Bölüm 4.2: Değişkenler

Mikroişlemciler

DB / DW / DD | Adlandırma kuralları | Bellek yerleşimi | ORG 100h ve .COM | Diziler ve DUP | LEA ve OFFSET | EQU sabitleri | Toplama ve karşılaştırma | Çift sözcük

Sercan KÜLCÜ · Giresun Üniversitesi · Elektrik-Elektronik Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Tanımlama

DB, DW, DD, DUP ve ? ile değişken ve dizi tanımlamak.

Bellek yerleşimi

Derleyicinin değişkenleri hangi bağıl konumlara, hangi bayt sırasıyla yerleştirdiğini çıkarmak.

Adresler

LEA ve OFFSET ile değişken adresini almak ve dizi elemanlarına indisle erişmek.

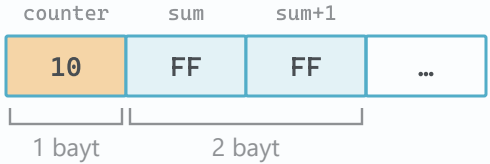
Programlar

Değişkenler arasında veri taşıyan, toplayan ve karşılaştıran programları adım adım izlemek.

Değişkenler

- Bir değişken bir **bellek konumunu** temsil eder.
- Programcı için `var1` adını kullanmak, `5A73:235B` adresini kullanmaktan çok daha kolaydır.
- Derleyici iki temel tür destekler:
  - **BYTE** (8 bit): `ad DB değer` (Define Byte)
  - **WORD** (16 bit): `ad DW değer` (Define Word)
- `DD` (Define Double word, 32 bit) de vardır: 4 bayt.

```
counter DB 10h      ; BYTE
sum     DW 0FFFFh   ; WORD
```



Adlandırma ve İlk Değer

- Değişken adları **harfle başlamalıdır**; sonrasında harf ve rakam kombinasyonları içerebilir.
- Yazmaç ve komut adları (`AX` , `MOV` ...) değişken adı olamaz.
- Ad belirtilmeyen değişkenler de olabilir: adları yoktur ama **adresleri** vardır.
- Herhangi bir sayısal değer (ikili, onluk, onaltılık, karakter) alabilir.
- İlk değer atanmamış değişkenler için `?` kullanılır (emu8086 belleğe 00 koyar).

Tanım	?
`sayi1 DB 5`	geçerli
`1sayi DB 5`	rakamla başlıyor
`ax DW 0`	yazmaç adı
`DB 0Ah`	adsız, ama adresi var
`sonuc DW ?`	ilk değersiz
`harf DB 'A'`	41h

Değer Aralıkları

Direktif	Boyut	İşaretsiz	İşaretli	Örnek
`DB`	1 bayt	0 … 255	−128 … 127	`x DB -1` → FFh
`DW`	2 bayt	0 … 65535	−32768 … 32767	`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

Bellekte yalnızca bit deseni saklanır; **işaretli mi işaretsiz mi** olduğu, değeri kullanan komuta bağlıdır (ör. JGE işaretli, JAE işaretsiz karşılaştırır). `DB 300` bir bayta sığmaz → assembler hata verir.

Kod İzleme: FFh 255 mi, −1 mi?

```
1  ORG 100h
2  MOV AL, x        ; FFh
3  ADD AL, 1        ; 255 + 1 ?
4  MOV AL, x
5  CBW              ; işaretli: -1
6  MOV BX, y        ; FFFEh
7  NEG BX           ; -(-2)
8  RET
9  x DB -1
10 y DW -2
```



Bellekte x = **FFh**, y = **FE FF** (little-endian FFFEh). AL = x = FFh: işaretsiz 255, işaretli −1.

Adım 1 / 6

DEĞİŞKENLER

AX = 00FFh | BX = 0000h | CF = 0 | ZF = 0

DB/DW yalnızca bit desenini saklar; işaretli mi işaretsiz mi olduğuna değeri kullanan komut karar verir.

Kod İzleme: FFh 255 mi, −1 mi?

```
1  ORG 100h
2  MOV AL, x          ; FFh
▶  ADD AL, 1          ; 255 + 1 ?
4  MOV AL, x
5  CBW                ; işaretli: -1
6  MOV BX, y          ; FFFEh
7  NEG BX             ; -(-2)
8  RET
9  x DB -1
10 y DW -2
```



`ADD AL, 1` : işaretsiz bakışla 255 + 1 = 256 bayta sığmaz → AL = **00h**, **CF = 1**. İşaretli bakışla −1 + 1 = 0 doğru: OF = 0.

Adım 2 / 6

DEĞİŞKENLER

AX = 0000h | BX = 0000h | CF = 1 | ZF = 1

DB/DW yalnızca bit desenini saklar; işaretli mi işaretsiz mi olduğuna değeri kullanan komut karar verir.

Kod İzleme: FFh 255 mi, −1 mi?

```
1  ORG 100h
2  MOV AL, x          ; FFh
3  ADD AL, 1          ; 255 + 1 ?
▶  MOV AL, x
5  CBW                ; işaretli: -1
6  MOV BX, y          ; FFFEh
7  NEG BX             ; -(-2)
8  RET
9  x DB -1
10 y DW -2
```



AL yeniden FFh.

