Üç komut, aygıt hazır olana kadar dönen bir bekleme döngüsü kurar.

Tanım 10.2.3

Yoklama, işlemcinin bir aygıtın durum bitini döngü içinde okuyarak aygıtın hazır olmasını beklemesidir. Aygıt ile işlemcinin hazır ve meşgul bitleri üzerinden veri aktarımını eşlemesine **el sıkışma** (*handshaking*) denir.

Yoklamanın artısı basitliğidir: ek donanım ya da kesme yordamı gerekmez, programın akışı tek bakışta izlenir. Eksisi, işlemcinin bekleme süresince başka hiçbir iş yapamamasıdır. Motor bir adımı 10 ms'de tamamlıyorsa 4,77 MHz'lik 8086 bu sürede yaklaşık 47.700 saat turunu boş döngüde harcar. Tek bir aygıt süren bir gömülü sistemde bu kabul edilebilir. Aynı anda klavye, ekran ve iletişim hattı izlenecekse kabul edilemez.

İkinci yol **kesmedir**. Aygıt hazır olunca işlemciye bir istek sinyali gönderir. İşlemci o anki komutu bitirir, bayrakları ve dönüş adresini yığına iter, aygıtın hizmet yordamını çalıştırır ve `IRET` ile kaldığı yere döner. PC'de aygıtların istekleri 8259A kesme denetleyicisinde toplanır ve önceliğe göre işlemciye iletilir. Kesmede işlemci beklemez, ama veriyi yine kendisi taşır: hizmet yordamındaki `IN` ve `OUT` komutlarını işlemci yürütür.

Üçüncü yol **doğrudan bellek erişimidir** (DMA, *direct memory access*). Diskten belleğe binlerce baytlık bir blok aktarılacaksa her bayt için bir kesme bile çok pahalıdır. Bu durumda 8237 gibi bir DMA denetleyicisi işlemciden yolu ister, yolu alınca veriyi aygıt ile bellek arasında kendisi taşır ve işi bitince yolu geri verir. İşlemci aktarım sırasında yalnız yolu kullanamaz; iç yazmaçlarındaki işine devam edebilir.

Yöntem	Kim bekler?	Veriyi kim taşır?	Uygun olduğu iş
Yoklama	işlemci, döngüde	işlemci (IN/OUT)	tek aygıt, basit gömülü sistem
Kesme	kimse; aygıt haber verir	işlemci (hizmet yordamında)	klavye, seri hat, zamanlayıcı
DMA	kimse; denetleyici yolu alır	DMA denetleyicisi (8237)	disk, ses kartı gibi blok aktarımı

Tablo 10.2.5: Aygıtla eşlemenin üç yolu.

emu8086'nın sanal aygıtlarının hiçbiri kesme üretmez. Bu notun bütün programları bu yüzden yoklama kullanır. Yoklama döngüsünün ortak kalıbı her aygıtta yinelenir: durum oku, ilgili biti sına, hazırsa veriyi yaz ya da oku, değilse yeniden oku.

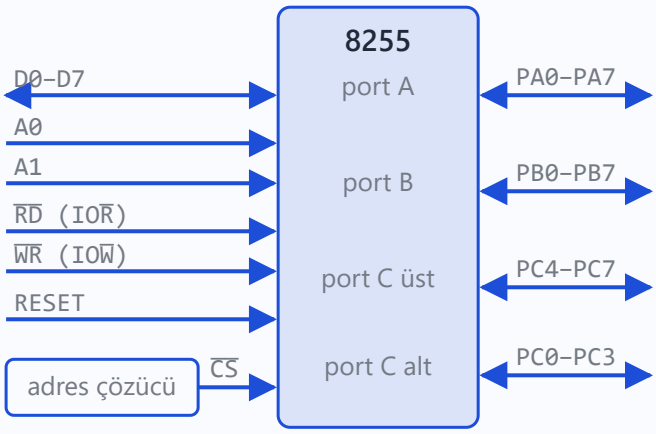
> **Sık yapılan hata**
>
> Yoklama döngüsünde `TEST` yerine `AND` kullanıp sonra AL'deki veriyi kullanmaya çalışmak. `AND AL, 80h` durumu sınar ama AL'nin öteki bitlerini siler. Durum baytında aygıttan gelen başka bilgi de varsa (hata biti, veri biti) kaybolur. `TEST` aynı sınamayı AL'yi değiştirmeden yapar.

10.2.2 8255 programlanabilir çevre arayüzü

10.2.2.1 Yapı ve adres seçimi

emu8086'nın aygıtları porta yazılan değeri kendisi yorumlar. Gerçek bir kartta ise işlemci ile lamba, anahtar ya da motor sürücüsü arasında bir arayüz yongası durur. 8086 sistemlerinde en yaygın paralel arayüz yongası Intel 8255'tir. Adı **programlanabilir çevre arayüzü**dür (*programmable peripheral interface*, PPI). “Programlanabilir” sözü, portlarının giriş mi çıkış mı olacağının yazılımla seçilmesinden gelir.

8255'in üç adet 8 bitlik portu vardır: A, B ve C. C portu iki 4 bitlik yarıya ayrılabilir: üst yarı PC7–PC4, alt yarı PC3–PC0. Bunların yanında bir de kontrol yazmacı bulunur. Dört yazmaçtan hangisine erişileceğini yonganın A1 ve A0 uçları seçer. Yonganın etkin olup olmayacağına $\overline{CS}$ (yonga seçme) ucu karar verir; bu uç adres çözücüden gelir. Okuma ve yazma, işlemcinin G/Ç okuma ve G/Ç yazma sinyalleriyle ($IO\overline{R}$, $IO\overline{W}$) yapılır. Şekil 10.2.4 bağlantıyı gösterir.



Şekil 10.2.4: 8255 PPI'nin işlemci tarafı (D0–D7, A0, A1, $\overline{RD}$, $\overline{WR}$, RESET, $\overline{CS}$) ve aygıt tarafı (PA0–PA7, PB0–PB7, PC7–PC4, PC3–PC0).

Kartın taban adresi 310h ise dört yazmacın adresleri Tablo 10.2.6'daki gibidir. Adres çözücü 310h–313h aralığını tanıyıp $\overline{CS}$'yi 0 yapar; adresin en düşük iki biti yonganın A1 ve A0 uçlarına gider.

A1 A0	Yazmaç	Adres
0 0	port A	310h
0 1	port B	311h
1 0	port C	312h
1 1	kontrol yazmacı	313h

Tablo 10.2.6: 8255'in yazmaçları (taban adres 310h, A1 ← A1, A0 ← A0 bağlantısı).

Bu ardışık adres düzeni, 8 bitlik veri yolu olan 8088 için doğrudur. 8086'nın veri yolu 16 bittir ve iki bankaya ayrılır: çift adresli baytlar D0–D7'den (alt banka), tek adresli baytlar D8–D15'ten (üst banka) aktarılır. 8255'in yalnız sekiz veri ucu vardır. Yonga D0–D7'ye bağlanırsa yalnız çift adreslerdeki yazmaçlarına ulaşılabilir. Çözüm, yonganın A0 ucunu işlemcinin A1'ine, A1 ucunu A2'sine bağlamaktır. Böylece dört yazmaç 310h, 312h, 314h ve 316h'ye, yani hep çift adreslere düşer. Bu notun programları ardışık 310h–313h düzenini kullanır; 8086'ya göre kurulmuş bir kartta adresler buna göre değiştirilir.

Adreslerin hepsi 255'ten büyüktür. 8255'e her erişim bu yüzden DX üzerinden yapılır: `MOV DX, 313h` · `OUT DX, AL`. RESET'ten sonra 8255'in bütün portları giriştir. Bunun bir güvenlik nedeni vardır: açılışta henüz programlanmamış bir yonganın çıkışları bağlı devreyi rastgele sürmemelidir. Program bu yüzden yongayı kullanmadan önce kontrol yazmacına bir kontrol sözcüğü yazar.

10.2.2.2 Kontrol sözcüğü: mod seçimi

Kontrol yazmacına yazılan baytın en yüksek biti (D7) baytın türünü belirler. D7 = 1 ise bayt bir **mod seçme sözcüğüdür**: portların modunu ve yönünü bir kerede ayarlar. D7 = 0 ise bayt bir **bit kur/sil** sözcüğüdür ve yalnız C portunun tek bir bitini değiştirir.

D7	D6 D5	D4	D3	D2	D1	D0
1 = mod seçimi	A grubu modu (00 = mod 0)	port A	PC7–PC4	B grubu modu (0 = mod 0)	port B	PC3–PC0

Tablo 10.2.7: 8255 mod seçme sözcüğü (D7 = 1). Yön bitlerinde 1 = giriş, 0 = çıkış.

8255'in portları iki gruba ayrılır. A grubu port A ile C'nin üst yarısıdır; B grubu port B ile C'nin alt yarısıdır. Mod 0, el sıkışmasız basit G/Ç'dir: porta yazılan değer çıkış uçlarında kalır, girişten okunan değer o anki uç durumudur. Mod 1 ve mod 2 el sıkışmalı aktarım içindir. Bu modlarda C portunun bazı uçları "veri hazır" (*strobe*) ve "alındı" sinyalleri olarak kullanılır; mod 2 port A'yı iki yönlü yapar. Lamba, anahtar ve motor sürücüleri için mod 0 yeterlidir.

Örnek 10.2.1

İstenen: port A çıkış (LED ya da röle sürer), port B giriş (anahtarlar okunur), C'nin iki yarısı çıkış, hepsi mod 0.

1. D7 = 1: bu bir mod seçme sözcüğüdür.
2. D6 D5 = 00 ve D2 = 0: iki grup da mod 0.
3. D4 = 0 (A çıkış), D3 = 0 (C üst çıkış), D1 = 1 (B giriş), D0 = 0 (C alt çıkış).
4. Bitler yan yana yazılır: 1 00 0 0 0 1 0b = 1000 0010b = **82h**.

Aynı yolla A giriş, B çıkış, C'nin tamamı giriş için 1 00 1 1 0 0 1b = 99h; A çıkış, B çıkış, C'nin tamamı giriş için 1 00 0 1 0 0 1b = 89h bulunur. Bütün portlar çıkış ise sözcük 1000 0000b = 80h olur.

Sözcüğü kurarken en sık karışan nokta yön bitlerinin anlamıdır. 8255'te 1 "giriş", 0 "çıkış" demektir. Bu, "1 = açık" sezgisinin tersidir ve ezberlenmesi gerekir. Sunumun etkileşimli kontrol sözcüğü oluşturucusu, her portun yönüne tıklandıkça bitlerin nasıl değiştiğini gösterir; sözcük kutusuna doğrudan bir değer yazılıp portların yönü de okunabilir.

10.2.2.3 Bit kur/sil (BSR)

C portu çoğu zaman tek tek denetim uçları olarak kullanılır: bir uç bir rölenin bobinini, öteki bir LED'i, bir üçüncüsü bir hane seçme transistörünü sürer. Yalnız bir ucu değiştirmek için portun tamamını yazmak, öteki uçların değerini programda ayrıca saklamayı gerektirir. 8255 bunun yerine **bit kur/sil** (*bit set/reset*, BSR) sözcüğünü sunar.

> **Tanım 10.2.4**
>
> **Bit kur/sil sözcüğü**, D7 = 0 olan kontrol baytıdır. D3–D1 değiştirilecek C bitinin numarasını (0–7), D0 o bitin yeni değerini verir; D6–D4 kullanılmaz (0 yazılır). Sözcük kontrol yazmacına yazılır, C portuna değil. Öteki C bitleri ve port yönleri değişmez.

PC7'yi 1 yapmak için bit numarası 7 = 111b ve değer 1 seçilir: 0000 1111b = 0Fh. PC7'yi 0 yapmak için yalnız D0 değişir: 0000 1110b = 0Eh. Aynı yolla PC3'ü 1 yapan sözcük 0000 0111b = 07h, PC0'ı 0 yapan sözcük 00h'dir. Sonuncusu ilginçtir: 00h bir mod sözcüğü gibi görünse de D7 = 0 olduğundan “PC0 = 0” anlamına gelir.

> **Sık yapılan hata**
>
> BSR sözcüğünü C portuna (312h) yazmak. Port C'ye yazılan 0Fh, PC3–PC0'ı 1, PC7–PC4'ü 0 yapar: sekiz bitin hepsi yeniden yazılır. Bit kur/sil sözcüğü yalnız kontrol yazmacına (313h) yazıldığında tek bit sözcüğü olarak yorumlanır.

10.2.2.4 Kod izleme: 8255 kurulumu

Aşağıdaki program dört temel işi art arda yapar: yongayı 82h ile programlar, port B'deki anahtarları okur, okunan değeri port A'daki LED'lere yazar ve BSR ile PC7'yi 1 yapar.

```
org 100h
mov dx, 313h   ; kontrol yazmacı
mov al, 82h    ; A,C çık; B gir
out dx, al
mov dx, 311h   ; port B
in  al, dx     ; anahtarları oku
mov dx, 310h   ; port A
out dx, al     ; LED'lere yaz
mov dx, 313h
mov al, 0Fh    ; BSR: PC7 = 1
out dx, al
ret
```

Programın her G/Ç erişimi aynı iki adımdan oluşur: önce DX'e adres, sonra `IN` ya da `OUT`. Port B'ye bağlı anahtarların 1001 0110b = 96h konumunda olduğu varsayılmıştır. Tablo 10.2.8 izlemeyi, son sütunda her G/Ç işlemini verir.

Adım	Komut	DX	AL	G/Ç	Açıklama
1	`mov dx, 313h`	0313h	00h		DX = 313h: kontrol yazmacının adresi; 255'ten büyük olduğu için DX'te tutulur
2	`mov al, 82h`	0313h	82h		AL = 82h = 1000 0010b: mod 0; A çıkış, C üst çıkış, B giriş, C alt çıkış
3	`out dx, al`	0313h	82h	313h ← 82h	kontrol sözcüğü yongaya yazıldı; port yönleri artık 82h'ye göre
4	`mov dx, 311h`	0311h	82h		DX = 311h: port B
5	`in al, dx`	0311h	96h	311h → 96h	anahtarlar okundu: AL = 96h = 1001 0110b
6	`mov dx, 310h`	0310h	96h		DX = 310h: port A
7	`out dx, al`	0310h	96h	310h ← 96h	LED'ler anahtarları kopyalar: bit 7, 4, 2 ve 1 yanar
8	`mov dx, 313h`	0313h	96h		DX yeniden kontrol yazmacını gösterir
9	`mov al, 0Fh`	0313h	0Fh		AL = 0Fh = 0000 1111b: D7 = 0, yani bit kur/sil sözcüğü
10	`out dx, al`	0313h	0Fh	313h ← 0Fh	D3–D1 = 111 (PC7), D0 = 1: yalnız PC7 = 1 olur; yönler değişmez
11	`ret`	0313h	0Fh		yığından 0000h → IP; PSP'deki INT 20h'ye dönülür
12	`INT 20h (PSP:0000)`	0313h	0Fh		program biter

Tablo 10.2.8: 8255 kurulum programının izlemesi (port B'deki anahtarlar 96h konumunda; her satır komut yürütüldükten sonraki durum).

İzlemede üç ayrıntı dikkat çeker. Birincisi, `OUT DX, AL` komutları hiçbir yazmacı değiştirmez; etkileri yalnız yongadadır. İkincisi, `IN AL, DX` AL'nin eski değerini (82h) tümüyle siler. Üçüncüsü, son `OUT` kontrol yazmacına yazdığı hâlde port yönlerini değiştirmez: D7 = 0 olduğu için yonga onu bir bit kur/sil sözcüğü olarak yorumlar. Kurulumdan sonra port A'nın LED'lerinde bit 7, 4, 2 ve 1 yanar; C portunda yalnız PC7 = 1'dir.

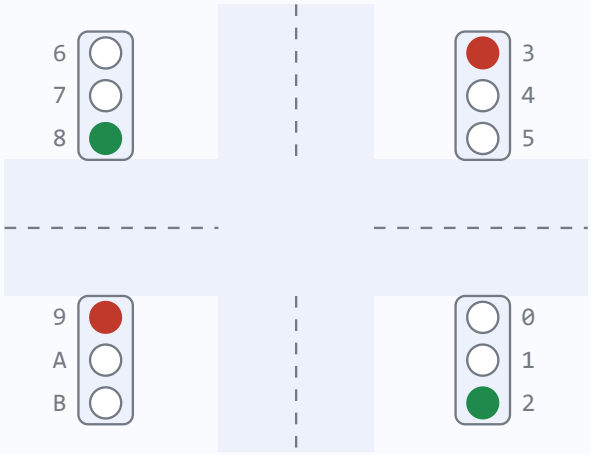
Programın makine kodu 21 bayttır. Dört `MOV DX` üçer, iki `MOV AL` ikişer bayt tutar; dört G/Ç komutu ve `RET` birer bayttır. Bu kısalık, DX'li biçimin bir başka yararıdır: port numarası tek bir `MOV`'da verilir ve G/Ç komutunun kendisi tek bayta iner.

10.2.3 Trafik ışıkları

10.2.3.1 Port 4 ve lamba bitleri

emu8086'nın trafik ışıkları aygıtı dört yönlü bir kavşaktır. Her köşede bir direk, her direkte üç lamba vardır: kırmızı, sarı ve yeşil. On iki lambanın her biri port 4'e yazılan 16 bitlik sözcüğün bir bitine bağlıdır. Bit 1 ise lamba yanar, 0 ise söner. Bit 12–15 kullanılmaz. Port 4'e bir sözcük yazmak, bu yüzden kavşağın tam bir fotoğrafını bir kerede çizmek demektir.

Bitlerin lambalara dağılımı düzenlidir. Direkler k = 0, 1, 2, 3 diye numaralanırsa k. direğin kırmızı lambası bit 3k, sarısı bit 3k + 1, yeşili bit 3k + 2'dir. Sağ alt direk bit 0–2'yi, sağ üst 3–5'i, sol üst 6–8'i, sol alt 9–11'i (onaltılık 9, A, B) kullanır.



Şekil 10.2.5: Port 4 = 030Ch = 0000 0011 0000 1100b. Yanan bitler 2 ve 8 (yeşil), 3 ve 9 (kırmızı): sağ alt ile sol üst yeşil, sağ üst ile sol alt kırmızı.

Şekil 10.2.5'teki değeri çözmek için sözcük dörtlü gruplara ayrılır: 0000 0011 0000 1100b. 1 olan bitler sağdan sayılarak bulunur: bit 2, 3, 8 ve 9. Bit 2 = 3 · 0 + 2, yani sağ alt direğin yeşilidir. Bit 3 = 3 · 1 + 0 sağ üst direğin kırmızısıdır. Bit 8 = 3 · 2 + 2 sol üst direğin yeşili, bit 9 = 3 · 3 + 0 sol alt direğin kırmızısıdır. Tersi de aynı kolaylıkla yapılır: istenen lambaların bit numaraları bulunur ve 2'nin bu kuvvetleri toplanır.

Örnek 10.2.2

Bütün yönlerde yalnız yeşil lambaları yakan sözcük nedir?

1. Yeşil lambalar bit 3k + 2'dedir: k = 0, 1, 2, 3 için bit 2, 5, 8, 11.
2. Sözcük $2^2 + 2^5 + 2^8 + 2^{11} = 4 + 32 + 256 + 2048 = 2340$'tır.

3. İkili yazımı 0000 1001 0010 0100b, onaltılık yazımı **0924h**'dir.

Kavşakta böyle bir durum elbette tehlikelidir; örnek yalnız bit hesabını gösterir. Bütün kırmızıları yakan sözcük aynı yolla bit 0, 3, 6, 9'dan 0000 0010 0100 1001b = 0249h çıkar. Programda bu değerin adı `all_red`'dir.

10.2.3.2 Durum tablosu

Kavşağın işleyişi birkaç durumun belirli bir sırayla tekrarlanmasıdır. Her durum port 4'e yazılacak bir sözcükle tanımlanır. Program bu sözcükleri bellekte ardışık bir tablo olarak tutar:

```
;               FEDC_BA98_7654_3210
situation  dw    0000_0011_0000_1100b
s1         dw    0000_0110_1001_1010b
s2         dw    0000_1000_0110_0001b
s3         dw    0000_1000_0110_0001b
s4         dw    0000_0100_1101_0011b
sit_end = $
all_red    equ   0000_0010_0100_1001b
```

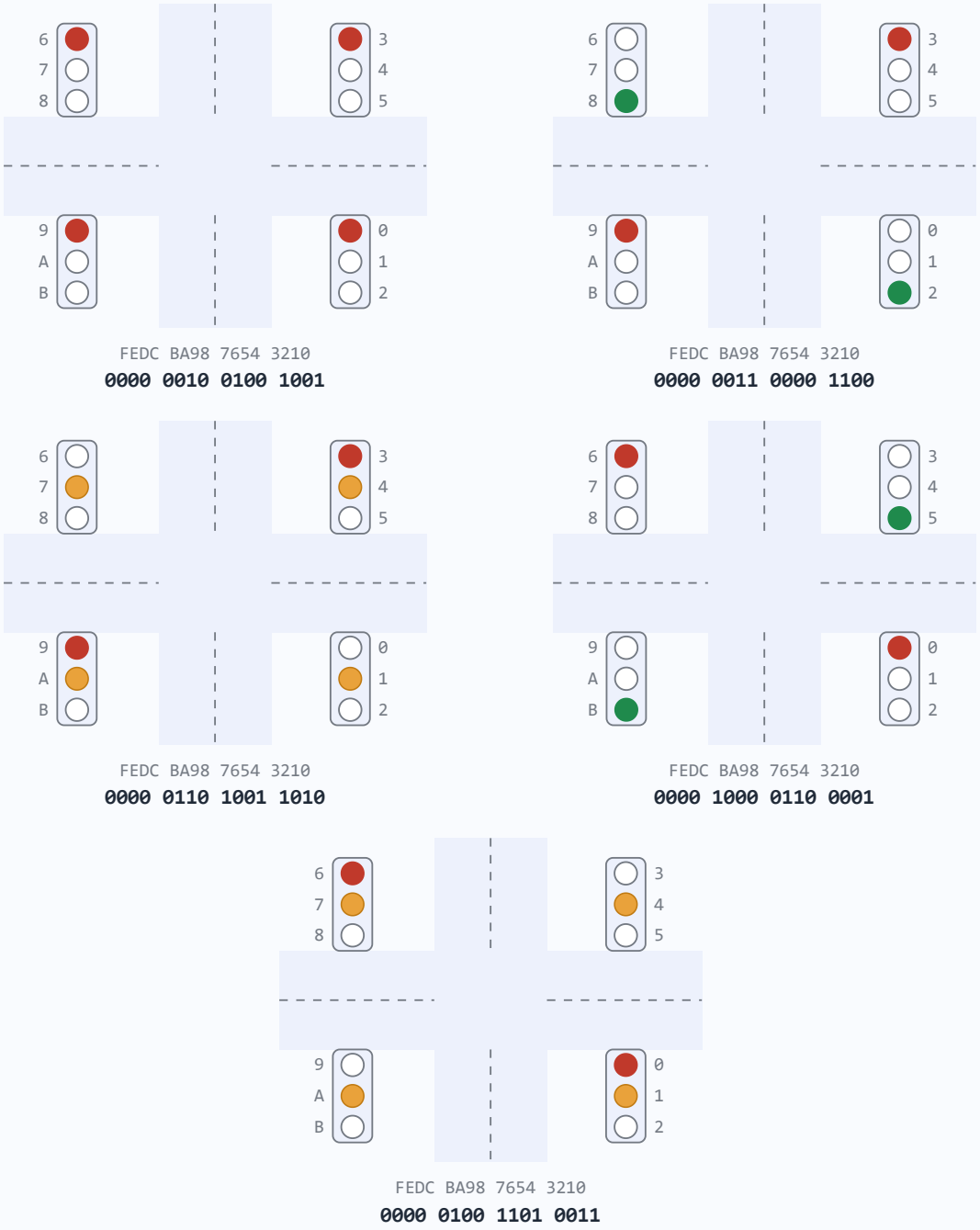
Sayıların içindeki alt çizgiler yalnız okunabilirlik içindir; emu8086 onları yok sayar. Açıklama satırındaki FEDC_BA98_7654_3210 dizisi her sütunun hangi bit olduğunu gösterir. `sit_end = $` satırı bellekte yer kaplamaz: `$`, assembler'ın o anki adres sayacıdır ve tablonun bittiği adresi verir. Program bu adresi "tablonun sonuna geldim mi?" sorusu için kullanır. `all_red` de EQU ile tanımlı bir sabittir, bellekte değil, komutun içinde durur.

Ad	Hex	Sağ alt	Sağ üst	Sol üst	Sol alt
situation	030Ch	yeşil	kırmızı	yeşil	kırmızı
s1	069Ah	sarı	kırmızı + sarı	sarı	kırmızı + sarı
s2	0861h	kırmızı	yeşil	kırmızı	yeşil
s3	0861h	kırmızı	yeşil	kırmızı	yeşil
s4	04D3h	kırmızı + sarı	sarı	kırmızı + sarı	sarı
all_red	0249h	kırmızı	kırmızı	kırmızı	kırmızı

Tablo 10.2.9: Trafik durumları ve yanan lambalar.

Tablodaki her satır bit hesabıyla doğrulanabilir. s1 = 069Ah = 0000 0110 1001 1010b'de 1 olan bitler 1, 3, 4, 7, 9 ve 10'dur. Bit 1 sağ alt direğin sarısı, bit 3 ve 4 sağ üst direğin kırmızısı ile sarısı, bit 7 sol üst direğin sarısı, bit 9 ve 10 sol alt direğin kırmızısı ile sarısıdır. s1, yeşilden kırmızıya geçecek yönlerde sarıyı, kırmızıdan yeşile geçecek yönlerde "kırmızı + sarı" uyarısını yakar. Kırmızı + sarı, "hazırlan" anlamına gelen geçiş işaretidir. s4 aynı geçişi ters yönde yapar.

s2 ile s3 aynı sözcüktür. Program her durumda aynı süre (5 saniye) beklediği için yan yolun yeşili iki aralık, yani 10 saniye sürer. Süreyi değiştirmek için programa dokunmak gerekmez; tabloya aynı sözcüğü bir kez daha yazmak yeter. Davranışın kodda değil tabloda durması bu tasarımın en güçlü yanıdır.



Şekil 10.2.6: Programın kavşağa sırayla yazdığı durumlar: all_red (0249h), situation (030Ch), s1 (069Ah), s2 = s3 (0861h) ve s4 (04D3h). Altta port 4'e yazılan sözcüğün ikili yazımı vardır.

Sunumun etkileşimli trafik bit düzenleyicisi aynı hesabı ters yönden yaptırır: bir bite tıklanınca o lamba yanar ya da söner ve sözcüğün onaltılık değeri güncellenir. Tablodaki adlara tıklanınca o durumun sözcüğü yüklenir.

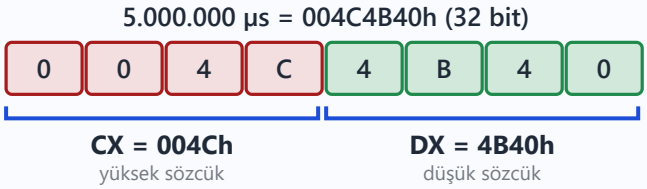
10.2.3.3 INT 15h ile bekleme

Her durum birkaç saniye sürmelidir. Programın bu süreyi ölçmesi gerekir. PC BIOS'u bunun için hazır bir hizmet sunar: **INT 15h, AH = 86h**. Hizmet, CX:DX çiftindeki 32 bitlik sayı kadar mikrosaniye bekler. CX yüksek sözcüğü, DX düşük sözcüğü taşır.

Örnek 10.2.3

5 saniyelik bekleme için CX ve DX ne olmalıdır?

1. Süre mikrosaniyeye çevrilir: 5 s = 5 × 1.000.000 = 5.000.000 µs.
2. Onluk sayı ardışık 16'ya bölmeyle onaltılığa çevrilir: 5.000.000 ÷ 16 = 312.500 kalan 0; 312.500 ÷ 16 = 19.531 kalan 4; 19.531 ÷ 16 = 1.220 kalan 11 (B); 1.220 ÷ 16 = 76 kalan 4; 76 ÷ 16 = 4 kalan 12 (C); 4 ÷ 16 = 0 kalan 4.
3. Kalanlar sondan başa okunur: 4C4B40h. 32 bitlik yazımı 004C4B40h'dir.
4. Sayı ikiye bölünür: CX = 004Ch, DX = 4B40h.



Şekil 10.2.7: 5.000.000 µs = 004C4B40h. Yüksek sözcük CX'e (004Ch), düşük sözcük DX'e (4B40h) yazılır.

```
; 5 saniye bekle (5 milyon µs)
mov cx, 4Ch     ; 004C4B40h = 5 000 000
mov dx, 4B40h
mov ah, 86h
int 15h
```

Aynı hesap başka süreler için de yapılır. 1 saniye 1.000.000 = 000F4240h'dir: CX = 000Fh, DX = 4240h. 0,2 saniye 200.000 = 00030D40h'dir: CX = 0003h, DX = 0D40h. İkinci örnekte DX'in onaltılık değeri harfle başlar; kaynak koda `0D40h` diye yazılır, başına 0 konmazsa assembler onu bir etiket adı sanar. Hizmetin bir çağrıda bekleyebileceği en uzun süre CX = DX = FFFFh, yani 4.294.967.295 µs'dir: yaklaşık 71,6 dakika.

Başarılı bir beklemeden sonra CF = 0 döner. emu8086'da hizmet AX'e dokunmaz; bazı gerçek BIOS'lar ise AH'ye bir durum kodu yazar. Program bu yüzden çağrıdan sonra AX'teki değere güvenmez. Gerçek PC'de bu hizmet zamanlayıcı yongasının kesmesiyle ölçülür ve süresi işlemcinin hızından bağımsızdır. emu8086'da da süreli bekleme bu hizmetle yapılır; öykünücü bekleme süresini kendi saatine göre uygular.

> **Sık yapılan hata**
>
> Süreyi CX ve DX'e onluk olarak bölüştürmek: 5.000.000 için CX = 5, DX = 0 yazmak. CX:DX tek bir 32 bitlik ikili sayıdır; CX = 5, DX = 0 değeri 5 × 65.536 = 327.680 µs, yani yaklaşık 0,33 saniyedir. Sayı önce bütün olarak onaltılığa çevrilir, sonra 16 bitlik iki yarıya ayrılır.

10.2.3.4 Yazılımla gecikme: LOOP ve saat turu

BIOS hizmeti olmayan bir gömülü sistemde bekleme, işlemcinin bilinen sayıda saat turu harcamasıyla yapılır. En basit yol boş bir `LOOP` döngüsüdür. Döngünün süresi, komutların saat turu sayılarından hesaplanır. 8086 kılavuzuna göre gereken süreler Tablo 10.2.10'dadır.

Komut	Saat turu
`MOV CX, sabit`	4
`LOOP` (CX ≠ 0, atlar)	17
`LOOP` (CX = 0, atlamaz)	5
`DEC r16`	2
`JNZ` (atlar / atlamaz)	16 / 4
`CALL` yakın / `RET` yakın	19 / 8

Tablo 10.2.10: Gecikme döngüsündeki komutların 8086 süreleri.

```
; ≈ 1 ms gecikme (4,77 MHz 8086)
mov  cx, 281    ; 4 tur
bekle:
loop bekle      ; 280 × 17 + 1 × 5 tur
```

CX = N ile başlayan döngüde `LOOP` N kez çalışır. İlk N − 1 seferde CX sıfıra inmemiştir ve komut atlar (17 tur). Son seferde CX = 0 olur ve komut atlamaz (5 tur). Toplam süre şöyledir:

> T = 4 + 17(N − 1) + 5 = 17N − 8 saat turu.

Saat turunu süreye çevirmek için frekans gerekir. Orijinal IBM PC'nin 8088'i 4,77 MHz'de çalışır: bir saat turu 1 / 4,77 MHz ≈ 0,2096 µs'dir. 1 ms bu yüzden 4.770 saat turudur. 17N − 8 = 4.770 denkleminden N ≈ 281,06 çıkar. CX'e yalnız tam sayı yazılabildiği için en yakın değer seçilir: N = 281, 17 ·

281 − 8 = 4.769 tur, yani ≈ 0,9998 ms.

Aynı döngü başka bir işlemcide başka bir süre verir. 5 MHz'de 1 ms 5.000 turdur; N ≈ 294,6 ve N = 295 seçilirse 5.007 tur ≈ 1,0014 ms elde edilir. 4,77 MHz için bulunan N = 281 ise 5 MHz'de yalnız 4.769 / 5 ≈ 954 µs sürer. Yazılımla gecikmenin en büyük kusuru budur: süre, saat hızına bağlıdır. Program daha hızlı bir işlemciye taşınınca bütün beklemeler kısalır.

Tek döngünün bir üst sınırı da vardır. CX = 0 ile başlayan `LOOP` önce CX'i FFFFh'ye indirir ve 65.536 kez çalışır. Bu, 17 · 65.536 − 8 = 1.114.104 tur, 4,77 MHz'de yaklaşık 233,6 ms'dir. Daha uzun bekleme için iç içe döngü gerekir:

```
        mov  dx, 100     ; 4 tur
dis:    mov  cx, 281     ; 4 tur
ic:     loop ic          ; 17 · 280 + 5 = 4 765 tur
        dec  dx          ; 2 tur
        jnz  dis         ; 16 tur, son seferde 4
```

Dış döngünün bir turu 4 + 4.765 + 2 = 4.771 tur ve ardından bir `JNZ` 'dir. Yüz turun 99'unda `JNZ` atlar (16), sonuncusunda atlamaz (4). Toplam 4 + 100 · 4.771 + 99 · 16 + 4 = 478.692 tur, 4,77 MHz'de yaklaşık 100,4 ms'dir. İç sayaç CX'te, dış sayaç DX'te durur; `LOOP` yalnız CX'i kullandığından iki sayaç birbirine karışmaz. Gecikme bir yordam olarak yazılıp çağrılırsa her çağrıya `CALL` (19) ve `RET` (8) turu da eklenir.

Bu hesaplar yaklaşıktır. Kesmeler döngünün arasına girip süreyi uzatabilir. Yavaş belleğin eklediği bekleme durumları ve komut kuyruğunun doluluğu da tur sayısını değiştirir. emu8086 komut sürelerini modellemez: döngü, bilgisayarın hızında ve öykünücünün adım gecikmesi ayarına göre koşar. Bu yüzden emu8086'da süreli bekleme INT 15h/86h ile yapılır; yazılımla gecikme hesabı gerçek donanım içindir.

10.2.3.5 Kod izleme: gecikme döngüsü

Formülün nereden geldiğini görmek için CX = 4 ile kısa bir döngü izlenir:

```
org 100h
mov cx, 4      ; 4 saat turu
bekle:
loop bekle     ; 17 tur, sonuncu 5
ret
```

Adım	Komut	CX	IP	Açıklama
1	`mov cx, 4`	0004h	0103h	4 tur; toplam 4
2	`loop bekle`	0003h	0103h	CX = 3 ≠ 0: aynı komuta atlar, 17 tur; toplam 21
3	`loop bekle`	0002h	0103h	CX = 2: 17 tur; toplam 38
4	`loop bekle`	0001h	0103h	CX = 1: 17 tur; toplam 55

Adım	Komut	CX	IP	Açıklama
5	`loop bekle`	0000h	0105h	CX = 0: atlamaz, 5 tur; toplam 60 = 17 · 4 − 8
6	`ret`	0000h	0000h	döngüden çıkıldı; 60 tur 4,77 MHz'de ≈ 12,6 µs

Tablo 10.2.11: Gecikme döngüsünün izlemesi; açıklamadaki tur toplamları 8086 komut sürelerinden hesaplanmıştır.

`LOOP bekle` komutu kendi adresine (0103h) atlar; makine kodu `E2 FE`'dir ve FEh = −2, komutun kendi uzunluğu kadar geri gitmek demektir. İlk üç çalıştırmada IP yine 0103h olur. Dördüncü çalıştırmada CX = 0'a iner, atlama olmaz ve IP 0105h'ye, `RET`'e geçer. Toplam 4 + 17 + 17 + 17 + 5 = 60 = 17 · 4 − 8 turdur. 60 tur, 4,77 MHz'de 60 × 0,2096 ≈ 12,6 µs sürer.

10.2.3.6 Trafik programı

Program dört parçadan oluşur: bütün yönleri kırmızıya alan hazırlık, tablodaki sözcüğü porta yazan döngü gövdesi, 5 saniyelik bekleme ve tablonun sonunda başa dönüş.

```
#start=Traffic_Lights.exe#
mov ax, all_red
out 4, ax
mov si, offset situation
next:
mov ax, [si]
out 4, ax
mov cx, 4Ch      ; 5 s bekle
mov dx, 4B40h
mov ah, 86h
int 15h
add si, 2        ; sonraki durum
cmp si, sit_end
jb  next
mov si, offset situation
jmp next
; ardından durum tablosu (situation … all_red)
```

Program ilk iş olarak bütün yönleri kırmızı yapar. Bu bir güvenlik önlemidir: denetleyici açıldığında kavşağın hangi durumda kaldığı bilinmez ve bir yön yeşilken öbürüne de yeşil yakılmamalıdır. `MOV AX, all_red` komutu sabiti doğrudan AX'e yükler (`B8 49 02`); `OUT 4, AX` sözcüğün iki baytını port 4 ve 5'e yazar.

SI, tablodaki sıradaki sözcüğün adresini tutar. `MOV AX, [SI]` o adresteki sözcüğü okur ve `OUT 4, AX` kavşağı o duruma geçirir. Ardından 5 saniye beklenir. `ADD SI, 2` bir sonraki sözcüğe geçer; tablo sözcüklerden oluştuğu için adım 2 bayttır. `CMP SI, sit_end` SI'yı tablonun bitiş adresiyle karşılaştırır. SI hâlâ küçükse CF = 1'dir ve `JB next` döngüye döner. Eşitlik olunca CF = 0'dır; SI tablonun başına alınır ve `JMP next` ile döngü yeniden başlar. Program hiç bitmez: gömülü sistemlerde ana program genellikle böyle sonsuz bir döngüdür.

Tablonun bellekteki yeri, kodun uzunluğuna bağlıdır. Program ORG 100h ile derlenirse kod 36 bayt tutar ve tablo 0124h'den başlar: situation 0124h, s1 0126h, s2 0128h, s3 012Ah, s4 012Ch'dedir; sit_end = 012Eh olur. Kaynakta tablonun yeri değişirse bu adresler de değişir, ama program yalnız etiketleri kullandığı için doğru çalışmayı sürdürür. emu8086 bir .COM programını yüklerken CX'e program boyutunu yükler; izlemede ilk `MOV CX` 'ten önceki CX 0000h alınmıştır. Program bu ilk değere dayanmadığı için sonuç değişmez.

Adım	Komut	AX	SI	CX	DX	CF	Port 4	Açıklama
1	`mov ax, all_red`	0249h	0000h	0000h	0000h	0	—	AX = all_red = 0249h; EQU sabiti, bellekte yer kaplamaz
2	`out 4, ax`	0249h	0000h	0000h	0000h	0	0249h	bütün yönler kırmızı
3	`mov si, offset situation`	0249h	0124h	0000h	0000h	0	0249h	SI = 0124h: tablonun ilk sözcüğü (situation)
4	`mov ax, [si]`	030Ch	0124h	0000h	0000h	0	0249h	AX ← [0124h] = 030Ch
5	`out 4, ax`	030Ch	0124h	0000h	0000h	0	030Ch	lambalar situation durumuna geçer
6	`mov cx, 4Ch`	030Ch	0124h	004Ch	0000h	0	030Ch	CX = 004Ch: 5.000.000 µs'nin yüksek sözcüğü
7	`mov dx, 4B40h`	030Ch	0124h	004Ch	4B40h	0	030Ch	DX = 4B40h: düşük sözcük
8	`mov ah, 86h`	860Ch	0124h	004Ch	4B40h	0	030Ch	AH = 86h: BIOS bekleme hizmeti; AL değişmez
9	`int 15h`	860Ch	0124h	004Ch	4B40h	0	030Ch	5 saniye beklenir; başarıda CF = 0
10	`add si, 2`	860Ch	0126h	004Ch	4B40h	0	030Ch	SI = 0126h: sonraki durum
11	`cmp si, sit_end`	860Ch	0126h	004Ch	4B40h	1	030Ch	0126h − 012Eh: borç var, CF = 1
12	`jb next`	860Ch	0126h	004Ch	4B40h	1	030Ch	JB atlar: next
13	`mov ax, [si]`	069Ah	0126h	004Ch	4B40h	1	030Ch	AX ← [0126h] = 069Ah (s1)
14	`out 4, ax`	069Ah	0126h	004Ch	4B40h	1	069Ah	sarı geçiş durumu
…	bekleme, SI = 0128h, CMP ve JB							
23	`out 4, ax`	0861h	0128h	004Ch	4B40h	1	0861h	s2 = 0861h; yan yol yeşil
…	bekleme, SI = 012Ah							
32	`out 4, ax`	0861h	012Ah	004Ch	4B40h	1	0861h	s3 = 0861h; aynı durum 5 saniye daha sürer
…	bekleme, SI = 012Ch							
41	`out 4, ax`	04D3h	012Ch	004Ch	4B40h	1	04D3h	s4 = 04D3h; ana yola dönüş hazırlığı
…	bekleme							
46	`add si, 2`	86D3h	012Eh	004Ch	4B40h	0	04D3h	SI = 012Eh = sit_end

Adım	Komut	AX	SI	CX	DX	CF	Port 4	Açıklama
47	`cmp si, sit_end`	86D3h	012Eh	004Ch	4B40h	0	04D3h	eşit: CF = 0, ZF = 1
48	`jb next`	86D3h	012Eh	004Ch	4B40h	0	04D3h	JB atlamaz: tablo bitti
49	`mov si, offset situation`	86D3h	0124h	004Ch	4B40h	0	04D3h	SI tablonun başına döner
50	`jmp next`	86D3h	0124h	004Ch	4B40h	0	04D3h	sonsuz döngü
51	`mov ax, [si]`	030Ch	0124h	004Ch	4B40h	0	04D3h	AX ← [0124h] = 030Ch
52	`out 4, ax`	030Ch	0124h	004Ch	4B40h	0	030Ch	situation yeniden: döngü baştan başlar

Tablo 10.2.12: Trafik ışıkları programının izlemesi (tablo 0124h'de; INT 15h bekleme dışında her satır komut yürütüldükten sonraki durum).

İzleme, `INT 15h` çağrısının AX'te iz bıraktığını da gösterir. `MOV AH, 86h` yalnız üst baytı değiştirir; AL, son okunan durumun düşük baytını korur. Bu yüzden 8. adımdan sonra AX = 860Ch'dir. Döngünün başındaki `MOV AX, [SI]` AX'in tamamını yeniden yazdığı için bu artık bir sorun oluşturmaz.

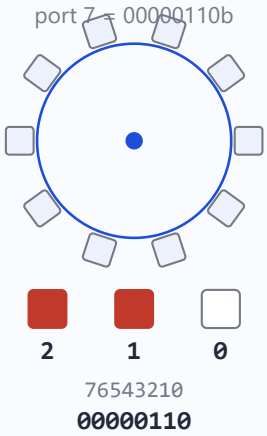
> **Sık yapılan hata**
>
> `ADD SI, 2` yerine `INC SI` yazmak. Tablo sözcüklerden oluşur; SI bir artırılırsa ikinci okuma bir sözcüğün ortasından yapılır. 0125h'den okunan sözcük, situation'ın yüksek baytı (03h) ile s1'in düşük baytının (9Ah) birleşimi olan 9A03h'dir. Bu değer bit 12–15'e de taşan anlamsız bir lamba düzeni yakar.

10.2.4 Step motor

10.2.4.1 emu8086 step motoru

Sıradan bir doğru akım motoru, gerilim verildiği sürece döner; ne kadar döndüğünü bilmek için ayrı bir algılayıcı gerekir. **Step motor** (*stepper motor*) ise sargılarına uygulanan her yeni uyarma düzeninde belirli bir açı kadar döner ve orada durur. Mil, sargılardan hangilerinin akım taşıdığına göre konum alır. Uyarma düzeni sırayla değiştirilirse motor adım adım döner. Bu yüzden konumu, gönderilen adım sayısından bilinir. Yazıcı kafaları, disket sürücüleri, 3B yazıcılar ve CNC tezgâhları step motorla sürülür.

emu8086'nın step motoru port 7'ye bağlıdır ve basitleştirilmiş bir modeldir. Porta yazılan baytın bit 0, 1 ve 2'si üç elektromıknatısı açar ya da kapatır. Motor hem saat yönünde (*clockwise*, CW) hem de ters yönde (CCW) döndürülebilir. Bir yarım adım 11,25°, bir tam adım 22,5°'dir; tam tur bu yüzden 32 yarım adım ya da 16 tam adımdır.



Şekil 10.2.8: emu8086 step motoru. Port 7 = 0000 0110b: bit 2 ve bit 1'e bağlı mıknatıslar açık, rotor ikisinin arasına çekilir.

Motorun bir de durum çıkışı vardır. Bir `OUT` komutundan sonra motor yeni konumuna gidene kadar meşguldür. Yeni komut almaya hazır olduğunda port 7'den okunan baytın bit 7'sini 1 yapar. Program her baytı göndermeden önce bu biti yoklar. Hazır olmayan motora gönderilen bayt kaybolur ya da motor adım kaçırır.

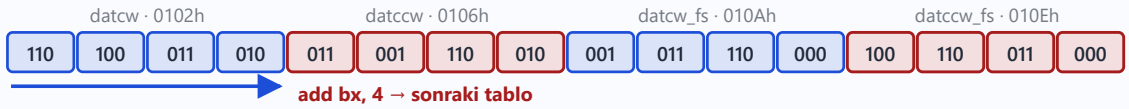
10.2.4.2 Veri tabloları ve mıknatıs dizileri

Motoru döndürmek, mıknatısları belirli bir sırayla açıp kapatmaktır. Bu sıra programda dört baytlık tablolar olarak tutulur. emu8086'nın örnek programı dört tablo tanımlar (Tablo 10.2.13).

Tablo	Adres	Anlam	1	2	3	4
`datcw`	0102h	CW yarım adım	110	100	011	010
`datccw`	0106h	CCW yarım adım	011	001	110	010
`datcw_fs`	010Ah	CW tam adım	001	011	110	000
`datccw_fs`	010Eh	CCW tam adım	100	110	011	000

Tablo 10.2.13: Step motor veri tabloları (bit 2–0 = mıknatıs 2–0; adresler ORG 100h'ye göre).

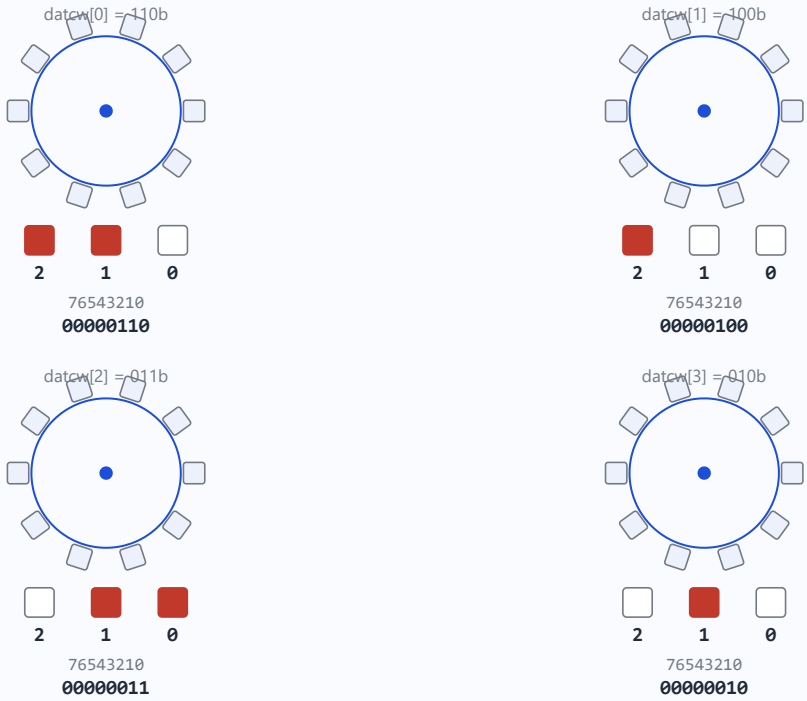
Baytın her biti bir mıknatısı gösterir: 110b, mıknatıs 2 ve 1'in açık, 0'ın kapalı olduğu anlamına gelir; 000b bütün mıknatısları kapatır. Yarım adım tablosu çift ve tek mıknatısı sırayla açar: 110 (iki), 100 (bir), 011 (iki), 010 (bir). İki mıknatıs açıkken rotor ikisinin arasında, tek mıknatıs açıkken o mıknatısın karşısında durur. Bu ara konumlar adımı yarıya böler. Tam adım tablosu ise tek, çift, çift mıknatıs ve “hepsi kapalı” baytını gönderir.



Şekil 10.2.9: Dört tablo bellekte arka arkaya durur ve her biri 4 bayttır. BX'e 4 eklemek bir sonraki tabloya geçirir.

Tabloların bellekte ardışık durması programın en kullanışlı ayrıntısıdır. datcw 0102h'de başlar; datccw 4 bayt sonra, 0106h'de; datcw_fs 010Ah'de; datccw_fs 010Eh'dedir. BX bir tablonun başını gösteriyorsa `ADD BX, 4` onu bir sonrakine taşır. Program 32 (20h) tur sonra tabloyu değiştirir. Yön ve adım türü böylece CW yarım → CCW yarım → CW tam → CCW tam → başa sırasıyla döner.

Kaynak koddaki yazım biçimi de değişebilir. emu8086'nın özgün örneğinde her bayt ayrı satırda ve sekiz basamakla yazılır: `datcw db 0000_0110b`, alt satırda `db 0000_0100b` … Notun programında tablo aynı değerlerle tek satıra sığdırılmıştır: `datcw db 110b,100b,011b,010b`. İki yazım da aynı dört baytı üretir.



Şekil 10.2.10: datcw tablosunun dört baytı sırayla: 110b (mıknatıs 2 ve 1), 100b (2), 011b (1 ve 0), 010b (1). Her bayt rotoru bir yarım adım, 11,25° döndürür.

10.2.4.3 Kontrol programı

```
ORG 100h
steps_before_direction_change = 20h
jmp start
datcw     db 110b,100b,011b,010b
datccw    db 011b,001b,110b,010b
datcw_fs  db 001b,011b,110b,000b
datccw_fs db 100b,110b,011b,000b
start:
mov bx, offset datcw
mov si, 0
mov cx, 0              ; tur sayacı
next_step:
wait: in   al, 7
      test al, 10000000b
      jz   wait
mov al, [bx][si]
out 7, al
inc si
cmp si, 4
jb  next_step
mov si, 0
inc cx
cmp cx, steps_before_direction_change
jb  next_step
mov cx, 0
add bx, 4              ; sonraki tablo
cmp bx, offset datccw_fs
jbe next_step
mov bx, offset datcw
jmp next_step
```

`steps_before_direction_change = 20h` bir sabittir. `=` yönergesi EQU gibi çalışır: bellekte yer kaplamaz, kullanıldığı yerde değeriyle değiştirilir. Tablolar kodun başında durduğu için ilk komut `JMP start` onların üzerinden atlar; aksi hâlde işlemci tablo baytlarını komut sanıp yürütürdü. Kısa atlamanın kodu `EB 10`'dur: 0102h'den 16 bayt ileri, 0112h'ye.

Programın üç sayacı vardır. BX kullanılan tablonun başını, SI tablo içindeki sırayı (0–3), CX tamamlanan tur sayısını tutar. Döngü gövdesinin ilk üç satırı yoklama döngüsüdür: bit 7 = 0 iken motor meşguldür ve port yeniden okunur. Bit 7 = 1 olunca `MOV AL, [BX][SI]` sıradaki baytı okur.

`[BX][SI]` yazımı `[BX + SI]` ile aynıdır: taban artı indis adreslemesi. BX tablonun başlangıç adresini, SI o tablodaki konumu verir; etkin adres ikisinin toplamıdır. BX = 0102h, SI = 2 için okunan bayt [0104h] = 011b'dir. Bu adresleme modu, iki boyutlu bir dizinin satır ve sütunu gibi düşünülebilir: BX hangi tablo, SI tablonun kaçıncı elemanı.

Bayt porta yazıldıktan sonra SI bir artar. SI 4'ten küçükse aynı tablonun sonraki baytına geçilir. SI = 4 olunca tablo bitmiştir: SI sıfırlanır ve CX bir artar. CX henüz 20h'ye ulaşmadıysa aynı tablo yeniden başlar. Ulaştıysa CX sıfırlanır ve BX 4 artırılır. Son satırlar BX'in tablolar dışına çıkmasını önler: `CMP BX, offset datccw_fs` · `JBE next_step` BX son tablonun adresine eşit ya da küçükse devam eder; değilse BX ilk tabloya döner.

Programın davranışı uzun bir koşuda görülebilir. Port 7 sürekli "hazır" döndürdüğünde porta yazılan ilk 128 bayt datcw'nin dört baytının 32 kez yinelenmesidir. 129. bayttan itibaren datccw (011, 001, 110, 010), 257. bayttan itibaren datcw_fs, 385. bayttan itibaren datccw_fs gelir; 513. baytta yeniden datcw başlar.

> **Sık yapılan hata**
>
> `CMP BX, offset datccw_fs` · `JBE` yerine `JB` yazmak. BX son tabloyu (010Eh) gösterdiğinde `JB` atlamaz ve BX hemen datcw'ye döner; CCW tam adım tablosu hiç kullanılmaz. Son tablonun da sayılması için eşitlik dahil edilir.

10.2.4.4 Kod izleme: ilk tur

İzlemede port 7'nin ilk okumada 00h (meşgul), sonraki bütün okumalarda 80h (hazır) döndürdüğü varsayılmıştır. Tablo 10.2.14 ilk dört baytı ve turun sonunu gösterir; yinelenen yoklama adımları "…" ile kısaltılmıştır.

Adım	Komut	AL	BX	SI	CX	ZF	CF	Port 7	Açıklama
1	`jmp start`	00h	0000h	0000h	0000h	0	0	—	veri tablolarının üzerinden atlanır
2	`mov bx, offset datcw`	00h	0102h	0000h	0000h	0	0	—	BX = 0102h: datcw (saat yönünde yarım adım)
3	`mov si, 0`	00h	0102h	0000h	0000h	0	0	—	SI = 0: tablodaki sıra
4	`mov cx, 0`	00h	0102h	0000h	0000h	0	0	—	CX = 0: tur sayacı
5	`wait: in al, 7`	00h	0102h	0000h	0000h	0	0	—	AL = 00h: bit 7 = 0, motor meşgul
6	`test al, 10000000b`	00h	0102h	0000h	0000h	1	0	—	00h AND 80h = 0: ZF = 1
7	`jz wait`	00h	0102h	0000h	0000h	1	0	—	ZF = 1: yeniden oku
8	`wait: in al, 7`	80h	0102h	0000h	0000h	1	0	—	AL = 80h: motor hazır
9	`test al, 10000000b`	80h	0102h	0000h	0000h	0	0	—	80h AND 80h ≠ 0: ZF = 0
10	`jz wait`	80h	0102h	0000h	0000h	0	0	—	atlamaz: bayt gönderilecek
11	`mov al, [bx][si]`	06h	0102h	0000h	0000h	0	0	—	AL ← [0102h + 0] = 06h = 110b

Adım	Komut	AL	BX	SI	CX	ZF	CF	Port 7	Açıklama
12	`out 7, al`	06h	0102h	0000h	0000h	0	0	06h	mıknatıs 2 ve 1 açık: yarım adım (11,25°)
13	`inc si`	06h	0102h	0001h	0000h	0	0	06h	SI = 1
14	`cmp si, 4`	06h	0102h	0001h	0000h	0	1	06h	1 − 4: CF = 1
15	`jb next_step`	06h	0102h	0001h	0000h	0	1	06h	JB atlar: sonraki bayt
…	hazır biti okunur (80h)								
19	`mov al, [bx][si]`	04h	0102h	0001h	0000h	0	0	06h	AL ← [0103h] = 04h = 100b
20	`out 7, al`	04h	0102h	0001h	0000h	0	0	04h	yalnız mıknatıs 2 açık
…	SI = 2, yoklama								
27	`mov al, [bx][si]`	03h	0102h	0002h	0000h	0	0	04h	AL ← [0104h] = 03h = 011b
28	`out 7, al`	03h	0102h	0002h	0000h	0	0	03h	mıknatıs 1 ve 0 açık
…	SI = 3, yoklama								
35	`mov al, [bx][si]`	02h	0102h	0003h	0000h	0	0	03h	AL ← [0105h] = 02h = 010b
36	`out 7, al`	02h	0102h	0003h	0000h	0	0	02h	yalnız mıknatıs 1 açık: ilk tur bitti
37	`inc si`	02h	0102h	0004h	0000h	0	0	02h	SI = 4
38	`cmp si, 4`	02h	0102h	0004h	0000h	1	0	02h	4 − 4 = 0: ZF = 1, CF = 0
39	`jb next_step`	02h	0102h	0004h	0000h	1	0	02h	JB atlamaz
40	`mov si, 0`	02h	0102h	0000h	0000h	1	0	02h	SI = 0: tablonun başına
41	`inc cx`	02h	0102h	0000h	0001h	0	0	02h	CX = 1: bir tur (4 bayt) tamam
42	`cmp cx, steps_before_direction_change`	02h	0102h	0000h	0001h	0	1	02h	1 − 20h: CF = 1
43	`jb next_step`	02h	0102h	0000h	0001h	0	1	02h	atlar: 32 tur dolmadı, aynı tabloyla devam

Tablo 10.2.14: Step motor programının ilk turu (port 7 önce 00h, sonra hep 80h döndürür).

İzlemenin ilk kısmı yoklamanın işleyişini gösterir. 5. adımda okunan 00h ile `TEST` sonucu sıfırdır ve ZF = 1 olur; `JZ` geri döner. 8. adımda okunan 80h'de bit 7 birdir; `TEST` sonucu sıfırdan farklıdır, ZF = 0 olur ve program ilerler. `TEST AL, 80h` komutu AL'yi değiştirmez; ama AL'nin değeri bir sonraki `MOV AL, [BX][SI]` ile zaten silinir. Durum baytına artık ihtiyaç kalmamıştır.

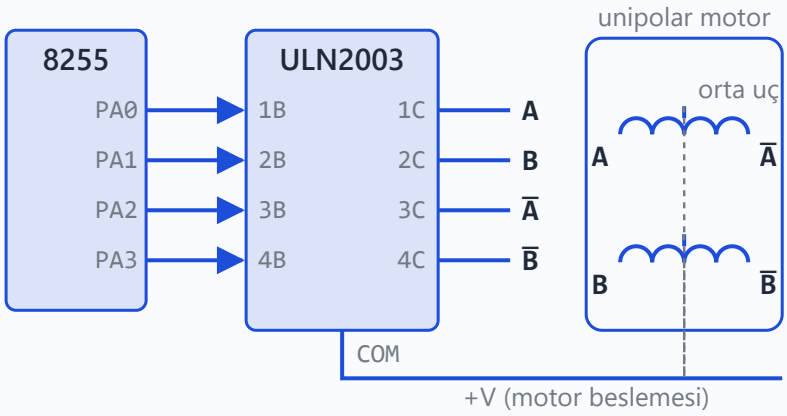
İkinci kısım tablonun taranmasıdır. SI = 0, 1, 2, 3 için port 7'ye sırasıyla 06h, 04h, 03h ve 02h yazılır. SI = 4 olduğunda `CMP SI, 4` sonucu sıfırdır: ZF = 1, CF = 0. `JB` yalnız CF = 1 iken atladığı için atlamaz. SI sıfırlanır, CX bire çıkar. `CMP CX, 20h` sonucunda CF = 1'dir (1 < 32) ve program aynı tabloyla ikinci

tura başlar. İlk turda motor dört yarım adım, yani 45° dönmüştür.

10.2.4.5 Gerçek donanım: unipolar motor ve ULN2003

emu8086'nın üç mıknatıslı motoru öğretim için basitleştirilmiş bir modeldir. Laboratuvarda en sık karşılaşılan step motor türü ise **unipolar** motordur. Bu motorda orta uçlu iki sargı vardır. Her sargı orta ucundan ikiye bölünür ve dört sargı yarısı oluşur: A, B, Ā ve B̄. Orta uçlar motorun besleme gerilimine (+V) bağlanır. Motorun beş ya da altı teli bu yüzdendir.

Bir sargı yarısından akım geçirmek için onun öbür ucu toprağa çekilir. Sargı akımı yüzlerce miliamperdir; 8255'in bir çıkış ucu ise ancak birkaç miliamper sürebilir. Araya bir sürücü yonga konur: **ULN2003**. ULN2003'te yedi Darlington transistör kanalı vardır ve her kanal 500 mA'e kadar akım çekebilir. Kanalın girişi 1 olunca çıkışı toprağa çeker ve o sargı yarısından akım geçer.



Şekil 10.2.11: 8255'in PA0–PA3 uçları ULN2003'ün dört kanalı üzerinden unipolar motorun A, B, Ā ve B̄ sargı yarılarını sürer. Orta uçlar ve ULN2003'ün COM ucu motor beslemesine (+V) bağlıdır.

Sargı bir bobindir ve akımı kesildiği anda uçlarında ters yönde yüksek bir gerilim oluşur. Bu gerilim transistörü bozabilir. ULN2003'ün her kanalında bu iş için bir diyot vardır; COM ucu +V'ye bağlanınca diyotlar ters gerilimi beslemeye boşaltır. COM ucunun bağlanmaması, sürücünün kısa sürede bozulmasının en yaygın nedenidir.

Gerçek motor **açık çevrimle** sürülür: emu8086'nın motorundaki gibi bir "hazır" biti yoktur. Motorun hızını, program adımlar arasında ne kadar beklediğiyle belirler. Bekleme çok kısa tutulursa rotor bir sonraki uyarmaya yetişemez ve **adım kaçırır**. Program kaçan adımı fark edemez; mil, programın sandığı konumda değildir. Bu yüzden adım aralığı motorun kataloğundaki en yüksek adım hızına göre seçilir.

Sürüş	Port A dizisi	Özellik
Tek faz (dalga), tam adım	01h 02h 04h 08h	her an bir sargı yarısı; en az akım
İki faz, tam adım	03h 06h 0Ch 09h	her an iki sargı yarısı; daha yüksek tork

Sürüş	Port A dizisi	Özellik
Yarım adım	01h 03h 02h 06h 04h 0Ch 08h 09h	iki diziyi karıştırır; adım açısı yarıya iner

Tablo 10.2.15: Unipolar motorun uyarma dizileri (port A'nın bit 3…0'ı = $\bar{B}$ $\bar{A}$ B A).

Dizilerin mantığı bit düzeyinde görülür. Tek faz dizisinde 1 biti A'dan B'ye, $\bar{A}$'ya, $\bar{B}$'ye kayar: 0001, 0010, 0100, 1000. İki faz dizisinde yan yana iki bit birlikte kayar: 0011, 0110, 1100, 1001. Son bayt 1001b, $\bar{B}$ ile A'nın birlikte açık olduğunu söyler; dizi bir halka gibi başa bağlanır. Yarım adım dizisi bu iki dizinin elemanlarını sırayla dizer: tek, çift, tek, çift. Böylece adım açısı yarıya iner; örneğin 1,8°'lik (turda 200 adım) bir motor 0,9° adımla döner. Dizi tersten uygulanırsa motor ters yönde döner.

emu8086 motorunun tabloları (110b, 100b …) bu dört fazlı motora uygulanamaz. Üç mıknatıslı modelin bit düzeni ile dört sargı yarılı motorunki farklıdır. Gerçek motor için tablo, sargıların porta bağlanış sırasına göre yeniden yazılır.

10.2.4.6 Kod izleme: yarım adım sürüşü

Aşağıdaki program yarım adım dizisini 8255'in port A'sına gönderir. Port A'nın daha önce 82h ile çıkış yapıldığı varsayılır. Dizideki konum bellekteki `adim` sözcüğünde tutulur.

```
org 100h
ileri:
inc  adim           ; geri için: dec adim
mov  bx, adim
and  bx, 7          ; dizin 0–7
mov  al, yarim[bx]
mov  dx, 310h       ; port A
out  dx, al
call bekle          ; adım aralığı = hız
jmp  ileri
bekle: ret          ; (gecikme)
yarim db 01h,03h,02h,06h
      db 04h,0Ch,08h,09h
adim  dw 6
```

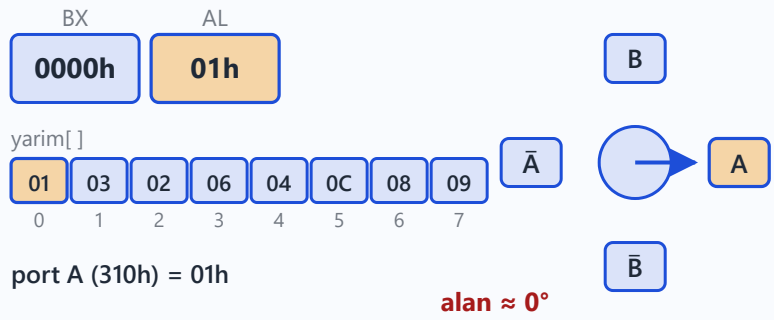
Programın püf noktası `AND BX, 7`'dir. `adim` sürekli artar: 6, 7, 8, 9 … Dizi ise yalnız sekiz elemanlıdır. 7 = 0000 0111b ile VE işlemi, sayının yalnız son üç bitini bırakır; sonuç her zaman 0–7 arasındadır. 8 = 1000b için sonuç 0, 9 için 1 olur ve dizi kendiliğinden başa sarar. Bu, 8'e bölümden kalanı bir bölme komutu olmadan bulmanın yoludur ve yalnız 2'nin kuvveti uzunluktaki dizilerde çalışır.

Geri döndürmek için `INC adim` yerine `DEC adim` yazmak yeter. `adim` 0'dan 1 azalınca FFFFh olur; `AND BX, 7` bunu 7'ye çevirir ve dizi sondan başa doğru okunur. Sayaç bellekte durduğu için `INC adim` bir bellek işlenenli komuttur (`FF 06 21 01`): işlemci sözcüğü okur, bir artırır ve geri yazar. Gecikme yordamı bu izlemede yalnız `RET`'tir; gerçek programda adım aralığını, yani motorun hızını belirler.

Adım	Komut	adim	BX	AL	SP	Port A	Açıklama
1	`inc adim`	0007h	0000h	00h	FFFEh	—	bellekteki sözcük 6 → 7
2	`mov bx, adim`	0007h	0007h	00h	FFFEh	—	BX ← adim = 7
3	`and bx, 7`	0007h	0007h	00h	FFFEh	—	dizin 0–7 aralığında kalır: 7
4	`mov al, yarim[bx]`	0007h	0007h	09h	FFFEh	—	AL ← yarim[7] = 09h = 1001b: $\bar{B}$ ve A
5	`mov dx, 310h`	0007h	0007h	09h	FFFEh	—	DX = 310h: 8255 port A
6	`out dx, al`	0007h	0007h	09h	FFFEh	09h	sargılar sürülür: alan A ile $\bar{B}$ arasında
7	`call bekle`	0007h	0007h	09h	FFFCh	09h	dönüş adresi (0116h) yığına: SP = FFFCh
8	`bekle: ret`	0007h	0007h	09h	FFFEh	09h	gecikme yordamı burada yalnız RET
9	`jmp ileri`	0007h	0007h	09h	FFFEh	09h	sonraki adım
10	`inc adim`	0008h	0007h	09h	FFFEh	09h	adim = 8
11	`mov bx, adim`	0008h	0008h	09h	FFFEh	09h	BX = 8
12	`and bx, 7`	0008h	0000h	09h	FFFEh	09h	8 AND 7 = 0: dizi başa sarar
13	`mov al, yarim[bx]`	0008h	0000h	01h	FFFEh	09h	AL ← yarim[0] = 01h: yalnız A
…	DX = 310h						
15	`out dx, al`	0008h	0000h	01h	FFFEh	01h	alan A sargısına döner: bir yarım adım
…	CALL, RET, JMP						
19	`inc adim`	0009h	0000h	01h	FFFEh	01h	adim = 9
20	`mov bx, adim`	0009h	0009h	01h	FFFEh	01h	BX = 9
21	`and bx, 7`	0009h	0001h	01h	FFFEh	01h	9 AND 7 = 1
22	`mov al, yarim[bx]`	0009h	0001h	03h	FFFEh	01h	AL ← yarim[1] = 03h: A ve B
…	DX = 310h						
24	`out dx, al`	0009h	0001h	03h	FFFEh	03h	alan A ile B arasına geçer: bir yarım adım daha

Tablo 10.2.16: Gerçek motor için yarım adım dizisini süren programın izlemesi (adim başlangıçta 6).

İzlemedeki açılar alanın sargılar arasındaki yönünü gösterir. 09h = 1001b, $\bar{B}$ ile A'nın birlikte açık olduğunu söyler; alan ikisinin arasına, yaklaşık 315°'ye yönelir. 01h'de yalnız A açıktır (0°), 03h'te A ve B (45°). Her `OUT` alanı 45°'lik bir sargı açısı kadar, rotoru bir yarım adım kadar ilerletir.



Şekil 10.2.12: adim = 8 olunca `AND BX, 7` dizini 0'a sarar; yarim[0] = 01h ile yalnız A sargısı açık kalır ve alan 0°'ye gelir.

10.2.5 LED ekran ve 7 bölütlü gösterge

10.2.5.1 emu8086 LED ekranı

emu8086'nın LED ekranı port 199'a bağlıdır ve beş basamaklı bir onluk sayı gösterir. Porta yazılan 16 bitlik sözcük, ikiye tümleyen gösterimde **işaretli** bir sayı olarak yorumlanır. Gösterilebilen aralık bu yüzden −32.768 … 32.767'dir. 199 ≤ 255 olduğundan port sabit olarak yazılır: `OUT 199, AX`. Ekran bir sözcük beklediği için AX kullanılır; `OUT 199, AL` yalnız düşük baytı gönderir.



Şekil 10.2.13: LED ekran: port 199'a yazılan AX = 0163h onluk 355 olarak görünür.

Ekranın işi, ikili sayıyı onluğa çevirmektir. Programcı sayıyı onluk olarak düşünür, assembler onu ikiliye çevirir, aygıt yeniden onluğa döndürür. `MOV AX, 1234` komutu AX'e 04D2h yazar (1234 = 4 · 256 + 210 = 04D2h). Ekrana giden de bu ikili sözcüktür.

Eksi sayılar ikiye tümleyenle yazılır. −5678 için önce 5678'in onaltılık değeri bulunur: 162Eh. İkiye tümleyen 10000h − 162Eh = E9D2h'dir. `MOV AX, -5678` yazıldığında bu hesabı assembler yapar ve `B8 D2 E9` kodunu üretir. Ekran E9D2h'nin en yüksek bitinin 1 olduğunu görür ve sayıyı −5678 olarak gösterir. Aynı sözcük işaretsiz okunsaydı 59.858 olurdu; ekranın yorumu, bitlerin anlamını belirler.

emu8086'nın LED ekran örneği `#make_bin#` yönergesiyle başlar. Bu yönerge DOS'a bağlı olmayan düz bir ikili dosya (.bin) üretir: başlık yok, PSP yok, yalnız makine kodu. Gömülü bir sistemin ROM'una yazılacak dosya tam olarak budur.

10.2.5.2 Kod izleme: LED ekran programı

```
#start=led_display.exe#
mov ax, 1234
out 199, ax
mov ax, -5678
out 199, ax
; porta sonsuz döngüde yaz:
mov ax, 0
x1:
out 199, ax
inc ax
jmp x1
hlt              ; hiç çalışmaz
```

Program önce iki sabit sayı gösterir, sonra 0'dan başlayıp durmadan artan bir sayaç kurar. Döngüde `INC AX` ve `JMP x1` vardır, ama hiçbir çıkış koşulu yoktur. Bu yüzden döngüden sonraki `HLT`'ye hiç ulaşılmaz. `HLT` yalnız programın sonunu belirtmek için yazılmıştır.

Adım	Komut	AX	Ekran	Açıklama
1	`mov ax, 1234`	04D2h	—	1234 = 04D2h
2	`out 199, ax`	04D2h	1234	ekranda 1234
3	`mov ax, -5678`	E9D2h	1234	−5678 = 10000h − 162Eh = E9D2h
4	`out 199, ax`	E9D2h	−5678	ekran sözcüğü işaretli okur: −5678
5	`mov ax, 0`	0000h	−5678	sayaç 0
6	`out 199, ax`	0000h	0	ekranda 0
7	`inc ax`	0001h	0	AX = 1
8	`jmp x1`	0001h	0	x1'e geri; HLT'ye hiç ulaşılmaz
9	`out 199, ax`	0001h	1	ekranda 1
10	`inc ax`	0002h	1	AX = 2
11	`jmp x1`	0002h	1	döngü
12	`out 199, ax`	0002h	2	ekranda 2

Tablo 10.2.17: LED ekran programının izlemesi; ekran sütunu port 199'a son yazılan sözcüğün işaretli değeridir.

Sayaç 32.767'ye (7FFFh) ulaştıktan sonra `INC AX` onu 8000h yapar. Ekran 8000h'yi işaretli okur ve −32.768 gösterir. Sayaç oradan −1'e (FFFFh) kadar artar, sonra 0'a sarar. `INC` bu geçişte OF'yi 1 yapar ama program bayraklara bakmadığı için hiçbir şey olmaz.

emu8086'da adım gecikmesi (*step delay*) 0 iken sayılar okunamayacak kadar hızlı akar. Döngüye INT 15h/86h ile bir bekleme eklenirse sayaç gözle izlenebilir. Örneğin her sayıdan sonra 0,2 saniye beklemek için `OUT`'tan sonra `MOV CX, 3`, `MOV DX, 0D40h`, `MOV AH, 86h`, `INT 15h` eklenir. Bekleme AH'yi değiştirdiği için sayaç AX'ten başka bir yazmaca alınmalıdır; örneğin `BX`'te sayılıp `OUT`'tan önce `MOV AX, BX` ile AX'e kopyalanır.

10.2.5.3 7 bölütlü gösterge

emu8086'nın LED ekranı sayıyı kendisi çözer. Gerçek donanımda bu işi program yapar. En yaygın sayısal gösterge **7 bölütlü göstergedir** (*seven-segment display*). Göstergede yedi çubuk biçimli LED (a–g) bulunur; ondalık nokta (dp) sekizinci LED'dir. Her LED'e bir **bölüt** denir. Bölütün bellek kesimiyle ilgisi yoktur; İngilizcede ikisine de *segment* dendiği için Türkçede ayrı adlar kullanılır.



Şekil 10.2.14: 7 bölütlü gösterge ve port A'ya bağlanışı (PA7 … PA0 → dp g f e d c b a). '3' rakamı için ortak katotta 4Fh, ortak anotta onun tersi B0h gönderilir.

Göstergenin sekiz LED'inin birer ucu ortak bir uçta birleşir. Bu ucun bağlanışı iki türü ayırır. **Ortak katotlu** (CC) göstergede ortak uç topraktadır; bölüt ucuna 1 verilince LED yanar. **Ortak anotlu** (CA) göstergede ortak uç +5 V'tadır; bölüt ucuna 0 verilince LED yanar. Ortak anotun kodu bu yüzden ortak katot kodunun bit bit tersidir ve `NOT AL` ile elde edilir. LED'ler doğrudan port ucuna değil, her biri bir akım sınırlayıcı direnç üzerinden bağlanır.

Bölüt kodu, yanması gereken bölütlerden hesaplanır. Port A'nın bitleri dp g f e d c b a sırasındadır: bit 0 = a, bit 6 = g, bit 7 = dp. '3' rakamında a, b, c, d ve g yanar; bit 0, 1, 2, 3 ve 6 birdir: 0100 1111b = 4Fh. Ortak anotta bunun tersi 1011 0000b = B0h gönderilir.

Rakam	Yanan bölütler	Ortak katot (CC)	Ortak anot (CA)
0	a b c d e f	3Fh	C0h
1	b c	06h	F9h
2	a b d e g	5Bh	A4h
3	a b c d g	4Fh	B0h
4	b c f g	66h	99h
5	a c d f g	6Dh	92h
6	a c d e f g	7Dh	82h
7	a b c	07h	F8h
8	a b c d e f g	7Fh	80h
9	a b c d f g	6Fh	90h

Tablo 10.2.18: Rakamların bölüt kodları (dp g f e d c b a sırası).

Tablo yalnız rakamlarla sınırlı değildir. Onaltılık rakamlar ve bazı harfler de yedi bölütle yazılabilir. 'A' için a b c e f g yanar: 0111 0111b = 77h. Küçük 'b' için c d e f g: 0111 1100b = 7Ch. Eksi işareti yalnız g bölütüdür: 0100 0000b = 40h.

10.2.5.4 Kod tablosu ve XLAT

Rakamı bölüt koduna çevirmek bir tablo aramasıdır. 8086'nın bu iş için tek baytlık bir komutu vardır: **XLAT** (*translate*). `XLAT`, AL ← DS:[BX + AL] işlemini yapar. BX tablonun başını, AL sıra numarasını gösterir; tablodaki bayt AL'ye yazılır.

```
; rakamı (0–9) bölüt koduna çevir
mov  bx, offset kod
mov  al, 3          ; rakam
xlat                ; AL = [BX + AL] = 4Fh
; not al            ; ortak anot ise
mov  dx, 310h       ; 8255 port A
out  dx, al
ret
kod  db 3Fh,06h,5Bh,4Fh,66h
     db 6Dh,7Dh,07h,7Fh,6Fh
```

Örnekte AL = 3'tür ve `XLAT` kod tablosunun 3 numaralı baytını (0'dan sayarak), 4Fh'yi getirir. Gösterge ortak anotluysa açıklama satırındaki `NOT AL` etkinleştirilir ve AL = B0h olur. Sonuç 8255'in port A'sına yazılır. `XLAT`'ın tek kısıtı tablonun en çok 256 bayt olmasıdır; AL 8 bitlik bir sıra numarasıdır. Komut bayraklara dokunmaz ve 11 saat turu sürer.

Tablo yönteminin hesaplama yöntemine üstünlüğü açıktır. Bölüt kodunu rakamdan bir formülle türetmenin kısa bir yolu yoktur; on ayrı karşılaştırma gerekirdi. Tablo ise her rakam için aynı sürede, tek komutla sonuç verir. Gösterge değişirse (ör. bölüt uçları porta başka sırayla bağlanırsa) yalnız tablo yeniden yazılır.

> **Sık yapılan hata**
>
> Rakamı ASCII olarak tabloya vermek. Klavyeden okunan '3' karakterinin kodu 33h = 51'dir; `XLAT` tablonun başından 51 bayt ötesini okur ve tablo dışından rastgele bir bayt getirir. ASCII rakam önce `SUB AL, 30h` ile sayıya çevrilir.

10.2.5.5 Çoklamalı tarama

Dört basamaklı bir göstergenin her basamağına ayrı bir 8 bitlik port bağlamak 32 uç gerektirir. **Çoklama** (*multiplexing*) bu sayıyı 12'ye indirir. Dört basamağın aynı adlı bölüt uçları birbirine bağlanır ve ortak port A'ya gelir. Her basamağın ortak ucu ise ayrı bir transistörle port C'nin bir bitine bağ-

lanır. Program her an yalnız bir basamağı yakar: o basamağın kodunu port A'ya yazar, port C'de yalnız o basamağın bitini 1 yapar.

Basamaklar sırayla ve hızla yakılınca göz dördünü birlikte yanıyor görür. Basamak başına yaklaşık 4 ms ayrılırsa tam bir tur 16 ms sürer; tarama sıklığı 1 / 16 ms ≈ 62 Hz olur. İnsan gözü yaklaşık 50 Hz'in üstündeki yanıp sönmeyi fark etmez. Tarama yavaşlarsa (ör. basamak başına 20 ms) göstergede belirgin bir titreme görülür.

```
tara:
mov  si, 0
hane:
mov  dx, 312h       ; port C: hane seçimi
mov  al, 0
out  dx, al         ; önce tüm haneler sönük
mov  al, rakam[si]
mov  bx, offset kod   ; yukarıdaki tablo
xlat
mov  dx, 310h       ; port A: bölütler
out  dx, al
mov  al, secim[si]  ; 01h, 02h, 04h, 08h
mov  dx, 312h
out  dx, al         ; yalnız bu hane yanar
call bekle_4ms
inc  si
cmp  si, 4
jb   hane
jmp  tara
bekle_4ms:          ; ≈ 4 ms gecikme
mov  cx, 1000
loop $
ret
rakam db 1, 2, 3, 4
secim db 01h, 02h, 04h, 08h
```

Döngünün en önemli satırı, hane değiştirilmeden önce bütün haneleri söndüren `OUT DX, AL`'dir (AL = 0, DX = 312h). Bu satır olmasaydı port A'ya yeni kod yazıldığı anda önceki hane hâlâ yanık olurdu. Yeni kod bir anlığına önceki hanede de görünür ve orada soluk bir iz bırakır; buna **gölge** (*ghosting*) denir. Doğru sıra şudur: hepsini söndür, bölüt kodunu yaz, yalnız istenen haneyi yak, bekle.

`secim` tablosu her hane için port C'ye yazılacak baytı tutar. 01h, 02h, 04h ve 08h, sırasıyla PC0, PC1, PC2 ve PC3'ü 1 yapar. Bu değerler 1'i SI kadar sola kaydırarak da üretilebilirdi (`MOV CX, SI` · `MOV AL, 1` · `SHL AL, CL`); tablo hem daha kısa hem daha hızlıdır.

Aşağıdaki kısaltılmış program göstergede 2026 sayısını bir tur tarar. Gecikme yordamı çıkarılmış, sonsuz döngü yerine tek tur ve `RET` konmuştur.

```
org 100h
mov  bx, offset kod
mov  si, 0
hane:
mov  dx, 312h      ; port C (hane)
mov  al, 0
out  dx, al        ; tümü sönük
mov  al, rakam[si]
xlat               ; bölüt kodu
mov  dx, 310h      ; port A (bölüt)
out  dx, al
mov  al, secim[si]
mov  dx, 312h
out  dx, al        ; tek hane yanar
inc  si
cmp  si, 4
jb   hane
ret
kod   db 3Fh,06h,5Bh,4Fh,66h
      db 6Dh,7Dh,07h,7Fh,6Fh
rakam db 2, 0, 2, 6
secim db 01h, 02h, 04h, 08h
```

BX döngünün dışında bir kez yüklenir, çünkü döngüde değişmez. Kod 36 bayttır ve tablolar onun hemen arkasından gelir: `kod` 0124h'de, `rakam` 012Eh'de, `secim` 0132h'dedir. Tablo 10.2.19 izlemenin önemli satırlarını verir; port sütunları o ana kadar porta yazılan son değerdir.