Adım 3 / 6

DEĞİŞKENLER

AX = 00FFh | BX = 0000h | CF = 1 | ZF = 1

DB/DW yalnızca bit desenini saklar; işaretli mi işaretsiz mi olduğuna değeri kullanan komut karar verir.

Kod İzleme: FFh 255 mi, −1 mi?

```
1  ORG 100h
2  MOV AL, x          ; FFh
3  ADD AL, 1          ; 255 + 1 ?
4  MOV AL, x
▶  CBW                ; işaretli: -1
6  MOV BX, y          ; FFFEh
7  NEG BX             ; -(-2)
8  RET
9  x DB -1
10 y DW -2
```



`CBW` AL'yi işaretli kabul eder: MSB = 1 → AH = FFh, AX = **FFFFh** = −1 (16 bit).

Adım 4 / 6

DEĞİŞKENLER

AX = **FFFFh** | BX = **0000h** | CF = **1** | ZF = **1**

DB/DW yalnızca bit desenini saklar; işaretli mi işaretsiz mi olduğuna değeri kullanan komut karar verir.

Kod İzleme: FFh 255 mi, −1 mi?

```
1  ORG 100h
2  MOV AL, x          ; FFh
3  ADD AL, 1          ; 255 + 1 ?
4  MOV AL, x
5  CBW                ; işaretli: -1
6  MOV BX, y          ; FFFEh
7  NEG BX             ; -(-2)
8  RET
9  x DB -1
10 y DW -2
```



BX = y = **FFFEh**: işaretsiz 65534, işaretli −2.

Adım 5 / 6

DEĞİŞKENLER

AX = **FFFFh** | BX = **FFFEh** | CF = **1** | ZF = **1**

DB/DW yalnızca bit desenini saklar; işaretli mi işaretsiz mi olduğuna değeri kullanan komut karar verir.

Kod İzleme: FFh 255 mi, −1 mi?

```
1  ORG 100h
2  MOV AL, x          ; FFh
3  ADD AL, 1          ; 255 + 1 ?
4  MOV AL, x
5  CBW                ; işaretli: -1
6  MOV BX, y          ; FFFEh
▶  NEG BX             ; -(-2)
8  RET
9  x DB -1
10 y DW -2
```



`NEG BX`: 0 − (−2) = **0002h**. Bit deseninin anlamını komut seçti: ADD/CF işaretsiz, CBW/NEG işaretli yorumladı.

Adım 6 / 6

DEĞİŞKENLER

AX = **FFFFh** | BX = **0002h** | CF = **1** | ZF = **0**

DB/DW yalnızca bit desenini saklar; işaretli mi işaretsiz mi olduğuna değeri kullanan komut karar verir.

Değişkenin Türü ve Boyut Denetimi

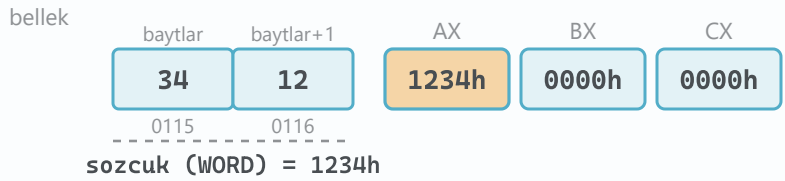
- Assembler her değişken adının yanında bir **tür** saklar: DB → BYTE (1), DW → WORD (2), DD → DWORD (4).
- Komuttaki diğer işlenen yazmaçsa boyutlar eşleşmelidir; eşleşmezse derleme durur.
- Sabit yazarken boyutu değişkenin türü belirler: `MOV w, 5` bir sözcük yazar.
- Değişken adı olmayan `[BX]` gibi işlenenlerin türü yoktur; sabitle kullanılırken `BYTE PTR` ya da `WORD PTR` gerekir.
- `BYTE PTR` türü yalnızca o komut için değiştirir; değişkenin tanımı aynı kalır.

Komut (w DW 1234h, b DB 5)	Sonuç
`MOV AL, w`	hata: 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 (sözcük)
`MOV AX, b`	hata: 2 bayt / 1 bayt
`MOV [BX], 5`	hata: boyut belirsiz
`MOV b, w`	hata: bellekten belleğe
`INC b`	b = 6 (bayt artırılır)

Tür denetimi derleme anında yapılır ve gözden kaçan pek çok hatayı yakalar. İşlemci çalışırken türü bilmez; yalnızca komutun kodladığı boyutta okur ya da yazar.

Kod İzleme: Aynı Belleğe İki Ad (LABEL)

```
1 ORG 100h
2 MOV AX, sozcuk      ; WORD
3 MOV BL, baytlar     ; düşük
4 MOV BH, baytlar+1   ; yüksek
5 MOV baytlar, 0FFh   ; tek bayt
6 MOV CX, sozcuk      ; sözcük
7 RET
8 sozcuk LABEL WORD
9 baytlar DB 34h, 12h
```



WORD olarak okundu

`sozcuk LABEL WORD` yer ayırmaz; yalnızca 0115h adresine WORD türünde ikinci bir ad verir. `MOV AX, sozcuk` iki baytı birden okur: AX = **1234h**.

Adım 1 / 6

DEĞİŞKENLER

AX = **1234h** BX = **0000h** CX = **0000h**

ad LABEL tür, kendisinden sonraki ilk bayta yeni bir ad ve tür verir. Bir tamponu hem bayt bayt hem sözcük sözcük işlemek için PTR yazmaktan daha okunaklıdır.