Adım	Komut	SI	AL	Port A	Port C	Açıklama
1	`mov bx, offset kod`	0000h	00h	—	—	BX = 0124h: kod tablosunun başı
2	`mov si, 0`	0000h	00h	—	—	SI = 0: ilk hane
3	`mov dx, 312h`	0000h	00h	—	—	DX = 312h: port C, hane seçimi
4	`mov al, 0`	0000h	00h	—	—	AL = 0
5	`out dx, al`	0000h	00h	—	00h	bütün haneler sönük: gölge oluşmaz
6	`mov al, rakam[si]`	0000h	02h	—	00h	AL ← rakam[0] = 2
7	`xlat`	0000h	5Bh	—	00h	AL ← [BX + 2] = 5Bh
8	`mov dx, 310h`	0000h	5Bh	—	00h	DX = 310h: port A, bölütler
9	`out dx, al`	0000h	5Bh	5Bh	00h	bölüt kodu porta çıkar
10	`mov al, secim[si]`	0000h	01h	5Bh	00h	AL ← secim[0] = 01h
11	`mov dx, 312h`	0000h	01h	5Bh	00h	DX = 312h
12	`out dx, al`	0000h	01h	5Bh	01h	yalnız 1. hane yanar: “2”
13	`inc si`	0001h	01h	5Bh	01h	SI = 1
…	CMP, JB; 2. hane: port C ← 00h, rakam 0					
20	`xlat`	0001h	3Fh	5Bh	00h	XLAT: AL = 3Fh
22	`out dx, al`	0001h	3Fh	3Fh	00h	port A ← 3Fh
25	`out dx, al`	0001h	02h	3Fh	02h	yalnız 2. hane yanar: “0”
…	3. hane: rakam 2					

Adım	Komut	SI	AL	Port A	Port C	Açıklama
33	`xlat`	0002h	5Bh	3Fh	00h	XLAT: AL = 5Bh
38	`out dx, al`	0002h	04h	5Bh	04h	yalnız 3. hane yanar: “2”
…	4. hane: rakam 6					
46	`xlat`	0003h	7Dh	5Bh	00h	XLAT: AL = 7Dh
51	`out dx, al`	0003h	08h	7Dh	08h	yalnız 4. hane yanar: “6”
52	`inc si`	0004h	08h	7Dh	08h	SI = 4
53	`cmp si, 4`	0004h	08h	7Dh	08h	eşit: CF = 0
54	`jb hane`	0004h	08h	7Dh	08h	döngü biter
55	`ret`	0004h	08h	7Dh	08h	gerçek programda tarama baştan başlar

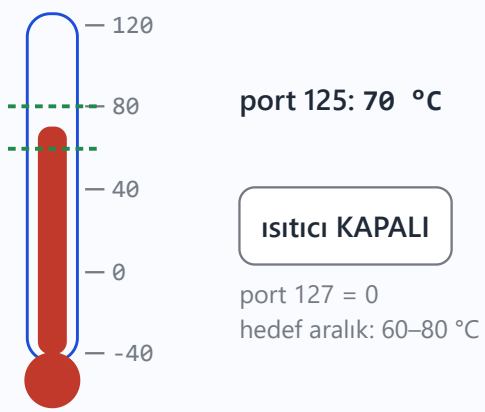
Tablo 10.2.19: Dört haneli göstergede 2026'yı çoklamalı tarayan programın izlemesi (ortak katot kodları).

İzlemede port A'nın değeri, port C 00h iken değişir. Örneğin 20. adımda XLAT 3Fh'yi getirdiğinde port A hâlâ 5Bh'dir, ama port C = 00h olduğu için hiçbir hane yanmaz. Port A 3Fh olduktan sonra (22. adım) port C 02h yapılır (25. adım). Her hane, kendi bölüt kodu porttayken ve yalnız kendisi seçiliyken yanar.

10.2.6 Termometre ve termostat

10.2.6.1 Aygıt

emu8086'nın termometre aygıtı bir odanın sıcaklığını ve odadaki ısıtıcıyı temsil eder. İki port kullanır. `IN AL, 125` anlık hava sıcaklığını santigrat derece cinsinden okur; değer işaretli bir bayttır, yani −128 ile 127 °C arasında olabilir. `OUT 127, AL` ısıtıcıyı yönetir: AL = 1 ısıtıcıyı açar, AL = 0 kapatır. Isıtıcı açıkken sıcaklık yavaş yavaş artar, kapalıyken düşer.



Şekil 10.2.15: Termometre aygıtı: port 125'ten okunan sıcaklık 70 °C, ısıtıcı kapalı (port 127 = 0). Hedef aralık 60–80 °C.

Amaç, sıcaklığı 60–80 °C aralığında tutan bir **termostattır**. Kural basittir: sıcaklık 60'ın altına düşerse ısıtıcı açılır, 80'in üstüne çıkarsa kapatılır, arada kalırsa ısıtıcıya dokunulmaz. Aradaki bu "dokunma" bölgesi bilerek bırakılmıştır. Tek bir eşik (ör. 70 °C) kullanılsaydı sıcaklık eşiğin çevresinde salınırken ısıtıcı saniyede birkaç kez açılıp kapanırdı. İki eşik arasındaki boşluğa **histerezis** denir; ev tipi termostatlar, buzdolapları ve klimalar aynı yöntemle çalışır.

10.2.6.2 Termostat programı

```
; DS'yi CS'ye eşitle:
mov ax, cs
mov ds, ax
start:
in  al, 125
cmp al, 60
jl  low
cmp al, 80
jle ok
jg  high
low:
mov al, 1
out 127, al      ; ısıtıcı açık
jmp ok
high:
mov al, 0
out 127, al      ; ısıtıcı kapalı
ok:
jmp start        ; sonsuz döngü
```

İlk iki satır DS'yi CS'ye eşitler. Kesim yazmacına sabit doğrudan yazılamadığı için değer önce AX'e alınır. Bu programda hiçbir değişken olmadığından satırlar zorunlu değildir; emu8086'nın özgün örneğinden kalmıştır. Bir .COM programında DS zaten CS'ye eşittir.

Döngü `IN AL, 125` ile başlar. Ardından iki karşılaştırma gelir. `CMP AL, 60` AL − 60 çıkarmasını yapar ve yalnız bayrakları kurar. `JL low`, sonuç "küçük" ise, yani SF ≠ OF ise atlar. Atlama olmazsa sıcaklık en az 60'tır. `CMP AL, 80` ve `JLE ok` ikinci eşiği sınar: ZF = 1 ya da SF ≠ OF ise sıcaklık en çok 80'dir ve program ısıtıcıya dokunmadan döngüye döner. İkisi de atlamazsa sıcaklık 80'den büyüktür; `JG high` ısıtıcıyı kapatan koda gider.

`JG high` satırının koşulu aslında hep doğrudur. `JLE` atlamadıysa sıcaklık 80'den büyük olmak zorundadır; `JG` bu yüzden her zaman atlar ve yerine `JMP high` da yazılabilirdi. Satır tümüyle silinemez: hemen altında `low` bloğu durur ve program sıcak odada ısıtıcıyı açardı. `JG`, okuyana koşulu açıkça söylediği için tercih edilmiştir.

Karşılaştırmaların hepsi **işaretli** atlamalarla (`JL`, `JLE`, `JG`) yapılır, çünkü sıcaklık eksi olabilir. İşaretsiz atlamalar (`JB`, `JBE`, `JA`) baytı 0–255 arasında bir sayı sayar. Bu fark eksi sıcaklıklarda programın davranışını tersine çevirir.

Teorem 10.2.1 (İşaretli karşılaştırma)

`CMP a, b` sonrasında a ile b işaretli sayılar olarak karşılaştırılıyorsa a < b ancak ve ancak SF ≠ OF iken doğrudur. a ≤ b, ZF = 1 ya da SF ≠ OF iken doğrudur. İşaretsiz karşılaştırmada a < b, ancak ve ancak CF = 1 iken doğrudur.

10.2.6.3 Kod izleme

İzlemede port 125'in sırayla 55, 70 ve 85 °C döndürdüğü, ısıtıcının başta kapalı olduğu varsayılmıştır. Üç değer, programın üç yolunu birer kez sınar: soğuk, aralıkta ve sıcak.

Adım	Komut	AL	ZF	SF	OF	CF	Isıtıcı	Açıklama
1	`mov ax, cs`	00h	0	0	0	0	kapalı	AX = CS = 0700h
2	`mov ds, ax`	00h	0	0	0	0	kapalı	DS = 0700h; kesim yazmacına sabit doğrudan yazılamaz
3	`in al, 125`	37h	0	0	0	0	kapalı	55 °C = 37h
4	`cmp al, 60`	37h	0	1	0	1	kapalı	55 − 60 = −5: SF = 1, OF = 0
5	`jl low`	37h	0	1	0	1	kapalı	SF ≠ OF: atlar (low)
6	`mov al, 1`	01h	0	1	0	1	kapalı	AL = 1
7	`out 127, al`	01h	0	1	0	1	açık	ısıtıcı açıldı
8	`jmp ok`	01h	0	1	0	1	açık	ok'a atla
9	`jmp start`	01h	0	1	0	1	açık	yeni ölçüm
10	`in al, 125`	46h	0	1	0	1	açık	70 °C = 46h
11	`cmp al, 60`	46h	0	0	0	0	açık	70 − 60 = 10: SF = 0, OF = 0
12	`jl low`	46h	0	0	0	0	açık	SF = OF: atlamaz
13	`cmp al, 80`	46h	0	1	0	1	açık	70 − 80 = −10: SF = 1, ZF = 0
14	`jle ok`	46h	0	1	0	1	açık	ZF = 1 ya da SF ≠ OF: atlar; ısıtıcıya dokunulmaz
15	`jmp start`	46h	0	1	0	1	açık	yeni ölçüm
16	`in al, 125`	55h	0	1	0	1	açık	85 °C = 55h
17	`cmp al, 60`	55h	0	0	0	0	açık	85 − 60 = 25
18	`jl low`	55h	0	0	0	0	açık	atlamaz
19	`cmp al, 80`	55h	0	0	0	0	açık	85 − 80 = 5: SF = 0, ZF = 0
20	`jle ok`	55h	0	0	0	0	açık	atlamaz
21	`jg high`	55h	0	0	0	0	açık	ZF = 0 ve SF = OF: atlar (high)
22	`mov al, 0`	00h	0	0	0	0	açık	AL = 0
23	`out 127, al`	00h	0	0	0	0	kapalı	ısıtıcı kapandı
24	`jmp start`	00h	0	0	0	0	kapalı	yeni ölçüm

Tablo 10.2.20: Termostat programının izlemesi; port 125 sırayla 55, 70 ve 85 °C döndürür, ısıtıcı başta kapalıdır.

Birinci ölçümde 55 − 60 = −5'tir. Sonuç eksi olduğu için SF = 1'dir; işaretli taşma olmadığından OF = 0'dır. SF ≠ OF olduğu için `JL` atlar ve ısıtıcı açılır. CF de 1'dir, çünkü işaretsiz olarak da 55 < 60'tır; bu değerler için işaretli ve işaretsiz yorum aynı sonucu verir.

İkinci ölçümde 70 − 60 = 10 ile SF = OF = 0'dır ve `JL` atlamaz. 70 − 80 = −10 ise SF = 1 yapar; `JLE` atlar ve program ısıtıcıya dokunmadan döner. Isıtıcı bir önceki turda açıldığı için açık kalır; sıcaklık artmaya devam eder. Üçüncü ölçümde 85 iki karşılaştırmayı da geçer: `JG`, ZF = 0 ve SF = OF koşuluyla atlar ve ısıtıcı kapanır.

Sunumun etkileşimli termostatı aynı programı zaman içinde çalıştırır: sıcaklık ısıtıcının durumuna göre yükselip alçalır ve programın hangi satırları yürüttüğü vurgulanır. Sıcaklığın 60 ile 80 arasında gidip geldiği, ısıtıcının ise yalnız eşiklerde durum değiştirdiği görülür.

10.2.6.4 Eksi sıcaklık ve işaretsiz atlama

Sıcaklık −10 °C olsun. Port 125, −10'un ikiye tümleyenini, F6h'yi döndürür. İşaretli program için `CMP AL, 60` −10 − 60 = −70 hesaplar. Sonuç −128'den büyük olduğu için taşma yoktur: SF = 1, OF = 0. `JL` atlar ve ısıtıcı açılır. Doğru davranış budur.

Program işaretsiz atlamalarla yazılmış olsaydı (`JB low`, `JBE ok`, `JA high`) F6h, 246 sayılırdı. 246 < 60 değildir: CF = 0 ve `JB` atlamaz. 246 ≤ 80 de değildir; `JA` atlar ve ısıtıcı *kapatılır*. Oda donarken termostat ısıtıcıyı kapalı tutar. Aynı hata −1 ile −128 arasındaki bütün sıcaklıklarda ortaya çıkar, çünkü bunların hepsi işaretsiz olarak 127'den büyüktür.

> **Sık yapılan hata**
>
> Aygıttan okunan değerin işaretli mi işaretsiz mi olduğuna bakmadan atlama komutu seçmek. Kural verinin anlamından gelir: sıcaklık, ivme, hız farkı gibi eksi olabilen büyüklükler `JL` / `JG` ailesiyle; adres, sayaç, ASCII kodu ve durum baytı gibi hiç eksi olmayan büyüklükler `JB` / `JA` ailesiyle karşılaştırılır.

10.2.7 Yazıcı

10.2.7.1 DOS yazıcı hizmeti ve kontrol karakterleri

emu8086'nın yazıcısı port 130'a bağlıdır, ama program porta doğrudan yazmaz. Bunun yerine DOS'un yazıcı hizmetini kullanır: **INT 21h, AH = 05h**, DL'deki karakteri yazıcıya (PRN aygıtına) gönderir. Hizmet, yazıcının hazır olmasını bekleme işini de kendisi yapar. Gerçek bir PC'de bu hizmet BIOS'un INT 17h yordamını, o da paralel portun yazmaçlarını kullanır. Program bu ayrıntıların hiçbirini bilmek zorunda değildir.

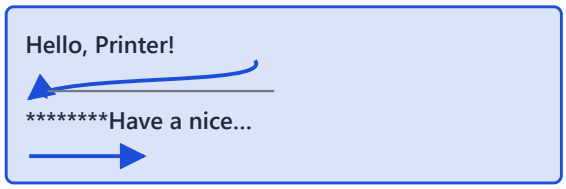
Yazıcıya giden baytların çoğu basılacak karakterlerdir. Bir kısmı ise **kontrol karakteridir**: kâğıda bir şey basmaz, yazıcının kafasını ya da kâğıdı hareket ettirir (Tablo 10.2.21).

Kod	Ad	Etki
0Ah (10)	LF (*line feed*)	bir satır aşağı

Kod	Ad	Etki
0Dh (13)	CR (*carriage return*)	satır başına dön
09h (9)	HT (*horizontal tab*)	sonraki yatay sekmeye (8 sütunda bir)
0Ch (12)	FF (*form feed*)	sayfayı çıkar, yeni sayfaya geç

Tablo 10.2.21: Yazıcıyı yöneten kontrol karakterleri.

Bu karakterlerin adları daktilodan gelir. CR, daktilonun şaryosunu (kâğıdı taşıyan arabayı) satırın başına döndürür; LF kâğıdı bir satır yukarı iter. Daktiloda iki hareket ayrı kollarla yapılırdı; yazıcı da onları ayrı karakterler olarak bekler. Yeni bir satıra geçmek için ikisi de gerekir. Alışılmış sıra CR + LF'dir, ama LF + CR de aynı sonucu verir: kafa önce aşağı, sonra başa gider. Yalnız CR gönderilirse kafa aynı satırın başına döner ve sonraki karakterler eskilerin üzerine basılır. Dikey sekme (VT) 0Bh'dir ve bu programda kullanılmaz.



Şekil 10.2.16: Yazıcı kâğıdında kontrol karakterleri: LF ve CR alt satırın başına götürür; tek başına CR aynı satırın başına döndürür; HT 8. sütuna atlar; FF sayfayı çıkarır.

10.2.7.2 Veri bölümü ve uzunluk hesabı

```
org 100h
jmp start
msg  db "Hello, Printer!", 0Ah, 0Dh
     db "***************"
     db 13, 9           ; CR ve yatay sekme (HT)
     db "Have a nice printing day!"
msg_end db 0
msg2 db "press any key to eject the page.$"
```

Metin birkaç `DB` satırına bölünmüştür, ama bellekte tek bir kesintisiz bayt dizisidir. Yazdırılacak karakter sayısı elle sayılmaz; assembler'a hesaplatılır:

> offset msg_end − offset msg = 15 + 2 + 15 + 2 + 25 = 59 = 3Bh bayt.

“Hello, Printer!” 15, LF ile CR 2, yıldızlar 15, CR ile HT 2, “Have a nice printing day!” 25 karakterdir. `msg` 0102h'de başladığına göre `msg_end` 0102h + 3Bh = 013Dh'dedir. `msg_end` 'deki 0 baytı yazdırılmaz; etiket yalnız son karakterden hemen sonraki adresi işaretlemek için vardır. Metin değişirse uzunluk kendiliğinden doğru hesaplanır.

`msg2` ise '$' ile biter. Bu dizgi yazıcıya değil ekrana gider ve INT 21h/09h ile yazdırılır; 09h hizmeti '$' karakterine kadar yazar. 33 baytlık msg2 013Eh'de başlar ve 015Eh'de biter; kod bu yüzden 015Fh'den başlar ve ilk satırdaki `JMP start` (`EB 5D`) oraya atlar.

10.2.7.3 Yazdırma döngüsü

```
start:
mov dl, 12        ; form feed: yeni sayfa
mov ah, 5
int 21h
mov si, offset msg
mov cx, offset msg_end - offset msg
print:
mov dl, [si]
mov ah, 5         ; DOS yazıcı işlevi
int 21h
inc si            ; sonraki karakter
loop print
mov dx, offset msg2
mov ah, 9
int 21h           ; ekrana mesaj
mov ax, 0
int 16h           ; tuş bekle
mov dl, 12        ; sayfayı çıkar
mov ah, 5
int 21h
ret
```

Program temiz bir sayfayla başlamak için önce bir form feed gönderir. Ardından SI dizginin başını, CX karakter sayısını alır. Döngü gövdesi bir karakteri DL'ye okur, AH'ye 05h yazar ve hizmeti çağırır. `MOV AH, 5` her turda yinelenir. INT 21h hizmetleri AH'yi değiştirebileceği için güvenli yol budur. `INC SI` sonraki karaktere geçer; `LOOP print` CX'i bir azaltır ve sıfır değilse başa döner. Döngü tam 59 kez çalışır.

Döngüden sonra ekrana “press any key to eject the page.” yazılır ve INT 16h/00h ile bir tuş beklenir. Kullanıcı tuşa basınca son bir form feed sayfayı yazıcıdan çıkarır. Program `RET` ile biter.

Adım	Komut	CX	DL	SI	Yazıcıya	Açıklama
1	`jmp start`	0000h	00h	0000h		veri baytlarının üzerinden 015Fh'ye atlanır
2	`mov dl, 12`	0000h	0Ch	0000h		DL = 12 = 0Ch: sayfa ilerletme
3	`mov ah, 5`	0000h	0Ch	0000h		AH = 05h: DOS yazıcı çıkışı
4	`int 21h`	0000h	0Ch	0000h	FF	yazıcı yeni sayfaya geçer
5	`mov si, offset msg`	0000h	0Ch	0102h		SI = 0102h: msg
6	`mov cx, offset msg_end - offset msg`	003Bh	0Ch	0102h		013Dh − 0102h = 3Bh = 59 karakter

Adım	Komut	CX	DL	SI	Yazıcıya	Açıklama
7	`mov dl, [si]`	003Bh	48h	0102h		DL ← [0102h] = 'H'
8	`mov ah, 5`	003Bh	48h	0102h		AH = 05h
9	`int 21h`	003Bh	48h	0102h	'H'	'H' yazıcıya gider
10	`inc si`	003Bh	48h	0103h		SI = 0103h
11	`loop print`	003Ah	48h	0103h		CX = 58 ≠ 0: print'e döner
...	karakter karakter devam					
84	`int 21h`	002Ch	0Ah	0111h	LF	16. bayt: LF, kâğıt bir satır ilerler
89	`int 21h`	002Bh	0Dh	0112h	CR	17. bayt: CR, satır başı
...	15 yıldız					
169	`int 21h`	001Bh	0Dh	0122h	CR	33. bayt: CR; LF yok, aynı satırın başına dönülür
174	`int 21h`	001Ah	09h	0123h	HT	34. bayt: HT, sekiz sütunluk sekmeye atlanır
...	“Have a nice printing day” yıldızların üzerine basılır					
297	`mov dl, [si]`	0001h	21h	013Ch		DL ← [013Ch] = '!'
299	`int 21h`	0001h	21h	013Ch	'!'	son karakter
300	`inc si`	0001h	21h	013Dh		SI = 013Dh = msg_end
301	`loop print`	0000h	21h	013Dh		CX = 0: döngü biter
302	`mov dx, offset msg2`	0000h	3Eh	013Dh		DX = msg2'nin adresi (013Eh)
304	`int 21h`	0000h	3Eh	013Dh		ekrana mesaj ($'a kadar)
306	`int 16h`	0000h	3Eh	013Dh		bir tuş beklenir
309	`int 21h`	0000h	0Ch	013Dh	FF	son sayfa ilerletme: kâğıt çıkar
310	`ret`	0000h	0Ch	013Dh		program biter

Tablo 10.2.22: Yazıcı programının izlemesi (seçilmiş adımlar; “Yazıcıya” sütunu o adımda yazıcıya giden karakterdir).

İzlemenin “Yazıcıya” sütunu her `INT 21h`'de yazıcıya giden baytı gösterir. 84. adımdaki LF ve 89. adımdaki CR kâğıdı ikinci satırın başına getirir. 169. adımdaki CR'den sonra LF yoktur: kafa aynı satırın, yıldızların başına döner. 174. adımdaki HT kafayı 8. sütuna (0'dan sayılınca) taşır. “Have a nice printing day!” bu yüzden ilk sekiz yıldızın arkasından başlar ve kalan yedi yıldızın üzerine basılır.

10.2.7.4 Kâğıtta ne görünür?

Program bittiğinde sayfada iki satır vardır:

```
Hello, Printer!
********Have a nice printing day!
```

İkinci satırda yıldızların ilk sekizi görünür; 9.–15. yıldızlar, “Have a ” sözcüklerinin harfleriyle aynı yere basılmıştır. Gerçek bir nokta vuruşlu yazıcıda bu yerlerde iki karakter üst üste görünür; emu8086'nın kâğıt görüntüsü ise üstteki karakteri gösterir. Satır başı ve sekme karakterlerinin bu birleşimi, daktilo döneminde altı çizili ya da kalın yazı elde etmek için bilerek kullanılırdı: aynı satır iki kez basılırdı.

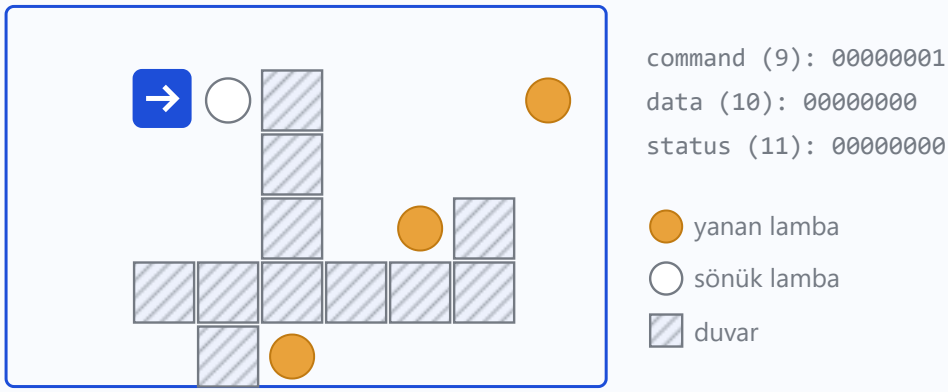
Sık yapılan hata

Karakter sayısını elle yazmak: `MOV CX, 59` . Metne tek bir harf eklenince sayı yanlış olur; ya son karakter basılmaz ya da `msg_end` 'in ötesindeki baytlar (burada msg2'nin başı) yazıcıya gider. `offset msg_end - offset msg` farkı derleme anında hesaplanır ve metinle birlikte kendiliğinden güncellenir.

10.2.8 Robot

10.2.8.1 Portlar, komutlar ve durum bitleri

emu8086'nın robotu bir ızgara dünyasında hareket eder. Dünyada duvarlar ve lambalar vardır. Robot önündeki hücreyi inceleyebilir, ileri gidebilir, sola ya da sağa dönebilir, önündeki lambayı yakıp söndürebilir. Dünya, aygıt penceresindeki araç kutusuyla düzenlenir: duvar ve lamba eklenir, robotun yeri değiştirilir.



Şekil 10.2.17: Robot dünyası: duvarlar (taralı), yanan ve sönük lambalar, robot ve üç portun o anki baytları.

Robot üç port kullanır ve bu üçlü, gerçek aygıtlarda da sık görülen bir düzendir. Port 9 **komut** portudur: program robota ne yapacağını buraya yazar (OUT). Port 10 **veri** portudur: inceleme komutunun sonucu buradan okunur (IN). Port 11 **durum** portudur: robotun meşgul olup olmadığı, yeni veri bulunup bulunmadığı ve son komutun hata verip vermediği buradan okunur (IN).

Komut	Etki	Veri	Önündeki
0	hiçbir şey yapma	255 (FFh)	duvar
1	ileri git	0	boş
2	sola dön	7	yanan lamba
3	sağa dön	8	sönük lamba
4	önündekini incele		
5	lambayı yak		
6	lambayı söndür		

Tablo 10.2.23: Robotun komutları (port 9) ve inceleme sonuçları (port 10).

Bit	1 ise
bit 0	veri yazmacında yeni veri var
bit 1	robot meşgul (komut sürüyor)
bit 2	son komutta hata

Tablo 10.2.24: Robotun durum baytı (port 11).

Durum baytının her biti ayrı bir bilgi taşır ve aynı anda birden fazlası 1 olabilir. 03h = 0000 0011b, robotun hem meşgul olduğunu hem de veri yazmacında okunmamış bir veri bulunduğunu söyler. 05h = 0000 0101b ise yeni veri ve hata bitlerinin birlikte kurulduğunu gösterir. Program bu yüzden durumu bir sayıyla karşılaştırmaz (`CMP AL, 2`), yalnız ilgilendiği biti sınar (`TEST AL, 010b`).

10.2.8.2 Lambayı bul ve yak

Aşağıdaki program robotu dünyada dolaştırır ve rastladığı sönük lambaları yakar. Her turda önündekini inceler. Duvar varsa sağa döner, sönük lamba varsa yakar, başka bir şey varsa ileri gider.

```
#start=robot.exe#
dongu:  CALL hazir_bekle
        MOV  AL, 4         ; incele
        OUT  9, AL
        CALL veri_bekle
        IN   AL, 10
        CMP  AL, 0FFh      ; duvar?
        JE   don
        CMP  AL, 8         ; sönük lamba?
        JNE  ilerle
        CALL hazir_bekle
        MOV  AL, 5         ; yak
        OUT  9, AL
        JMP  dongu
don:    MOV  AL, 3         ; sağa dön
        JMP  gonder
ilerle: MOV  AL, 1         ; ileri
gonder: PUSH AX            ; komutu koru
        CALL hazir_bekle
        POP  AX
        OUT  9, AL
        JMP  dongu

hazir_bekle PROC
b1:     IN   AL, 11
        TEST AL, 010b      ; meşgul?
        JNZ  b1
        RET
hazir_bekle ENDP

veri_bekle PROC
v1:     IN   AL, 11
        TEST AL, 001b      ; yeni veri?
        JZ   v1
        RET
veri_bekle ENDP
```

Program iki bekleme yordamına dayanır. `hazir_bekle` durum baytının bit 1'ini sınar: meşgul biti 1 olduğu sürece (ZF = 0) `JNZ` geri döner. Robot önceki komutu bitirince yordam döner. `veri_bekle` bit 0'ı sınar ve ters koşulla döner: yeni veri biti 0 olduğu sürece (ZF = 1) `JZ` bekler. İki yordamın yapısı aynıdır; yalnız maske ve atlama koşulu farklıdır.

Her komuttan önce meşgul bitinin sıfırlanması beklenir. İnceleme (4) komutundan sonra ise veri hazır biti beklenir; ancak ondan sonra port 10 okunur. Okunan veri FFh ise `JE don` sağa dönme koduna gider. 8 ise `JNE` atlamaz ve lamba yakılır. Başka bir değerde `JNE ilerle` ileri gitme koduna gider. “Sağa dön” ve “ileri git” yolları `gonder` etiketinde birleşir: AL'deki komut porta yazılır ve döngü baştan başlar.

Bekleme işleri `CALL` ile çağrılan yordamlardır. Her çağrıda dönüş adresi yığına itilir (SP 2 azalır), `RET` onu geri alır. Aynı bekleme döngüsünün programa üç kez yazılması yerine tek bir yordamın üç yerden çağrılması kodu kısaltır ve hata olasılığını azaltır.

10.2.8.3 AL'nin korunması

`gonder` yolunda bir incelik vardır. AL robotun komutunu (1 ya da 3) taşır. Ama komut gönderilmeden önce `hazir_bekle` çağrılır ve o yordam `IN AL, 11` ile AL'yi durum baytıyla ezer. Yordam döndüğünde AL artık komut değil, son okunan durumdur. Bu yüzden komut çağrıdan önce `PUSH AX` ile yığına konur ve çağrıdan sonra `POP AX` ile geri alınır. `PUSH` 8 bitlik bir yazmaç kabul etmediği için AL yerine AX itilir.

Tablo 10.2.25 robotun önünde duvar varken bir turu izler. Port 11'in sırayla 00h (hazır), 01h (veri var), 03h (meşgul ve veri var) ve 01h döndürdüğü, port 10'un FFh (duvar) döndürdüğü varsayılmıştır.

Adım	Komut	AL	SP	ZF	Port 9	Açıklama
1	`dongu: CALL hazir_bekle`	00h	FFFCh	0	—	dönüş adresi 0103h yığına
2	`b1: IN AL, 11`	00h	FFFCh	0	—	durum = 00h
3	`TEST AL, 010b`	00h	FFFCh	1	—	bit 1 = 0: meşgul değil, ZF = 1
4	`JNZ b1`	00h	FFFCh	1	—	atlamaz
5	`RET`	00h	FFFEh	1	—	dönüş: SP = FFFEh
6	`MOV AL, 4`	04h	FFFEh	1	—	komut 4: incele
7	`OUT 9, AL`	04h	FFFEh	1	04h	robot önündekini inceler
8	`CALL veri_bekle`	04h	FFFCh	1	04h	veri beklenir
9	`v1: IN AL, 11`	01h	FFFCh	1	04h	durum = 01h: yeni veri var
10	`TEST AL, 001b`	01h	FFFCh	0	04h	bit 0 = 1: ZF = 0
11	`JZ v1`	01h	FFFCh	0	04h	atlamaz
12	`RET`	01h	FFFEh	0	04h	dönüş
13	`IN AL, 10`	FFh	FFFEh	0	04h	veri = FFh: duvar
14	`CMP AL, 0FFh`	FFh	FFFEh	1	04h	eşit: ZF = 1
15	`JE don`	FFh	FFFEh	1	04h	don'a atlar
16	`don: MOV AL, 3`	03h	FFFEh	1	04h	komut 3: sağa dön
17	`JMP gonder`	03h	FFFEh	1	04h	gonder
18	`gonder: PUSH AX`	03h	FFFCh	1	04h	AX = 0003h yığına: SP = FFFCh
19	`CALL hazir_bekle`	03h	FFFAh	1	04h	dönüş adresi yığına: SP = FFFAh
20	`b1: IN AL, 11`	03h	FFFAh	1	04h	durum = 03h: AL ezildi
21	`TEST AL, 010b`	03h	FFFAh	0	04h	bit 1 = 1: meşgul, ZF = 0
22	`JNZ b1`	03h	FFFAh	0	04h	yeniden oku
23	`b1: IN AL, 11`	01h	FFFAh	0	04h	durum = 01h
24	`TEST AL, 010b`	01h	FFFAh	1	04h	bit 1 = 0: ZF = 1
25	`JNZ b1`	01h	FFFAh	1	04h	atlamaz
26	`RET`	01h	FFFCh	1	04h	dönüş: SP = FFFCh, AL = 01h
27	`POP AX`	03h	FFFEh	1	04h	AL = 03h geri geldi
28	`OUT 9, AL`	03h	FFFEh	1	03h	robota “sağa dön” gider

Adım	Komut	AL	SP	ZF	Port 9	Açıklama
29	`JMP dongu`	03h	FFFEh	1	03h	döngü yeniden başlar

Tablo 10.2.25: Robotun önünde duvar varken programın izlemesi; port 11 sırayla 00h, 01h, 03h, 01h, port 10 FFh döndürür.

18. adımda AX = 0003h yığına itilir ve SP FFFCh'ye iner. 19. adımdaki `CALL` dönüş adresini de iter: SP = FFFAh. Yordamın içinde AL önce 03h (meşgul), sonra 01h olur. 26. adımdaki `RET` dönüş adresini alır; AL'de 01h vardır. 27. adımdaki `POP AX` komutu 03h'yi geri getirir ve 28. adımda porta doğru değer, 3, gider. Yığın da dengelidir: SP başladığı yere, FFFEh'ye döner.

`PUSH AX` ve `POP AX` silinseydi aynı senaryoda porta 01h giderdi: "ileri git". Robot duvara doğru yürümeye çalışır ve hata biti (bit 2) kurulur. Program bu hatayı denetlemediği için bir sonraki turda yeniden inceler, yeniden duvarı görür ve aynı hatayı yineler: robot duvarın önünde takılır kalır.

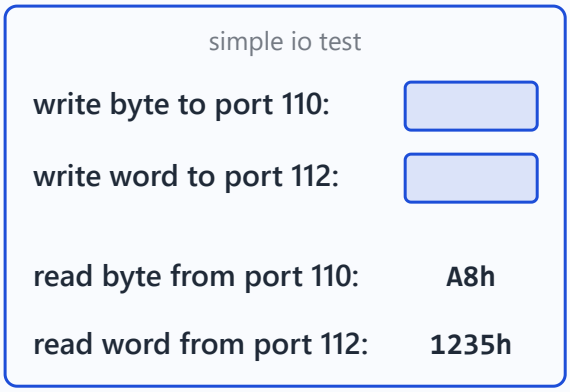
> **Sık yapılan hata**
>
> Bir yordam çağrısından sonra, yordamın kullandığı bir yazmacın eski değerine güvenmek. `hazir_bekle` AL'yi değiştirir; bunu yordamın adından anlamak mümkün değildir. Ya yordam değiştirdiği yazmaçları kendisi saklar (girişte `PUSH`, çıkışta `POP`), ya da çağıran taraf korumak istediği değeri yığına atar. Robot programı ikinci yolu kullanır.

Programın iki eksiği daha vardır. Birincisi, hata biti hiç okunmaz. Duvara doğru "ileri" komutu bit 2'yi kurar; sağlam bir program her komuttan sonra durum baytını okuyup bu biti sınamalıdır. İkincisi, yoklamanın bedelidir: işlemci robotun her hareketini beklerken tümüyle meşguldür. Gerçek bir robot denetleyicisinde bu bekleme kesmeyle yapılır ve işlemci arada algılayıcı verilerini işler.

10.2.9 Basit G/Ç test aygıtı

10.2.9.1 Port 110 ve 112

Test aygıtı (*simple io test*), `IN` ve `OUT` komutlarını denemek için yapılmış bir penceredir. Port 110 bir bayt, port 112 bir sözcük tutar. Pencereye yazılan değerler programa giriş olur; onaltılık değer için sonuna `h` eklenir (ör. `A7h`). Programın porta yazdığı değerler de pencerede görünür. Giriş portları her 100 ms'de bir güncellenir. Bu yüzden pencereye yazılan yeni değer, program çalışırken bile en geç 100 ms içinde okunur.



Şekil 10.2.18: Test aygıtı penceresi. Program port 110'a A8h, port 112'ye 1235h yazmıştır; pencere bu değerleri gösterir.

Pencerenin bir hata ayıklama aracı olarak değeri büyüktür. Bir aygıt programı yazılırken porta giden değerin doğru olup olmadığı burada görülebilir; giriş değerleri de aygıt olmadan elle verilebilir. Sunumdaki örnekte pencereye port 110 için A7h, port 112 için 1234h girilir. Program birincisine 1 ekleyip A8h'yi, ikincisine 1 ekleyip 1235h'yi geri yazar.

10.2.9.2 Kod izleme: bayt ve sözcük G/Ç

```
org 100h
in  al, 110      ; bayt oku
inc al
out 110, al      ; bayt yaz
in  ax, 112      ; sözcük oku
add ax, 1
out 112, ax      ; sözcük yaz
ret
```

Aynı program sınır değerlerle çalıştırılınca bayrakların davranışı ortaya çıkar. Pencereye port 110 için FFh, port 112 için 7FFFh girilmiş olsun.

Adım	Komut	AX	ZF	SF	OF	CF	G/Ç	Açıklama
1	`in al, 110`	00FFh	0	0	0	0	110 → FFh	bayt okuma: AL = FFh
2	`inc al`	0000h	1	0	0	0		FFh + 1 = 00h: ZF = 1; INC CF'ye dokunmaz
3	`out 110, al`	0000h	1	0	0	0	110 ← 00h	pencerede port 110 = 00h
4	`in ax, 112`	7FFFh	1	0	0	0	112 → 7FFFh	sözcük okuma: AL ← port 112, AH ← port 113
5	`add ax, 1`	8000h	0	1	1	0		7FFFh + 1 = 8000h: iki pozitifin toplamı negatif, OF = 1
6	`out 112, ax`	8000h	0	1	1	0	112 ← 8000h	pencerede port 112 = 8000h
7	`ret`	8000h	0	1	1	0		yığından 0000h → IP

Adım	Komut	AX	ZF	SF	OF	CF	G/Ç	Açıklama
8	`INT 20h (PSP:0000)`	8000h	0	1	1	0		program biter

Tablo 10.2.26: Test aygıtı programının izlemesi; pencerede port 110 = FFh, port 112 = 7FFFh girilmiştir.

İlk kısım 8 bitlik bir taşmadır. FFh + 1 = 100h sonucunun 9. biti AL'ye sığmaz ve AL = 00h olur. ZF = 1'dir. `INC` komutu CF'yi değiştirmez; CF önceki değerinde, 0'da kalır. Aynı işlem `ADD AL, 1` ile yapılsaydı CF = 1 olurdu. Porta 00h yazılır.

İkinci kısım işaretli taşmadır. `IN AX, 112` AL'yi port 112'den, AH'yi port 113'ten alır: AX = 7FFFh, işaretli olarak +32.767. `ADD AX, 1` sonucu 8000h'dir. İki pozitif sayının toplamı negatif bir sayı olarak görünmüştür: OF = 1, SF = 1. İşaretsiz toplamda taşma yoktur (32.767 + 1 = 32.768 < 65.536), bu yüzden CF = 0'dır. Pencerede port 112 = 8000h görünür. Aynı sözcük LED ekrana yazılsaydı −32.768 gösterilirdi.

10.2.10 Ortak desen

Yedi aygıt birbirinden çok farklı görünür: biri lamba yakar, biri motor döndürür, biri sıcaklık okur. Programları ise aynı iskeletin çeşitlemeleridir (Tablo 10.2.27).

Aygıt	Port	Temel fikir
Trafik lambaları	4	Her bit bir lamba; durum tablosu ve INT 15h/86h bekleme.
Step motor	7	Bit 0–2 mıknatıs, bit 7 hazır; yarım adım 11,25°, tam adım 22,5°.
LED ekran	199	Sözcük, işaretli onluk gösterim.
Termometre	125, 127	Okuma, işaretli karşılaştırma, ısıtıcıyı aç/kapa.
Yazıcı	130	INT 21h/05h; kontrol karakterleri LF, CR, HT, FF.
Robot	9, 10, 11	Komut, veri, durum; meşgul ve veri bitlerini yokla.
Test aygıtı	110, 112	Bayt ve sözcük G/Ç denemesi.
8255 PPI (gerçek)	310h–313h	Kontrol sözcüğü (ör. 82h) port yönlerini belirler; port > 255 → DX.

Tablo 10.2.27: Aygıtlar, portları ve temel fikirleri.

Ortak desen dört adımdır: durumu oku, ilgili biti sına, hazırsa veriyi yaz ya da oku, döngüye dön. Trafik ışıklarında “durum” zamandır ve sınama yerine bekleme yapılır. Termostatta durum, okunan sıcaklığın kendisidir. Step motor ve robotta durum ayrı bir porttaki bitlerdir. Her durumda `IN` ve `OUT` yalnız AL/AX ile çalışır, port 255'ten büyükse DX kullanılır. Aygıtların çoğu sonsuz bir döngüde sürülür.

10.2.11 Alıştırmalar ve çözümleri

Alıştırma 1. Bütün yönlerde yalnız sarı lambaları yakan port 4 değeri nedir?

Sarı lambalar bit 3k + 1'dedir: bit 1, 4, 7 ve 10. Sözcük 2 + 16 + 128 + 1024 = 1170 = 0000 0100 1001 0010b = **0492h**'dir.

Alıştırma 2. INT 15h/86h ile 2,5 saniyelik bekleme için CX ve DX ne olmalıdır?

2,5 s = 2.500.000 µs = 002625A0h. Yüksek sözcük CX = **0026h**, düşük sözcük DX = **25A0h**'dir. Kaynak koda `mov cx, 26h` ve `mov dx, 25A0h` yazılır.

Alıştırma 3. LED ekranda −1 göstermek için AX'e hangi onaltılık değer yazılır?

−1'in 16 bitlik ikiye tümleyeni 10000h − 1 = **FFFFh**'dir. `mov ax, -1` yazıldığında assembler aynı değeri üretir.

Alıştırma 4. Termostat 20–25 °C aralığına nasıl uyarlanır? Sıcaklık −10 °C iken `JL` yerine `JB` kullanılsaydı ne olurdu?

Yalnız iki sabit değişir: `cmp al, 20` · `jl low` · `cmp al, 25` · `jle ok` · `jg high`. −10 °C porttan F6h olarak gelir. İşaretli karşılaştırmada −10 < 20'dir ve ısıtıcı açılır. `JB` ise F6h'yi 246 sayar; 246 < 20 olmadığı için atlamaz ve ısıtıcı açılmaz. İşaretsiz `JA` da kullanılmışsa ısıtıcı kapatılır.

Alıştırma 5. Step motor programında motoru yalnız saat yönünde ve tam adımla döndürmek için hangi satırlar değişmelidir?

BX başta tam adım tablosunu göstermelidir: `mov bx, offset datcw_fs`. Tablo değiştiren kısım (`inc cx`'ten son `jmp next_step`'e kadar) silinir ve `mov si, 0`'dan sonra doğrudan `jmp next_step` yazılır. CX artık kullanılmadığı için `mov cx, 0` satırı da gereksizdir.

Alıştırma 6. Yazıcıya “Merhaba” yazıp sayfayı çıkaran en kısa program nasıl yazılır?