Kod İzleme: Aynı Belleğe İki Ad (LABEL)

```
1 ORG 100h
2 MOV AX, sozcuk      ; WORD
▶ MOV BL, baytlar     ; düşük
4 MOV BH, baytlar+1   ; yüksek
5 MOV baytlar, 0FFh   ; tek bayt
6 MOV CX, sozcuk      ; sözcük
7 RET
8 sozcuk LABEL WORD
9 baytlar DB 34h, 12h
```



`baytlar` aynı adresin BYTE türündeki adıdır: BL = **34h**. PTR yazmaya gerek kalmadı.

Adım 2 / 6

DEĞİŞKENLER

AX = **1234h** | BX = **0034h** | CX = **0000h**

ad LABEL tür, kendisinden sonraki ilk bayta yeni bir ad ve tür verir. Bir tamponu hem bayt bayt hem sözcük sözcük işlemek için PTR yazmaktan daha okunaklıdır.

Kod İzleme: Aynı Belleğe İki Ad (LABEL)

```
1 ORG 100h
2 MOV AX, sozcuk      ; WORD
3 MOV BL, baytlar     ; düşük
▶ MOV BH, baytlar+1   ; yüksek
5 MOV baytlar, 0FFh   ; tek bayt
6 MOV CX, sozcuk      ; sözcük
7 RET
8 sozcuk LABEL WORD
9 baytlar DB 34h, 12h
```



`baytlar+1` ikinci bayt: BH = **12h**. BX = 1234h; little-endian sıranın sonucu.

Adım 3 / 6

DEĞİŞKENLER

AX = **1234h** | BX = **1234h** | CX = **0000h**

ad LABEL tür, kendisinden sonraki ilk bayta yeni bir ad ve tür verir. Bir tamponu hem bayt bayt hem sözcük sözcük işlemek için PTR yazmaktan daha okunaklıdır.

Kod İzleme: Aynı Belleğe İki Ad (LABEL)

```
1 ORG 100h
2 MOV AX, sozcuk      ; WORD
3 MOV BL, baytlar     ; düşük
4 MOV BH, baytlar+1   ; yüksek
▶ MOV baytlar, 0FFh   ; tek bayt
6 MOV CX, sozcuk      ; sözcük
7 RET
8 sozcuk LABEL WORD
9 baytlar DB 34h, 12h
```



tek bayt yazıldı

Tür BYTE olduğu için `MOV baytlar, 0FFh` yalnızca 0115h adresine yazar: bellek **FF 12**.

Adım 4 / 6

DEĞİŞKENLER

AX = **1234h** BX = **1234h** CX = **0000h**

ad LABEL tür, kendisinden sonraki ilk bayta yeni bir ad ve tür verir. Bir tamponu hem bayt bayt hem sözcük sözcük işlemek için PTR yazmaktan daha okunaklıdır.

Kod İzleme: Aynı Belleğe İki Ad (LABEL)

```
1 ORG 100h
2 MOV AX, sozcuk      ; WORD
3 MOV BL, baytlar     ; düşük
4 MOV BH, baytlar+1   ; yüksek
5 MOV baytlar, 0FFh   ; tek bayt
▶ MOV CX, sozcuk      ; sözcük
7 RET
8 sozcuk LABEL WORD
9 baytlar DB 34h, 12h
```



Aynı bellek WORD adıyla okununca değişiklik görülür: CX = **12FFh**. İki ad, tek bellek konumu.

Adım 5 / 6

DEĞİŞKENLER

AX = **1234h** BX = **1234h** CX = **12FFh**

ad LABEL tür, kendisinden sonraki ilk bayta yeni bir ad ve tür verir. Bir tamponu hem bayt bayt hem sözcük sözcük işlemek için PTR yazmaktan daha okunaklıdır.

Kod İzleme: Aynı Belleğe İki Ad (LABEL)

```
1 ORG 100h
2 MOV AX, sozcuk      ; WORD
3 MOV BL, baytlar     ; düşük
4 MOV BH, baytlar+1   ; yüksek
5 MOV baytlar, 0FFh   ; tek bayt
6 MOV CX, sozcuk      ; sözcük
▶ RET
8 sozcuk LABEL WORD
9 baytlar DB 34h, 12h
```



RET. `LABEL` bellekte bayt üretmedi; program boyutu iki baytlık tanım kadar arttı.

Adım 6 / 6

DEĞİŞKENLER

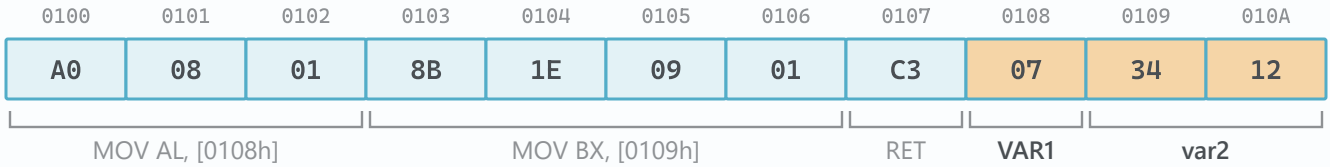
`AX = 1234h` `BX = 1234h` `CX = 12FFh`

ad LABEL tür, kendisinden sonraki ilk bayta yeni bir ad ve tür verir. Bir tamponu hem bayt bayt hem sözcük sözcük işlemek için PTR yazmaktan daha okunaklıdır.