Her karakter için `MOV DL, 'M'` · `MOV AH, 5` · `INT 21h` yazılabilir; dizgi ve döngüyle daha kısa olur:

```
org 100h
mov  si, offset m
mov  cx, 8          ; "Merhaba" + FF
yaz: mov dl, [si]
mov  ah, 5
int  21h
inc  si
loop yaz
ret
m    db "Merhaba", 12
```

Dizginin sonuna eklenen 12 (FF) sayfayı çıkarır; böylece ayrı bir form feed çağrısına gerek kalmaz.

Alıştırma 7. 8 MHz'lik bir 8086'da `mov cx, N` / `loop` ile yaklaşık 1 ms gecikme için N kaçtır?

1 ms, 8.000 saat turudur. 17N − 8 = 8.000 denkleminden N ≈ 471,06 çıkar. N = **471** ile 17 · 471 − 8 = 7.999 tur, yani 0,999875 ms elde edilir.

Alıştırma 8. 8255'te A giriş, B çıkış, C'nin tamamı giriş (mod 0) için kontrol sözcüğü nedir?

D7 = 1, D6 D5 = 00, D4 = 1 (A giriş), D3 = 1 (C üst giriş), D2 = 0, D1 = 0 (B çıkış), D0 = 1 (C alt giriş): 1 00 1 1 0 0 1b = **99h**.

Alıştırma 9. Ortak anotlu göstergede '5' rakamının ve 'H' harfinin kodları nedir (dp g f e d c b a sırası)?

'5' için ortak katot kodu 6Dh = 0110 1101b'dir; tersi 1001 0010b = **92h**. 'H' için b, c, e, f ve g yanar: 0111 0110b = 76h; ortak anot kodu 1000 1001b = **89h**'dir.

10.2.12 Özet

- Çevre birimi işlemciye yazmaçlarıyla görünür; bu yazmaçların adresine G/Ç portu denir. Port numaralarını işlemci değil, sistemin adres çözücüsü belirler.
- 8086'da porta yalnız `IN` ve `OUT` ile erişilir. Veri yalnız AL (bayt) ya da AX (sözcük) üzerinden geçer. Port 0–255 ise komuta sabit olarak, büyükse DX ile verilir. Sözcük biçimi port n ve n + 1'i birlikte kullanır.
- Port eşlemeli G/Ç ayrı bir 64 K'lık adres uzayı kullanır (M/$\overline{IO}$ = 0); bellek eşlemeli G/Ç'de aygıt bellek adreslerindedir ve her bellek komutuyla erişilir (ör. B800h metin ekranı).
- Aygıtla eşlemenin üç yolu vardır: yoklama (işlemci durum bitini döngüde okur), kesme (aygıt işlemciyi uyarır) ve DMA (denetleyici veriyi işlemcisiz taşır).
- 8255 PPI'nin üç 8 bitlik portu ve bir kontrol yazmacı vardır. D7 = 1 olan kontrol sözcüğü portların mod ve yönünü (1 = giriş) belirler; D7 = 0 olan bit kur/sil sözcüğü C'nin tek bir bitini değiştirir.
- Trafik ışıklarında her lamba port 4'teki sözcüğün bir bitidir (direk k: kırmızı 3k, sarı 3k + 1, yeşil 3k + 2). Durumlar bir tabloda tutulur ve SI ile sırayla porta yazılır.
- INT 15h/86h, CX:DX mikrosaniye bekler. `MOV CX, N` / `LOOP` gecikmesi 17N − 8 saat turudur ve saat hızına bağlıdır; 4,77 MHz'de 1 ms için N = 281.
- emu8086 step motorunda bit 0–2 mıknatısları, bit 7 hazır bilgisini taşır. Program her bayttan önce bit 7'yi yoklar ve tabloyu `[BX][SI]` ile okur.
- Gerçek unipolar motor ULN2003 gibi bir sürücüyle sürülür; tek faz, iki faz ve yarım adım dizileri vardır. Motor açık çevrimdir ve hızı adım aralığıyla belirlenir.
- LED ekran port 199'daki sözcüğü işaretli onluk gösterir. 7 bölütlü göstergede kod tablodan `XLAT` ile alınır; ortak anotta kod tersine çevrilir. Çoklamalı taramada hane değiştirmeden önce bütün haneler söndürülür.
- Termostat eksi sıcaklık okuyabildiği için işaretli atlamalar (`JL`, `JLE`, `JG`) kullanır; işaretsiz atlamalar eksi değerleri büyük pozitif sayı sanır.
- Yazıcıya INT 21h/05h ile karakter gönderilir; LF, CR, HT ve FF kâğıdı ve kafayı yönetir. Uzunluk `offset msg_end - offset msg` ile derleme anında hesaplanır.

- Robot komut, veri ve durum portlarıyla çalışır; bekleme yordamları AL'yi ezdiği için komut çağrıdan önce yığında korunur.

10.2.13 Kavram listesi

çevre birimi *peripheral* · G/Ç portu *I/O port* · port eşlemeli / bellek eşlemeli G/Ç *port-mapped / memory-mapped I/O* · adres çözücü *address decoder* · yoklama *polling* · el sıkışma *handshaking* · kesme *interrupt* · doğrudan bellek erişimi (DMA) *direct memory access* · programlanabilir çevre arayüzü (8255 PPI) *programmable peripheral interface* · kontrol sözcüğü *control word* · yazılımla gecikme *software delay* · saat turu *clock cycle* · step motor *stepper motor* · 7 bölütlü gösterge *seven-segment display* · bölüt *segment (display)* · kontrol karakteri *control character* · öykünücü *emulator*

10.2.14 Kendini Dene

1. Sağ alt ve sol üst direklerde kırmızı + sarı, öteki iki direkte yalnız kırmızı yakan port 4 sözcüğü nedir?

   Cevap

   Sağ alt: bit 0 ve 1; sağ üst: bit 3; sol üst: bit 6 ve 7; sol alt: bit 9. Toplam 1 + 2 + 8 + 64 + 128 + 512 = 715 = 0000 0010 1100 1011b = **02CBh**.

2. INT 15h/86h ile 3 saniye beklemek için CX ve DX ne olmalıdır?

   Cevap

   3.000.000 µs = 002DC6C0h: CX = **002Dh**, DX = **C6C0h**. Kaynakta DX değeri harfle başladığı için `0C6C0h` diye yazılır.

3. `IN AL, 300h` neden derlenmez? Aynı iş nasıl yazılır?

   Cevap

   Sabit portlu biçimde port numarası tek bir bayta yazılır ve 300h bu bayta sığmaz. Doğrusu `MOV DX, 300h` · `IN AL, DX`'tir.

4. 8255'in kontrol yazmacına 9Bh yazılırsa portlar ne olur? PC4'ü 1 yapan bit kur/sil sözcüğü nedir?

Cevap

9Bh = 1 00 1 1 0 1 1b: mod 0, A giriş, C üst giriş, B giriş, C alt giriş; bütün portlar giriştir. PC4 = 1 için D3–D1 = 100, D0 = 1: 0000 1001b = **09h**.

5. 4,77 MHz'lik 8086'da `mov cx, 500` / `bekle: loop bekle` ne kadar sürer?

Cevap

17 · 500 − 8 = 8.492 saat turu; 8.492 ÷ 4,77 ≈ 1.780 µs ≈ **1,78 ms**.

6. 'E' harfinin ortak katot ve ortak anot bölüt kodları nedir?

Cevap

'E' için a, d, e, f ve g yanar: bit 0, 3, 4, 5, 6. Ortak katot 0111 1001b = **79h**, ortak anot 1000 0110b = **86h**.

7. Termostat programında sıcaklık tam 80 °C okunursa hangi atlama gerçekleşir, ısıtıcıya ne olur?

Cevap

`CMP AL, 60` sonucu pozitiftir ve `JL` atlamaz. `CMP AL, 80` sonucu sıfırdır: ZF = 1 ve `JLE ok` atlar. Isıtıcıya dokunulmaz; önceki durumunda kalır.

8. Robotun durum portu 06h döndürüyor. `hazir_bekle` ve `veri_bekle` yordamları ne yapar?

Cevap

06h = 0000 0110b: bit 1 (meşgul) ve bit 2 (hata) 1, bit 0 (veri) 0. `hazir_bekle` meşgul biti 1 olduğu için beklemeyi sürdürür; `veri_bekle` de veri biti 0 olduğu için bekler. İki yordam da hata bitine bakmaz.

9. Test aygıtında port 110'a 7Fh girilmişken `IN AL, 110` · `INC AL` çalışıyor. AL ve OF, SF, ZF ne olur?

Cevap

AL = **80h**. +127'den bir fazlası 8 bitte −128 olarak görünür: OF = 1, SF = 1, ZF = 0. CF'ye `INC` dokunmaz.

10. Bellek eşlemeli bir aygıtın durum yazmacı B800:0010'da olsaydı, bit 7'yi beklemek için hangi ko-

mutlar yeterdi?

Cevap

ES = B800h kurulduktan sonra `bekle: TEST BYTE PTR ES:[0010h], 80h` · `JZ bekle`. Bellek eşlemeli aygıtta `TEST` doğrudan bellek işlenenine uygulanır; AL'ye okumak gerekmez.

10.2.15 İleri okuma

- Barry B. Brey, *The Intel Microprocessors*, 8. baskı: temel G/Ç arayüzü bölümü IN/OUT, adres çözme, 8255 ve 7 bölütlü gösterge sürmeyi devre şemalarıyla anlatır.
- Intel, *8255A Programmable Peripheral Interface* veri sayfası: mod 0, 1 ve 2'nin zamanlama diyagramları ve kontrol sözcüğünün özgün tanımı.
- Texas Instruments, *ULN2003A* veri sayfası: Darlington kanallarının akım sınırları ve endüktif yüklerde COM ucunun bağlanışı.
- emu8086 yardım belgeleri, *I/O ports and hardware interrupts* ve *virtual devices*: sanal aygıtların port düzeni ve örnek programları.

10.2.16 Kaynaklar

1. emu8086 yardım belgeleri: *I/O ports and hardware interrupts*, *Virtual devices* ve örnek programlar (traffic_lights.asm, stepper_motor.asm, led_display_test.asm, thermometer.asm, printer.asm, robot.asm).
2. Intel, *The 8086 Family User's Manual*, 1979 (IN/OUT komutları, G/Ç adres uzayı ve komut süreleri).
3. Intel, *8255A Programmable Peripheral Interface* veri sayfası.
4. Texas Instruments, *ULN2003A* veri sayfası.
5. M. Bodur, *Mikroişlemcilere Giriş: Assembler ile Yazılım ve Arayüz*, EMO, 2014.
6. R. Brown, *Ralf Brown's Interrupt List* (INT 15h/86h, INT 21h/05h, INT 16h/00h).
7. B. B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson, 2009, Bölüm 11 (Temel G/Ç arayüzü).

KISIM IV · BÖLÜM 11

Komut Kodlama ve Dizgi Komutları

Assembly dilinde yazılan her satır, sonunda birkaç bayta dönüşür. İşlemci yalnız bu baytları görür: bellekten okur, önden getirme kuyruğuna koyar, çözer ve yürütür. Bu not önce 8086'nın komut kümesini işlevsel sınıflara ayırır. Ardından bir komutun makine koduna nasıl dönüştüğünü bit bit gösterir: önek, işlem kodu, D ve W bitleri, ModR/M baytı, yer değiştirme ve anlık değer. Aynı kuralların tersi, yani bellekteki baytlardan komutu geri çıkarmak da adım adım işlenir. Yanlış bir adresten başlayan çözmenin aynı baytları bambaşka komutlara çevirdiği de bir örnekle gösterilir. Sonra veri taşıma komutlarından XLAT, LEA, LDS ve LES ele alınır. Notun ikinci yarısı dizgi komutlarına ayrılmıştır. MOVS, CMPS, SCAS, LODS ve STOS'un SI, DI, CX ve yön bayrağıyla birlikte çalışması anlatılır. Ardından REP, REPE ve REPNE önekleri, örtüşen bloklar ve süreler gelir. Son kesimler JMP'nin kodlamasını, işlemci kontrolü komutlarını ve hangi komutun hangi bayrağı değiştirdiğini gösteren başvuru tablolarını içerir. Her kod parçası yazmaçlar, bayraklar ve bellekle birlikte izlenir.

Kod (8086): `github.com/sercankulcu/microprocessors-8086/Bolum11`

ÖĞRENME ÇIKTILARI

Komut sınıflarını tanımak
8086'nın yaklaşık yüz komutunu veri taşıma, aritmetik, mantık-kaydırma, dizgi, kontrol aktarma ve işlemci kontrolü sınıflarına ayırmak; her sınıfın tipik kullanımını açıklamak.

Komut kodlamak
Bir assembly komutunun işlem kodunu, D ve W bitlerini, ModR/M baytını, yer değiştirmesini ve anlık değerini bulup makine kodunu bayt bayt yazmak; kısa özel biçimleri ve önekleri tanımak.

Makine kodunu çözmek
Bellekteki bir bayt dizisini doğru başlangıç adresinden komutlara ayırmak; yanlış adresten başlayan çözmenin neden başka komutlar ürettiğini göstermek.

Adres yükleme komutlarını kullanmak
XLAT ile tablo çevirisi yapmak; LEA ile adresi, MOV ile içeriği almak; LDS ve LES ile bellekteki uzak işaretçiyi kesim ve bağıl konum yazmaçlarına yüklemek.

Dizgi komutlarını izlemek
MOVS, CMPS, SCAS, LODS ve STOS komutlarını REP, REPE ve REPNE önekleriyle kullanmak; her adımda SI, DI, CX, ZF ve DF'nin değişimini göstermek; örtüşen bloklarda doğru yönü seçmek.

Başvuru tablolarını okumak
JMP biçimlerinin kodlamasını ve göreli uzaklığı hesaplamak; işlemci kontrolü komutlarının işlemci uçlarıyla ilişkisini açıklamak; bayrak etkileri tablosundan bir komutun hangi bayrağı değiştirdiğini okumak.

11.1 Komut kümesine genel bakış

11.1.1 Komut kümesi nedir?

Bir işlemcinin yürütebildiği bütün komutların listesine **komut kümesi** (*instruction set*) denir. Komut kümesi, işlemci ile onu programlayan kişi arasındaki sözleşmedir. Programcı hangi komutların bulunduğunu, her komutun hangi işlenenleri kabul ettiğini ve yazmaçlara, belleğe ve bayraklara ne yaptığını bilir. İşlemcinin içindeki devrelerin nasıl kurulduğunu bilmesi gerekmez. Aynı komut kümesini yürüten iki farklı işlemci aynı programı çalıştırır; 8086, 8088 ve sonraki x86 işlemciler bu yüzden 8086 için yazılmış programları çalıştırabilir.

8086'nın komut kümesinde, sayılış biçimine göre, yaklaşık yüz komut vardır. Programcı bu komutları `MOV`, `ADD`, `JMP` gibi **anımsatıcılarla** (*mnemonic*) yazar. Anımsatıcı, insan için kolay hatırlanan bir addır; işlemci onu tanımaz. Assembler her satırı işlemcinin anladığı sayısal biçime, yani **makine koduna** çevirir. Aynı anımsatıcı, işlenenlerine göre birçok farklı makine koduna dönüşebilir. Örneğin `MOV` yazmaçtan yazmaca, bellekten yazmaca, anlık değerden belleğe ve kesim yazmacına taşıma için ayrı kodlar kullanır. Bu yüzden 8086'nın anımsatıcı sayısı yüz kadarken farklı işlem kodu sayısı birkaç yüzü bulur.

8086'nın komutları değişken uzunluktadır. En kısa komut tek bayttır (`NOP` = 90h, `INC AX` = 40h). En uzun komut, öneksiz olarak altı bayta ulaşır (`MOV WORD PTR [1234h], 5678h` = C7 06 34 12 78 56). Başa bir ya da daha çok önek baytı da eklenebilir. Değişken uzunluk belleği verimli kullanır: sık kullanılan komutlar kısa, seyrek kullanılanlar uzun tutulur. Bedeli, işlemcinin bir komutun nerede bittiğini ancak baytlarını sırayla çözerek anlayabilmesidir.

11.1.2 Altı işlevsel sınıf

Komutları ezberlemenin en kolay yolu, onları yaptıkları işe göre gruplamaktır. Tablo 11.1 8086 komutlarını altı sınıfa ayırır. Sınıflar yalnız bir ezber kolaylığı değildir: aynı sınıftaki komutlar bayrakları benzer biçimde etkiler ve çoğu zaman benzer biçimde kodlanır. Veri taşıma komutları bayraklara dokunmaz; aritmetik ve mantık komutları sonucu anlatan bayrakları yazar; kontrol aktarma komutları bayrakları okur ama değiştirmez.

Sınıf	Yaptığı iş	Komutlar
1. Veri taşıma	yazmaç, bellek, yığın ve G/Ç arasında veri kopyalama	`MOV PUSH POP XCHG IN OUT XLAT LEA LDS LES LAHF SAHF PUSHF POPF`
2. Aritmetik	dört işlem, BCD düzeltme, işaret genişletme	`ADD ADC INC AAA DAA SUB SBB DEC NEG CMP AAS DAS MUL IMUL AAM DIV IDIV AAD CBW CWD`
3. Mantık, kaydırma, döndürme	bit düzeyinde işlemler	`NOT AND OR XOR TEST SHL SAL SHR SAR ROL ROR RCL RCR`
4. Dizgi işleme	bellek bloklarını SI ve DI ile tarama ve kopyalama	`MOVSB/W CMPSB/W SCASB/W LODSB/W STOSB/W` ; önekler `REP REPE REPZ REPNE REPNZ`
5. Kontrol aktarma	dallanma, döngü, prosedür ve kesme	`JMP` , `Jcc` (`JE JNE JA JB JG JL` …), `LOOP LOOPE LOOPNE JCXZ CALL RET RETF INT INTO IRET`
6. İşlemci kontrolü	bayrakları kurma ve temizleme, işlemcinin durumunu yönetme	`CLC STC CMC CLD STD CLI STI HLT WAIT ESC LOCK NOP`

Tablo 11.1: 8086 komut kümesinin altı işlevsel sınıfı ve her sınıftaki komutlar.

Bu not, birinci sınıftan adres yükleme komutlarını (XLAT, LEA, LDS, LES), dördüncü sınıfın tamamını, beşinci sınıftan JMP'nin kodlamasını ve altıncı sınıfın tamamını ayrıntılı işler. Aritmetik ve mantık komutlarının bayrak etkileri, notun sonundaki başvuru tablolarında bir arada verilir. Kodlama kuralları ise bütün sınıflar için ortaktır.

11.1.3 Komut baytları işlemciye nasıl ulaşır?

Kodlama kurallarının neden böyle olduğunu anlamak için komutun işlemcinin içinde izlediği yola bakmak gerekir. 8086'nın içi iki birime ayrılır. **Yol arayüz birimi** (*bus interface unit*, BIU) dış yolları yönetir. CS:IP'nin gösterdiği adresten komut baytlarını okur ve altı baytlık **önden getirme kuyruğuna** (*prefetch queue*) koyar. Kuyrukta iki bayt boşalır boşalmaz yeni bir okuma başlatır. 8086'nın veri yolu 16 bit olduğundan, çift adresten başlayan bir okuma tek yol döngüsünde iki bayt getirir.

Yürütme birimi (*execution unit*, EU) baytları kuyruğun başından tek tek alır. İlk bayt bir önekse onu bir mandala kaydeder ve sonraki bayta geçer. İlk gerçek işlem kodu baytı, komutun türünü ve ardından kaç bayt geleceğini belirler. İşlem kodu bir ModR/M baytı gerektiriyorsa EU kuyruktan ikinci baytı alır. ModR/M'nin MOD alanı yer değiştirmenin 0, 1 ya da 2 bayt olduğunu söyler. İşlem kodundaki W biti ve komutun türü de anlık değerin uzunluğunu belirler. EU bu baytları aldıktan sonra komutun nerede bittiğini bilir; bir sonraki bayt yeni komutun ilk baytıdır.

Bu sıralı çözme, makine kodunun iki önemli özelliğini açıklar. Birincisi, komutun uzunluğu ilk bir-iki bayttan çıkar; bu yüzden önemli bilgiler (işlem kodu, D, W, MOD) hep başta durur. İkincisi, işlemci kodla veriyi ayırt etmez. CS:IP nereyi gösteriyorsa oradaki bayt işlem kodu sayılır. Yanlış bir adrese atlanırsa aynı baytlar bambaşka komutlar olarak yürütülür. Bu konu makine kodunu çözme kesiminde bir örnekle ayrıntılı işlenir.

Tanım 11.1

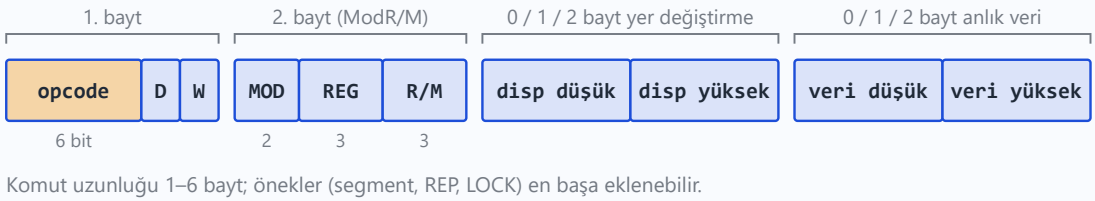
Makine kodu (*machine code*), bir komutun bellekte saklanan ve işlemcinin doğrudan yürüttüğü bayt dizisidir. **Komut biçimi** (*instruction format*) bu dizinin alanlarını ve sırasını tanımlar. **Kodlama**, assembly komutundan makine kodunu üretmek; **çözme**, makine kodundan assembly komutunu geri bulmaktır.

11.2 Komut biçimi

11.2.1 Alanlar

8086'nın genel komut biçimi Şekil 11.1 ile gösterilmiştir. Bir komut en çok beş alandan oluşur ve bu alanlar her zaman aynı sırayla dizilir:

1. **Önekler** (isteğe bağlı): kesim öneki (26h, 2Eh, 36h, 3Eh), yineleme öneki (F3h, F2h) ya da LOCK (F0h). Komutun davranışını değiştirir.
2. **İşlem kodu baytı** (her zaman var): komutun ne yaptığını söyler. Çoğu iki işlenenli komutta bu baytın 6 biti işlemi, son iki biti D ve W bitlerini taşır.
3. **ModR/M baytı** (işlenenlerden biri yazmaç ya da bellek olabiliyorsa): MOD, REG ve R/M alanlarıyla iki işleneni seçer.
4. **Yer değiştirme** (0, 1 ya da 2 bayt): bellek adresine eklenen sabit; düşük bayt önce gelir.
5. **Anlık değer** (0, 1 ya da 2 bayt): komutun içinde verilen sabit işlenen; düşük bayt önce gelir.



Şekil 11.1: 8086 komut biçimi. Birinci bayt 6 bitlik işlem kodu ile D ve W bitlerini, ikinci bayt (ModR/M) MOD, REG ve R/M alanlarını taşır. Ardından 0–2 bayt yer değiştirme ve 0–2 bayt anlık değer gelir.

Yer değiştirme ve anlık değer baytları her zaman **little-endian** sırayla yazılır: 16 bitlik bir sayının düşük baytı düşük adrese, yüksek baytı bir sonraki adrese konur. `MOV AX, 1234h` bu yüzden B8 **34 12** olarak kodlanır. Assembly satırında sayılar insanların okuduğu sırayla yazılır; bellekte ve makine kodu listelerinde ise ters dizilmiş görünür. Bu ters diziliş 8086'nın 16 bitlik veri yolunun düzeninden gelir. Çift adresteki bayt veri yolunun düşük yarısında (D0–D7), tek adresteki bayt yüksek yarısında (D8–D15) taşınır. Sözcüğün düşük baytı çift adreste durunca yazmacın düşük yarısına doğrudan bağlanır.

11.2.2 D ve W bitleri

İki işlenenli komutların çoğunda işlenenlerden biri yazmaç, öteki yazmaç ya da bellektir. 8086 bu iki işleneni tek bir ModR/M baytıyla tanımlar: REG alanı yazmacı, R/M alanı ise yazmacı ya da bellek adresini seçer. Geriye iki soru kalır: işlem bayt mı, sözcük mü; sonuç REG'e mi, R/M'ye mi yazılacak? Bu iki soruyu işlem kodu baytının son iki biti cevaplar.

> **Tanım 11.2**
>
> **W biti** (*word*, işlem kodunun bit 0'ı) işlenen boyutunu seçer: W = 0 bayt (8 bit), W = 1 sözcük (16 bit) işlemdir. REG alanındaki aynı üç bit, W = 0 iken AL…BH, W = 1 iken AX…DI yazmaçlarından birini gösterir.
>
> **D biti** (*direction*, işlem kodunun bit 1'i) yönü seçer: D = 1 ise REG alanı *hedef*, R/M alanı kaynaktır; D = 0 ise REG alanı *kaynak*, R/M alanı hedeftir.

D biti, iki yazmaçlı komutlarda ilginç bir sonuç doğurur. `MOV AX, BX` komutunda iki işlenen de yazmaçtır; hangisinin REG, hangisinin R/M alanına yazılacağı serbesttir. D = 1 seçilirse hedef AX REG alanına, kaynak BX R/M alanına girer: 100010 **1 1** = 8Bh, ModR/M = 11 000 011 = C3h. D = 0 seçilirse kaynak BX REG alanına, hedef AX R/M alanına girer: 100010 **0 1** = 89h, ModR/M = 11 011 000 = D8h. Tablo 11.2 iki kodlamayı ve aynı durumdaki `ADD AX, BX` komutunu yan yana koyar.

Komut	D	İşlem kodu	ModR/M (MOD REG R/M)	Makine kodu
`MOV AX, BX`	1	100010 1 1 = 8Bh	11 000 (AX) 011 (BX) = C3h	`8B C3`
`MOV AX, BX`	0	100010 0 1 = 89h	11 011 (BX) 000 (AX) = D8h	`89 D8`
`ADD AX, BX`	1	000000 1 1 = 03h	11 000 (AX) 011 (BX) = C3h	`03 C3`
`ADD AX, BX`	0	000000 0 1 = 01h	11 011 (BX) 000 (AX) = D8h	`01 D8`

Tablo 11.2: İki yazmaçlı komutların iki geçerli kodlaması. İkisi de aynı işi yapar; fark yalnız D bitinde ve REG ile R/M alanlarının yer değiştirmesindedir.

İki kodlama da geçerlidir ve işlemci ikisini de aynı biçimde yürütür: sonuç AX'e yazılır, BX değişmez. Hangi biçimin üretileceği assembler'a bağlıdır: MASM genellikle D = 1 biçimini (`8B C3`, `03 C3`), NASM ise D = 0 biçimini (`89 D8`, `01 D8`) üretir. İkisi de aynı işi yapar ve aynı uzunluktadır. Bu yüzden ders malzemesinde aynı komut iki farklı kodla görülebilir: 8086 mimarisi anlatılırken `ADD AX, BX` için 01 D8, assembler çıktısında 03 C3 yazılması bir çelişki değildir. İşlenenlerden biri bellek olduğunda ise seçim kalmaz. `MOV [BX], AL` komutunda bellek yalnız R/M alanında gösterilebildiği için AL REG alanına girer ve D = 0 olmak zorundadır (88 07). `MOV AL, [BX]` komutunda ise D = 1'dir (8A 07).

Sık yapılan hata

D bitini "ilk işlenen" ya da "soldaki işlenen" diye okumak. D bitinin anlamı REG alanına bağlıdır: D = 1, REG'deki yazmacın hedef olduğunu söyler. Komut satırında hedef her zaman soldadır; ama makine kodunda hedefin REG mi, R/M mi olduğu D'ye göre değişir.

11.2.3 REG alanı

ModR/M baytının ortadaki üç biti REG alanıdır. Üç bit sekiz yazmacı seçmeye yeter. Hangi sekiz yazmacın kastedildiği W bitine bağlıdır. Tablo 11.3 bu kodları verir. Aynı tablo, kesim yazmaçlarının iki bitlik kodlarını da içerir. Kesim yazmaçları yalnız iki komut ailesinde (8Ch ve 8Eh ile başlayan MOV, ayrıca PUSH ve POP) yer alır ve REG alanına başında 0 ile yazılır.

Kod	W = 0 (bayt)	W = 1 (sözcük)	Kesim yazmacı
`000`	AL	AX	ES (00)
`001`	CL	CX	CS (01)
`010`	DL	DX	SS (10)
`011`	BL	BX	DS (11)
`100`	AH	SP	–
`101`	CH	BP	–
`110`	DH	SI	–
`111`	BH	DI	–

Tablo 11.3: REG alanının (ve MOD = 11 iken R/M alanının) yazmaç kodları. Kesim yazmaçlarının kodu iki bittir.

Sıranın bir mantığı vardır. İlk dört kod AX, CX, DX, BX'i ve onların düşük yarılarını verir. Son dört kod, sözcük işlemde SP, BP, SI, DI'yi, bayt işlemde ilk dört yazmacın yüksek yarılarını (AH, CH, DH, BH) seçer. Yani 100 kodu W = 0 iken AH, W = 1 iken SP demektir. Bir çözmede W bitini gözden kaçırmak, AH'yi SP sanmaya yol açar. Bu kodlar yalnız ModR/M içinde kullanılmaz. Tek baytlık kısa biçimlerde de işlem kodunun son üç biti aynı yazmaç kodunu taşır: `INC` 40h + r, `PUSH` 50h + r, `MOV r8, imm8` B0h + r, `MOV r16, imm16` B8h + r. `PUSH BX` bu yüzden 50h + 3 = 53h, `MOV SI, 1000h` ise B8h + 6 = BEh, ardından 00 10'dur.

11.2.4 MOD alanı

ModR/M baytının en soldaki iki biti MOD alanıdır. MOD, R/M alanının nasıl yorumlanacağını ve komutta kaç baytlık yer değiştirme bulunduğunu söyler. Tablo 11.4 dört durumu verir.

MOD	Anlamı	Yer değiştirme
`00`	bellek işleneni, yer değiştirme yok; tek istisna R/M = 110: 16 bitlik doğrudan adres	0 bayt (R/M = 110 ise 2 bayt)

MOD	Anlamı	Yer değiştirme
`01`	bellek işleneni + 8 bitlik yer değiştirme; d8 işaretle 16 bite genişletilir	1 bayt
`10`	bellek işleneni + 16 bitlik yer değiştirme	2 bayt
`11`	yazmaç işleneni: R/M alanı da bir yazmaç kodudur	yok

Tablo 11.4: MOD alanının dört değeri.

MOD = 01 durumundaki işaret genişletmenin pratik bir yararı vardır. Yer değiştirme −128 ile +127 arasındaysa assembler tek bayt kullanır. `MOV AX, [BX+DI-2]` komutunda −2 = FEh tek bayta sığar ve komut 8B 41 FE olarak üç bayt tutar. İşlemci FEh'yi FFFEh'ye genişletip BX + DI'ye ekler; 16 bitlik toplamada FFFEh eklemek 2 çıkarmakla aynı sonucu verir. Yer değiştirme bu aralığın dışındaysa MOD = 10 seçilir ve iki bayt gerekir: `MOV CL, [BX+SI+1234h]` = 8A 88 34 12.

11.2.5 R/M alanı ve etkin adres

MOD = 11 değilse R/M alanı bir adresleme biçimini seçer. İşlemci bu biçime göre **etkin adresi** (*effective address*, EA) hesaplar. Etkin adres, işlenenin kesim içindeki bağıl konumudur. Fiziksel adres, kesim yazmacının 16 ile çarpımına EA eklenerek bulunur. Tablo 11.5 sekiz R/M kodunu, üç MOD değeriyle ve varsayılan kesimle birlikte verir.

R/M	MOD = 00	MOD = 01	MOD = 10	Varsayılan kesim
`000`	[BX+SI]	[BX+SI+d8]	[BX+SI+d16]	DS
`001`	[BX+DI]	[BX+DI+d8]	[BX+DI+d16]	DS
`010`	[BP+SI]	[BP+SI+d8]	[BP+SI+d16]	SS
`011`	[BP+DI]	[BP+DI+d8]	[BP+DI+d16]	SS
`100`	[SI]	[SI+d8]	[SI+d16]	DS
`101`	[DI]	[DI+d8]	[DI+d16]	DS
`110`	**[d16] doğrudan**	[BP+d8]	[BP+d16]	DS / SS
`111`	[BX]	[BX+d8]	[BX+d16]	DS

Tablo 11.5: R/M alanı ve etkin adres. BP içeren bütün biçimlerde varsayılan kesim SS'tir; MOD = 00, R/M = 110 ise doğrudan adrestir ve kesim DS'dir.

Tablodan üç kural çıkar. Birincisi, yalnız BX, BP, SI ve DI adres yazmacı olabilir; bir taban yazmacı (BX ya da BP) ile bir indis yazmacı (SI ya da DI) birlikte kullanılabilir. `[AX]`, `[CX]`, `[SI+DI]` ya da `[BX+BP]` için R/M kodu yoktur ve assembler bunları hata sayar. İkincisi, BP içeren her biçim yığın kesimini (SS) kullanır. BP, yığın çerçevesindeki parametrelere ve yerel değişkenlere erişmek için tasarlanmıştır. Üçüncüsü, MOD = 00 ve R/M = 110 birleşimi [BP]'ye değil 16 bitlik doğrudan adrese ayrılmıştır. Yalnız `[BP]` yazıldığında assembler bu yüzden MOD = 01 ve d8 = 0 kullanır: `MOV AL, [BP]` üç bayttır (8A 46 00), oysa `MOV AL, [BX]` iki bayttır (8A 07).

Etkin adresin hesaplanması zaman alır ve bu süre komutun süresine eklenir. 8086'da EA'yı yürütme biriminin mikrokodu ALU ile adım adım hesaplar; bu yüzden süre adresleme biçimine göre değişir. Tablo 11.6 Intel 8086 kılavuzundaki EA sürelerini verir. Bir bellek işlenenli komutun toplam süresi "temel süre + EA" biçiminde yazılır. Örneğin `MOV reg, mem` 8 + EA saat turu sürer; `MOV CL, [BX+SI+12h]` için bu 8 + 11 = 19 turdur. 5 MHz'lik bir 8086'da bir saat turu 200 ns olduğundan komut yaklaşık 3,8 µs sürer.

Adresleme biçimi	Örnek	EA süresi
yalnız taban ya da indis	`[BX]`, `[SI]`, `[DI]`	5
yalnız yer değiştirme (doğrudan)	`[1234h]`, `degisken`	6
taban + indis	`[BP+DI]`, `[BX+SI]` / `[BP+SI]`, `[BX+DI]`	7 / 8
yer değiştirme + taban ya da indis	`[BX+4]`, `[BP+2]`, `dizi[SI]`	9
yer değiştirme + taban + indis	`[BP+DI+d]`, `[BX+SI+d]` / `[BP+SI+d]`, `[BX+DI+d]`	11 / 12

Tablo 11.6: 8086'da etkin adres hesaplama süreleri (saat turu). Kesim öneki varsa 2 tur eklenir.

Bellekten okunan ya da belleğe yazılan her sözcük için bir sayı daha önemlidir. Sözcük çift adresteyse BIU onu tek yol döngüsünde (4 saat turu) taşır. Tek adresteyse iki ayrı bayt döngüsü gerekir ve süreye 4 tur eklenir. 8086, çift adres için A0 = 0 ile düşük bayt şeridini, tek adres için $\overline{BHE}$ = 0 ile yüksek bayt şeridini seçer. Tek adresteki bir sözcüğün iki baytı farklı şeritlere ve farklı sözcük adreslerine düştüğünden iki döngü kaçınılmazdır. Assembler'daki `EVEN` yönergesi sözcük dizilerini çift adrese hizalamak için bu yüzden kullanılır.

11.2.6 Kısa özel biçimler

Genel biçim her komutu kodlayabilir ama her zaman en kısa kodu vermez. 8086'nın tasarımcıları, en sık kullanılan işlemler için ModR/M baytı gerektirmeyen ayrı işlem kodları ayırmıştır. Bu kısa biçimler bir ya da iki bayt kazandırır ve bazen daha hızlıdır. Tablo 11.7 en önemlilerini toplar.

Biçim	İşlem kodu	Örnek	Kod	Genel biçim
INC / DEC r16	40h+r / 48h+r	`INC AX`	`40`	`FF C0` (2 bayt)
PUSH / POP r16	50h+r / 58h+r	`PUSH BX`	`53`	`FF F3` (2 bayt)
XCHG AX, r16	90h+r	`XCHG AX, BX`	`93`	`87 C3` (2 bayt)
MOV r8, imm8	B0h+r	`MOV AL, 5`	`B0 05`	`C6 C0 05` (3 bayt)
MOV r16, imm16	B8h+r	`MOV AX, 1234h`	`B8 34 12`	`C7 C0 34 12` (4 bayt)
MOV AL/AX ↔ bellek	A0h–A3h	`MOV AX, [2000h]`	`A1 00 20`	`8B 06 00 20` (4 bayt)
ADD AL/AX, imm	04h / 05h	`ADD AL, 7`	`04 07`	`80 C0 07` (3 bayt)
ADD r/m16, imm8 (işaret genişletmeli)	83h /0	`ADD BX, 5`	`83 C3 05`	`81 C3 05 00` (4 bayt)

Tablo 11.7: ModR/M baytı kullanmayan kısa biçimler. "r" işlem kodunun son üç bitine yazılan yazmaç kodudur.

Son satırdaki 83h biçimi bir başka fikri de gösterir. 80h, 81h ve 83h işlem kodları sekiz farklı aritmetik ve mantık komutunu (ADD, OR, ADC, SBB, AND, SUB, XOR, CMP) anlık değerle yapar. Hangi işlemin yapılacağını ModR/M'nin REG alanı söyler, çünkü bu komutlarda ikinci bir yazmaç işleneni yoktur. Kılavuzlarda bu durum `83 /0` biçiminde yazılır: "/0", REG alanına 000 konacağını, yani işlemin ADD olduğunu belirtir. `ADD BX, 5` için ModR/M = 11 000 011 = C3h olur ve ardından tek bayt 05 gelir. İşlemci 05h'yi işaretle 0005h'ye genişletip BX'e ekler. Aynı "işlem seçici" düzen FEh ve FFh ailesinde de vardır: `FF /4` dolaylı yakın JMP, `FF /6` PUSH bellek, `FE /0` bayt INC'dir. Bu düzen JMP'nin kodlaması anlatılırken yeniden karşımıza çıkar.

> **Sık yapılan hata**
>
> Bir komutun makine kodunun tek olduğunu sanmak. `MOV AL, [1000h]` hem A0 00 10 (3 bayt) hem 8A 06 00 10 (4 bayt) olarak kodlanabilir; ikisi de doğrudur. Assembler kısa biçimi seçer. Elle kodlama yaparken önce kısa bir biçim olup olmadığına bakmak, sınavda ve liste (listing) çıktısıyla karşılaştırmada karışıklığı önler.

11.3 Komut kodlama adım adım

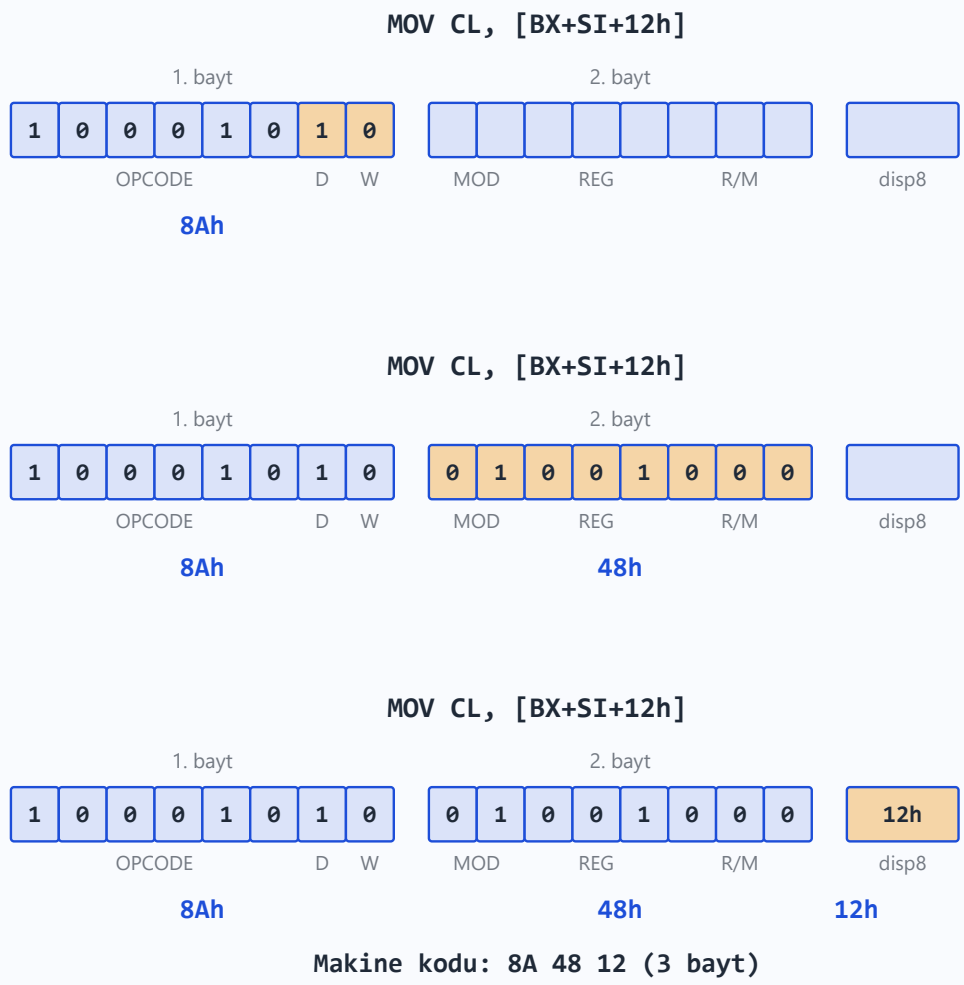
11.3.1 Kodlama yöntemi

Bir assembly komutunu elle kodlamak, assembler'ın yaptığı işi kâğıt üzerinde tekrarlamaktır. İş her seferinde aynı sırayla yürür:

1. **Biçimi seç.** İşlenenlerin türüne bakılır: yazmaç–yazmaç, yazmaç–bellek, anlık–yazmaç, anlık–bellek. Kısa özel bir biçim varsa (AL/AX ile bellek, anlık değerle yazmaç, INC r16 ...) o kullanılır; yoksa genel biçime geçilir.
2. **İşlem kodunu yaz.** Komut ailesinin 6 bitlik kodu yazılır (MOV için 100010, ADD için 000000). Ardından D ve W bitleri eklenir. D, REG alanındaki yazmacın hedef olup olmadığına göre; W, işlenen boyutuna göre seçilir.
3. **ModR/M baytını kur.** REG alanına yazmaç işleneninin kodu yazılır. Öteki işlenen yazmaçsa MOD = 11 olur ve R/M'ye onun kodu girer. Bellekse adresleme biçimine göre R/M seçilir ve yer değiştirmenin boyutuna göre MOD = 00, 01 ya da 10 olur.
4. **Yer değiştirmeyi ve anlık değeri ekle.** Önce yer değiştirme, sonra anlık değer gelir; ikisi de düşük bayt önce yazılır.
5. **Önekleri başa koy.** Varsayılan kesim dışında bir kesim kullanılıyorsa kesim öneki, dizgi komutu yineleniyorsa REP öneki, yol kilitleniyorsa LOCK en başa eklenir.