Örnek Kod

```
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
```

- Kod önce, değişkenler **RET'ten sonra**: CPU onlara hiç ulaşmaz.
- Derleyici kaynağı bir **bayt dizisine** çevirir ve değişken adlarını bellek konumlarıyla değiştirir.
- Büyük-küçük harfe duyarsızdır: `VAR1` ile `var1` aynı değişkendir.



Kod İzleme: var1 ve var2

```
▶ ORG 100h
2 MOV AL, var1    ; AL ← var1
3 MOV BX, var2    ; BX ← var2
4 RET             ; programı bitir
5 VAR1 DB 7       ; BYTE, 7
6 var2 DW 1234h   ; WORD, 1234h
```

AX	00	00	BX	00	00	IP	0100
	AH	AL		BH	BL		

0B56:0107 … 010A

0107	0108	0109	010A
C3	07	34	12

Program 0B56:0100'e yüklendi; DS = CS = 0B56h, IP = 0100h.

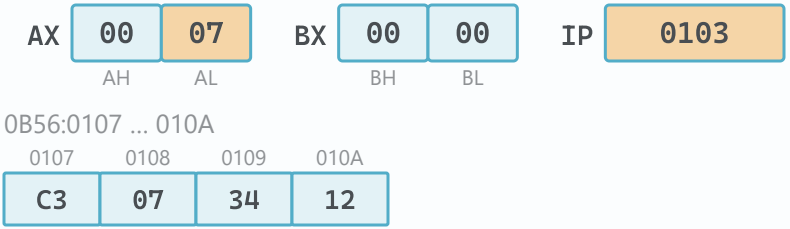
Adım 1 / 4

DEĞİŞKENLER

IP = 0100h AL = 00h BX = 0000h

Kod İzleme: var1 ve var2

```
1 ORG 100h
▶ MOV AL, var1   ; AL ← var1
3 MOV BX, var2   ; BX ← var2
4 RET            ; programı bitir
5 VAR1 DB 7      ; BYTE, 7
6 var2 DW 1234h  ; WORD, 1234h
```



`MOV AL, [0108h]` : VAR1'in değeri → AL = **07h**. IP = 0103h.

Adım 2 / 4

DEĞİŞKENLER

`IP = 0103h` `AL = 07h` `BX = 0000h`

Kod İzleme: var1 ve var2

```
1 ORG 100h
2 MOV AL, var1    ; AL ← var1
▶ MOV BX, var2    ; BX ← var2
4 RET             ; programı bitir
5 VAR1 DB 7       ; BYTE, 7
6 var2 DW 1234h   ; WORD, 1234h
```

AX 00 07 (AH AL) BX 12 34 (BH BL) IP 0107

0B56:0107 … 010A

0107	0108	0109	010A
C3	07	34	12

`MOV BX, [0109h]` : 34h düşük, 12h yüksek bayt → BX = **1234h**. IP = 0107h.

Adım 3 / 4

DEĞİŞKENLER

IP = 0107h AL = 07h BX = 1234h

Kod İzleme: var1 ve var2

```
1 ORG 100h
2 MOV AL, var1    ; AL ← var1
3 MOV BX, var2    ; BX ← var2
▶ RET             ; programı bitir
5 VAR1 DB 7       ; BYTE, 7
6 var2 DW 1234h   ; WORD, 1234h
```

AX	AH	AL	BX	BH	BL	IP
	00	07	12	34		0000

0B56:0107 … 010A

0107	0108	0109	010A
C3	07	34	12

RET (0107h'de C3) yığından 0000h'yi alır: IP = **0000h**, PSP'deki INT 20h programı bitirir. 0108h'den sonrası veridir.

Adım 4 / 4

DEĞİŞKENLER

IP = **0000h** · AL = **07h** · BX = **1234h**

BELLEK YERLEŞİMİ

Derleyici Ne Yapar?

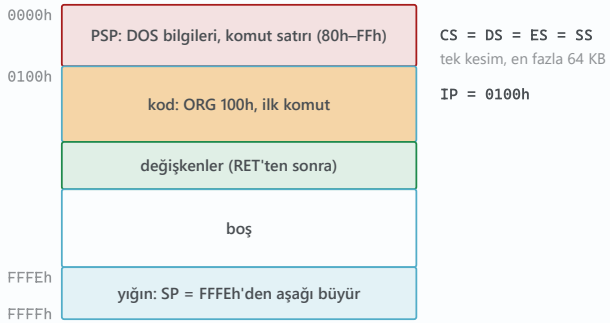
- Kaynak kodu bir dizi bayta dönüştürür; değişken adlarını **bağıl konumlarla** değiştirir: `MOV AL, var1` → `MOV AL, [0108h]`.
- .COM dosyaları yüklendiğinde **DS, CS yazmacının değerini alır** (ES ve SS de aynıdır).
- **Bağıl konum** (offset), bellek kesiminin başlangıcından itibaren olan uzaklıktır.
- `VAR1`'in bağıl konumu **0108h**, tam adresi **0B56:0108**'dir.
- `var2`'nin bağıl konumu **0109h**, tam adresi **0B56:0109**'dur.
  - WORD türünde olduğu için 2 bayt kaplar (0109h ve 010Ah).
  - Düşük bayt düşük adreste saklanır: **34h, 12h'den önce** yer alır (little-endian).

Kesim değeri (0B56h) DOS'un programı nereye yüklediğine bağlıdır ve her çalıştırmada farklı olabilir; bağıl konumlar ise derleme anında sabittir.

BELLEK YERLEŞİMİ

ORG 100h Direktifi

- Derleyiciye çalıştırılabilir dosyanın **yükleneceği adresi** söyler; değişken adları adreslerle değiştirilirken dikkate alınır.
- **.COM** dosyalarında kullanılır; bu dersteki örneklerin çoğu emu8086 .COM şablonuyla yazılır.
- İşletim sistemi kesimin **ilk 256 baytında** (PSP) programla ilgili verileri tutar; komut satırı parametreleri bu alandadır.
- **.EXE** dosyaları 0000h bağıl konumuna yüklenir ve değişkenler için ayrı bir kesim kullanır.
- Direktifler gerçek makine koduna **dönüştürülmez**.



BELLEK YERLEŞİMİ

Hizalama: Sözcük Hangi Adreste?




```
a   DB 5          ; 0200h
    EVEN          ; 0201h: dolgu
w   DW 1234h      ; 0202h (çift)
```

- 8086'nın veri yolu 16 bittir: çift adresten başlayan bir sözcük tek yol döngüsünde okunur.
- Tek adresteki sözcük için iki yol döngüsü gerekir; 8086 bunu kendiliğinden yapar, yalnızca daha yavaştır (sözcük erişimi başına 4 saat turu).
- MASM ve TASM'daki `EVEN` direktifi konum sayacını çift sayıya yuvarlar; gerekirse araya bir dolgu baytı koyar. Tanımlar yer değiştirmez, yalnızca kayar.
- 8 bitlik veri yolu olan 8088'de her sözcük zaten iki döngüdür; hizalama orada fark yaratmaz.

Örnek: `MOV AX, w` 8086'da 10 saat turudur; w tek adresteyse 14 olur. Bir döngüde 1000 kez okunan sözcük için fark 4000 tur, 5 MHz'de 0,8 ms eder. Bayt değişkenler için hizalama gerekmez.

BELLEK YERLEŞİMİ

Değişkenler Nereye Yazılır?

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.

DS yüklenmezse

.EXE programı başladığında DS, PSP'yi gösterir. `MOV AL, x` doğru bağıl konumu kullanır ama yanlış kesimden okur: hata vermeden yanlış değer gelir.

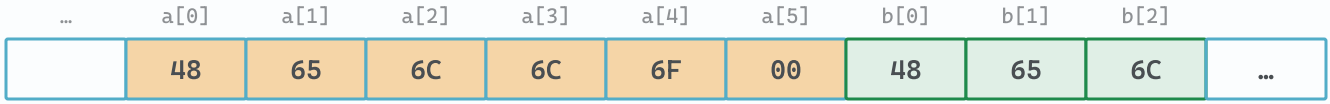
Kesimin önemi

.COM'da CS = DS olduğu için kod ile verinin aynı kesimde durması sorun değildir. .EXE'de kod CS'de, veri DS'de durur; değişken adresi DS'ye göre hesaplanır.

Diziler

- Diziler **değişken zincirleri** olarak düşünülebilir: ardışık bellek konumlarında saklanan veri gruplarıdır.
- Tırnak içindeki metin otomatik olarak **bayt dizisine** (ASCII kodları) dönüştürülür.
- Dizi elemanlarına bellek adresi üzerinden erişilir: ilk eleman dizinin adresinde, sonrakiler ardından.

```
a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h
b DB 'Hello', 0
```



a ve b bellekte aynı baytları tutar: 48h 'H', 65h 'e', 6Ch 'l', 6Ch 'l', 6Fh 'o', 00h (dizgi sonu).

Kod İzleme: Dizi Elemanlarına Erişim

```
1 MOV AL, a[3]   ; a + 3
2 MOV SI, 0
3 MOV BL, a[SI]  ; a + 0
4 INC SI
5 MOV BH, a[SI]  ; a + 1
6 MOV a[SI], 'E' ; a[1] ← 45h
7 RET
8 a DB 48h,65h,6Ch,6Ch,6Fh,0
```

AX	BX	SI
AH: 00 / AL: 6C	BH: 00 / BL: 00	0000

a dizisi

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
48	65	6C	6C	6F	00

`a[3]`, **a + 3** bayt demektir: AL = **6Ch** ('l').

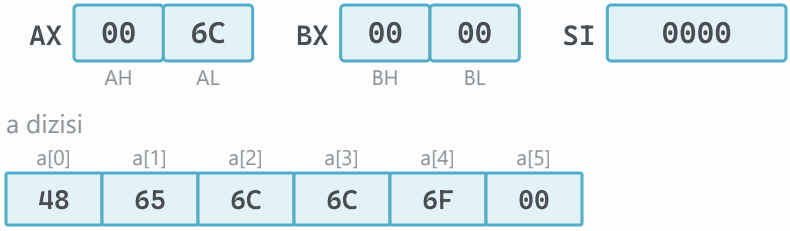
Adım 1 / 7

DEĞİŞKENLER

`AL = 6Ch` `SI = 0000h` `BL = 00h` `BH = 00h`

Kod İzleme: Dizi Elemanlarına Erişim

```
1 MOV AL, a[3]   ; a + 3
2 MOV SI, 0
3 MOV BL, a[SI]  ; a + 0
4 INC SI
5 MOV BH, a[SI]  ; a + 1
6 MOV a[SI], 'E' ; a[1] ← 45h
7 RET
8 a DB 48h,65h,6Ch,6Ch,6Fh,0
```



İndis yazmacı SI = 0.