11.3.2 Örnek: `MOV CL, [BX+SI+12h]`

Bu komut DS:(BX + SI + 12h) adresindeki bir baytı CL'ye kopyalar. İşlenenlerden biri yazmaç (CL), öteki bellektir. Kısa bir biçim yoktur, çünkü kısa bellek biçimi yalnız AL ve AX için ve yalnız doğrudan adres için vardır. Genel biçim kullanılır: MOV reg, r/m. Şekil 11.2 kodlamanın üç aşamasını gösterir.



Şekil 11.2: `MOV CL, [BX+SI+12h]` komutunun kodlanması. Üstte birinci bayt (işlem kodu 100010, D = 1, W = 0 → 8Ah), ortada ModR/M baytı (MOD = 01, REG = 001, R/M = 000 → 48h), altta 8 bitlik yer değiştirme (12h) ve sonuç.

1. **İşlem kodu.** MOV'un yazmaç ile r/m arasındaki ailesi 100010'dur.
2. **D ve W.** Hedef CL bir yazmaçtır ve REG alanına girecektir; REG hedef olduğu için D = 1. CL bayt yazmacıdır; W = 0. Birinci bayt 1000 1010b = 8Ah.
3. **REG.** CL'nin kodu, W = 0 sütununda 001'dir.
4. **R/M ve MOD.** [BX+SI+d] biçiminin R/M kodu 000'dır. Yer değiştirme 12h, −128...+127 aralığına sığar; MOD = 01 (8 bitlik yer değiştirme). ModR/M = 01 001 000b = 48h.
5. **Yer değiştirme.** 12h tek bayt olarak eklenir.

Sonuç **8A 48 12**'dir ve komut üç bayttır. Yer değiştirme 1234h gibi bir sözcük olsaydı MOD = 10 seçilecek, ModR/M 10 001 000b = 88h olacak ve komut 8A 88 34 12 biçiminde dört bayt tutacaktı. Komutun süresi 8 + EA'dır; [BX+SI+d] biçiminin EA süresi 11 tur olduğundan toplam 19 saat turu eder. Bu sürenin büyük kısmı adres hesabına ve BIU'nun bellekten baytı getirmesine gider.

Örnek 11.1

`MOV [SI+20h], CX` komutunu kodlayalım. Kaynak CX REG alanına girer, hedef bellek R/M'dedir: REG kaynak olduğundan D = 0, sözcük işlem olduğundan W = 1. Birinci bayt 100010 0 1 = 89h. REG = 001 (CX), R/M = 100 ([SI+d]), 20h tek bayta sığdığından MOD = 01. ModR/M = 01 001 100b = 4Ch. Sonuç **89 4C 20**.

Aynı yöntemle `ADD BX, [1234h]` : ADD ailesi 000000, BX hedef ve REG'de olduğundan D = 1, W = 1 → 03h. REG = 011 (BX). Doğrudan adres için MOD = 00, R/M = 110 → ModR/M = 00 011 110b = 1Eh. Ardından adres 34 12 gelir: **03 1E 34 12**.

11.3.3 Örnek komutlar ve uzunlukları

Tablo 11.8 farklı uzunluktaki on komutu ve makine kodlarını bir arada verir. Tablo, komut biçiminin bütün alanlarının ne zaman ortaya çıktığını gösterir: yalnız işlem kodu (1 bayt), işlem kodu ve anlık değer, işlem kodu ve ModR/M, önek ve ModR/M, ModR/M ile yer değiştirme ve anlık değer birlikte.

Komut	Makine kodu	Bayt	Açıklama
`NOP`	`90`	1	aslında `XCHG AX, AX` (90h + 0)
`INC AX` / `PUSH BX`	`40` / `53`	1	yazmaç kodu işlem kodunun içinde (40h + r, 50h + r)
`MOV AL, 5`	`B0 05`	2	B0h + r, anlık bayt
`INT 21h`	`CD 21`	2	kesme numarası komutun içinde
`MOV AX, 1234h`	`B8 34 12`	3	düşük bayt önce (little-endian)
`MOV CL, [BX+SI+12h]`	`8A 48 12`	3	ModR/M + d8
`MOV ES:[BX], AL`	`26 88 07`	3	26h = ES öneki
`ADD [BX+DI+1000h], DX`	`01 91 00 10`	4	MOD = 10, d16
`MOV WORD PTR [1234h], 5678h`	`C7 06 34 12 78 56`	6	en uzun öneksiz biçim: d16 + imm16
`REP MOVSB`	`F3 A4`	2	F3h = REP öneki

Tablo 11.8: Örnek komutlar, makine kodları ve uzunlukları.

Birkaç satırı ayrıca açmak yararlıdır. `ADD [BX+DI+1000h], DX` komutunda kaynak DX REG alanındadır (D = 0, W = 1 → 01h). ModR/M = 10 010 001b = 91h: MOD = 10 (16 bitlik yer değiştirme), REG = 010 (DX), R/M = 001 ([BX+DI+d]). Yer değiştirme 1000h, little-endian olarak 00 10 diye eklenir. Bu komut belleği hem okur hem yazar: BIU önce sözcüğü okur, ALU toplamayı yapar, BIU sonucu aynı adrese geri yazar. 8086'da bu biçimin süresi 16 + EA, yani 16 + 12 = 28 saat turudur.

`MOV WORD PTR [1234h], 5678h` komutunda bütün alanlar bir aradadır. C7h, "anlık değeri r/m'ye taşı" ailesinin sözcük biçimidir. ModR/M = 00 000 110b = 06h: REG alanı burada yazmaç değil işlem seçicidir ve MOV için 000'dır; MOD = 00, R/M = 110 doğrudan adres demektir. Ardından adres (34 12) ve en sonda anlık değer (78 56) gelir. Bu komut, öneksiz bir 8086 komutunun ulaşabildiği en büyük uzunluk olan altı bayttır. Assembly satırındaki `WORD PTR` gereklidir; o olmadan assembler 5678h'nin bayt mı sözcük mü yazılacağını bilemez. Bu örnekte 5678h bayta sığmadığı için sözcük tahmin edilebilir, ama `MOV [1234h], 5` gibi bir satırda boyut gerçekten belirsizdir ve hata verilir.

11.3.4 MOV'un kodlama biçimleri

MOV, 8086'nın en çok kullanılan komutudur ve tek bir anımsatıcının arkasında sekiz ayrı kodlama biçimi barındırır. Tablo 11.9 bu biçimleri, ardından gelen alanları, birer örneği ve Intel 8086 kılavuzundaki süreleri verir.

Biçim	İşlem kodu (8 / 16 bit)	Ardından	Örnek	Kod	Süre
yazmaç → r/m	`88` / `89`	ModR/M (+ disp)	`MOV [BX], AL`	`88 07`	2 (yazmaç), 9 + EA
r/m → yazmaç	`8A` / `8B`	ModR/M (+ disp)	`MOV CX, [SI]`	`8B 0C`	2 (yazmaç), 8 + EA
anlık → r/m	`C6` / `C7`	ModR/M (REG = 000) + disp + imm	`MOV BYTE PTR [BX], 5`	`C6 07 05`	10 + EA
anlık → yazmaç	`B0+r` / `B8+r`	imm8 / imm16	`MOV SI, 1000h`	`BE 00 10`	4
bellek → AL / AX	`A0` / `A1`	16 bit adres	`MOV AX, [2000h]`	`A1 00 20`	10
AL / AX → bellek	`A2` / `A3`	16 bit adres	`MOV [1000h], AL`	`A2 00 10`	10
kesim yazmacı → r/m16	`8C`	ModR/M (REG = sreg)	`MOV AX, ES`	`8C C0`	2, 9 + EA
r/m16 → kesim yazmacı	`8E`	ModR/M (REG = sreg)	`MOV DS, AX`	`8E D8`	2, 8 + EA

Tablo 11.9: MOV'un kodlama biçimleri. Süreler saat turudur; EA etkin adres süresidir.

Tablodaki ilk iki satır aynı ailenin D = 0 ve D = 1 hâlleridir: 88h/89h ile 8Ah/8Bh arasındaki tek fark bit 1'dir. `MOV CX, [SI]` komutunda CX hedef ve REG'dedir: 8Bh; ModR/M = 00 001 100b = 0Ch. `MOV [BX], AL` komutunda AL kaynaktır: 88h; ModR/M = 00 000 111b = 07h. Üçüncü satırda REG alanı yazmaç seçmez, "000 = MOV" anlamına gelir; ikinci işlenen anlık değer olduğundan REG boştur. AL/AX ile doğrudan adres arasındaki taşımanın A0h–A3h kısa biçimi, ModR/M baytını atlar ve bir bayt kazandırır. Kısa biçim 10 tur sürer; genel biçim okumada 8 + 6 = 14, yazmada 9 + 6 = 15 tur tutar.

11.3.5 Kesim yazmaçlı MOV

Kesim yazmaçları genel amaçlı yazmaçlardan ayrı tutulur ve yalnız iki MOV biçimiyle taşınır. 8Ch kesim yazmacından r/m16'ya, 8Eh r/m16'dan kesim yazmacına kopyalar. Her iki biçimde de REG alanının ilk biti 0, son iki biti kesim yazmacının kodudur: ES = 00, CS = 01, SS = 10, DS = 11. Şekil 11.3 `MOV DS, AX` komutunun ikinci baytını açar: MOD = 11 (yazmaç), REG = 011 (DS), R/M = 000 (AX), yani 1101 1000b = D8h. Komut **8E D8**'dir.



Şekil 11.3: `MOV DS, AX` = 8E D8. ModR/M baytının REG alanında 0 + kesim yazmacı kodu (DS = 11), R/M alanında kaynak yazmaç (AX = 000) bulunur.

Bazı biçimler hiç yoktur. Kesim yazmacına anlık değer yazan bir işlem kodu bulunmaz; `MOV DS, 1234h` geçersizdir ve değer önce AX gibi bir genel yazmaca yüklenir. Bellekten belleğe taşıyan bir MOV da yoktur, çünkü ModR/M baytı yalnız bir bellek işleneni tanımlayabilir. İki bellek işleneni gerektiren iş, dizgi komutlarının (MOVS, CMPS) özel görevidir. `MOV CS, AX` ise kodlanabilir görünse de kullanılmaz: CS'yi IP ile birlikte değiştirmeyen bir komut, programın bir sonraki komutu bambaşka bir kesimden getirmesine yol açar. CS yalnız uzak JMP, uzak CALL, RETF, INT ve IRET ile, IP ile birlikte değişir. 8086'nın bir özelliği daha vardır: SS'ye yazan bir MOV ya da POP'tan sonra, bir sonraki komut bitene kadar kesme kabul edilmez. Böylece `MOV SS, AX` ile hemen ardından gelen `MOV SP, …` arasına bir kesmenin girip yanlış yığını kullanması önlenir.

Örnek 11.2

`MOV ES, DX` : 8Eh ailesi, REG = 0 + 00 (ES) = 000, MOD = 11, R/M = 010 (DX). ModR/M = 11 000 010b = C2h. Komut **8E C2**. Ters yöndeki `MOV AX, ES` 8Ch ile kodlanır ve ModR/M = 11 000 000b = C0h olur: **8C C0**.

11.3.6 Doğrudan adres: MOD = 00, R/M = 110

R/M tablosundaki tek düzensizlik, MOD = 00 ve R/M = 110 birleşimidir. Düzenli olsaydı bu birleşim [BP] anlamına gelecekti. 8086 tasarımcıları onu 16 bitlik doğrudan adrese ayırmıştır, çünkü değişkenlere adıyla erişmek [BP]'den çok daha sıktır. `MOV [1000h], BL` komutu bu yüzden şöyle kodlanır:

Bayt	Bitler	Anlamı
`88h`	`100010 0 0`	MOV r/m ↔ reg; D = 0 (REG kaynak), W = 0 (bayt)
`1Eh`	`00 011 110`	MOD = 00, REG = 011 (BL), R/M = 110 → doğrudan adres

Bayt	Bitler	Anlamı
`00 10`	`0001 0000 0000 0000`	16 bitlik adres 1000h (düşük bayt önce)

Tablo 11.10: `MOV [1000h], BL` = 88 1E 00 10 komutunun bitleri.

Komut dört bayttır ve kesim DS'dir. BL için A2h benzeri bir kısa biçim yoktur; kısa biçim yalnız AL ve AX içindir. Aynı adrese AL yazılsaydı assembler A2 00 10 (3 bayt) üretirdi. Genel biçimle yazılmış `88 06 00 10` de aynı işi yapar: ModR/M 06h = 00 000 110b, REG = 000 (AL). Tablo 11.11 bu biçimleri [BP] ile karşılaştırır.

Komut	Kod	Açıklama
`MOV [1000h], AL`	`A2 00 10`	AL için kısa biçim, 3 bayt
`MOV [1000h], AL`	`88 06 00 10`	genel biçim, 4 bayt (aynı iş): ModR/M 06h = 00 000 110
`MOV [1000h], BL`	`88 1E 00 10`	BL için kısa biçim yok
`MOV [BP], BL`	`88 5E 00`	5Eh = 01 011 110: [BP+d8], d8 = 0; kesim SS

Tablo 11.11: Doğrudan adres ve [BP] kodlamaları.

Bu kural yalnız assembler'ı değil çözücüyü de ilgilendirir. 1Eh baytını gören bir çözücü, MOD = 00 ve R/M = 110 olduğunu fark edip arkasından iki adres baytı okumak zorundadır. Kuralı unutan bir çözücü 88 1E'yi iki baytlık `MOV [BP], BL` sanır. O zaman adres baytları yeni bir komutun başı olur: 00 10, `ADD [BX+SI], DL` diye okunur (00h = ADD r/m8, r8; 10h = 00 010 000b). Bundan sonraki bütün komut sınırları kayar. Bu, bir sonraki kesimde işlenen genel sorunun küçük bir örneğidir.

11.3.7 Önek baytları

Önek, kendi başına bir komut değildir; arkasından gelen komutun davranışını değiştirir. 8086'da üç tür önek vardır ve Tablo 11.12 hepsini listeler.

Önek	Bayt	Etkisi
ES: / CS: / SS: / DS:	`26` / `2E` / `36` / `3E`	bellek işleneninin varsayılan kesimini değiştirir (001*ss*110 düzeni)
REP / REPE / REPZ	`F3`	dizgi komutunu CX ≠ 0 (ve CMPS/SCAS için ZF = 1) iken yineler
REPNE / REPNZ	`F2`	dizgi komutunu CX ≠ 0 ve ZF = 0 iken yineler
LOCK	`F0`	komut süresince LOCK̄ ucunu etkin tutar; yolu başka işlemcilere kapatır

Tablo 11.12: 8086 önek baytları.

Kesim öneklerinin bit düzeni ilginçtir: 26h = 001 **00** 110, 2Eh = 001 **01** 110, 36h = 001 **10** 110, 3Eh = 001 **11** 110. Ortadaki iki bit, MOV biçimlerinde kullanılan kesim yazmacı kodunun aynısıdır. `MOV AL, ES:[BX]` bu yüzden 26 8A 07 olarak kodlanır: önek, ardından `MOV AL, [BX]`'in kendi kodu. Assembler gereksiz öneki üretmez. `MOV AL, DS:[BX]` yazılırsa DS zaten varsayılan kesim olduğu için önek eklenmez ve kod 8A 07 kalır. `MOV AL, DS:[BP]` ise öneki gerektirir, çünkü BP'nin varsayılanı SS'tir.

Önek, komutun süresine 2 saat turu ekler ve bir bayt yer kaplar. Bir başka bedeli kesmelerle ortaya çıkar. 8086, REP önekli bir dizgi komutu yürütülürken kesme gelirse işi yarıda bırakıp kesmeye gidebilir. Dönüşte komutu yeniden başlatmak için yığına önekin adresini iter. Ancak 8086 yalnız *son* öneki hatırlar: `REP` ile kesim öneki birlikte kullanılmışsa (`REP ES: MOVSB` gibi) dönüşte önceki önek kaybolur. Bu yüzden 8086 programlarında REP'li bir dizgi komutuna kesim öneki eklenmez; gerekiyorsa kesmeler CLI ile kapatılır.

Örnek 11.3

`2E FF 27` baytlarını çözelim. 2Eh, CS önekidir. FFh, işlemi ModR/M'nin REG alanının seçtiği bir ailedir. 27h = 00 100 111b: MOD = 00, REG = 100 (/4: dolaylı yakın JMP), R/M = 111 ([BX]). Komut `JMP WORD PTR CS:[BX]` 'tir: IP, CS:[BX] adresindeki sözcükle değişir. Atlama tablosu kod kesiminde tutulduğunda bu önek gerekir.

11.4 Makine kodunu çözme

11.4.1 Çözme yöntemi

Çözme (*disassembly*), kodlamanın tersidir: bellekteki baytlardan assembly komutlarını geri bulmak. Hata ayıklayıcılar, emu8086'nın bellek penceresi ve liste çıktısı bu işi yapar. Elle çözme, bir programın bellekte nasıl durduğunu anlamanın ve sınavlarda sık sorulan “bu baytlar hangi komutlar?” sorusunu cevaplamanın yoludur. Yöntem, işlemcinin EU'sunun yaptığını taklit eder:

1. **Başlangıç noktasını bil.** Çözme, bir komutun ilk baytından başlamalıdır. Program başlangıcı (.COM için 0100h) ya da bir atlama hedefi güvenilir başlangıçtır.
2. **İlk baytı sınıflandır.** Bayt bir önekse (26h, 2Eh, 36h, 3Eh, F0h, F2h, F3h) not edilir ve sonraki bayta geçilir. Değilse işlem kodu tablosundan komut ailesi bulunur.
3. **Kısa biçim mi?** 40h–5Fh, 90h–97h, B0h–BFh gibi kodlarda yazmaç işlem kodunun içindedir; ModR/M yoktur. Ardından gelecek anlık değerin uzunluğu ailenin kuralından bilinir.
4. **ModR/M'yi aç.** Aile ModR/M gerektiriyorsa sonraki bayt MOD, REG, R/M alanlarına ayrılır. MOD ve R/M'den yer değiştirmenin uzunluğu çıkar (MOD = 00 ve R/M = 110 özel durumu unutulmaz).
5. **Kalan baytları al.** Yer değiştirme ve anlık değer little-endian olarak okunur. Komut burada biter; bir sonraki bayt yeni komutun ilk baytıdır.

11.4.2 Örnek: on iki baytlık bir parça

Bellekte 0100h'den başlayan şu on iki bayt bir programın parçası olsun: `8B 4E 04 03 C3 A3 00 02 F3 AB EB FE`. Çözme 0100h'den başlar. Tablo 11.13 sonucu verir; adımlar aşağıda açıklanmıştır.

Adres	Baytlar	Çözüm	Komut
0100h	`8B 4E 04`	8Bh = 100010 1 1; 4Eh = 01 001 110 → MOD = 01, REG = CX, R/M = [BP+d8]; d8 = 04h	`MOV CX, [BP+04h]`
0103h	`03 C3`	03h = 000000 1 1 (ADD, D = 1, W = 1); C3h = 11 000 011 → AX ← AX + BX	`ADD AX, BX`
0105h	`A3 00 02`	A3h = AX → bellek kısa biçimi; adres 0200h	`MOV [0200h], AX`
0108h	`F3 AB`	F3h = REP öneki; ABh = STOSW	`REP STOSW`
010Ah	`EB FE`	EBh = kısa JMP; FEh = −2 → hedef 010Ch − 2 = 010Ah	`JMP 010Ah` (kendine)

Tablo 11.13: 8B 4E 04 03 C3 A3 00 02 F3 AB EB FE baytlarının çözümü.

İlk bayt 8Bh, MOV r/m ↔ reg ailesindendir ve D = 1, W = 1 taşır: REG alanı 16 bitlik hedef yazmaçtır. İkinci bayt 4Eh = 0100 1110b, alanlarına ayrılınca 01 | 001 | 110 olur. MOD = 01 olduğundan bir bayt yer değiştirme gelecektir. REG = 001, W = 1 ile CX'tir. R/M = 110, MOD = 01 iken [BP+d8] demektir; MOD = 00 olsaydı doğrudan adres olurdu. Üçüncü bayt 04h yer değiştirmedir. Komut `MOV CX, [BP+04h]`, kesim SS'tir. Bu biçim, yığından bir prosedür parametresi okumanın tipik kalıbıdır.

Dördüncü bayt 03h, ADD ailesinin D = 1, W = 1 biçimidir. Beşinci bayt C3h = 11 000 011b'de MOD = 11'dir, yani iki işlenen de yazmaçtır: REG = AX (hedef), R/M = BX. Komut `ADD AX, BX`'tir. Aynı komutun 01 D8 biçiminde de kodlanabileceği yukarıda gösterilmişti; burada D = 1 biçimiyle karşılaşıyoruz. Altıncı bayt A3h, ModR/M kullanmayan kısa biçimdir: AX'i doğrudan adrese yazar. Ardından gelen iki bayt adresi verir: 00 02 → 0200h. Dokuzuncu bayt F3h bir önektir; komut bir sonraki baytla tamamlanır. ABh = 1010 101 1b, STOS ailesinin W = 1 biçimidir. Komut `REP STOSW`'dir ve yalnız iki bayt tutar. Son iki bayt EB FE'dir. EBh kısa JMP'dir ve FEh işaretli olarak −2'dir. Hedef, bir sonraki komutun adresi olan 010Ch'ye −2 eklenerek bulunur: 010Ah, yani komutun kendisi. Bu iki bayt, bir programı olduğu yerde durduran sonsuz döngüdür.

Sonuç: on iki bayt beş komuttur. Komutların uzunlukları 3, 2, 3, 2 ve 2 bayttır ve her uzunluk yalnız ilk bir-iki bayta bakılarak çıkarılmıştır. Çözmenin sonunda bayt sayısının tam tutması (0100h + 12 = 010Ch), sınırların doğru bulunduğunun bir denetimidir.

11.4.3 Yanlış bağıl konumdan çözme

Çözme yönteminin ilk adımı, başlangıç noktasını bilmekti. Bu adımın neden vazgeçilmez olduğunu bir örnek gösterir. Bellekte bağıl konum 0000h'den başlayan şu on bir bayt bulunsun: `B4 02 B2 41 CD 21 B8 00 4C CD 21`. Konum 0'dan çözüldüğünde bu baytlar, ekrana 'A' yazıp programı bitiren tanıdık bir programdır:

```
MOV AH, 02h     ; B4 02
MOV DL, 41h     ; B2 41  ('A')
INT 21h         ; CD 21  karakter yaz
MOV AX, 4C00h   ; B8 00 4C
INT 21h         ; CD 21  programı bitir
```

B4h = B0h + 4, `MOV AH, imm8`'dir ve bir anlık bayt alır: 02h. B2h = B0h + 2, `MOV DL, imm8`'dir: 41h = 'A'. CDh, INT komutudur ve kesme numarasını ardından gelen bayttan alır. B8h, `MOV AX, imm16`'dır ve iki bayt alır: 00 4C → 4C00h. Son CD 21, AH = 4Ch ile DOS'un program bitirme hizmetini çağırır. Çözüm, programcının yazdığı kodun ta kendisidir.

Şimdi IP'nin bir bayt kaydığını düşünelim; örneğin hatalı bir atlama 0001h'ye gitsin. İşlemci 02h'yi işlem kodu sayar. Şekil 11.4 iki çözümü üst üste gösterir ve Tablo 11.14 ikinci çözümün bayt bayt açıklamasını verir.



Şekil 11.4: Aynı on bir baytın bağıl konum 0'dan (üstte) ve bağıl konum 1'den (altta) çözülmesi. Konum 1'den çözümde 'A' ve INT'in işlem kodu bir yer değiştirmenin içinde kaybolur; son CD 21'de iki çözüm yeniden aynı sınıra düşer.

Konum	Baytlar	Çözüm	Komut
0001h	`02 B2 41 CD`	02h = ADD r8, r/m8 (D = 1, W = 0); B2h = 10 110 010 → MOD = 10 (d16), REG = DH, R/M = [BP+SI]; d16 = CD41h	`ADD DH, [BP+SI+CD41h]`
0005h	`21 B8 00 4C`	21h = AND r/m16, r16 (D = 0, W = 1); B8h = 10 111 000 → MOD = 10, REG = DI, R/M = [BX+SI]; d16 = 4C00h	`AND [BX+SI+4C00h], DI`
0009h	`CD 21`	INT 21h	`INT 21h`

Tablo 11.14: Bağıl konum 1'den başlayan çözüm.

Konum 1'den çözüm üç komut verir. İlki B2h'yi ModR/M sayar ve MOD = 10 yüzünden iki bayt yer değiştirme okur. Böylece 'A' karakteri (41h) ile INT'in işlem kodu (CDh), CD41h diye bir sayının içinde kaybolur. Bu komut SS:(BP + SI + CD41h) adresindeki baytı DH'ye ekler; yalnız okur. İkinci komut daha tehlikelidir: 21h, D = 0 olan AND'dir ve sonucu bellekteki r/m işlenene yazar. DS:(BX + SI +

4C00h) adresindeki sözcük, DI ile VE işlemine girip bozulur. Son CD 21'de iki çözüm yeniden aynı sınıra düşer; çözme kendiliğinden yeniden eşlenmiştir. Ama bu kez AX = 4C00h hiç yüklenmemiştir. INT 21h, AH'de o an ne varsa o hizmetle çağrılır ve sonucu kestirilemez.

Örnek iki şey öğretir. Birincisi, baytların anlamını baytlar değil, IP'nin başladığı yer belirler. Aynı on bir bayt bir başlangıçta zararsız bir "A yaz ve çık" programıdır, bir bayt ötede belleği bozan bir komut dizisidir. İkincisi, x86 kodunda çözme genellikle birkaç komut sonra kendiliğinden doğru sınıra döner. Bu, yanlış çözümün fark edilmesini zorlaştırır: birkaç anlamsız komuttan sonra her şey yeniden düzgün görünür.

Örnek 11.4

`8A 1E 34 12` baytlarını konum 0'dan çözelim. 8Ah = 100010 1 0 (D = 1, W = 0), 1Eh = 00 011 110 (MOD = 00, REG = BL, R/M = 110 → doğrudan adres) olur. Komut `MOV BL, [1234h]`'tür. Konum 1'den başlanırsa 1Eh tek baytlık `PUSH DS`'dir (06h + 8 × 3 düzenindeki kesim PUSH'u), ardından 34 12 iki baytlık `XOR AL, 12h` olur. Dört bayt bu kez iki ayrı komut verir ve yığına beklenmedik bir sözcük itilir.

11.4.4 Kod ve veri: işlemcinin bakışı

8086'da kodla veriyi ayıran bir donanım yoktur. Kod kesimi ile veri kesimi, kesim yazmaçlarındaki birer sayıdan ibarettir ve .COM programlarında dört kesim yazmacı da aynı kesimi gösterir. Program, verilerini kodun hemen arkasına koyar. Bir JMP yanlışlıkla veri alanına atlarsa ya da yığındaki dönüş adresi bozulup RET oraya dönerse, işlemci veri baytlarını komut diye yürütmeye başlar. `sayi db 3Ch` gibi bir değişken, işlem kodu olarak okunursa 3Ch = `CMP AL, imm8` olur ve ardından gelen baytı anlık değer diye yutar.

Çözücü programlar da aynı sorunla karşılaşır. Bir çözücü, bellekteki baytların hangisinin kod, hangisinin veri olduğunu ancak programın akışını izleyerek tahmin edebilir. Listing dosyası bu sorunu yaşamaz, çünkü assembler her satırın ne olduğunu kaynak koddan bilir. Bu nedenle bir programın makine kodunu incelemenin en güvenilir yolu, assembler'ın ürettiği liste çıktısına bakmaktır. emu8086'da derleme sonrasında oluşan .lst dosyası, her satırın adresini ve baytlarını yan yana verir.

Sık yapılan hata

Programın sonundaki `RET` ya da `INT 21h/4Ch`'yi unutmak. Kod bittiğinde işlemci durmaz; IP ilerlemeye devam eder ve hemen arkadan gelen değişkenleri komut olarak yürütür. Program o zaman anlamsız davranır ya da kilitlenir. Değişkenleri koddan önce tanımlayıp üzerinden bir JMP ile atlamak ya da her çıkış yolunun açıkça programı bitirdiğinden emin olmak bu hatayı önler.

11.5 Veri taşıma komutları

11.5.1 Genel bakış

Veri taşıma komutları (*data transfer instructions*) bir değeri bir yerden ötekine kopyalar: yazmaçtan yazmaca, bellekten yazmaca, yığına ya da bir G/Ç portuna. Bu komutların ortak özelliği ALU'dan bir hesap sonucu geçirmemeleridir. Bu yüzden bayrakları *değiştirmezler*. İki istisna, bayrakları doğrudan yazmak için tasarlanmış SAHF ve POPF'tur. Tablo 11.15 bu sınıfın komutlarını özetler.

Komut	İşlem	Örnek	Kod
`MOV h, k`	h ← k (kopyalama)	`MOV BX, [SI]`	`8B 1C`
`PUSH k` / `POP h`	SP ← SP − 2, [SS:SP] ← k / h ← [SS:SP], SP ← SP + 2	`PUSH AX`	`50`
`XCHG a, b`	a ↔ b (biri yazmaç olmalı)	`XCHG AL, [BX]`	`86 07`
`IN` / `OUT`	AL/AX ↔ G/Ç portu	`IN AL, 60h`	`E4 60`
`XLAT`	AL ← [DS:BX + AL]	tablo çevirisi	`D7`
`LEA r, m`	r ← m'nin **adresi** (EA)	`LEA SI, [BX+DI+4]`	`8D 71 04`
`LDS` / `LES r, m`	r ← [m], DS/ES ← [m + 2] (uzak işaretçi)	`LES DI, uzak`	`C4 3E …`
`LAHF` / `SAHF`	AH ← FLAGS'ın düşük baytı / tersi		`9F` / `9E`
`PUSHF` / `POPF`	FLAGS'ı yığına at / yığından al		`9C` / `9D`

Tablo 11.15: Veri taşıma komutları.

MOV'un kodlama biçimleri yukarıda ayrıntılı işlendi. PUSH ve POP'un yazmaçlı biçimleri tek bayttır (50h + r, 58h + r); kesim yazmaçlarınınki de tek bayttır (PUSH ES = 06h, PUSH CS = 0Eh, PUSH SS = 16h, PUSH DS = 1Eh). XCHG iki işlenenin yerini değiştirir ve ara yazmaca gerek bırakmaz; AX ile bir yazmacın takası 90h + r biçiminde tek bayttır. IN ve OUT, G/Ç adres alanına erişen tek komutlardır. Port numarası 0–255 ise komutun içinde sabit olarak verilir (E4h, E5h, E6h, E7h), daha büyük port numaraları DX ile verilir (ECh, EDh, EEh, EFh). Bu komutlar yürütülürken 8086'nın M/IO̅ ucu 0'a çekilir ve adres yolu bellek yerine bir G/Ç portunu seçer.

LAHF (*load AH from flags*) bayrak yazmacının düşük baytını AH'ye kopyalar. Bu baytta SF (bit 7), ZF (bit 6), AF (bit 4), PF (bit 2) ve CF (bit 0) bulunur. SAHF bunun tersini yapar ve AH'deki bu beş biti bayraklara yazar. İki komut, 8080 işlemcisinden aktarılan programların bayrak işlemlerini kolaylaştırmak için konmuştur. OF, DF, IF ve TF'ye ise yalnız PUSHF/POPF ile toplu olarak erişilir.

11.5.2 XLAT: tablo çevirisi

Birçok program bir değeri tablodan başka bir değere çevirir: bir onaltılık basamağı ASCII karakterine, bir tuş kodunu bir harfe, bir sayıyı yedi bölütlü göstergenin bit desenine. Bu işin genel yolu, değeri bir indis yazmacına koyup tablodan okumaktır. 8086 bu sık işe tek baytlık özel bir komut ayırmıştır.

Tanım 11.3

XLAT (*translate*), AL'deki değeri bir **çeviri tablosunda** indis olarak kullanır: AL ← [DS:BX + AL]. BX tablonun başlangıç adresini tutar; AL işaretsiz bir sayı (0–255) olarak eklenir. Tablo en çok 256 bayt olabilir. Komut tek bayttır (D7h), bayrakları değiştirmez ve 11 saat turu sürer.

XLAT'ın etkin adres hesabı sıradan adresleme biçimlerinin hiçbirine uymaz: 8 bitlik AL, 16 bitlik BX'e eklenir. Bu yüzden XLAT'ın ModR/M baytı yoktur ve işlenenleri örtüktür. Bazı assembler'lar `XLAT tablo` ya da `XLATB` yazımını da kabul eder; işlenen yalnız belgeleme içindir ve kodu değiştirmez. Tek fark, işlenen başka bir kesimdeyse (`XLAT ES:tablo`) bir kesim önekinin eklenmesidir. BX'e tablonun adresini koymak programcının işidir.

11.5.3 Kod izleme: XLAT ile baytı onaltılık yazdırma

Aşağıdaki program, bellekteki bir baytı iki onaltılık basamak olarak ekrana yazar. Bayt 3Ch'dir; ekranda “3C” görülmelidir. Bir bayt iki yarıma (dörtlüye) ayrılır; her yarım 0–15 arasında bir sayıdır ve XLAT ile '0'...'F' karakterlerinden birine çevrilir.

```
org 100h
mov bx, offset tablo
mov al, sayi       ; AL = 3Ch
mov cl, 4
shr al, cl         ; üst dörtlü
xlat               ; AL = [BX+AL]
mov dl, al
mov ah, 2
int 21h            ; DL'yi yaz
mov al, sayi
and al, 0Fh        ; alt dörtlü
xlat
mov dl, al
int 21h
ret
sayi  db 3Ch
tablo db '0123456789ABCDEF'
```

Program 0100h'den başlar. Komutların toplam uzunluğu 28 bayttır; bu yüzden `sayi` 011Ch'ye, `tablo` 011Dh'ye yerleşir. Tablo 11.16 her komuttan sonraki durumu verir.

Adres	Kod	Komut	AL	AH	BX	CL	DL	Çıktı
0100h	`BB 1D 01`	`MOV BX, OFFSET tablo`	00h	00h	**011Dh**	00h	00h	
0103h	`A0 1C 01`	`MOV AL, sayi`	**3Ch**	00h	011Dh	00h	00h	
0106h	`B1 04`	`MOV CL, 4`	3Ch	00h	011Dh	**04h**	00h	
0108h	`D2 E8`	`SHR AL, CL`	**03h**	00h	011Dh	04h	00h	

Adres	Kod	Komut	AL	AH	BX	CL	DL	Çıktı
010Ah	`D7`	`XLAT`	**33h** ('3')	00h	011Dh	04h	00h	
010Bh	`8A D0`	`MOV DL, AL`	33h	00h	011Dh	04h	**33h**	
010Dh	`B4 02`	`MOV AH, 2`	33h	**02h**	011Dh	04h	33h	
010Fh	`CD 21`	`INT 21h`	33h	02h	011Dh	04h	33h	**3**
0111h	`A0 1C 01`	`MOV AL, sayi`	**3Ch**	02h	011Dh	04h	33h	3
0114h	`24 0F`	`AND AL, 0Fh`	**0Ch**	02h	011Dh	04h	33h	3
0116h	`D7`	`XLAT`	**43h** ('C')	02h	011Dh	04h	33h	3
0117h	`8A D0`	`MOV DL, AL`	43h	02h	011Dh	04h	**43h**	3
0119h	`CD 21`	`INT 21h`	43h	02h	011Dh	04h	43h	**3C**
011Bh	`C3`	`RET`	43h	02h	011Dh	04h	43h	3C

Tablo 11.16: XLAT programının izlemesi. Her satır komut yürütüldükten sonraki durumdur; başlangıçta bütün yazmaçlar 0'dır.

İlk satır BX'e tablonun bağıl konumunu yükler: 011Dh. `MOV AL, sayi` AL için kısa biçimi (A0h) kullanır ve doğrudan adres 011Ch'yi okur. 8086'da 1'den büyük kaydırma sayısı yalnız CL ile verilebildiğinden CL'ye 4 konur. `SHR AL, CL`, 0011 1100b'yi dört bit sağa kaydırır: 0000 0011b = 03h. Dışarı çıkan son bit 1 olduğundan CF = 1 olur; kaydırma sayısı 1'den büyük olduğu için OF tanımsızdır. Bu bayraklar programın akışını etkilemez.

Birinci XLAT, AL = 03h'yi indis olarak kullanır. Okunan adres DS:(011Dh + 03h) = DS:0120h'dir; orada tablonun dördüncü karakteri '3' (33h) durur. AL artık 33h'dir. DL'ye kopyalanır, AH'ye 02h konur ve INT 21h çağrılır. DOS'un 02h hizmeti DL'deki karakteri yazar ve AL'ye de aynı değeri koyar; ekranda “3” görülür.

İkinci yarıda bayt yeniden okunur ve `AND AL, 0Fh` üst dörtlüyü siler: 3Ch AND 0Fh = 0Ch. AND, CF ile OF'yi 0 yapar. İkinci XLAT DS:(011Dh + 0Ch) = DS:0129h'yi okur; orada 'C' (43h) vardır. AH hâlâ 02h olduğundan yeniden yüklenmez; DL = 43h ile INT 21h çağrılır ve ekranda “3C” görülür. RET, programı PSP'deki INT 20h üzerinden bitirir.

Bu programda XLAT'ın kazancı açıktır. Aynı çeviri `MOV SI, AX` benzeri hazırlık ve `MOV AL, tablo[SI]` gibi bir komutla da yapılabilirdi. Ama AL'yi 16 bitlik bir indis yazmacına genişletmek ek komut ister. XLAT bu işi tek bayt ve 11 turda yapar. Tablo yönteminin bir başka yararı, karşılaştırma ve dallanma gerektirmemesidir: “basamak 10'dan küçükse '0' ekle, değilse 'A' − 10 ekle” mantığı yerine tek bir tablo okuması yeter.

Örnek 11.5

DS = 1000h, BX = 0200h ve AL = 05h iken XLAT hangi fiziksel adresi okur? Etkin adres BX + AL = 0200h + 05h = 0205h'dir. Fiziksel adres 1000h × 10h + 0205h = 10000h + 0205h = **10205h**'dir. O adresteki bayt AL'ye yüklenir; BX değişmez.

Sık yapılan hata

XLAT'tan önce AH'nin temizlenmesi gerektiğini sanmak ya da tam tersine, AL'deki fazla bitleri temizlemeyi unutmak. XLAT yalnız AL'yi kullanır; AH'ye bakmaz. Ama AL'nin bütün sekiz biti indise katılır. Yukarıdaki programda `AND AL, 0Fh` unutulsaydı indis 3Ch = 60 olur ve tablonun çok ötesindeki bir bayt okunurdu.

11.5.4 LEA: adres mi, içerik mi?

Bir bellek işleneninin iki yönü vardır: adresi ve o adreste duran içerik. MOV içeriği okur. Bazen ise yalnız adres gerekir: bir dizginin başlangıcını SI'ya koymak, bir dizinin elemanının adresini bir prosedüre vermek. 8086 bunun için ayrı bir komut sunar.

Tanım 11.4

LEA (*load effective address*), bellek işleneninin etkin adresini hesaplar ve hedef yazmaca yazar: r16 ← EA. Belleğe erişmez ve bayrakları değiştirmez. Kodu 8Dh + ModR/M'dir; süresi 2 + EA saat turudur.

Adresi derleme anında belli olan bir değişken için `LEA SI, metin` ile `MOV SI, OFFSET metin` aynı sonucu verir. emu8086 doğrudan adresli LEA'yı bu yüzden `MOV r16, imm16` ile aynı üç baytlık koda derler: `LEA BX, veri` = BB 14 01. Gerçek MASM aynı satırı 8D 1E ile başlayan dört baytlık bir kodla derler; aradaki bir bayt, arkadan gelen değişkenlerin adreslerini de bir kaydırır. LEA'nın asıl gücü, adresi çalışma anında yazmaçlardan hesaplanan işlenenlerde ortaya çıkar. `LEA SI, [BX+DI+4]` komutu, BX ve DI'nın o anki değerleriyle SI = BX + DI + 4 yapar (8D 71 04). OFFSET işleci bunu yapamaz, çünkü değeri derleme anında hesaplanan bir sabittir.

LEA'nın belleğe erişmemesi ve bayrak değiştirmemesi onu bir toplama aracı da yapar. `LEA AX, [BX+SI]`, AX = BX + SI işlemini tek komutta yapar; BX ve SI değişmez, bayraklar korunur. `ADD` ile aynı iş iki komut ister (`MOV AX, BX` / `ADD AX, SI`) ve bayrakları bozar. Bu özellik, bir CMP ile koşullu atlama arasında bir adresi artırmak gerektiğinde işe yarar.

```
mov bx, 0010h
mov di, 0004h
lea ax, [bx+di+2]  ; AX = 0016h (adres)
mov ax, [bx+di+2]  ; AX = [0016h] (içerik)
```

Bu iki satır aynı işleneni kullanır ve aynı etkin adresi (0010h + 0004h + 2 = 0016h) hesaplar. LEA bu sayıyı AX'e yazar ve durur. MOV ise BIU'ya DS:0016h'den bir sözcük okutur ve okunan değeri AX'e yazar. Makine kodları da yalnız işlem kodunda ayrılır: 8D 41 02 ve 8B 41 02.

11.5.5 LDS ve LES: uzak işaretçi

Bağıl konum tek başına bir bellek yerini belirtmeye yetmez; kesim de gerekir. Hem kesimi hem bağıl konumu tutan dört baytlık değere **uzak işaretçi** (*far pointer*) denir. Uzak işaretçi bellekte belirli bir sırayla saklanır: önce bağıl konum (düşük sözcük), sonra kesim (yüksek sözcük). Her sözcük de kendi içinde little-endian'dır. B800:0000 adresini gösteren bir uzak işaretçi bellekte 00 00 00 B8 baytları olarak durur.

> **Tanım 11.5**
>
> **LDS r16, m32** bellekteki uzak işaretçiyi yükler: r16 ← [m], DS ← [m + 2]. **LES r16, m32** aynı işi ES ile yapar. İşlem kodları C5h (LDS) ve C4h (LES), ardından ModR/M gelir; ikinci işlenen bellek olmak zorundadır. Bayraklar değişmez; süre 16 + EA saat turudur.

LDS ve LES, iki sözcüğü tek komutta okur. Aynı işi MOV ile yapmak iki komut ister: biri bağıl konumu genel yazmaca, öteki kesimi kesim yazmacına yükler. Dizgi komutlarıyla birlikte kullanımları özellikle yaygındır: LDS SI kaynağın, LES DI hedefin tam adresini tek komutta hazırlar. Ekran belleği gibi programın kendi kesimi dışındaki bir alana yazmak için tipik kalıp `LES DI, ekran` ve ardından `STOSB` ya da `STOSW`'dir.

> **Örnek 11.6**
>
> Bellekteki `uzak dd` değişkeninin baytları 34 12 00 B8 olsun. `LES DI, uzak` sonrası ilk sözcük (34 12 → 1234h) DI'ya, ikinci sözcük (00 B8 → B800h) ES'ye yüklenir: **DI = 1234h, ES = B800h**. ES:DI artık fiziksel B8000h + 1234h = B9234h adresini gösterir. Baytlar 00 20 00 10 olsaydı `LDS SI, …` ile SI = 2000h, DS = 1000h olur ve DS:SI 12000h'yi gösterirdi.

11.5.6 Kod izleme: LEA, MOV ve LES

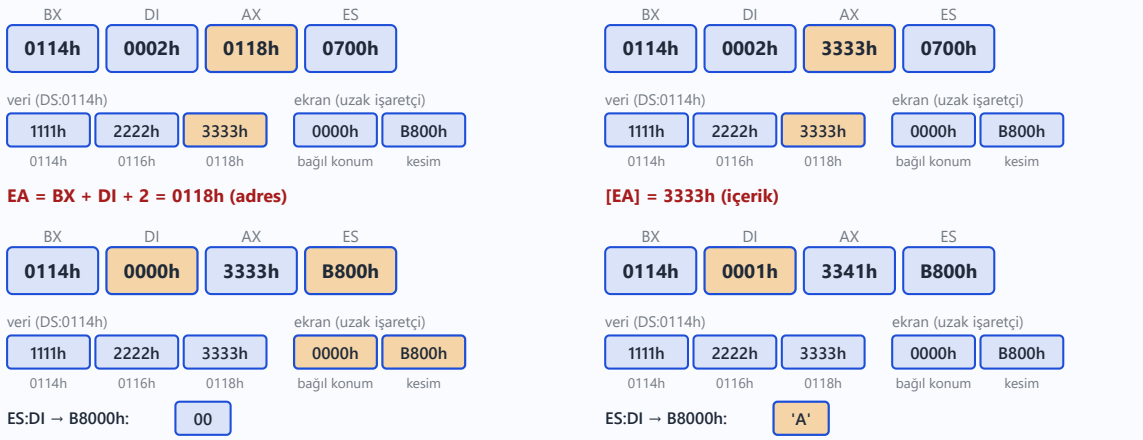
Aşağıdaki program LEA ile MOV'un farkını ve LES ile uzak işaretçinin yüklenmesini bir arada gösterir. Program sonunda ekranın sol üst köşesine 'A' yazar.

```
org 100h
lea bx, veri        ; BX ← adres
mov di, 2
lea ax, [bx+di+2]   ; AX ← EA
mov ax, [bx+di+2]   ; AX ← [EA]
les di, ekran       ; uzak işaretçi
mov al, 'A'
stosb               ; ES:[DI] ← AL
ret
veri  dw 1111h, 2222h, 3333h
ekran dw 0000h, 0B800h
```

Komutların uzunlukları 3 + 3 + 3 + 3 + 4 + 2 + 1 + 1 = 20 bayttır. Program 0100h'den başladığından `veri` 0114h'ye, `ekran` 0114h + 6 = 011Ah'ye yerleşir. Tablo 11.17 izlemeyi, Şekil 11.5 ise LEA ile MOV'un aynı EA'dan farklı şeyler aldığını gösterir.

Adres	Kod	Komut	AX	BX	DI	ES	Bellek
0100h	`BB 14 01`	`LEA BX, veri`	0000h	**0114h**	0000h	0700h	
0103h	`BF 02 00`	`MOV DI, 2`	0000h	0114h	**0002h**	0700h	
0106h	`8D 41 02`	`LEA AX, [BX+DI+2]`	**0118h**	0114h	0002h	0700h	okunmaz
0109h	`8B 41 02`	`MOV AX, [BX+DI+2]`	**3333h**	0114h	0002h	0700h	DS:0118h okunur
010Ch	`C4 3E 1A 01`	`LES DI, ekran`	3333h	0114h	**0000h**	**B800h**	DS:011Ah–011Dh okunur
0110h	`B0 41`	`MOV AL, 'A'`	**3341h**	0114h	0000h	B800h	
0112h	`AA`	`STOSB`	3341h	0114h	**0001h**	B800h	[B8000h] ← 41h
0113h	`C3`	`RET`	3341h	0114h	0001h	B800h	

Tablo 11.17: LEA, MOV ve LES programının izlemesi. Başlangıçta CS = DS = ES = SS = 0700h.



Şekil 11.5: LEA ve LES programının dört anı. Üst sol: LEA AX, [BX+DI+2] yalnız adresi (0118h) verir. Üst sağ: aynı EA ile MOV içeriği (3333h) okur. Alt sol: LES DI, ekran uzak işaretçiyi yükler (DI = 0000h, ES = B800h). Alt sağ: STOSB 'A'yı B8000h'ye yazar ve DI'yı 1 artırır.

Birinci komut, emu8086'da üç baytlık `MOV BX, 0114h` olarak derlenmiştir; BX = 0114h olur ve bellek okunmaz. `MOV DI, 2`'den sonra iki komut aynı işleneni kullanır. EA = BX + DI + 2 = 0114h + 0002h + 0002h = 0118h. LEA bu sayıyı AX'e yazar. MOV ise DS:0118h'deki sözcüğü okur: `veri`'nin üçüncü elemanı olan 3333h. Fark, makine kodunda yalnız işlem kodundadır (8Dh ve 8Bh); ModR/M (41h = 01 000 001: [BX+DI+d8], REG = AX) ve yer değiştirme (02h) aynıdır.

`LES DI, ekran` komutunun ModR/M baytı 3Eh = 00 111 110b'dir: REG = 111 (DI), MOD = 00 ve R/M = 110 (doğrudan adres 011Ah). İşlemci DS:011Ah'den dört bayt okur: 00 00 00 B8. İlk sözcük 0000h DI'ya, ikincisi B800h ES'ye gider. Bu noktada DI'nın önceki değeri (2) kaybolur. `MOV AL, 'A'` yalnız

AL'yi değiştirir; AH'de önceki MOV'dan kalan 33h durur ve AX = 3341h olur. STOSB, AL'yi ES:DI adresine yazar. Fiziksel adres B800h × 10h + 0000h = B8000h'dir; bu, metin modunda ekranın sol üst köşesindeki karakter hücresidir. DF = 0 olduğundan DI bir artar.

> **Sık yapılan hata**
>
> LDS ile DS'yi değiştirdikten sonra programın kendi değişkenlerine erişmeye devam etmek. `LDS SI, ptr` sonrasında DS başka bir kesimi gösterir; `MOV AL, sayac` gibi bir komut artık yanlış kesimden okur. Çözüm, DS'yi LDS'den önce PUSH DS ile saklayıp iş bitince POP DS ile geri almak ya da hedef için ES:DI kullanan LES'i tercih etmektir.

11.6 Aritmetik, mantık ve kaydırma komutları: özet

Aritmetik ve mantık komutları ayrıntılarıyla ayrı bir konudur; burada yalnız komut kümesi içindeki yerleri ve kodlamalarının ortak düzeni özetlenir. Tablo 11.18 grupları verir.

Grup	Komutlar	Açıklama
Toplama	`ADD ADC INC`	ADC eldeyi (CF) de ekler; INC CF'yi değiştirmez
Çıkarma	`SUB SBB DEC NEG CMP`	SBB ödüncü (CF) de çıkarır; NEG = 0 − h; CMP sonucu yazmaz
Çarpma	`MUL IMUL`	işaretsiz / işaretli; örtük AL ya da AX
Bölme	`DIV IDIV`	işaretsiz / işaretli; bölüm ve kalan
BCD düzeltme	`DAA DAS AAA AAS AAM AAD`	ikili sonucu onluğa düzeltir
İşaret genişletme	`CBW CWD`	AL → AX, AX → DX:AX
Mantık	`AND OR XOR NOT TEST`	bit temizleme, kurma, tersleme; TEST sonucu yazmaz
Kaydırma / döndürme	`SHL SAL SHR SAR ROL ROR RCL RCR`	kaydırma sayısı 1 ya da CL; çıkan bit CF'ye

Tablo 11.18: Aritmetik, mantık ve kaydırma komutları.

İki işlenenli sekiz aritmetik-mantık komutu (ADD, OR, ADC, SBB, AND, SUB, XOR, CMP) aynı kodlama kalıbını paylaşır. İşlem kodunun bit 5–3'ü hangi işlemin yapılacağını seçer: ADD = 000, OR = 001, ADC = 010, SBB = 011, AND = 100, SUB = 101, XOR = 110, CMP = 111. Yazmaç–r/m biçimleri *00ooo0dw* düzenindedir: ADD için 00–03h, SUB için 28–2Bh, CMP için 38–3Bh. Anlık değerli biçimler ise 80h, 81h ve 83h'tir ve aynı üç bitlik işlem numarası bu kez ModR/M'nin REG alanına yazılır. Böylece `SUB BX, 5` = 83 EB 05 (11 101 011: REG = 101 = SUB) ve `CMP BX, 5` = 83 FB 05 olur. Bu düzen bir kez görülünce, sekiz komutun yirmi dört biçimi tek bir tabloya iner.

İşlemci toplama ve çıkarmayı tek bir toplayıcı devresiyle yapar ve sayının işaretli mi işaretsiz mi olduğunu bilmez. Bu yüzden her iki yorum için bayrak üretir: CF işaretsiz taşmayı, OF işaretli taşmayı gösterir. Hangi bayrağın anlamlı olduğuna programcı karar verir. Bu komutların hangi bayrağı nasıl değiştirdiği notun sonundaki başvuru tablosunda toplanmıştır.

11.7 Dizgi komutları

11.7.1 Dizgi komutları neden var?

Bellekte ardışık duran baytlar ya da sözcükler üzerinde aynı işi tekrar tekrar yapmak programlarda çok sık görülür. Örnekler: bir metni bir yerden ötekine kopyalamak, iki parolayı karşılaştırmak, bir metinde bir karakter aramak, ekranı boşlukla doldurmak. Bu işleri MOV, CMP ve INC ile de yapmak mümkündür. Ama her eleman için birkaç komut gerekir: kaynaktan oku, hedefe yaz, iki adresi artır, sayacı azalt, döngünün başına dön. Üstelik MOV bellekten belleğe kopyalayamadığı için her eleman bir ara yazmaçtan geçer.

8086 bu tür işler için beş **dizgi komutu** (*string instructions*) sunar. Bu komutların işlenenleri örtüktür ve komutun içinde yazılmaz. Kaynak her zaman DS:SI, hedef her zaman ES:DI'dır. Komut, işini yaptıktan sonra SI'yı ve DI'yı kendiliğinden bir sonraki elemana ilerletir. Bu sayede her komut tek bayttır ve bir döngünün bütün adres işlerini kendi içinde halleder. Bir yineleme önekiyle birlikte kullanıldığında, tek bir iki baytlık komut binlerce baytlık bir bloğu işleyebilir.

Tanım 11.6

Dizgi komutu, DS:SI ile gösterilen kaynakta ve/veya ES:DI ile gösterilen hedefte bir bayt (B sürümü) ya da bir sözcük (W sürümü) işleyen komuttur. Komut işini bitirince kullandığı indis yazmaçlarını yön bayrağına göre 1 ya da 2 artırır veya azaltır. Kaynağın kesimi bir kesim önekiyle değiştirilebilir; hedef her zaman ES'tedir.

11.7.2 Beş dizgi komutu

Tablo 11.19 beş dizgi komutunu, yaptıkları işi, güncelledikleri yazmaçları, bayrak etkilerini, işlem kodlarını ve tek başına yürütüldüklerindeki sürelerini verir.

Komut	İşlem	Güncellenen	Bayrak	Kod	Süre
`MOVSB / MOVSW`	[ES:DI] ← [DS:SI]	SI, DI	–	`A4` / `A5`	18
`CMPSB / CMPSW`	[DS:SI] − [ES:DI]	SI, DI	CMP gibi	`A6` / `A7`	22
`SCASB / SCASW`	AL/AX − [ES:DI]	DI	CMP gibi	`AE` / `AF`	15
`LODSB / LODSW`	AL/AX ← [DS:SI]	SI	–	`AC` / `AD`	12
`STOSB / STOSW`	[ES:DI] ← AL/AX	DI	–	`AA` / `AB`	11

Tablo 11.19: Dizgi komutları. Kod sütununda bayt / sözcük sürümü; süre tek yürütme için saat turudur.

Komutları iki gruba ayırmak kolaylık sağlar. MOVS, LODS ve STOS veri taşır ve bayraklara dokunmaz. CMPS ve SCAS ise karşılaştırır: sonucu bir yere yazmadan bir çıkarma yapar ve bayrakları CMP gibi kurar. Bütün işlem kodları 1010 xxxw düzenindedir; son bit W'dir ve bayt ile sözcük sürümünü ayırır.

Kodların yalnız tek bayt olması, dizgi komutlarının neden bu kadar sık kullanıldığını da açıklar: önden getirme kuyruğu tek okumada birkaç dizgi komutunu birden alabilir.

Donanım açısından bakınca dizgi komutları BIU'yu yoğun kullanır. MOVSB bir bellek okuma ve bir bellek yazma yol döngüsü gerektirir; her döngü en az dört saat turu sürer. CMPSB iki okuma, SCASB ve LODSB birer okuma, STOSB bir yazma yapar. MOVSW çift adreslerde aynı iki döngüde iki bayt taşır; bu yüzden büyük blokları sözcük sürümüyle kopyalamak yaklaşık iki kat hızlıdır. Kaynak ya da hedef tek adresteyse her sözcük iki bayt döngüsüne bölünür ve bu kazanç azalır.

Sözdizimi konusunda bir not: MASM ve emu8086, işlenenli yazımı da kabul eder. `MOVS hedef, kaynak` yazıldığında assembler değişkenlerin türüne bakarak MOVSB ya da MOVSW üretir. Kaynak işlenene bir kesim öneki yazılmışsa (`MOVS hedef, ES:kaynak`) o önek koda eklenir. İşlenenler yalnız boyutu ve öneki belirler; adresleri SI ve DI'ya yüklemek yine programcının işidir.

11.7.3 Yön bayrağı ve adım

Dizgi komutunun SI ve DI'yı artırması mı, azaltması mı gerektiğini **yön bayrağı** (*direction flag*, DF) belirler. DF, bayrak yazmacının bit 10'udur ve bir kontrol bayrağıdır: aritmetik komutlar onu değiştirmez. İki komutla kurulur:

- `CLD` (*clear direction flag*, FCh): DF = 0. Dizgi komutları SI ve DI'yı **artırır**; işlem düşük adresten yükseğe ilerler.
- `STD` (*set direction flag*, FDh): DF = 1. Dizgi komutları SI ve DI'yı **azaltır**; işlem yüksek adresten düşüğe ilerler.

Adım büyüklüğü komutun sürümüne bağlıdır: B sürümünde ±1, W sürümünde ±2. Güncelleme her zaman işlemden *sonra* yapılır. Bu yüzden bir dizgi komutu bittikten sonra SI ve DI, işlenmiş son elemanı değil, bir sonrakini gösterir. Bir arama komutu bir eşleşme bulduğunda eşleşmenin adresi DI − 1 (bayt, DF = 0) olarak bulunur.

> **Sık yapılan hata**
>
> DF'nin değerini varsaymak. DOS programı başlatırken DF'yi genellikle 0 bırakır, ama başka bir prosedür, bir kütüphane ya da bir kesme işleyicisi DF'yi 1 yapmış olabilir. Dizgi komutu kullanan her kod parçası, işe başlamadan önce yönü açıkça `CLD` ya da `STD` ile kurmalıdır. STD kullanan bir prosedür de dönmeden önce `CLD` ile DF'yi geri almalıdır.

Dizgi komutlarının kaynağı ve hedefi farklı kesimlerde olabilir. .COM programlarında CS, DS, ES ve SS aynı kesimi gösterir; DS:SI ve ES:DI aynı belleğe bakar. .EXE programlarında DOS, ES'yi programın veri kesimine değil PSP'ye ayarlar. Bu programlarda DS gibi ES de açıkça kurulmalıdır:

```
mov ax, @data
mov ds, ax
mov es, ax       ; dizgi hedefi için
```

11.7.4 Kod izleme: MOVSB ile kopyalama

Aşağıdaki program dört baytlık 'ABCD' metnini, nokta ile dolu dört baytlık bir alana kopyalar. Kopyalama bir LOOP döngüsüyle, her turda tek bir MOVSB ile yapılır; böylece komutun her yürütmesi ayrı ayrı izlenebilir.

```
org 100h
cld
lea si, kaynak     ; DS:SI kaynak
lea di, hedef      ; ES:DI hedef
mov cx, 4
kopya:
movsb              ; [DI]←[SI], SI++, DI++
loop kopya         ; = rep movsb
ret
kaynak db 'ABCD'
hedef  db 4 dup('.')
```

Kodun uzunluğu 14 bayttır: CLD (1), iki LEA (3 + 3), MOV CX (3), MOVSB (1), LOOP (2), RET (1). `kaynak` bu yüzden 010Eh'de, `hedef` 0112h'de başlar. LOOP'un kodu E2 FD'dir: FDh = −3, sonraki komutun adresi 010Dh'den 3 geri, yani MOVSB'nin adresi 010Ah'dir.

Adım	Komut	SI	DI	CX	hedef	Açıklama
1	`CLD`	0000h	0000h	0000h	`....`	DF = 0: ileri yön
2	`LEA SI, kaynak`	**010Eh**	0000h	0000h	`....`	BE 0E 01
3	`LEA DI, hedef`	010Eh	**0112h**	0000h	`....`	BF 12 01
4	`MOV CX, 4`	010Eh	0112h	**0004h**	`....`	sayaç
5	`MOVSB`	**010Fh**	**0113h**	0004h	`A...`	[0112h] ← [010Eh] = 41h
6	`LOOP kopya`	010Fh	0113h	**0003h**	`A...`	CX ≠ 0 → atla
7	`MOVSB`	**0110h**	**0114h**	0003h	`AB..`	[0113h] ← 42h
8	`LOOP kopya`	0110h	0114h	**0002h**	`AB..`	atla
9	`MOVSB`	**0111h**	**0115h**	0002h	`ABC.`	[0114h] ← 43h
10	`LOOP kopya`	0111h	0115h	**0001h**	`ABC.`	atla
11	`MOVSB`	**0112h**	**0116h**	0001h	`ABCD`	[0115h] ← 44h
12	`LOOP kopya`	0112h	0116h	**0000h**	`ABCD`	CX = 0 → düş
13	`RET`	0112h	0116h	0000h	`ABCD`	program biter

Tablo 11.20: MOVSB döngüsünün izlemesi. Bayraklar hiçbir adımda değişmez.

Her MOVSB adımında iki şey olur. BIU önce DS:SI'daki baytı okur, sonra onu ES:DI'ya yazar; bayt hiçbir genel yazmaçtan geçmez. Ardından SI ve DI birer artar. Son MOVSB'den sonra SI = 0112h olur. Bu, kaynağın hemen arkasındaki bayttır ve tesadüfen `hedef`'in başlangıcıdır. DI = 0116h ise hedefin bir ötesini gösterir. Bayrak yazmacı baştan sona değişmez. LOOP da bayraklara dokunmaz; CX'i azaltıp sıfır değilse atlar.

Aynı iş `MOV CX, 4` ve `REP MOVSB` ile tek komutta yapılır ve sonuç yazmaçları aynı olur: SI = 0112h, DI = 0116h, CX = 0. Fark süredir. LOOP'lu sürüm her bayt için 18 tur MOVSB ve atlayan LOOP için 17 tur harcar; son LOOP atlamadığı için 5 tur sürer. Dört bayt için 4 × 18 + 3 × 17 + 5 = 128 tur eder. REP MOVSB ise 9 + 17 × 4 = 77 tur sürer. Fark, blok büyüdükçe iki katına yaklaşır.

11.7.5 Kod izleme: LODSB ve STOSB ile büyük harfe çevirme

LODS ile STOS birlikte kullanıldığında bir dizgi okunur, her eleman bir yazmaçta işlenir ve sonuç yazılır. Bu kalıp, bir diziyi yerinde dönüştürmenin tipik yoludur. Aşağıdaki program 'mikro' metnini büyük harfe çevirir. ASCII'de küçük harf ile büyük harf arasındaki fark yalnız bit 5'tir ('m' = 6Dh = 0110 1101b, 'M' = 4Dh = 0100 1101b). `AND AL, 0DFh` bu biti temizler.

```
org 100h
cld               ; DF = 0: ileri
lea si, metin
mov di, si        ; aynı yere yaz
mov cx, 5
don:
lodsb             ; AL = [SI], SI++
and al, 0DFh      ; büyük harf
stosb             ; [DI] = AL, DI++
loop don
ret
metin db 'mikro'
```

Kod 16 bayttır ve `metin` 0110h'ye yerleşir. `MOV DI, SI` (8B FE) hedefi kaynakla aynı yere koyar. Bir .COM programında ES = DS olduğu için DS:SI ve ES:DI aynı baytı gösterir. LODSB okuma adresini bir ilerletir, STOSB bir adım arkadan gelerek aynı baytın üzerine yazar. Tablo 11.21 ilk turu ayrıntılı, sonraki turları özet olarak verir.

Komut	AL	SI	DI	CX	metin
`CLD`, `LEA SI, metin`	00h	0110h	0000h	0000h	`mikro`
`MOV DI, SI`, `MOV CX, 5`	00h	0110h	0110h	0005h	`mikro`
`LODSB`	**6Dh** ('m')	**0111h**	0110h	0005h	`mikro`
`AND AL, 0DFh`	**4Dh** ('M')	0111h	0110h	0005h	`mikro`
`STOSB`	4Dh	0111h	**0111h**	0005h	`Mikro`
`LOOP don`	4Dh	0111h	0111h	**0004h**	`Mikro`

Komut	AL	SI	DI	CX	metin
2. tur sonu	49h ('I')	0112h	0112h	0003h	`MIkro`
3. tur sonu	4Bh ('K')	0113h	0113h	0002h	`MIKro`
4. tur sonu	52h ('R')	0114h	0114h	0001h	`MIKRo`
5. tur sonu	4Fh ('O')	0115h	0115h	0000h	`MIKRO`

Tablo 11.21: LODSB / STOSB döngüsünün izlemesi.

Bu programda bayrakları yalnız AND değiştirir: CF ve OF'yi 0, SF ile ZF'yi sonuca göre kurar, PF'yi sonucun eşliğine göre ayarlar. 4Dh = 0100 1101b'de dört tane 1 olduğundan ilk turda PF = 1, 49h = 0100 1001b'de üç tane 1 olduğundan ikinci turda PF = 0 olur. LODSB, STOSB ve LOOP bu bayrakları olduğu gibi bırakır. Program büyük harfle başlayan bir metinde de doğru çalışır: 'M' zaten bit 5'i 0 olan bir harftir ve AND onu değiştirmez. Ama harf olmayan karakterlerde dikkat gerekir. Örneğin bir rakam ('1' = 31h) bu işlemden 11h olarak çıkar; gerçek bir programda önce karakterin 'a'–'z' aralığında olup olmadığı sınanır.

Her turun süresi LODSB (12) + AND AL, imm8 (4) + STOSB (11) + LOOP (17) = 44 turdur. Son turda LOOP 5 tur sürdüğü için toplam 5 × 27 + 4 × 17 + 5 = 208 saat turu eder. Bu döngüye REP konamaz, çünkü REP yalnız tek bir dizgi komutunu yineler; arada bir AND işlemi gerektiği için döngü LOOP ile kurulur.

11.7.6 LODSW ile sözcük dizisinin toplamı

W sürümlerinde adım 2'dir. Aşağıdaki program dört sözcükten oluşan bir diziyi LODSW ile okuyup BX'te toplar.

```
org 100h
cld
lea si, dizi
mov cx, 4
xor bx, bx        ; toplam = 0
top:
lodsw             ; AX = [SI], SI += 2
add bx, ax
loop top
ret
dizi dw 100h, 200h, 300h, 400h
```

Tur	LODSW okunan adres	AX	SI	BX	CX (LOOP sonrası)
1	010Fh	0100h	0111h	0100h	0003h
2	0111h	0200h	0113h	0300h	0002h
3	0113h	0300h	0115h	0600h	0001h

Tur	LODSW okunan adres	AX	SI	BX	CX (LOOP sonrası)
4	0115h	0400h	0117h	**0A00h**	0000h

Tablo 11.22: LODSW toplamının izlemesi (her turun ADD sonrası durumu). `dizi` 010Fh'de başlar.

Kod 15 bayt tuttuğu için dizi tek adreste, 010Fh'de başlar. Her LODSW, SI'yı 2 artırır ve dizinin bir sonraki sözcüğüne geçer. Sonuç BX = 0A00h'dir (256 + 512 + 768 + 1024 = 2560 = 0A00h). Dizi tek adreste durduğu için her LODSW iki ayrı bayt yol döngüsü gerektirir. Dizinin önüne `EVEN` yönergesi konsaydı dizi 0110h'ye kayar ve her sözcük tek döngüde okunurdu.

11.7.7 CMPS ve SCAS: karşılaştırma

CMPS ve SCAS, CMP gibi bir çıkarma yapar, sonucu atar ve altı durum bayrağını kurar. İşlenenlerin sırası, sonuçtaki CF ve SF'nin yorumu için önemlidir:

- **CMPS**: [DS:SI] − [ES:DI]. Kaynak eksi hedeftir. 'R' (52h) ile 'S' (53h) karşılaştırıldığında 52h − 53h = FFh çıkar: CF = 1, SF = 1, ZF = 0. CF = 1, işaretsiz yorumla kaynağın hedeften küçük olduğunu gösterir.
- **SCAS**: AL (ya da AX) − [ES:DI]. Aranan değer eksi bellektir. AL = 'S' ve bellekte 'R' varken 53h − 52h = 01h olur: CF = 0, ZF = 0.

Bu sıra, Intel sözdiziminde hedefin solda yazıldığı genel kurala aykırı görünebilir: CMPS'de sol taraf kaynaktır. CMPS'den sonra JA, JB gibi işaretsiz atlamalar “kaynak, hedeften büyük/küçük” diye okunur. SCAS yalnız DI'yı ilerletir, çünkü kaynak işleneni bir yazmaçtır. CMPS ise MOVS gibi iki bellek dizgisini aynı anda adresler ve SI ile DI'yı birlikte ilerletir.

11.8 Yineleme önekleri

11.8.1 REP, REPE ve REPNE

Dizgi komutunun asıl gücü, bir **yineleme önekiyle** (*repeat prefix*) birlikte ortaya çıkar. Önek, ardından gelen dizgi komutunu CX sayacı sıfırlanana kadar ya da bir koşul bozulana kadar yineler. Tablo 11.23 üç öneki ve sürdürme koşullarını verir.

Önek	Bayt	Sürdürme koşulu	Kullanım
`REP`	`F3`	CX ≠ 0	MOVS, STOS (LODS ile anlamsız)
`REPE / REPZ`	`F3`	CX ≠ 0 ve ZF = 1	CMPS, SCAS: “eşitken sürdür”
`REPNE / REPNZ`	`F2`	CX ≠ 0 ve ZF = 0	CMPS, SCAS: “eşit değilken sürdür”

Tablo 11.23: Yineleme önekleri.

REP ile REPE aynı bayttır (F3h). İşlemci farkı önekten değil, arkasından gelen komuttan anlar: MOVS, LODS ve STOS bayrak üretmediği için ZF koşulu bunlarda sınanmaz; CMPS ve SCAS'ta sınanır. F2h'nin bit 0'ı 0, F3h'ninki 1'dir ve bu bit, sürdürme için ZF'nin hangi değeri alması gerektiğini söyler. Önekli bir dizgi komutunun yürütülmesi şu sırayı izler:

1. CX = 0 ise dur. Komut hiç yürütülmeyebilir.
2. Bekleyen bir kesme varsa onu kabul et (dönüş adresi önekin adresidir).
3. Dizgi komutunu bir kez yürüt; SI ve DI'yı güncelle.
4. CX ← CX − 1. Bu azaltma bayrakları değiştirmez.
5. REPE ya da REPNE ise ve komut CMPS ya da SCAS ise ZF'yi sına; koşul sağlanmıyorsa dur. Sağlanıyorsa 1. adıma dön.

İlk adımın önemi büyüktür. CX = 0 ile başlayan bir `REP MOVSB` hiç bayt kopyalamaz. LOOP ise bunun tersine davranır: CX = 0 iken girilen bir LOOP döngüsü önce CX'i FFFFh'ye indirir ve 65.536 tur döner. Bu fark, REP'in uzunluğu hesaplanan bir blokta güvenle kullanılabilmesini sağlar: uzunluk 0 çıkarsa hiçbir şey olmaz.

Tanım 11.7

Yineleme öneki, ardından gelen dizgi komutunu CX kez ya da bir ZF koşulu bozulana kadar yineleten önektir. Her yinelemede CX bir azalır. Komut bittiğinde CX, yapılmamış yineleme sayısını; SI ve DI işlenmiş son elemanın bir sonrasını gösterir.

11.8.2 Blok kopyalama ve ekranı doldurma

REP'in en yaygın iki kullanımı bir bellek bloğunu kopyalamak ve bir bloğu tek bir değerle doldurmaktır. Aşağıdaki ilk parça 100 baytı kopyalar, ikincisi ekranın tamamını boşlukla doldurur.

```
; 100 baytlık blok kopyalama
cld
lea si, kaynak
lea di, hedef
mov cx, 100
rep movsb

; ekranı boşlukla doldur
mov ax, 0B800h
mov es, ax
xor di, di
mov ax, 0720h   ; ' ' + öznitelik
mov cx, 2000    ; 80×25
rep stosw
```

İlk parçada REP MOVSB yüz kez yürür. Sonunda CX = 0'dır, SI ve DI başlangıç değerlerinden 100 (64h) fazladır ve kaynak bloğun bir kopyası hedeftedir. Bu parça bir programa yerleştirilip emu8086'da çalıştırıldığında, örneğin kaynak 010Dh'de ve hedef 0171h'de başlıyorsa, sonunda SI = 0171h ve DI = 01D5h görülür. 8086 kılavuzuna göre süre 9 + 17 × 100 = 1709 saat turudur. Aynı blok sözcük sürümüyle (`MOV CX, 50` / `REP MOVSW`) kopyalansaydı süre 9 + 17 × 50 = 859 tura iner; MOVSW'nin turu MOVSB'ninkiyle aynıdır ama her turda iki bayt taşır.

İkinci parça, metin modundaki ekran belleğini doldurur. Renkli metin ekranı B800:0000'dan başlar. Her karakter hücresi iki bayttır: çift adreste karakterin ASCII kodu, tek adreste öznitelik (renk) baytı. AX = 0720h'nin düşük baytı AL = 20h (boşluk), yüksek baytı AH = 07h'dir (siyah zemin üzerine açık gri). STOSW, AX'i little-endian olarak yazdığı için 20h çift adrese, 07h tek adrese düşer; yani her STOSW bir hücreyi tam olarak siler. 80 × 25 = 2000 hücre için CX = 2000'dir. Komut bittiğinde DI = 2000 × 2 = 4000 = 0FA0h, CX = 0 olur ve ekranın son hücresi B8F9Eh–B8F9Fh'de 20h, 07h değerlerini taşır. Süre 9 + 10 × 2000 = 20.009 turdur; 5 MHz'de yaklaşık 4 ms.

11.8.3 Kod izleme: REPNE SCASB ile karakter arama

REPNE SCASB, bir dizgide bir karakteri arar: “eşit değilken sürdür” koşulu, karakter bulunduğunda ya da CX bittiğinde durur. Aşağıdaki program 'merhaba' metninde 'h' harfini arar ve bulduğu konumu (0'dan sayarak) DI'ya koyar.

```
org 100h
cld
lea di, metin
mov cx, 7          ; uzunluk
mov al, 'h'        ; aranan
repne scasb        ; ara
jne yok
dec di             ; bulunan konum
sub di, offset metin
yok:
ret
metin db 'merhaba'
```

Kod 19 bayttır ve `metin` 0113h'de başlar. `REPNE SCASB` iki bayttır (F2 AE). Tablo 11.24 komutun dört yinelemesini ayrı ayrı gösterir. Her yinelemede AL ile ES:DI'daki bayt karşılaştırılır, DI bir artar ve CX bir azalır.

Komut	Karşılaştırma	DI	CX	ZF	CF	Açıklama
`CLD`		0000h	0000h	0	0	ileri yön
`LEA DI, metin`		**0113h**	0000h	0	0	BF 13 01
`MOV CX, 7`		0113h	**0007h**	0	0	en çok 7 karşılaştırma
`MOV AL, 'h'`		0113h	0007h	0	0	AL = 68h
`REPNE SCASB` (1)	68h – 'm' (6Dh)	0114h	0006h	0	1	eşit değil → sürdür

Komut	Karşılaştırma	DI	CX	ZF	CF	Açıklama
`REPNE SCASB` (2)	68h – 'e' (65h)	0115h	0005h	0	0	sürdür
`REPNE SCASB` (3)	68h – 'r' (72h)	0116h	0004h	0	1	sürdür
`REPNE SCASB` (4)	68h – 'h' (68h)	**0117h**	**0003h**	**1**	0	eşit → dur
`JNE yok`		0117h	0003h	1	0	ZF = 1 → atlama yok
`DEC DI`		**0116h**	0003h	0	0	eşleşmenin adresi
`SUB DI, OFFSET metin`		**0003h**	0003h	0	0	'h' 3. indiste

Tablo 11.24: REPNE SCASB programının izlemesi. REPNE SCASB'nin her yinelemesi ayrı satırda gösterilmiştir.

Dördüncü yinelemede 'h' bulunur: 68h − 68h = 0 ve ZF = 1 olur. REPNE'nin koşulu (ZF = 0) bozulduğu için komut durur. DI, eşleşmenin bir sonrasını (0117h) gösterir, çünkü güncelleme karşılaştırmadan sonra yapılmıştır. CX = 3, dizgide taranmadan kalan karakter sayısıdır. Komutun bitmesinin iki nedeni olabileceği için ardından bir sınama gerekir. ZF = 1 ise karakter bulunmuştur. ZF = 0 ise CX sıfırlanmış ve dizginin tamamı boşuna taranmıştır; o durumda `JNE yok` atlar. CX = 0 sınaması tek başına yetmez: aranan karakter tam son konumdaysa hem CX = 0 hem ZF = 1 olur.

`DEC DI`, DI'yı eşleşmenin kendisine geri götürür: 0116h. `SUB DI, OFFSET metin` başlangıç adresini çıkarır ve 0116h − 0113h = 3 bulunur: 'h', 0'dan sayınca 3. indistedir. DEC ve SUB bayrakları kendi sonuçlarına göre değiştirir; sonuç sıfır olmadığı için ZF = 0'a döner. Komutun süresi 9 + 15 × 4 = 69 saat turudur.

11.8.4 REPE CMPSB ile iki dizgiyi karşılaştırma

REPE CMPSB, iki dizgiyi “eşit oldukları sürece” karşılaştırır ve ilk farkta durur. Aşağıdaki program 'ANKARA' ile 'ANTALY' dizgilerinin ilk altı karakterini karşılaştırır ve ilk farkın indisini SI'da bırakır.

```
org 100h
cld
lea si, ad1
lea di, ad2
mov cx, 6
repe cmpsb
je ayni            ; tamamı eşit
dec si
sub si, offset ad1 ; ilk farkın indisi
ayni:
ret
ad1 db 'ANKARA'
ad2 db 'ANTALY'
```

Kod 20 bayttır; `ad1` 0114h'de, `ad2` 011Ah'de başlar. İlk yineleme 'A' ile 'A'yı karşılaştırır: fark 0, ZF = 1, SI = 0115h, DI = 011Bh, CX = 5. İkinci yineleme 'N' ile 'N'yi karşılaştırır: ZF = 1, CX = 4. Üçüncü yinelemede 'K' (4Bh) − 'T' (54h) = F7h çıkar. ZF = 0 olur ve REPE durur. Bu anda SI = 0117h, DI = 011Dh,

CX = 3'tür. Sonucun işaretsiz yorumu için CF'ye bakılır: CF = 1, yani 'K' < 'T'. Bu, sözlük sırasında ANKARA'nın ANTALY'dan önce geldiğini söyler. `JE ayni` atlamaz; `DEC SI` ve `SUB SI, OFFSET ad1` ile 0116h − 0114h = 2 bulunur: ilk fark 2. indistedir.

Burada da komutun bitmesinin iki nedeni vardır. Dizgiler tamamen eşit olsaydı altı yinelemenin hepsi ZF = 1 ile biter ve CX = 0 olurdu; o zaman `JE ayni` atlardı. Bir parola ya da komut adı karşılaştırması bu kalıpla yapılır.

11.8.5 Örtüşen bloklar ve yön seçimi

Kaynak ile hedef bellekte üst üste biniyorsa kopyalama yönü sonucu değiştirir. Bir metni iki bayt sağa kaydırmak istediğimizi düşünelim: `metin db '12345', 2 dup('.')`. Hedef, kaynağın iki bayt sağındadır ve iki blok üç baytta örtüşür.

```
std                 ; sondan başa
lea si, metin+4     ; kaynağın son baytı
lea di, metin+6     ; hedefin son baytı
mov cx, 5
rep movsb
```

STD ile kopyalama sondan başa ilerler: önce '5' 6. konuma, sonra '4' 5. konuma, '3' 4. konuma yazılır. Her kaynak bayt, üzerine bir şey yazılmadan önce okunur. Sonuç '1212345' olur: metin iki sağa kaymıştır ve ilk iki konumda eski karakterler kalmıştır. Aynı iş CLD ile, SI = metin ve DI = metin+2 verilerek yapılsaydı ilk adım '1'i 2. konuma, yani '3'ün üzerine yazardı. Üçüncü adım bu yeni '1'i okur ve ileri taşır. Sonuç '1212121' olur ve metin bozulur. Tablo 11.25 iki sonucu karşılaştırır.

Yön	Başlangıç SI / DI	Bitiş SI / DI	Sonuç
STD (doğru)	0111h / 0113h	010Ch / 010Eh	`1212345`
CLD (yanlış)	010Dh / 010Fh	0112h / 0114h	`1212121`

Tablo 11.25: '12345..' metninin iki bayt sağa kopyalanması. Program 0100h'den başladığında `metin` 010Dh'dedir.

> **Kural (örtüşen kopyalama)**
>
> Hedef kaynağın sağında (daha yüksek adreste) örtüşüyorsa kopyalama sondan başa yapılır: STD, SI ve DI blokların son elemanını gösterir. Hedef kaynağın solunda örtüşüyorsa baştan sona yapılır: CLD, SI ve DI blokların ilk elemanını gösterir. Bloklar örtüşmüyorsa iki yön de doğrudur.

11.8.6 Süre, kesmeler ve donanım

REP'li bir dizgi komutu, aynı işi yapan bir döngüden belirgin biçimde hızlıdır. Tablo 11.26 Intel 8086 kılavuzundaki sürelere göre birkaç karşılaştırma verir. Kazancın kaynağı, döngü denetiminin (CX azaltma, sınama, atlama) mikrokod içinde yapılması ve önden getirme kuyruğunun her turda boşaltılmamasıdır. Bir LOOP ya da JMP atladığında kuyruk boşaltılır ve BIU yeni adresten baştan okumak zorunda kalır; REP'li bir komut ise IP'yi hiç değiştirmez.

İş	Döngü ile	REP ile
4 bayt kopyalama	MOVSB + LOOP: 128	REP MOVSB: 77
100 bayt kopyalama	MOVSB + LOOP: 3488	REP MOVSB: 1709; REP MOVSW (50 sözcük): 859
2000 sözcük doldurma	–	REP STOSW: 20.009
7 baytta 4. konumda bulunan karakteri arama	–	REPNE SCASB: 69

Tablo 11.26: Dizgi işlerinin süreleri (saat turu). REP'li komut için süre 9 + n × tur başına süredir.

Uzun bir REP komutu binlerce tur sürebilir. 8086, bu süre boyunca kesmeleri bekletmez. Her yinelemenin başında bekleyen bir kesme olup olmadığına bakar. Varsa o ana kadarki işi bırakır; SI, DI ve CX o anki değerlerini korur. Yığına itilen dönüş adresi, önekin adresidir. Kesme işleyicisi IRET ile döndüğünde işlemci önekli komutu yeniden başlatır ve iş kalan CX değeriyle sürer. Bu düzen kesme gecikmesini kısa tutar ama yukarıda anlatılan sınırlamayı da getirir: 8086 yalnız bir öneki hatırladığından REP ile kesim öneki birlikte kullanılmamalıdır.