Adım 2 / 7

DEĞİŞKENLER

AL = 6Ch | SI = 0000h | BL = 00h | BH = 00h

Kod İzleme: Dizi Elemanlarına Erişim

```
1 MOV AL, a[3]   ; a + 3
2 MOV SI, 0
▶ MOV BL, a[SI]  ; a + 0
4 INC SI
5 MOV BH, a[SI]  ; a + 1
6 MOV a[SI], 'E' ; a[1] ← 45h
7 RET
8 a DB 48h,65h,6Ch,6Ch,6Fh,0
```

AX	00	6C	BX	00	48	SI	0000
	AH	AL		BH	BL		

a dizisi

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
48	65	6C	6C	6F	00

`a[SI]` = [a + SI]: BL = **48h** ('H').

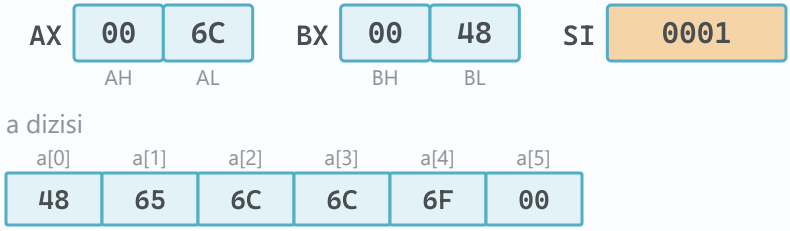
Adım 3 / 7

DEĞİŞKENLER

AL = **6Ch** | SI = **0000h** | BL = **48h** | BH = **00h**

Kod İzleme: Dizi Elemanlarına Erişim

```
1 MOV AL, a[3]   ; a + 3
2 MOV SI, 0
3 MOV BL, a[SI]  ; a + 0
▶ INC SI
5 MOV BH, a[SI]  ; a + 1
6 MOV a[SI], 'E' ; a[1] ← 45h
7 RET
8 a DB 48h,65h,6Ch,6Ch,6Fh,0
```



SI = 1: bir sonraki bayt.

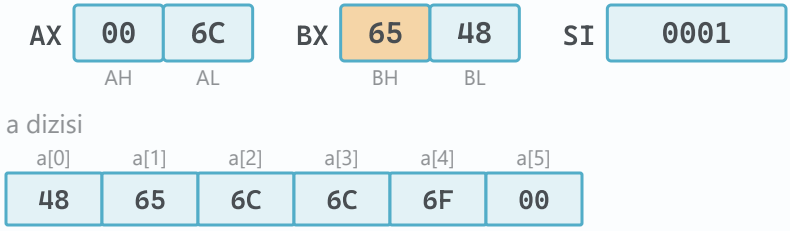
Adım 4 / 7

DEĞİŞKENLER

AL = 6Ch | SI = 0001h | BL = 48h | BH = 00h

Kod İzleme: Dizi Elemanlarına Erişim

```
1 MOV AL, a[3]   ; a + 3
2 MOV SI, 0
3 MOV BL, a[SI]  ; a + 0
4 INC SI
▶ MOV BH, a[SI]  ; a + 1
6 MOV a[SI], 'E' ; a[1] ← 45h
7 RET
8 a DB 48h,65h,6Ch,6Ch,6Fh,0
```



BH = a[1] = **65h** ('e'). BX = 6548h.

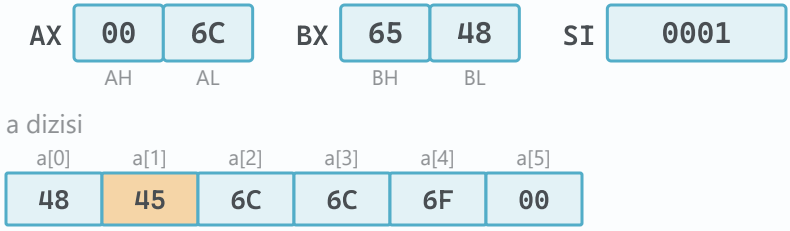
Adım 5 / 7

DEĞİŞKENLER

AL = **6Ch** | SI = **0001h** | BL = **48h** | BH = **65h**

Kod İzleme: Dizi Elemanlarına Erişim

```
1 MOV AL, a[3]   ; a + 3
2 MOV SI, 0
3 MOV BL, a[SI]  ; a + 0
4 INC SI
5 MOV BH, a[SI]  ; a + 1
▶ MOV a[SI], 'E' ; a[1] ← 45h
7 RET
8 a DB 48h,65h,6Ch,6Ch,6Fh,0
```



Dizi bayt türünde olduğu için boyut belli: a[1] ← **45h** ('E'). Dizi artık “HEllo”.