> **Sık yapılan hata**
>
> REP'i LODS ile kullanmak ya da REPE'yi MOVS ile kullanmak. `REP LODSB` geçerli bir koddur ama her yinelemede AL'nin üzerine yazar; sonunda AL'de yalnız son bayt kalır. `REPE MOVSB` ise REP MOVSB ile aynıdır, çünkü MOVS ZF üretmez. Önek ile komutun eşleşmesi: MOVS ve STOS için REP, CMPS ve SCAS için REPE ya da REPNE.

11.9 Kontrol aktarma: JMP'nin kodlaması

11.9.1 JMP türleri

Koşulsuz atlama JMP, IP'ye (uzak atlamada CS'ye de) yeni bir değer yükler. 8086'da JMP'nin beş kodlama biçimi vardır. Biçimler hedefin nasıl verildiğine göre ayrılır: komutun içinde bir uzaklık olarak, komutun içinde mutlak bir adres olarak ya da bir yazmaçta veya bellekte. Tablo 11.27 beşini ve sürelerini verir.

Tür	Kod	Erim	Örnek	Süre
Kısa (*short*)	`EB d8`	−128 … +127 bayt	`JMP SHORT dongu`	15
Yakın (*near*)	`E9 d16`	aynı kesim, ±32 K	`JMP basla`	15
Uzak (*far*)	`EA IP CS`	herhangi bir kesim	`JMP FAR PTR hedef`	15
Dolaylı yakın	`FF /4`	IP ← yazmaç ya da bellek	`JMP BX`, `JMP tablo[BX]`	11 / 18 + EA
Dolaylı uzak	`FF /5`	CS:IP ← bellek (4 bayt)	`JMP DWORD PTR [SI]`	24 + EA

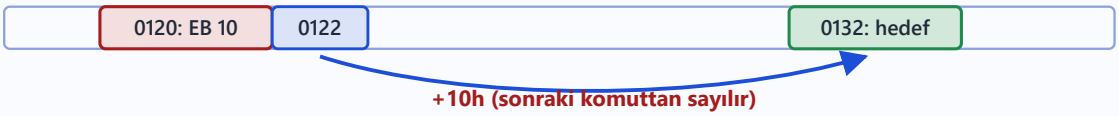
Tablo 11.27: JMP'nin kodlama biçimleri. Süreler saat turudur.

Kısa ve yakın atlamalar **göreli** kodlanır: komut hedefin adresini değil, hedefin bir sonraki komuta olan uzaklığını taşır. İşlemci bu uzaklığı IP'ye ekler. IP, komut yürütülürken zaten bir sonraki komutu gösterdiği için hesap şudur: hedef = sonraki komutun adresi + yer değiştirme. Kısa biçimde uzaklık tek bayttır ve işaretle 16 bite genişletilir. Uzak atlama ise hedefin mutlak adresini taşır: önce IP (2 bayt), sonra CS (2 bayt), ikisi de little-endian. Uzak JMP bu yüzden beş bayttır. `JMP FAR PTR uzak` komutu, hedef 0700:0256h ise EA 56 02 00 07 olarak kodlanır.

Dolaylı biçimler FFh ailesindendir ve hangi işlemin yapılacağını ModR/M'nin REG alanı söyler: /4 dolaylı yakın JMP, /5 dolaylı uzak JMP. `JMP BX` = FF E3 (11 100 011: MOD = 11, REG = 100, R/M = BX). `JMP DWORD PTR [SI]` = FF 2C (00 101 100: REG = 101, R/M = [SI]). Bu iki satırda REG alanının bir yazmacı değil, işlemin türünü gösterdiği açıkça görülür.

11.9.2 Göreli uzaklığın hesabı

Şekil 11.6 bir kısa atlamayı gösterir. 0120h'deki `EB 10` iki bayttır; bir sonraki komut 0122h'dedir. Hedef 0122h + 10h = 0132h olur.



Şekil 11.6: Göreli atlama: hedef = sonraki komutun adresi + yer değiştirme. 0120h'deki EB 10 için 0122h + 10h = 0132h.

Ters yöndeki hesap, assembler'ın her atlama için yaptığı iştir: yer değiştirme = hedef − sonraki komutun adresi. Sonuç negatifse ikiye tümleyenle yazılır. Üç örnek:

1. **Kendine atlama.** 010Ah'deki iki baytlık bir kısa JMP kendine atlıyorsa sonraki komut 010Ch'dir. 010Ah − 010Ch = −2 = FEh. Kod **EB FE**'dir; bir programı olduğu yerde durduran sonsuz döngüdür.
2. **Geri atlama.** 0150h'deki kısa JMP 0130h'ye atlıyorsa sonraki komut 0152h'dir. 0130h − 0152h = −22h. −22h'nin ikiye tümleyeni 100h − 22h = DEh'dir. Kod **EB DE**.
3. **Yakın atlama.** 0122h'deki bir JMP, 0251h'deki etikete gidiyorsa uzaklık kısa erime sığmaz. Yakın biçim üç bayttır ve sonraki komut 0125h'dedir. 0251h − 0125h = 012Ch. Kod **E9 2C 01**.

Göreli kodlamanın önemli bir sonucu vardır: kod bellekte başka bir adrese taşındığında kısa ve yakın atlamaların baytları değişmez, çünkü atlayan komut ile hedef birlikte kayar. Bu, programın yükleneceği adresi bilmeyen DOS gibi bir sistem için kolaylıktır. Uzak atlama ise mutlak CS değeri taşıdığından, .EXE programlarında DOS'un yükleme sırasında düzeltmesi gerekir. 8086'da koşullu atlamaların (Jcc) yalnız kısa biçimi vardır. Hedef −128…+127 eriminin dışındaysa assembler koşulu tersine çevirir ve arkasına yakın bir JMP koyar.

Her atlama, süresinin yanında bir bedel daha öder. Atlama gerçekleştiğinde BIU'nun önden getirme kuyruğundaki baytlar artık geçersizdir, çünkü sıradaki komut başka bir yerdedir. Kuyruk boşaltılır ve BIU yeni CS:IP'den okumaya başlar. JMP'nin 15 turluk süresinin büyük kısmı bu yeniden doldurmadan gelir. Koşullu atlamaların atlamadığında yalnız 4 tur sürmesinin nedeni de kuyruğun korunmasıdır.

11.9.3 Dolaylı atlama ve atlama tablosu

Dolaylı yakın JMP, hedefi çalışma anında seçmeyi sağlar. Bir menü ya da bir durum makinesi, olası hedeflerin adreslerini bir sözcük dizisinde tutar ve bir indisle bunlardan birine atlar. Bu yapıya **atlama tablosu** denir; yüksek düzeyli dillerdeki `switch` komutunun makine düzeyindeki karşılığıdır.

```
mov bx, 2           ; seçim (0, 1, 2)
shl bx, 1           ; sözcük indisi: × 2
jmp tablo[bx]       ; IP ← tablo[BX]
tablo dw s0, s1, s2
s0: mov dx, 0
    ret
s1: mov dx, 1
    ret
s2: mov dx, 2
    ret
```

Tablodaki her eleman iki bayttır; bu yüzden seçim numarası SHL ile ikiyle çarpılır. Program 0100h'den başladığında `JMP tablo[BX]` FF A7 09 01 olarak kodlanır: ModR/M A7h = 10 100 111b (MOD = 10, REG = 100 = /4, R/M = [BX+d16]), ardından tablonun adresi 0109h. BX = 4 iken işlemci DS:010Dh'deki sözcüğü okur; orada s2'nin adresi 0117h vardır. IP ← 0117h olur ve program DX = 2 ile biter. Seçim numarasının tablo boyutunu aşmadığı önceden sınanmalıdır; aksi hâlde tablonun arkasındaki kod baytları adres sanılır ve işlemci rastgele bir yere atlar.

11.10 İşlemci kontrolü komutları

11.10.1 Bayrak komutları

İşlemci kontrolü komutları (*processor control instructions*) hesap yapmaz ve veri taşımaz; işlemcinin durumunu değiştirir. Bunların bir kısmı tek bir bayrağı kurar ya da temizler; bu bayrak komutlarının hepsi tek bayttır ve 2 saat turu sürer. Tablo 11.28 komutların tamamını kodları ve süreleriyle verir.

Komut	Kod	Etki	Süre
`CLC` / `STC`	`F8` / `F9`	CF ← 0 / CF ← 1	2
`CMC`	`F5`	CF ← ¬CF	2
`CLD` / `STD`	`FC` / `FD`	DF ← 0 / 1 (dizgi yönü)	2
`CLI` / `STI`	`FA` / `FB`	IF ← 0 / 1 (INTR kesmeleri)	2
`HLT`	`F4`	kesme ya da RESET gelene kadar dur	2
`WAIT`	`9B`	$\overline{\text{TEST}}$ ucu etkinleşene kadar bekle (8087 eşlemesi)	3 + 5n
`ESC`	`D8` – `DF`	komutu yardımcı işlemciye (8087) ilet	2 / 8 + EA
`LOCK`	`F0`	önek: sonraki komut süresince $\overline{\text{LOCK}}$ ucu etkin, yol kilitli	2
`NOP`	`90`	hiçbir şey yapma	3

Tablo 11.28: İşlemci kontrolü komutları.

Bayrak komutlarının her biri tek bir bayrağa dokunur; ötekiler değişmez. Böyle küçük komutların ayrı olarak bulunmasının nedeni, bu üç bayrağı (CF, DF, IF) programın kendisinin bilerek kurmasıdır. CF, çok sözcüklü toplamaya başlamadan önce temizlenebilir (CLC, ardından bir ADC zinciri). DF, dizgi komutlarından önce yönü seçer. IF ise kesmelerin kabul edilip edilmeyeceğini belirler. OF, SF, ZF, AF ve PF için ayrı kurma/temizleme komutu yoktur; bunlara POPF ile (OF dışındakilere SAHF ile de) yazılır.

CLC ve STC'nin yaygın bir kullanımı, bir prosedürden başarı ya da hata bilgisi döndürmektir. Prosedür hata bulursa STC, bulmazsa CLC ile döner. Çağıran taraf hemen ardından `JC hata` ya da `JNC tamam` ile karar verir. DOS'un INT 21h hizmetlerinin çoğu da hatayı bu yolla bildirir. Aşağıdaki prosedür AX'i BL'ye böler, ama BL = 0 ise bölme hatasına yol açmadan CF = 1 ile döner:

```
bol:
    cmp bl, 0
    jne tamam
    stc          ; hata: bölen sıfır
    ret
tamam:
    div bl       ; AL = bölüm, AH = kalan
    clc          ; başarı
    ret
```

AX = 7 ve BL = 0 ile çağrıldığında prosedür CF = 1 ile döner ve AX değişmez. AX = 7 ve BL = 3 ile çağrıldığında DIV, AL = 2 (bölüm) ve AH = 1 (kalan) üretir; AX = 0102h olur ve CF = 0 ile dönülür. DIV bütün bayrakları tanımsız bıraktığı için CLC burada gereklidir: o olmadan CF'nin değeri kestirilemez. RET bayrakları değiştirmediğinden CF çağırana olduğu gibi ulaşır.

11.10.2 Kesme bayrağı: CLI ve STI

IF, işlemcinin INTR ucundan gelen maskelenebilir kesmeleri kabul edip etmeyeceğini belirler. CLI ile IF = 0 yapıldığında INTR yok sayılır; NMI ucundan gelen maskelenemez kesme ve INT n ile yazılımla çağrılan kesmeler ise yine çalışır. CLI ile STI arasındaki bölge, bir kesmenin yarıda bırakılmış bir veri yapısını görmemesi gereken kritik bölgedir. Tipik örnek yığının taşınmasıdır:

```
cli              ; kesmeler kapalı
mov ss, ax
mov sp, 0FFFEh
sti              ; kesmeler açık
```

SS değiştiği hâlde SP henüz eski değerindeyken bir kesme gelirse, işlemci FLAGS, CS ve IP'yi yanlış bir adrese iter ve belleği bozar. 8086, SS'ye yazan komuttan sonraki tek komut boyunca kesmeleri zaten bekletir. Ama bu koruma yalnız bir komutluktur; CLI/STI çifti niyeti açıkça gösterir ve arada daha çok komut olsa da güvenlidir. STI'nin de benzer bir özelliği vardır: kesmeler STI'dan sonraki komut bittikten sonra kabul edilir. Böylece bir kesme işleyicisinin sonundaki `STI` / `IRET` çiftinde, IRET yürütülmeden yeni bir kesme araya giremez.

11.10.3 HLT, WAIT, ESC, LOCK ve NOP

Bu komutlar işlemcinin dış dünyayla ilişkisini yönetir ve doğrudan 8086'nın uçlarıyla bağlantılıdır.

- **HLT** (*halt*) işlemciyi durma durumuna sokar. Komut yürütülmez, yol döngüsü başlatılmaz; işlemci bir sonraki komutu getirmeden bekler. Durumdan çıkmanın üç yolu vardır: INTR ucundan bir kesme (IF = 1 ise), NMI ucundan bir kesme ya da RESET. Kesme ile çıkıldığında yığına HLT'den

sonraki komutun adresi itilir; işleyici döndüğünde program HLT'nin arkasından sürer. Bu yüzden HLT, "bir sonraki kesmeye kadar bekle" anlamında kullanılır. CLI'dan sonra bir HLT ise işlemciyi yalnız NMI ya da RESET ile uyandırılabilir hâle getirir.

- **WAIT**, 8086'nın $\overline{\text{TEST}}$ girişini yoklar. Uç 1 olduğu sürece işlemci beklemeye devam eder ve her beş saat turunda ucu yeniden örnekler. Bu uç genellikle 8087 kayan nokta yardımcı işlemcisinin BUSY çıkışına bağlanır. WAIT, 8087 bir hesabı bitirmeden 8086'nın onun sonucunu okumasını önler.
- **ESC** (*escape*), 11011xxx (D8h–DFh) ile başlayan komutlardır. 8086 bu komutları kendisi yürütmez. Bellek işleneni varsa etkin adresi hesaplar ve o adresten bir okuma yapar. Yolu izleyen 8087 hem komutu hem adresi yakalar ve hesabı kendisi yapar. İşlem kodundaki üç bit ile ModR/M'nin REG alanı birleşerek 8087'nin altı bitlik komut kodunu verir.
- **LOCK** bir önektir. Arkasından gelen komut yürütüldüğü sürece 8086'nın $\overline{\text{LOCK}}$ çıkışı (en büyük modda) 0 tutulur. Yol hakem devresi bu süre boyunca başka bir işlemcinin ya da DMA denetleyicisinin yolu almasına izin vermez. Böylece okuma ve yazmadan oluşan bir komut, araya başka bir yol ustası girmeden tamamlanır.
- **NOP** hiçbir şey yapmaz ve 3 tur sürer. Kodu 90h'dir, yani aslında `XCHG AX, AX`'tir. Zamanlama döngülerinde kısa gecikme için, kod hizalamada boşluk doldurmak için ve hata ayıklamada bir komutu silmeden etkisiz kılmak için kullanılır.

LOCK'un en tanınan kullanımı, birden çok işlemcinin paylaştığı bir kaynağı korumak için bir **semafor** kurmaktır. Bellekteki `kilit` baytı 0 ise kaynak boştur. Bir işlemci kaynağı almak için AL = 1 ile `LOCK XCHG AL, kilit` yürütür. Takas tek komutta yapılır ve LOCK yüzünden araya başka bir işlemci giremez. AL'ye gelen eski değer 0 ise kaynak alınmıştır; 1 ise başkası almıştır ve döngü yeniden dener:

```
bekle:
    mov al, 1
    lock xchg al, kilit   ; F0 86 06 …
    test al, al
    jnz bekle               ; başkası almış
    ; kritik bölge
    mov kilit, 0            ; kaynağı bırak
```

Okuma ve yazma iki ayrı komutla yapılsaydı (`MOV AL, kilit` / `MOV kilit, 1`), iki işlemci aynı anda 0 okuyup ikisi birden kaynağı aldığını sanabilirdi. Tek işlemcili bir sistemde aynı korumayı CLI/STI sağlar; LOCK çok işlemcili sistemler içindir.

11.11 Bayrak etkileri

11.11.1 Hesaplama komutları

Bir programın doğru çalışması çoğu zaman hangi komutun hangi bayrağı değiştirdiğini bilmeye bağlıdır. Koşullu atlamalar en son bayrak değiştiren komutun sonucuna bakar; araya giren her komut bu sonucu bozabilir. Tablo 11.29 aritmetik, mantık ve kaydırma komutlarının altı durum bayrağına etkisini verir.

Komut	OF	SF	ZF	AF	PF	CF
`ADD ADC SUB SBB CMP NEG`	*	*	*	*	*	*
`CMPS SCAS`	*	*	*	*	*	*
`INC DEC`	*	*	*	*	*	–
`MUL IMUL`	*	?	?	?	?	*
`DIV IDIV`	?	?	?	?	?	?
`DAA DAS`	?	*	*	*	*	*
`AAA AAS`	?	?	?	*	?	*
`AAM AAD`	?	*	*	?	*	?
`AND OR XOR TEST`	0	*	*	?	*	0
`SHL SAL SHR SAR`	*[1]	*	*	?	*	*
`ROL ROR RCL RCR`	*[1]	–	–	–	–	*

Tablo 11.29: Hesaplama komutlarının bayrak etkileri. * sonuca göre değişir · 0 temizlenir · – değişmez · ? tanımsız · [1] OF yalnız kaydırma sayısı 1 iken tanımlı. DF, IF ve TF bu komutlardan etkilenmez.

Tablonun birkaç satırı ayrıca dikkat ister. INC ve DEC, CF'yi korur; bu özellik çok sözcüklü toplama döngülerinde bir sayacı ya da bir indisi artırırken ADC'nin eldesini bozmamak için tasarlanmıştır. MUL ve IMUL yalnız CF ile OF'yi anlamlı kurar: ikisi birlikte sonucun üst yarısının gerekli olup olmadığını söyler. DIV ve IDIV'den sonra hiçbir bayrağa güvenilmez. Mantık komutları CF ile OF'yi her zaman 0 yapar. `TEST AX, AX` bu yüzden sık kullanılan bir sınamadır: AX'i değiştirmez, ZF, SF ve PF'yi AX'e göre kurar ve CF'yi temizler. Döndürme komutları yalnız CF ve OF'yi değiştirir. Kaydırma ve döndürmede sayı 0 ise (CL = 0) hiçbir bayrak değişmez.

“Tanımsız” (?) bayrak, işlemcinin o bayrağa bir değer yazdığı ama bu değerin belgelenmediği anlamına gelir. emu8086'da ya da gerçek bir işlemcide bu bayrakta belirli bir değer görülür, ama başka bir işlemci modelinde farklı olabilir. Doğru yazılmış bir program tanımsız bir bayrağa bakarak karar vermez.

11.11.2 Diğer komutlar

Tablo 11.30 geri kalan komutların bayraklara etkisini toplar. Bu tablonun ana mesajı, komutların büyük çoğunluğunun bayraklara hiç dokunmamasıdır.

Komutlar	Etkilenen bayraklar
`MOV XCHG LEA LDS LES XLAT PUSH POP IN OUT LAHF PUSHF CBW CWD NOT`	hiçbiri
`MOVS LODS STOS`, `REP`	hiçbiri
`JMP Jcc LOOP LOOPE LOOPNE JCXZ CALL RET`	hiçbiri (yalnız okunur)
`SAHF`	SF ZF AF PF CF ← AH
`POPF`, `IRET`	tümü (yığından)
`INT n`, `INTO`	IF ← 0, TF ← 0 (eski FLAGS yığında)
`CLC STC CMC`	CF
`CLD STD`	DF
`CLI STI`	IF

Tablo 11.30: Diğer komutların bayrak etkileri.

İki satır şaşırtıcı gelebilir. NOT bir mantık komutudur ama hiçbir bayrağı değiştirmez; bayrakları kurmak isteyen bir program onun yerine `XOR AX, 0FFFFh` kullanır. LOOP da CX'i azaltır ama DEC'in aksine bayraklara dokunmaz. LOOPE ve LOOPNE ZF'yi okur, yazmaz. INT n ise bayrakları değiştirmeden önce yığına iter; IRET onları geri yüklediği için kesme işleyicisinden dönünce programın bayrakları eski hâline gelir.

11.11.3 CMP ile koşullu atlama arasına ne girebilir?

Bayrak tablolarının en sık kullanıldığı yer, bir karşılaştırma ile ona bağlı koşullu atlamanın arasıdır. Araya bayrak değiştiren bir komut girerse atlama karşılaştırmanın değil o komutun sonucuna bakar. Aşağıdaki parçada AL = 20 ve BX = FFFFh'dir:

```
cmp al, 10
add bx, 2       ; hata: bayrakları bozar
jb kucuk         ; AL < 10 ise
mov dx, 1
ret
kucuk:
mov dx, 2
ret
```

CMP, 20 − 10 hesabından CF = 0 üretir; AL 10'dan küçük değildir ve JB atlamamalıdır. Ama araya giren `ADD BX, 2` FFFFh + 2 = 1 0001h hesaplar: sonuç 16 bite sığmaz ve CF = 1 olur. JB bu CF'ye bakar, atlar ve program yanlış dalda DX = 2 ile biter. ADD'nin yerine `LEA BX, [BX+2]` yazılırsa BX yine

0001h olur ama bayraklar değişmez; JB CMP'nin sonucunu görür, atlamaz ve DX = 1 doğru sonucu çıkar. Genel kural: CMP ile Jcc arasına yalnız MOV, LEA, XCHG, PUSH, POP gibi bayrak etkilemeyen komutlar girebilir. Bir sayacın artırılması gerekiyorsa ya CMP'den önceye alınır ya da LEA ile yapılır.

11.12 Sık yapılan hatalar

Bu kesim, not boyunca karşılaşılan hataları bir arada listeler. Her birinin ardından neden ortaya çıktığı ve nasıl önleneceği verilmiştir.

- **D bitini "soldaki işlenen" sanmak.** D = 1, REG alanındaki yazmacın hedef olduğunu söyler. İki yazmaçlı komutlarda iki kodlama da geçerlidir (8B C3 ve 89 D8); bellekli komutlarda D'yi belleğin hangi tarafta olduğu belirler.
- **W bitini atlayıp yanlış yazmacı okumak.** REG = 100, W = 0 iken AH, W = 1 iken SP'dir. Çözmede önce W'ye bakılır.
- **[BP]'yi MOD = 00 ile kodlamak.** MOD = 00 ve R/M = 110 doğrudan adrestir. [BP], MOD = 01 ve d8 = 0 ile kodlanır ve üç bayttır.
- **Çözmeye yanlış adresten başlamak.** Bir bayt kayması bütün sınırları değiştirir; çözme bir program başlangıcından ya da bilinen bir atlama hedefinden başlar.
- **Little-endian sırayı unutmak.** Yer değiştirme, anlık değer ve uzak işaretçinin her sözcüğü düşük bayt önce yazılır. Uzak işaretçide önce bağıl konum, sonra kesim gelir.
- **LEA ile MOV'u karıştırmak.** `LEA AX, [BX+DI+2]` adresi, `MOV AX, [BX+DI+2]` içeriği verir.
- **DF'yi varsaymak.** Dizgi komutundan önce yön CLD ya da STD ile açıkça kurulur.
- **.EXE programında ES'yi kurmamak.** Dizgi hedefi her zaman ES:DI'dır ve DOS, ES'yi veri kesimine ayarlamaz.
- **REPNE/REPE'den sonra yalnız CX'e bakmak.** Komutun neden durduğunu ZF söyler; aranan son konumda bulunduğunda CX = 0 ve ZF = 1 birlikte olur.
- **Örtüşen bloklarda yanlış yön.** Hedef kaynağın sağındaysa STD ile sondan başa kopyalanır.
- **CMP ile Jcc arasına bayrak bozan komut koymak.** INC, ADD, SUB, AND gibi komutlar bayrakları yeniden kurar; araya yalnız bayrak etkilemeyen komutlar girer.
- **Tanımsız bayrağa güvenmek.** MUL'den sonra ZF, DIV'den sonra herhangi bir bayrak, sayı 1'den büyük kaydırmadan sonra OF tanımsızdır.

11.13 Alıştırmalar ve çözümleri

Bu kesimdeki dokuz alıştırma sunumun sonundaki alıştırmalardır. Her birinin çözümü, notta anlatılan yöntemle adım adım verilmiştir.

Alıştırma 1. `MOV DX, [BP+DI+8]` komutunu kodlayın (işlem kodu 100010, D = 1, W = 1).

Çözüm. Birinci bayt 100010 1 1 = 8Bh. Hedef DX, REG = 010. [BP+DI+d] biçiminin R/M kodu 011'dir. 8, tek bayta sığdığından MOD = 01. ModR/M = 01 010 011b = 53h. Yer değiştirme 08h. Kod **8B 53 08**; BP içerdiği için varsayılan kesim SS'tir.

Alıştırma 2. DS = 1000h, BX = 0200h, AL = 05h iken `XLAT` hangi fiziksel adresteki baytı AL'ye yükler?

Çözüm. EA = BX + AL = 0200h + 05h = 0205h. Fiziksel adres 1000h × 10h + 0205h = **10205h**.

Alıştırma 3. Bellekteki `uzak dd` değişkeninin baytları 34 12 00 B8. `LES DI, uzak` sonrası ES ve DI nedir?

Çözüm. Uzak işaretçide önce bağıl konum, sonra kesim saklanır. İlk sözcük 34 12 → 1234h, ikinci sözcük 00 B8 → B800h. **DI = 1234h, ES = B800h**.

Alıştırma 4. CX = 0 iken `REP MOVSB` kaç bayt kopyalar?

Çözüm. **Hiç**. REP her yinelemeden önce CX'i sınar; CX = 0 ise dizgi komutu bir kez bile yürütülmez. SI, DI ve bellek değişmez.

Alıştırma 5. CLD, CX = 5, SI = 0100h, DI = 0200h; `REPE CMPSB` 3. baytta fark buluyor. Sonra CX, SI, DI ve ZF nedir?

Çözüm. Üç karşılaştırma yapılır: ilk ikisi eşittir (ZF = 1), üçüncüsü farklıdır (ZF = 0) ve REPE durur. Her karşılaştırmada SI ve DI bir artar, CX bir azalır: **CX = 2, SI = 0103h, DI = 0203h, ZF = 0**. Farklı bayt SI − 1 = 0102h'dedir.

Alıştırma 6. 0150h adresindeki `JMP SHORT` komutu 0130h'ye atlıyor. Makine kodu nedir?

Çözüm. Komut iki bayttır; sonraki komut 0152h'dedir. Uzaklık 0130h − 0152h = −22h. Bir baytta −22h = 100h − 22h = DEh. Kod **EB DE**.

Alıştırma 7. Bir dizgideki boşlukları sayan kodu `REPNE SCASB` ile yazın.

Çözüm. REPNE SCASB bir boşluk bulduğunda durur; DI bulunan boşluğun arkasını, CX kalan uzunluğu gösterir. Sayaç artırılıp komut yeniden çalıştırılırsa arama kaldığı yerden sürer. Döngü, arama bulamadan bittiğinde (ZF = 0) ya da dizgi tükendiğinde (CX = 0) sona erer.

```
org 100h
cld
xor bx, bx          ; sayaç = 0
lea di, metin
mov cx, 19          ; uzunluk
mov al, ' '
ara:
repne scasb
jne bitti           ; boşluk kalmadı
inc bx              ; bir boşluk daha
jcxz bitti          ; dizgi bitti
jmp ara
bitti:
ret
metin db 'bu bir deneme metni'
```

emu8086'da çalıştırıldığında program BX = 3 ile biter: 'bu bir deneme metni' 19 karakterdir ve üç boşluk içerir. JCXZ sınaması, son karakterin boşluk olduğu durumu karşılar. O durumda REPNE SCASB boşluğu son yinelemede bulur; ZF = 1 ve CX = 0 olur. Sayaç artırılır ve JCXZ döngüden çıkar. Bu sınama olmasaydı CX = 0 ile yeniden giren REPNE hiç yürümez ve ZF = 1 olarak kalırdı; JNE atlamaz, sayaç durmadan artar ve döngü hiç bitmezdi. Metnin sonu 'met ' olarak değiştirildiğinde program dört boşluğu doğru sayar.

Alıştırma 8. `8A 1E 34 12` baytları hangi komuttur? Aynı baytlar bir bayt kaydırılarak (bağıl konum 1'den) çözülürse ne olur?

Çözüm. 8Ah = 100010 1 0: D = 1, W = 0. 1Eh = 00 011 110: MOD = 00, REG = 011 (BL), R/M = 110 → doğrudan adres. Ardından 34 12 → 1234h. Komut `MOV BL, [1234h]`. Konum 1'den çözüldüğünde 1Eh tek baytlık `PUSH DS`, ardından 34 12 iki baytlık `XOR AL, 12h` olur.

Alıştırma 9. `MOV ES, DX` ve `MOV AL, [0020h]` komutlarını kodlayın.

Çözüm. `MOV ES, DX`: 8Eh ailesi; ModR/M = 11 000 010b = C2h (REG = 0 + 00 = ES, R/M = DX). Kod **8E C2**. `MOV AL, [0020h]`: AL ile doğrudan adres arasındaki kısa biçim A0h, ardından adres little-endian: **A0 20 00**.

11.14 Özet

- **Komut kümesi**: 8086'nın yaklaşık yüz komutu altı sınıfa ayrılır: veri taşıma, aritmetik, mantık-kaydırma, dizgi, kontrol aktarma, işlemci kontrolü. Anımsatıcı tek, işlem kodu çoktur.
- **Komut biçimi**: önekler + işlem kodu (6 bit, D, W) + ModR/M (MOD, REG, R/M) + 0–2 bayt yer değiştirme + 0–2 bayt anlık değer. Öneksiz komut 1–6 bayttır; sayılar little-endian yazılır.
- **D ve W**: W = 0 bayt, W = 1 sözcük; D = 1 REG hedef, D = 0 REG kaynak. İki yazmaçlı komutların iki geçerli kodu vardır: `MOV AX, BX` = 8B C3 ya da 89 D8; `ADD AX, BX` = 03 C3 ya da 01 D8.

- **ModR/M**: MOD = 11 yazmaç, 00/01/10 bellek (0/1/2 bayt yer değiştirme). MOD = 00, R/M = 110 doğrudan adrestir; [BP] bu yüzden d8 = 0 ile kodlanır. BP içeren biçimlerin kesimi SS'tir. EA süresi 5–12 turdur.
- **Kısa biçimler**: 40h+r, 50h+r, 90h+r, B0h+r, B8h+r, A0h–A3h, 04h/05h, 83h; aynı işin genel biçiminden bir-iki bayt kısadır. 80h/81h/83h ve FEh/FFh ailelerinde REG alanı işlemi seçer (/0 … /7).
- **Çözme**: başlangıç noktasından ilk bayt, ModR/M, yer değiştirme ve anlık değer sırasıyla okunur. Yanlış adresten başlayan çözme aynı baytları başka komutlara çevirir; işlemci kodla veriyi ayırt etmez.
- **Veri taşıma**: bayrakları değiştirmez (SAHF ve POPF hariç). XLAT: AL ← [DS:BX + AL]. LEA adresi, MOV içeriği verir; emu8086'da doğrudan adresli LEA üç baytlık MOV r16, imm16 olarak derlenir. LDS/LES uzak işaretçiyi (önce bağıl konum, sonra kesim) yükler.
- **Dizgi komutları**: MOVS, CMPS, SCAS, LODS, STOS; kaynak DS:SI, hedef ES:DI. DF = 0 (CLD) artırır, DF = 1 (STD) azaltır; adım B'de 1, W'de 2. CMPS = [SI] − [DI], SCAS = AL/AX − [DI].
- **Yineleme önekleri**: REP (F3h) CX ≠ 0 iken; REPE (F3h) ve REPNE (F2h) ZF koşuluyla. CX = 0 ise komut hiç yürümez. Bittikten sonra ZF komutun neden durduğunu söyler. Örtüşen bloklarda hedef sağdaysa STD kullanılır.
- **JMP**: kısa (EB d8) ve yakın (E9 d16) göreli; hedef = sonraki komut + uzaklık. Uzak (EA IP CS) ve dolaylı (FF /4, FF /5) mutlak adres kullanır. Atlama önden getirme kuyruğunu boşaltır.
- **İşlemci kontrolü**: CLC/STC/CMC, CLD/STD, CLI/STI tek bayrak; HLT, WAIT, ESC, LOCK işlemcinin uçlarıyla (INTR/NMI/RESET, TEST̄, 8087, LOCK̄) çalışır; NOP = XCHG AX, AX.
- **Bayrak etkileri**: taşıma, dizgi taşıma, atlama ve NOT bayrak değiştirmez; INC/DEC CF'yi korur; mantık komutları CF = OF = 0 yapar; DIV sonrası bayraklar tanımsızdır. CMP ile Jcc arasına bayrak etkileyen komut girmemelidir.

11.15 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

komut kümesi *instruction set* · anımsatıcı *mnemonic* · makine kodu *machine code* · komut biçimi *instruction format* · işlem kodu *opcode* · ModR/M baytı *ModR/M byte* · yer değiştirme *displacement* · anlık değer *immediate value* · etkin adres (EA) *effective address* · varsayılan kesim *default segment* · önek, kesim öneki *prefix, segment override prefix* · little-endian *little-endian* · önden getirme kuyruğu *prefetch queue* · veri taşıma komutu *data transfer instruction* · çeviri tablosu (XLAT) *translation table* · uzak işaretçi *far pointer* · dizgi komutu *string instruction* · yön bayrağı (DF) *direction flag* · yineleme öneki (REP) *repeat prefix* · kısa / yakın / uzak atlama *short / near / far jump* · göreceli adresleme *relative addressing* · işlemci kontrolü komutu *processor control instruction* · saat turu *clock cycle* · liste (listing) çıktısı *listing file*

11.16 Kendini Dene

1. `MOV [BX+SI+0Ah], DX` komutunun makine kodu nedir?

   Cevap

   DX kaynaktır ve REG alanına girer: D = 0, W = 1 → 89h. REG = 010 (DX), R/M = 000 ([BX+SI+d]), 0Ah tek bayta sığdığından MOD = 01. ModR/M = 01 010 000b = 50h. Kod **89 50 0A**.

2. `C6 46 FE 07` baytları hangi komuttur? Bellek işleneni hangi kesimdedir?

   Cevap

   C6h, anlık baytı r/m8'e taşıyan MOV'dur (REG alanı 000 olmalı). 46h = 01 000 110b: MOD = 01 (d8), R/M = 110 → [BP+d8]. d8 = FEh = −2, anlık değer 07h. Komut `MOV BYTE PTR [BP−2], 7`; BP içerdiği için kesim SS'tir. Bu, yığın çerçevesindeki bir yerel değişkene değer atamanın kalıbıdır.

3. `MOV AL, [BX]` iki bayt iken `MOV AL, [BP]` neden üç bayttır?

   Cevap

   MOD = 00 ve R/M = 110 birleşimi doğrudan adrese ayrılmıştır. [BP] bu yüzden MOD = 01 ve d8 = 00 ile kodlanır: 8A 46 00. [BX] için böyle bir çakışma yoktur: 8A 07.

4. 0200h'deki yakın JMP (E9) 0100h'ye atlıyor. Makine kodu nedir?

   Cevap

   Komut üç bayttır; sonraki komut 0203h'dedir. 0100h − 0203h = −103h; 16 bitte 10000h − 103h = FEFDh. Little-endian yazılır: **E9 FD FE**.

5. `kare db 0, 1, 4, 9, 16, 25, 36, 49, 64, 81` tanımlı. `LEA BX, kare`, `MOV AL, 7`, `XLAT` sonrası AL nedir?

   Cevap

   XLAT AL ← [DS:BX + 7] yapar; tablonun 7. elemanı (0'dan sayarak) 49'dur. **AL = 49 = 31h**. Tablo yöntemi, küçük sayıların karesini çarpma yapmadan, 11 turda verir.

6. Bellekte `ptr` değişkeninin baytları 00 20 00 10'dur. `LDS SI, ptr` sonrası SI ve DS nedir? DS:SI

hangi fiziksel adresi gösterir?

Cevap

İlk sözcük bağıl konum: **SI = 2000h**; ikinci sözcük kesim: **DS = 1000h**. Fiziksel adres 10000h + 2000h = **12000h**. Bu andan sonra DS programın kendi verisini göstermez.

7. CLD, SI = 0200h, DI = 0300h, CX = 3 iken `REP MOVSW` kaç bayt kopyalar? Sonunda SI, DI ve CX nedir?

Cevap

Üç sözcük, yani **6 bayt** kopyalanır. Her yinelemede SI ve DI 2 artar: **SI = 0206h, DI = 0306h, CX = 0**.

8. STD, DI = 0309h, CX = 10, AL = 'x'. Bellekte 0300h–0309h arasında '0123x5x789' var. `REPNE SCASB` sonrası DI, CX ve ZF nedir?

Cevap

Tarama sondan başa ilerler: 0309h '9', 0308h '8', 0307h '7', 0306h 'x'. Dördüncü karşılaştırmada eşleşme bulunur, ZF = 1 olur ve komut durur. DI her karşılaştırmadan sonra azaldığı için **DI = 0305h**, **CX = 6**, **ZF = 1**. Eşleşmenin adresi DI + 1 = 0306h'dir; STD ile tarama ilk 'x'i değil, en sondakini bulur.

9. `CMP AX, BX` ile `JE esit` arasına şu komutlardan hangileri konabilir: `INC CX`, `MOV CX, 0`, `XOR CX, CX`, `LEA CX, [BX+1]`, `PUSH AX`?

Cevap

`MOV CX, 0`, `LEA CX, [BX+1]` **ve** `PUSH AX` bayrakları değiştirmez ve konabilir. `INC CX` ZF, SF, OF, AF, PF'yi yeniden kurar; `XOR CX, CX` ZF = 1 yapar ve JE'yi her zaman atlatır. CX'i sıfırlamanın kısa yolu olan XOR'un bu yan etkisi burada hataya yol açar.

10. `REP STOSB` ile `MOV CX, 0` / `dongu: STOSB` / `LOOP dongu` arasında CX = 0 iken davranış farkı nedir?

Cevap

REP önce CX'i sınar; CX = 0 iken STOSB hiç yürümez ve bellek değişmez. LOOP'lu döngüde STOSB bir kez yürür, LOOP CX'i FFFFh'ye indirir ve döngü 65.536 kez döner; ES:DI'dan başlayan 64 KB'lık bir alan AL ile dolar.

11.17 İleri okuma

- Intel, *The 8086 Family User's Manual* (1979): komut kümesi bölümü her komutun bayrak etkisini ve saat turu süresini, komut kodlama bölümü işlem kodu tablolarını ve ModR/M düzenini birinci elden verir.
- B. B. Brey, *The Intel Microprocessors* (8. baskı), veri taşıma ve dizgi komutları bölümleri: makine dili kodlaması, MOVS/STOS/LODS örnekleri ve sonraki x86 işlemcilere uzanan ayrıntılar.
- K. R. Irvine, *Assembly Language for x86 Processors* (7. baskı): dizgi komutları ve dizi işleme bölümü; REPE CMPSB ile karşılaştırma ve tablo tabanlı arama için çok sayıda örnek.
- emu8086 yardım belgeleri, "8086 instructions": her komutun kısa tanımı, geçerli işlenen biçimleri ve bayrak tablosu; sunumdaki örneklerin denendiği ortamın kendi açıklamaları.

11.18 Kaynaklar

1. Intel, *The 8086 Family User's Manual*, 1979 — Bölüm 2: komut kümesi, Bölüm 4: komut kodlama.
2. B. B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson, 2009.
3. K. R. Irvine, *Assembly Language for x86 Processors*, 7. baskı, Pearson, 2014.
4. Ş. Ç. Öztürk, O. Ergin, *Intel 8086 ile Mikroişlemci Programlamaya Giriş*, 2009.
5. emu8086 yardım belgeleri: "8086 instructions" ve "8086 assembler tutorial".

EK A

Terim Sözlüğü

Kitapta geçen İngilizce terimler ve her biri için kullanılan tek Türkçe karşılık. Sağdaki sayı, terimin ayrıntılı anlatıldığı bölümdür. Sözlüğün aranabilir hâli ders sayfasındadır.

A

above / below → büyük / küçük (işaretsiz; above / below) 7
İşaretsiz karşılaştırmanın sonuçları (JA, JB).

arithmetic instruction → aritmetik komut 6
Toplama, çıkarma, çarpma, bölme yapan komutlar.

B

BCD adjustment → BCD düzeltmesi 6
İkili toplama/çıkarma sonucunu geçerli BCD'ye çeviren işlem (DAA, AAA…).

C

comparison (CMP) → karşılaştırma (CMP) 7
İki işleneni çıkarıp yalnız bayrakları ayarlayan işlem.

D

data transfer instruction → veri taşıma komutu 11
Yazmaç, bellek, yığın ve G/Ç arasında veri kopyalayan komutlar (MOV, XCHG, PUSH…).

dividend / divisor / quotient / remainder → bölünen / bölen / bölüm / kalan 6
Bölme işleminin öğeleri.

F

far pointer → uzak işaretçi 11
Bağıl konumu ve kesimi birlikte tutan 4 baytlık adres; bellekte önce bağıl konum, sonra kesim saklanır (LDS, LES).

flow control → akış kontrolü 7
Komutların yürütülme sırasını atlama ve döngülerle değiştirme.

G

greater / less → büyük / küçük (işaretli; greater / less) 7
İşaretli karşılaştırmanın sonuçları (JG, JL).

I

instruction format → komut biçimi 11
Bir makine komutunun alanlarının (işlem kodu, ModR/M, yer değiştirme, anlık veri) düzeni.

instruction set → komut kümesi 11
Bir işlemcinin yürütebildiği komutların tümü.

intrasegment / intersegment jump → kesim içi / kesimler arası atlama 7
Yalnız IP'yi değiştiren (kısa ve yakın) atlama ile CS'yi de değiştirip başka bir kesime geçen (uzak) atlama.