Adım 6 / 7

DEĞİŞKENLER

AL = **6Ch** SI = **0001h** BL = **48h** BH = **65h**

Kod İzleme: Dizi Elemanlarına Erişim

```
1 MOV AL, a[3]   ; a + 3
2 MOV SI, 0
3 MOV BL, a[SI]  ; a + 0
4 INC SI
5 MOV BH, a[SI]  ; a + 1
6 MOV a[SI], 'E' ; a[1] ← 45h
▶ RET
8 a DB 48h,65h,6Ch,6Ch,6Fh,0
```

AX	AH	AL	BX	BH	BL	SI
	00	6C		65	48	0001

a dizisi

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
48	45	6C	6C	6F	00

RET. Dizi tanımı kodun sonunda, yürütülmeyen bölgede.

Adım 7 / 7

DEĞİŞKENLER

AL = 6Ch · SI = 0001h · BL = 48h · BH = 65h

Erişim Yolları ve DUP

Elemana erişim

- Köşeli parantezle sabit indis: `MOV AL, a[3]`
- Bellek indis yazmaçları (BX, SI, DI, BP) ile: `MOV SI, 3` · `MOV AL, a[SI]`
- Eşdeğer yazımlar: `a[SI]` = `[a + SI]` = `[SI + a]`
- İndis **bayt** cinsindendir, eleman sayısı değil.

DUP işleci

Büyük dizi için: `sayı DUP(değer(ler))`

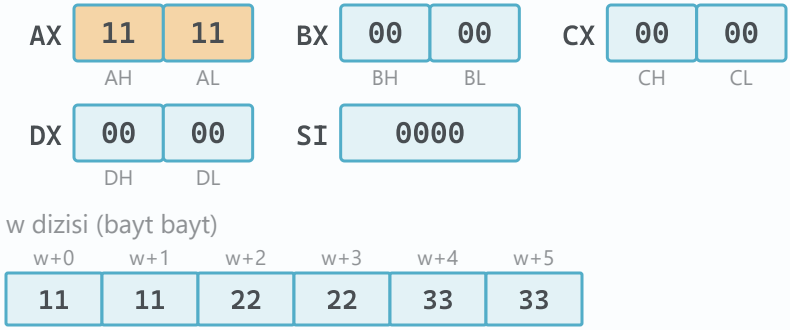
- `c DB 5 DUP(9)` = `c DB 9, 9, 9, 9, 9`
- `d DB 5 DUP(1, 2)` = `d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2`
- `tampon DB 100 DUP(?)` : 100 baytlık ilk değersiz alan
- `tablo DW 10 DUP(0)` : 10 sözcük = 20 bayt



DW ile **sayı dizileri** tanımlanabilir (`w DW 1000h, 2000h`); kullanılamadığı şey **metin dizgileridir** (`DW 'Hello'` geçersiz). Değerler 255'ten büyük ya da −128'den küçükse DW gerekir.

Kod İzleme: DW Dizisinde İndis

```
▶ MOV AX, w[0]   ; 1. eleman
2 MOV BX, w[2]   ; 2. eleman
3 MOV SI, 4
4 MOV CX, w[SI]  ; 3. eleman
5 MOV DX, w[1]   ; dikkat!
6 RET
7 w DW 1111h, 2222h, 3333h
```



w[0]: ilk sözcük → AX = **1111h**.

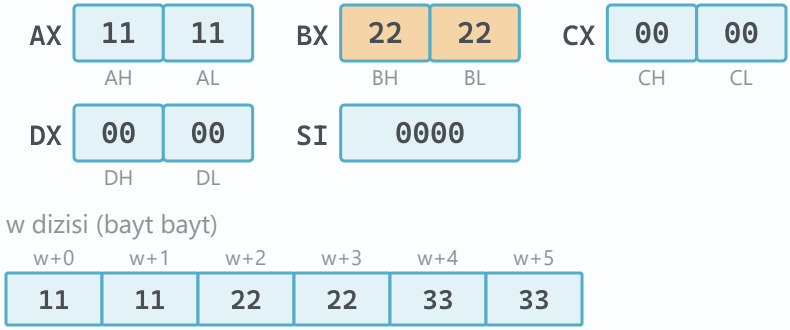
Adım 1 / 5

DEĞİŞKENLER

`AX = 1111h` `BX = 0000h` `SI = 0000h` `CX = 0000h` `DX = 0000h`

Kod İzleme: DW Dizisinde İndis

```
1 MOV AX, w[0]   ; 1. eleman
▶ MOV BX, w[2]   ; 2. eleman
3 MOV SI, 4
4 MOV CX, w[SI]  ; 3. eleman
5 MOV DX, w[1]   ; dikkat!
6 RET
7 w DW 1111h, 2222h, 3333h
```



İkinci eleman **w[2]**'dedir (her eleman 2 bayt): BX = 2222h.

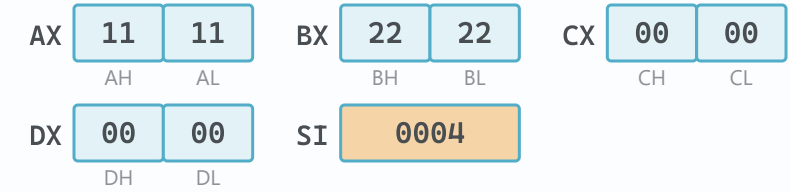
Adım 2 / 5

DEĞİŞKENLER

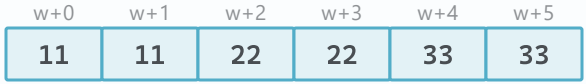
AX = **1111h** · BX = **2222h** · SI = **0000h** · CX = **0000h** · DX = **0000h**

Kod İzleme: DW Dizisinde İndis

```
1 MOV AX, w[0]   ; 1. eleman
2 MOV BX, w[2]   ; 2. eleman
▶ MOV SI, 4
4 MOV CX, w[SI]  ; 3. eleman
5 MOV DX, w[1]   ; dikkat!
6 RET
7 w DW 1111h, 2222h, 3333h
```



w dizisi (bayt bayt)



i. elemanın indisi = **2 × i**: üçüncü eleman için SI = 4.