J

jump → atlama 7
Yürütmenin başka bir adresten devam etmesi (JMP, Jcc).

jump displacement → uzaklık 7
Atlama hedefinin, atlamadan sonraki komutun adresine göre işaretli farkı.

L

label → etiket 7
Bir adrese verilen ve atlama/çağrı hedefi olarak kullanılan ad.

logical / arithmetic shift → mantıksal / aritmetik kaydırma 6
Boşalan bite 0 koyan ve işaret bitini koruyan sağa kaydırmalar (SHR / SAR).

logical instruction → mantık komutu 6
Bit düzeyinde AND, OR, XOR, NOT, TEST işlemleri yapan komutlar.

loop → döngü 7
Bir komut grubunun koşul sağlanana kadar yinelenmesi (LOOP, CX).

M

masking → maskeleme 6
AND/OR/XOR ile bir değerin seçilen bitlerini silme, kurma ya da ters çevirme.

ModR/M byte → ModR/M baytı 11
Komut kodlamasında işlenenleri ve adresleme modunu belirten MOD, REG, R/M alanlarından oluşan bayt.

multiplication / division → çarpma / bölme 6
MUL/IMUL ve DIV/IDIV komutlarının işlemi; sonuç AX ya da DX:AX'te.

N

nested loop → iç içe döngü 7
Bir döngünün gövdesinde başka bir döngü bulunması.

O

opcode → işlem kodu (opcode) 11
Makine komutunun hangi işlemin yapılacağını belirten kısmı.

P

packed / unpacked BCD → paketli / paketsiz BCD 6
Bir baytta iki ya da bir BCD basamağı tutan biçimler.

prefix → önek 11
Komutun başına eklenen ve davranışını değiştiren bayt (kesim, REP, LOCK).

processor control instruction → işlemci kontrolü komutu 11
Bayrakları kuran/silen ve işlemciyi durduran komutlar (CLC, STI, HLT, NOP…).

R

repeat prefix (REP) → yineleme öneki (REP) 11
Dizgi komutunu CX kez ya da koşul bozulana kadar yineleten önek.

rotate → döndürme 6
Dışarı çıkan bitin öbür uçtan geri girdiği kaydırma (ROL, ROR, RCL, RCR).

S

shift → kaydırma 6
Bitleri sola ya da sağa kaydırma (SHL, SHR, SAR); boşalan bitlere 0 ya da işaret biti girer.

short / near / far jump → kısa / yakın / uzak atlama 11
8 bitlik uzaklıkla, aynı kesimde 16 bitlik uzaklıkla ve başka kesime yapılan atlamalar.

sign extension → işaret genişletme 6
İşaretli bir değeri işaret bitini kopyalayarak daha geniş bir yazmaca taşıma (CBW, CWD).

string instruction → dizgi komutu 11
DS:SI ve ES:DI ile ardışık bellek öğeleri üzerinde çalışan komutlar (MOVS, CMPS, SCAS, LODS, STOS).

T

translation table (XLAT) → çeviri tablosu (XLAT) 11
AL'yi BX'in gösterdiği tablodaki karşılığıyla değiştiren komut ve tablosu.

U

unconditional / conditional jump → koşulsuz / koşullu atlama 7
Her zaman ya da yalnız bayraklar koşulu sağladığında yapılan atlama.

EK B

Mikroişlemciler İçin Ön Bilgiler

MİKROİŞLEMCİLER · ÖN BİLGİLER

Mikroişlemci programlama; sayı sistemlerini, sayısal mantığı, bilgisayar mimarisini, assembly dilini, bellek adreslemeyi, kesmeleri, yığına dayalı yürütmeyi, giriş/çıkışı ve temel elektroniği bir araya getirir.

Temel hazırlık

En sağlam hazırlık; ikili ve onaltılık sayılarla, Boole mantığıyla, programlamayla, CPU ile belleğin düzeniyle ve temel sayısal elektronikle uygulamaya dönük bir tanışıklıktır.

SAYILAR · MİMARİ · ASSEMBLY

Temel ön bilgiler

Bu konular; 8086 programlama modeline, yazmaçlara ve pinlere, kesim:bağıl konum ile bellek erişimine, aritmetik ve mantık komutlarına, prosedürlere, yığına, kesmelere, G/Ç portlarına ve BIOS/DOS hizmetlerine zemin hazırlar.

İKİ

İkili sayı sistemi

Bitler · ikinin kuvvetleri · işaretsiz değerler

Mikroişlemciler veriyi ve komutları ikili bit örüntüleri olarak gösterir.

TEKNİK ÖNEMİ

İkili aritmetik; yazmaçların, belleğin, işlem kodlarının, bayrakların ve sayısal mantığın temelidir.

- Onluk ve ikili tabanlar arasında dönüşüm.
- İkinin kuvvetlerini tanıma.
- 8 ve 16 bitlik işaretsiz değerlerin aralıkları.

Bağlantılı konular: yazmaçlar, işlem kodları, maskeler, adresler, bayraklar.

HEX

Onaltılık sayı sistemi

16 tabanı · ikili değerin kısa yazımı

Onaltılık (hexadecimal) sistem, ikili değerleri kısa biçimde yazmanın bir yoludur.

TEKNİK ÖNEMİ

8086 adresleri, makine kodu baytları, yazmaç değerleri ve bit maskeleri genellikle onaltılık yazılır.

- Onluk↔onaltılık ve ikili↔onaltılık dönüşüm.
- Bir onaltılık basamağın dört bite karşılık gelmesi.
- Bayt ve sözcük değerlerini onaltılık okuma.

Bağlantılı konular: adresler, bellek dökümleri, işlem kodları, hata ayıklama.

İT

İşaretli sayılar ve ikiye tümleyen

Negatif tam sayılar · taşma

İşlemcilerin negatif tam sayıları ikili olarak kodlamanın bir yoluna ihtiyacı vardır.

TEKNİK ÖNEMİ

İkiye tümleyen (two's complement) gösterim, aynı toplayıcı donanımının hem işaretli hem işaretsiz aritmetikte kullanılmasını sağlar.

- İşaret biti ve işaretli değer aralıkları.
- Negatif değeri mutlak değerinden bulma.
- Elde (carry) ile işaretli taşmayı (overflow) ayırt etme.

Bağlantılı konular: ADD/SUB, işaret bayrağı (SF), taşma bayrağı (OF), koşullu atlamalar.

ASC

ASCII karakter kodları

Karakter · bayt · 30h–39h

Klavyeden okunan ve ekrana yazılan her karakter, bir baytlık ASCII koduyla taşınır.

TEKNİK ÖNEMİ

Bir sayıyı ekrana yazmak ya da klavyeden okumak, rakam ile ASCII kodu arasında dönüşüm gerektirir. DOS ve BIOS hizmetleri karakterleri bu kodlarla alır ve verir.

- Rakamlar: '0'–'9' = 30h–39h. Bir rakamın değeri, kodundan 30h çıkarılarak bulunur.
- Büyük harfler 'A'–'Z' = 41h–5Ah. Küçük harfler 'a'–'z' = 61h–7Ah. Aradaki fark 20h, yani yalnızca 5. bit farklıdır.
- Denetim karakterleri: satır başı (CR) 0Dh, satır besleme (LF) 0Ah. Boşluk 20h, '$' ise 24h'dir.

Bağlantılı konular: INT 21h karakter hizmetleri, sayı okuma ve yazdırma yordamları, dizgiler, '$' ile biten DOS dizgileri.

MHz

Ölçü birimleri ve önekler

Frekans · bayt · KB ve MB

Saat hızları, komut süreleri ve bellek boyutları önekli birimlerle verilir.

TEKNİK ÖNEMİ

Bir program parçasının süresini hesaplamak saat periyodunu, adres yolunun genişliğini yorumlamak ise ikili öneklerle bellek boyutunu bilmeyi gerektirir.

- Periyot frekansın tersidir: T = 1/f. 5 MHz'lik saatte bir saat periyodu 200 ns'dir.
- Zaman önekleri: 1 ms = 10^{-3} s, 1 µs = 10^{-6} s, 1 ns = 10^{-9} s.
- Süre = saat turu sayısı × saat periyodu.
- Bellekte ikili önekler kullanılır: 1 KB = 2^{10} = 1024 bayt, 1 MB = 2^{20} bayt. n adres hattı 2^n farklı adres verir; 20 hat 1 MB'lık adres uzayı demektir.

Bağlantılı konular: saat sinyali, komut tur sayıları, gecikme döngüleri, adres yolu genişliği, bellek haritası.

BM

Boole mantığı

AND · OR · XOR · NOT

Aritmetik ve kontrol devreleri Boole işlemlerinden kurulur.

TEKNİK ÖNEMİ

Mantık komutları bu işlemleri assembly programlarına doğrudan sunar.

- Temel doğruluk tabloları.
- Bit maskeleme.
- XOR ile bitleri tersine çevirme, bir yazmacı kendisiyle XOR'layarak sıfırlama.

Bağlantılı konular: maskeler, bayraklar, aygıt kontrolü, bit alanları.

SM

Sayısal mantığın temelleri

Kapılar · çoklayıcılar · flip-floplar

Bir CPU, ikili değerleri saklayan ve dönüştüren sayısal devrelerden oluşur.

TEKNİK ÖNEMİ

Temel mantık kavramları; yazmaçları, ALU'ları, yolları ve kontrol sinyallerini anlamayı kolaylaştırır.

- Birleşimsel ve ardışıl mantık arasındaki fark.
- Kapılar, çoklayıcılar ve flip-floplar.
- Yazmacın saklanan bir ikili durum olması.

Bağlantılı konular: ALU, yazmaç kümesi, komut çözücü, kontrol birimi.

Z

Kilitler ve üç durumlu tamponlar

Kilit · tampon · yüksek empedans

Bir yola birden çok aygıt bağlandığında, yolu hangi aygıtın sürdüğünü ve bir değerin ne zaman yakalandığını bu yapı taşları belirler.

TEKNİK ÖNEMİ

Adres ve verinin aynı pinlerden sırayla verildiği işlemcilerde adresin bir kilitte yakalanması, veri yolunun da üç durumlu tamponlarla ayrılması gerekir.

- Kilit (latch), izin girişi etkinken girişini çıkışa geçirir; izin kalkınca son değeri saklar.
- Üç durumlu tampon, çıkışını yüksek empedans (Z) durumuna da geçebilir. Bu durumda aygıt yolu sürmez.

PRG

Temel programlama

Değişkenler · döngüler · koşullar

Algoritma ve akış kontrolü

Bir yolda ayak sesi duyan bir aygıt sürmesi yol çakışmasıdır.

TEKNİK ÖNEMİ

Yüksek düzeyli programlama kavramları, düşük düzeyli komut dizilerine karşılık gelir.

- Atama ve aritmetik ifadeler.
- if/else yapılarını ve döngüleri adım adım izleme.
- Diziler ve fonksiyonlar.

Bağlantılı konular: CMP/Jcc, döngüler, bellekteki değişkenler, prosedürler.

C

C programlama kavramları

Tipler · diziler · fonksiyonlar · bellek

C, yüksek düzeyli kod ile assembly arasında kullanışlı bir köprüdür.

TEKNİK ÖNEMİ

İşlemciye ilişkin birçok kavram; C'deki değişkenler, işaretçiler, fonksiyon çağrıları ve veri boyutlarıyla ilişkilendirildiğinde daha anlaşılır olur.

- Tam sayı tipleri ve diziler.
- Temel kontrol yapıları ve döngüler.
- Kavramsal düzeyde fonksiyon çağrıları ve yerel değişkenler.

Bağlantılı konular: yığın çerçeveleri, çağırma kuralları (calling conventions), veri gösterimi.

BİT

Bit düzeyinde işlemler

Maskeler · kaydırma · döndürme

Bitlerle çalışmak, düşük düzeyli programlamanın temelidir.

TEKNİK ÖNEMİ

8086, bit düzeyinde kontrol için özel mantık, kaydırma ve döndürme komutlarına sahiptir.

- Maske oluşturma ve uygulama.
- Tek tek bit test etme ve değiştirme.
- Mantıksal ve aritmetik kaydırma arasındaki kavramsal fark.

Bağlantılı konular: SHL/SHR, ROL/ROR, aygıt yazmaçları, bayraklar.

BMT

Bilgisayar mimarisinin temelleri

CPU · bellek · yollar

Mikroişlemci programlama, CPU'nun bellek ve G/Ç ile nasıl veri alışverişi yaptığını bilmeye dayanır.

TEKNİK ÖNEMİ

- Bir yolda aynı anda yalnızca bir aygıt sürmesi izin verilen bir kaynaktır; çakışmayı önlemek için İşletim sistemi ve donanım bu yolu yönetir.
- Aktif düşük (active-low) sinyal 0 iken etkindir; adının sonundaki ' işaretiyle ya da üstüne çizilen çizgiyle gösterilir.
- ALE ve adres kilitleri, veri alıcı-vericileri, RD' ve WR' sinyalleri, bellek bankaları.

Yazmaçlar, yollar, bellek ve kontrol sinyalleri, assembly kodunun gördüğü yürütme ortamını oluşturur.

- CPU, bellek ve giriş/çıkışın (G/Ç) temel görevleri.
- Kavramsal düzeyde adres, veri ve kontrol yolları.
- Getir–çöz–yürüt döngüsü.

Bağlantılı konular: 8086 yol arayüzü, bellek erişimi, komut yürütme.

YAZ

Yazmaçlar

Genel amaçlı · kesim · indis · işaretçi

8086, adlandırılmış küçük bir yazmaç kümesini programların doğrudan kullanımına açar.

TEKNİK ÖNEMİ

Assembly komutları çoğunlukla AX, BX, CX, DX yazmaçları ile kesim, indis ve işaretçi yazmaçları üzerinde çalışır.

- Yazmaçların CPU içindeki küçük ve hızlı saklama alanları olması.
- Genel amaçlı, kesim, indis ve işaretçi görevlerin ayrımı.
- 16 bitlik yazmaçlar ve uygun olanlarda 8 bitlik yüksek/düşük yarılar.

BEL

Bellek adresleme

Adresler · baytlar · sözcükler

Assembly kodu, adresi açıkça belirtilen bellek gözlerini okur ve yazar.

Bağlantılı konular: AX/AH/AL, BX, CX, DX, SP, BP, SI, DI, CS, DS, SS, ES.

TEKNİK ÖNEMİ

Belleğin düzenini bilmek; değişkenler, diziler, işaretçiler ve komutlar için vazgeçilmezdir.

- Bayt düzeyinde adreslenen bellek.
- Kavramsal doğrusal adres uzayı gösterim.
- Etkin adresi tanıma.

KES

8086'da kesimleme

Kesim:bağıl konum · fiziksel adres

8086, 20 bitlik fiziksel adresleri 16 bitlik kesim (segment) ve bağıl konum (offset) değerlerinden oluşturur.

Bağlantılı konular: veri değişkenleri, diziler, işaretçiler, komut işlenenleri.

TEKNİK ÖNEMİ

Kesimleme, yazmaçları 16 bitlik olan ilk 8086'nın nasıl 1 MB'a kadar belleği adresleyebildiğini açıklar.

- Fiziksel adres = kesim × 16 + bağıl konum.
- Örtüşen kesim adresleri.
- CS, DS, SS ve ES yazmaçlarının görevleri.

ALU

Aritmetik ve bayraklar

ADD · SUB · CMP · durum bayrakları

Aritmetik komutlar hem hedefteki değeri hem de işlemcinin durum bayraklarını günceller.

TEKNİK ÖNEMİ

Bağlantılı konular: kod, veri, yığın ve ek kesimler, 20 bitlik adresleme.
Bayraklar karşılaştırmalarda, işaretli/işaretsiz kararlarda, döngülerde ve hata algılamada kullanılır.

- Kavramsal düzeyde elde, sıfır, işaret ve taşma bayrakları.
- CMP'nin çıkarma sonucunu saklamadan yalnız bayrakları güncellemesi.
- Bayraklarla koşullu atlamalar arasındaki ilişki.

Bağlantılı konular: ADD, SUB, CMP, Jcc, işaretli ve işaretsiz karşılaştırmalar.

AK

Assembly'de akış kontrolü

Atlamalar · döngüler · etiketler

Assembly, kararları ve tekrarları etiketler ile dallanma komutları aracılığıyla ifade eder.

TEKNİK ÖNEMİ

Yüksek düzeyli kontrol yapıları, derlendiğinde karşılaştırmalara ve atlamalara dönüşür.

- Koşullu ve koşulsuz atlamaları adım adım izleme.
- Etiketlerin dallanma hedefi olarak kullanılması.
- Basit if/while mantığını düşük düzeyli adımlara dönüştürme.

YIĞ

Yığının temelleri

PUSH · POP · SP · LIFO

Yığın; geçici saklama, prosedürler, dönüş adresleri ve kesmeler için kullanılır.
Bağlantılı konular: JMP, JE/JZ, JNE/JNZ, işaretli/işaretsiz koşullu atlamalar.

TEKNİK ÖNEMİ

Yığının düzenli kullanımı, prosedür çağrılarının ve kesme işlemenin doğru çalışması için temeldir.

- LIFO (son giren ilk çıkar) sırası.
- Kavramsal bir yığın üzerinde PUSH ve POP.
- 8086'da yığının durumunu tutan SP ve SS yazmaçları.

PROC

Prosedürler ve çağrılar

CALL · RET · parametreler

Prosedürler, assembly programlarını yeniden kullanılabilir bloklara ayırır.
Bağlantılı konular: prosedürler, dönüş adresleri, parametreler, kesmeler.

TEKNİK ÖNEMİ

CALL ve RET, dönüş bilgisini korumak için yığına dayanır.

INT

Kesmeler

Donanım · yazılım · vektör tablosu

Kesmeler, donanım ya da yazılım olaylarına karşılık denetimi bir hizmet yordamına aktarır.

TEKNİK ÖNEMİ

BIOS/DOS hizmetleri, zamanlayıcılar, klavyeler ve aygıtlarla etkileşim kesmelere dayanır.

- CALL'in dönüş adresini yığına itmesi.
- RET ile çağıran koda geri dönülmesi.
- Kavramsal düzeyde yazmaçlar ve yığınla parametre geçirme.
- Kesme vektörü kavramı.
- İşlemci durumunun saklanıp geri yüklenmesi gerekliliği.

Bağlantılı konular: INT komutu, kesme vektör tablosu (IVT), kesme hizmet yordamı (ISR), BIOS/DOS hizmetleri.

G/Ç

Giriş/çıkış

Portlar · aygıtlar · IN/OUT

Mikroişlemciler çevre birimleriyle açık G/Ç mekanizmaları üzerinden haberleşir.

TEKNİK ÖNEMİ

Port eşlemeli ve bellek eşlemeli G/Ç, yazılımdaki komutları donanım aygıtlarına bağlar.

- Kavramsal düzeyde aygıt yazmaçları.
- x86'da IN ve OUT komutlarının amacı.
- Yoklamalı (polling) ve kesmeli G/Ç.

Bağlantılı konular: portlar, çevre birimleri, klavyeler, ekranlar, denetleyiciler.

ELK

Temel elektronik

Gerilim · sinyaller · zamanlama

Mikroişlemciler yalnız soyut komut makineleri değil, aynı zamanda elektrikle çalışan sayısal sistemlerdir.

TEKNİK ÖNEMİ

Temel sinyal ve zamanlama kavramları; saat sinyalini, yolları, pinleri ve arabağlantıyı açıklamaya yardımcı olur.

- Kavramsal düzeyde ikili gerilim düzeyleri.
- Saat sinyalleri.
- Giriş/çıkış yönü ve sinyal zamanlaması.

Bağlantılı konular: saat, sıfırlama (RESET), yol zamanlaması, sayısal arayüzler.

ARÇ

Assembly araçları

Assembler · bağlayıcı · hata ayıklayıcı

Düşük düzeyli programlama, simgesel assembly kodunu çalıştırılabilir makine koduna dönüştüren bir araç zinciri gerektirir.

TEKNİK ÖNEMİ

Araç zincirinin aşamalarını bilmek; söz dizimi, sembol ve çalışma zamanı hatalarının kaynağını bulmayı kolaylaştırır.

- Assembler ile bağlayıcının (linker) görevleri.
- Basit bir bellek/yazmaç dökümünü okuma.
- Kavramsal düzeyde tek adım çalıştırma ve kesme noktaları.

Bağlantılı konular: MASM/TASM/NASM tarzı çalışma düzeni, bağlayıcı, hata ayıklayıcı.

MK

Makine kodunun temelleri

İşlem kodları · işlenenler · kodlama

Assembly komutları, kodlanmış makine komutlarının simgesel gösterimidir.

TEKNİK ÖNEMİ

Komut kodlama; işlenenlerin, adresleme modlarının ve komut boyutlarının neden önemli olduğunu açıklar.

- İşlem kodu (opcode) ile işlenen (operand) arasındaki fark.
- Komutun bellekte kapladığı alan.
- Adresleme modunun kodlamayı etkilediği.

Bağlantılı konular: makine kodu, tersine çevirme (disassembly), komut boyutu.

BUS

Yol ve zamanlamanın temelleri

Adres · veri · kontrol

İşlemcinin dış dünyayla haberleşmesi, birbirine uyumlu adres, veri ve kontrol sinyalleriyle gerçekleşir.

TEKNİK ÖNEMİ

Yol zamanlaması, bellek ve çevre birimlerinin ne zaman okuyacağını, yazacağını ya da yanıt vereceğini nasıl bildiğini açıklar.

- Adres bilgisi ile veri bilgisinin ayrımı.
- Kavramsal düzeyde okuma/yazma kontrol sinyalleri.
- Genel hatlarıyla bekleme durumu ve zamanlama.

Bağlantılı konular: bellek döngüleri, G/Ç döngüleri, kontrol pinleri, arabağlantı.

EK C

Mikroişlemcilerde Önemli İsimler

MİKROİŞLEMCİLER · BİLİŞİM TARİHİ

Mikroişlemcilerin tarihi; yarı iletken teknolojisini, komut kümesi mimarisini, CPU gerçeklemesini, bellek sistemlerini, başarım mühendisliğini ve ilk tek yongalı işlemcilerden günümüzün x86 ve RISC tasarımlarına uzanan gelişimi bir arada içerir.

Tümleşik devreden günümüz işlemcilerine

Bu kartlar transistörün icadına, yarı iletken tümleştirmeye, ilk mikroişlemcilere, 8086/x86 ailesine, RISC mimarilerine, bellek teknolojisine, işlemci başarımına, paralel işlemcilerin sınıflandırılmasına ve ilk yaygın yüksek düzeyli dile katkıda bulunan kişileri tanıtır.

İSİMLER · YONGALAR · MİMARİ

Öne çıkan katkılar

Tümleşik devrelerin, ilk ticari mikroişlemcilerin, Intel 8086/x86 ailesinin, RISC ve ARM mimarilerinin, bellek teknolojisinin ve işlemci başarım mühendisliğinin arkasındaki başlıca isimler.

FF

Federico Faggin

Intel 4004 · 8080 · silisyum kapılı tasarım

Intel 4004'ün tasarımını yönetti, ilk genel amaçlı mikroişlemcilerin geliştirilmesinde rol aldı.

TEKNİK ÖNEMİ

Çalışmaları, kart dolusu mantık devresiyle kurulan CPU işlevlerinin tek yongalı, yüksek yoğunlukta tümleşik işlemcilere taşınmasına katkı sağladı.

4004 **8080** **MOS tümleştirme**

Faggin, silisyum kapılı MOS teknolojisini geliştirdi ve 4004'ü ticari bir mikroişlemci olarak kullanılabilir kılan gerçekleme çalışmasını yönetti. Ardından 8008 ve 8080 dönemine de katkıda bu-

lundu.

Teknik kavramlar: 4004, 8080, MOS tümleştirme, komut çözme, veri işleme yolunun (data-path) tümleştirilmesi, yarı iletken ölçeklemesi.

TH

Ted Hoff

Mikroişlemci mimarisi · Intel 4004

Intel 4004'e dönüşen proje için genel amaçlı, programlanabilir bir işlemci mimarisi önerdi.

TEKNİK ÖNEMİ

Pek çok özel amaçlı mantık devresinin yerini programlanabilir bir CPU'nun alması, mikroişlemciyi farklı ürünlerde yeniden kullanılabilen bir hesaplama bileşeni hâline getirdi.

4004 **mimari** **depolanan programla kontrol**

Hoff'un mimari katkısı, hesap makinesi için düşünülmüş yonga takımını sadeleştirip programlanabilir bir işlemci ile bellek ve G/Ç bileşenlerine indirgemekti.

SM

Stanley Mazor

Intel 4004 mimarisi

Intel 4004'ün mimari tanımı ve komut kümesi üzerinde çalıştı.

TEKNİK ÖNEMİ

Komut kümesindeki tercihler, yazılımın veri taşımayı, aritmetiği, dallanmayı ve kontrolü nasıl ifade edeceğini belirler.

ISA **4004** **işlem kodu tasarımı**

Mazor, Intel'in ilk mikroişlemci ailesinde komutların ve mimari davranışın tanımlanmasına katkıda bulundu.

MS

Masatoshi Shima

4004 gerçeklemesi · CPU tasarımı

Intel 4004'ün ve sonraki mikroişlemcilerin ayrıntılı mantık tasarımında merkezi rol oynadı.

Teknik kavramlar: komut kümesi mimarisi (ISA), 4004, işlem kodu (opcode) tasarımı, komut biçimleri, akış kontrolü.

TEKNİK ÖNEMİ

Mimari bir tanımı çalışan bir yongaya dönüştürmek; somut kontrol mantığı, veri işleme yolları, zamanlama ve arayüzler gerektirir.

4004 **mantık tasarımı** **veri işleme yolu**

SM

Stephen Morse

Intel 8086 mimarisi · 1970'ler

Intel 8086'nın komut kümesinin ve programlama modelinin baş mimarıdır.

TEKNİK ÖNEMİ

8086, derste incelenen x86 mimarisinin temelini attı. Shima, mimari gereksinimleri ayrıntılı işlemci mantığına dönüştürdü; daha sonra başka etkili CPU'ların tasarımında da çalıştı.

8086 **x86** **ISA**

Teknik kavramlar: 4004, mantık tasarımı, veri işleme yolu, zamanlama, yazmaçlar arası aktarım, yonga gerçeklemesi.

8086; 16 bitlik yazmaçları, kesimli adreslemeyi, genel amaçlı yazmaçlarla kesim yazmaçlarını ve günümüz x86 işlemcilerinin atası olan komut kümesini getirdi.

Teknik kavramlar: 8086, x86, ISA, kesim:bağıl konum (segment:offset) adreslemesi, yazmaçlar, bayrak yazmacı (FLAGS).

BR

Bruce Ravenel

8086 mimarisi ve gerçeklemesi

8086 tasarım ekibinde ve x86'nın ilk geliştirme döneminde çalıştı.

TEKNİK ÖNEMİ

İşlemci mimarisi; komut kümesi tasarımını, gerçekleme kısıtlarını ve sistem uyumluluğunu bir araya getiren bir ekip işidir.

8086 **x86** **yürütme birimi**

8086 ekibi, 1970'lerin sonundaki yarı iletken kısıtları içinde transistör bütçesi, başarım, kod yoğunluğu, yol tasarımı ve uyumluluk hedefleri arasında denge kurdu.

BP

Bill Pohlman

Intel 8086 proje yönetimi

Teknik kavramlar: 8086, x86, yürütme birimi (EU), yol arayüz birimi (BIU), komutları önden getirme (prefetch), kesimli bellek.

Intel 8086 geliştirme programının mühendislik çalışmalarını yönetti.

TEKNİK ÖNEMİ

Başarılı bir CPU tasarımı; mimari, mantık tasarımı, doğrulama, üretim ve ürün kısıtları arasında eşgüdüm gerektirir.

8086 **Intel** **mühendislik**

8086 sıkı kısıtlar altında yürütülen bir mühendislik çalışmasının ürünüdür: sınırlı transistör sayısı, 16 bitlik aritmetik, 20 bitlik adres yolu ve önceki işlemcilerden geçişi kolaylaştıran bir tasarım.

SW

Sophie Wilson

ARM komut kümesi · RISC

İlk ARM komut kümesi mimarisini tasarladı.

TEKNİK ÖNEMİ

ARM, küçük ve düzenli bir komut kümesinin düşük donanım karmaşıklığı ve düşük güç tüketimiyle yüksek başarım sağlayabileceğini gösterdi.

ARM **RISC** **ISA**

Wilson'ın ARM mimarisindeki çalışması; basit komut çözmeyi, yükle/sakla (load/store) tasarımını, koşullu yürütmeyi ve yazmaçların verimli kullanımını öne çıkardı.

Teknik kavramlar: ARM, RISC, ISA, yükle/sakla mimarisi, yazmaçlar, koşullu yürütme.

SF

Steve Furber

ARM işlemci gerçeklemesi

İlk ARM işlemcisinin gerçeklemesini birlikte tasarladı.

TEKNİK ÖNEMİ

Çalışmaları RISC mimarisinin fikirlerini kullanışlı ve az güç tüketen bir işlemci tasarımına bağladı.

ARM **RISC** **veri işleme yolu**

Furber, ARM komut kümesinin çalışan bir yongaya dönüşmesinde rol aldı; daha sonra işlemci ve nöromorfik sistem araştırmalarına önemli katkılar yaptı.

JH

John Hennessy

RISC mimarisi · MIPS

Teknik kavramlar: ARM, RISC, veri işleme yolu, kontrol mantığı, düşük güçlü tasarım.

RISC işlemci mimarisi ve MIPS ailesi üzerine çığır açan çalışmalara öncülük etti.

TEKNİK ÖNEMİ

RISC araştırmaları; işlemcilerin komut karmaşıklığı, boru hattı, derleyici desteği ve gerçekleme hızı arasında kurduğu dengeyi yeniden biçimlendirdi.

MIPS **RISC** **boru hattı**

Hennessy'nin çalışmaları; yükle/sakla komut kümelerinin, geniş yazmaç kümelerinin ve boru hatlı yürütmenin yaygın tasarım ilkeleri olarak yerleşmesine katkı sağladı.

DP

David Patterson

RISC · Berkeley RISC

Teknik kavramlar: MIPS, RISC, boru hattı (pipelining), yükle/sakla, derleyici ile mimarinin birlikte tasarımı.

Berkeley RISC projesini yönetti ve günümüzdeki indirgenmiş komut kümeli bilgisayar anlayışının tanımlanmasına katkıda bulundu.

TEKNİK ÖNEMİ

RISC tasarımları, daha basit komutların daha hızlı boru hatlarına ve daha verimli gerçeklemelere olanak verdiğini gösterdi.

RISC **RISC-V** **boru hattı**

Patterson ayrıca nicel bilgisayar mimarisi yaklaşımının yerleşmesine katkıda bulundu, daha sonra da RISC-V'in geliştirilmesinde yer aldı.

Teknik kavramlar: RISC, RISC-V, boru hattı, nicel mimari, komut kümesi tasarımı.

MF

Michael J. Flynn

Flynn sınıflandırması · 1966

1966'da işlemcileri aynı anda işlenen komut akışı ve veri akışı sayısına göre SISD, SIMD, MISD ve MIMD olarak dört sınıfa ayırdı.

TEKNİK ÖNEMİ

Skalar çekirdek, vektör ve dizi işlemcisi, GPU ve çok çekirdekli işlemci arasındaki fark bu sınıflandırmayla tek bir tabloda anlatılır. x86'daki SSE ve AVX komutları SIMD sınıfına girer.

SIMD **MIMD** **paralel mimari**

Sınıflandırma “Very High-Speed Computing Systems” (*Proceedings of the IEEE*, 1966) makalesinde yayımlandı. Flynn, Stanford Üniversitesi'nde bilgisayar mimarisi alanında çalıştı.

Teknik kavramlar: SISD, SIMD, MISD, MIMD, komut akışı, veri akışı, vektör işleme.

JB

John Bardeen

Transistör · 1947 · Bell Laboratuvarları

Walter Brattain ile birlikte 1947 nin sonunda Bell Laboratuvarları nda ilk çalışan transistörü, nokta temaslı transistörü gerçekleştirdi.

TEKNİK ÖNEMİ

Transistör, elektron lambasının yaptığı anahtarlama ve yükseltme işini çok daha küçük, az güç harcayan ve dayanıklı bir yarı iletken aygıtla yapar. Bir mikroişlemcideki her mantık kapısı transistörlerden kurulur.

transistör **yarı iletken** **Bell Laboratuvarları**

Bardeen, Brattain ve Shockley transistör çalışmaları için 1956 Nobel Fizik Ödülü'nü paylaştı. Bardeen 1972'de süperiletkenlik kuramıyla (BCS kuramı) ikinci bir Nobel Fizik Ödülü aldı; bu ödülü iki kez kazanan tek kişidir.

WB

Walter Brattain

Transistör · 1947 · deneysel fizik

Bell Laboratuvarları'nda Bardeen ile birlikte germanyum kristali üzerinde ilk nokta temaslı transistörü kurup çalıştırdı (1947).

Teknik kavramlar: nokta temaslı transistör, germanyum, yükseltme, anahtarlama, katı hâl elektroniği.

TEKNİK ÖNEMİ

Kuramsal fikrin çalışan bir aygıta dönüşmesi, yarı iletken yüzeyler üzerindeki deneysel çalışmayla mümkün oldu. Transistör 1948'de duyuruldu ve mikroelektronik çağını başlattı.

transistör **germanyum** **yüzey fiziği**

Brattain, yarı iletken yüzeylerin elektriksel davranışı üzerine çalışan deneysel bir fizikçiydi. 1956 Nobel Fizik Ödülü'nü Bardeen ve Shockley ile paylaştı.

Teknik kavramlar: nokta temaslı transistör, yarı iletken yüzeyi, germanyum kristali, akım kazancı.

WS

William Shockley

Eklem transistörü · Silikon Vadisi'nin başlangıcı

Bell Laboratuvarları'nda transistör grubunu yönetti ve eklem (jonksiyon) transistörünün kuramını geliştirdi; aygıt 1951'de duyuruldu.

TEKNİK ÖNEMİ

Eklem transistörü, nokta temaslı transistörden daha sağlam ve seri üretime daha uygundu. Tümleşik devrelerdeki transistörler bu çizgiden gelir.

eklem transistörü **yarı iletken sanayii** **Fairchild**

Shockley 1956'da Mountain View'da (Kaliforniya) Shockley Semiconductor Laboratory'yi kurdu. 1957'de buradan ayrılan sekiz araştırmacı, aralarında Robert Noyce ve Gordon Moore'un da bulunduğu Fairchild Semiconductor'ı kurdu. Intel daha sonra bu ekipten doğdu. Shockley 1956 Nobel Fizik Ödülü'nü Bardeen ve Brattain ile paylaştı.

RN

Robert Noyce

Tümleşik devreler · Intel'in kurucularından

Tümleşik devrenin mucitlerinden biridir; Intel'i birlikte kurdu.

TEKNİK ÖNEMİ

Tümleşik devreler, yoğun ve güvenilir işlemci yongalarının ekonomik olarak üretilebilmesini sağladı.

tümleşik devre **Intel** **yarı iletken üretimi**

Noyce'un düzlemsel tümleşik devreler ve yarı iletken üretimi üzerindeki çalışmaları, ticari mikroişlemcilerin teknolojik altyapısını oluşturdu.

GM

Gordon Moore

Yarı iletken ölçeklemesi · Intel

Intel'in kurucularındandır; sonradan Moore yasası adını alan ölçekleme eğilimini dile getirdi.

Teknik kavramlar: tümleşik devre (IC), Intel, yarı iletken üretimi, düzlemsel (planar) süreç.

TEKNİK ÖNEMİ

Transistör yoğunluğunun hızla artması, onlarca yıl boyunca işlemci başarımını, tümleştirme düzeyini, önbellek boyutunu ve mimari karmaşıklığı belirledi.

Moore yasası **transistör yoğunluğu** **ölçekleme**

Moore'un gözlemleri, sektörde yarı iletken tümleştirme ve ürün yol haritaları için bir planlama ilkesine dönüştü.

Teknik kavramlar: Moore yasası (Moore's law), transistör yoğunluğu, ölçekleme, tümleştirme, Intel.

JK

Jack Kilby

Tümleşik devre

İlk tümleşik devrelerden birini yapıp çalıştırdı.

TEKNİK ÖNEMİ

Tümleştirme, elle kablolanan düzeneklerin yerine küçük yarı iletken devreleri koydu ve giderek karmaşıklaşan sayısal sistemlerin önünü açtı.

tümleşik devre **yarı iletken tümleştirme** **minyatürleştirme**

Kilby'nin çalışmaları, mikroişlemcilerin doğduğu tümleşik devre çağını başlattı.

Teknik kavramlar: tümleşik devre, yarı iletken tümleştirme, minyatürleştirme.

RD

Robert Dennard

DRAM · ölçekleme

Tek transistörlü DRAM hücresini icat etti ve Dennard ölçeklemesini formüle etti.

TEKNİK ÖNEMİ

DRAM baskın ana bellek teknolojisi oldu; ölçekleme kuralları ise işlemci frekansının ve güç tüketiminin gidişatını etkiledi.

DRAM **bellek** **Dennard ölçeklemesi**

Dennard'ın katkıları, işlemcilerin gelişimini doğrudan bellek yoğunluğuna ve transistörlerin güç özelliklerine bağlar.

FW

Fred Weber

AMD K8 · x86-64 dönemi

K8 kuşağı döneminde AMD'deki büyük işlemci mimarisi çalışmalarını yönetti.

Teknik kavramlar: DRAM, bellek, Dennard ölçeklemesi (Dennard scaling), güç yoğunluğu, yarı iletken fiziği.

TEKNİK ÖNEMİ

AMD64, var olan x86 yazılımlarıyla uyumluluğu koruyarak x86'yı 64 bitlik adreslemeye genişletti.

AMD64 **x86-64** **tümleşik bellek denetleyicisi**

K8 mimarisi 64 bitlik bir ISA uzantısı getirdi ve bellek denetleyicisini işlemci yongasının üzerine taşıdı.

Teknik kavramlar: AMD64, x86-64, tümleşik bellek denetleyicisi, uyumluluk, 64 bitlik ISA.

DM

Dirk Meyer

AMD işlemcileri · x86 gerçeklemesi

AMD işlemcilerinin geliştirilmesinde kilit yöneticilik ve mühendislik görevleri üstlendi.

TEKNİK ÖNEMİ

Günümüz x86 işlemcileri, karmaşık bir ISA'yı derin boru hatlı ve sırasız (out-of-order) iç yürütmeye dönüştürmek zorundadır.

x86 **AMD** **mikromimari**

AMD'nin tasarımları, mimari uyumluluğun mikromimarideki köklü yeniliklerle bir arada var olabileceğini gösterir.

TG

Tom Gunter

Motorola 68000 ailesi

Motorola 68000 projesinin baş mimarıydı; tasarım ekibini 1977'de kurdu.

Teknik kavramlar: x86, AMD, mikromimari, sırasız yürütme, uyumluluk.

TEKNİK ÖNEMİ

68000 ailesi iş istasyonlarında, kişisel bilgisayarlarda ve gömülü sistemlerde etkili oldu.

Motorola 68000 **ISA tasarımı** **yazmaçlar**

Görece dik (orthogonal) komut kümesi ve programcının görebildiği geniş yazmaç kümesiyle ilk x86 işlemcilerinden farklı bir tasarım anlayışı sundu.

CP

Chuck Peddle

MOS 6502

MOS Technology 6502'nin geliştirilmesini yönetti.

Teknik kavramlar: Motorola 68000, ISA tasarımı, yazmaçlar, gömülü sistemler.

TEKNİK ÖNEMİ

Ucuz 6502, mikroişlemcilerin ilk kişisel bilgisayarlarda ve gömülü ürünlerde kullanılabilmesine katkı sağladı.

6502 **düşük maliyetli CPU** **kişisel bilgisayarlar**

6502; Apple II, Commodore bilgisayarları ve pek çok oyun konsolu gibi sistemlerin işlemcisiydi.

BM

Bill Mensch

Motorola 6800 · 6502

Teknik kavramlar: 6502, düşük maliyetli CPU, kişisel bilgisayarlar, gömülü sistemler.

Motorola 6800'ün geliştirilmesinde çalıştı ve 6502'yi birlikte tasarladı.

TEKNİK ÖNEMİ

Çalışmaları, ilk dönemin iki etkili 8 bitlik işlemci ailesini kapsar.

6800 **6502** **8 bitlik işlemciler**

Mensch daha sonra 6502 türevi işlemcilerin geliştirilmesini sürdüren Western Design Center şirketini kurdu.

Teknik kavramlar: 6800, 6502, 8 bitlik işlemciler, CPU gerçeklemesi.

IS

Ivan Sutherland

Bilgisayar sistemleri · VLSI eğitimine etkisi

Araştırmaları ve eğitim çalışmalarıyla bilgisayar sistemleri ve VLSI alanındaki düşüncenin gelişmesine katkıda bulundu.

TEKNİK ÖNEMİ

VLSI tasarım yöntemleri, giderek karmaşıklaşan işlemci yongalarının sistematik biçimde ele alınmasını mümkün kıldı.

VLSI **bilgisayar sistemleri** **tümleşik tasarım**

Daha geniş kapsamlı çalışmaları, günümüz işlemci gerçeklemesinin dayandığı tümleşik sayısal sistem tasarımı kültürünü etkiledi.

Teknik kavramlar: VLSI (çok büyük ölçekli tümleştirme), bilgisayar sistemleri, tümleşik tasarım, yonga tasarım yöntemleri.

JB

John Backus

Fortran · 1957 · IBM

IBM'de ilk yaygın yüksek düzeyli dil olan Fortran'ı ve derleyicisini geliştiren ekibi yönetti (1957).

TEKNİK ÖNEMİ

Fortran derleyicisi, formül biçiminde yazılmış programlardan elle yazılmış koda yakın verimlilikte makine kodu üretilebileceğini gösterdi. Programcılığın assembly'den yüksek düzeyli dillere geçişi bu çalışmayla başladı.

Fortran **derleyici** **yüksek düzeyli dil**

Programlama dillerinin söz dizimini tanımlamakta kullanılan Backus–Naur biçimi (BNF) de onun adını taşır. 1977'de Turing Ödülü'nü aldı.

Teknik kavramlar: derleyici, kaynak kod, makine kodu, söz dizimi, BNF.