Adım 3 / 5

DEĞİŞKENLER

AX = **1111h** · BX = **2222h** · SI = **0004h** · CX = **0000h** · DX = **0000h**

Kod İzleme: DW Dizisinde İndis

```
1 MOV AX, w[0]   ; 1. eleman
2 MOV BX, w[2]   ; 2. eleman
3 MOV SI, 4
▶ MOV CX, w[SI]  ; 3. eleman
5 MOV DX, w[1]   ; dikkat!
6 RET
7 w DW 1111h, 2222h, 3333h
```



CX = w[SI] = **3333h**.

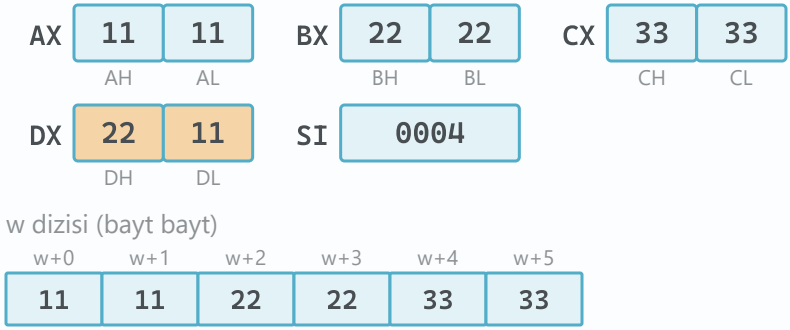
Adım 4 / 5

DEĞİŞKENLER

AX = **1111h** BX = **2222h** SI = **0004h** CX = **3333h** DX = **0000h**

Kod İzleme: DW Dizisinde İndis

```
1 MOV AX, w[0]   ; 1. eleman
2 MOV BX, w[2]   ; 2. eleman
3 MOV SI, 4
4 MOV CX, w[SI]  ; 3. eleman
▶ MOV DX, w[1]   ; dikkat!
6 RET
7 w DW 1111h, 2222h, 3333h
```



w[1] elemanın ortasından okur: baytlar 11h, 22h → DX = **2211h**. Sık yapılan hata!

Adım 5 / 5

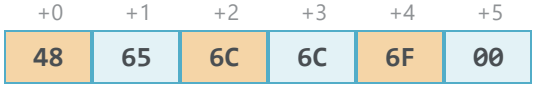
DEĞİŞKENLER

`AX = 1111h` `BX = 2222h` `SI = 0004h` `CX = 3333h` `DX = 2211h`

Tanımdan Bayta

```
b DB 'Hello', 0
```

DB liste | DW | DUP | DW ?, −1 | DD | hata



b: 6 bayt · öğe boyutu 1 bayt · renk değişimi = bir öğe

Yazılan DB / DW / DD satırı baytlara açılır. Sayılar (onluk, `h`, `b`, `o`), `'metin'` (yalnız DB), `?` ve `n DUP(…)` desteklenir. Hücre üstü: değişkenin başından bayt uzaklığı.

Dizgiler ve Sonlandırıcılar

```
s1 DB 'Merhaba', 0              ; sıfırla biten dizgi
s2 DB 'Merhaba$'                ; '$' ile biten (INT 21h / AH = 09h)
s3 DB 13, 10, 'Satır', 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)

- Dizginin bellekte kapladığı yer = karakter sayısı + sonlandırıcı: s1 için 8 bayt. Sonlandırıcı unutulursa yazdırma, dizginin ardındaki baytları da karakter sanıp devam eder.
- '$' ile biten bir dizgi '$' karakterini yazdıramaz. Türkçe harflerin (ç, ğ, ş…) kodu kullanılan kod sayfasına bağlıdır; ASCII'nin 0–127 aralığında değildir.

Karakter Sabitleri ve ASCII Aritmetiği

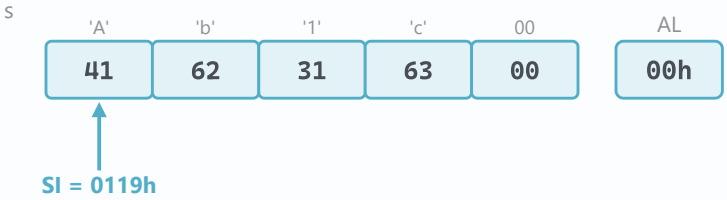
Karakterler	ASCII kodları
'0' … '9'	30h … 39h
'A' … 'Z'	41h … 5Ah
'a' … 'z'	61h … 7Ah
boşluk · '$'	20h · 24h
CR · LF	0Dh · 0Ah

- Tırnak içindeki karakter assembler için yalnızca bir sayıdır: `MOV AL, 'A'` = `MOV AL, 41h`.
- Rakam karakterinden sayıya: `SUB AL, '0'` ('7' = 37h → 7). Ters yön: `ADD AL, '0'`.
- Büyük ve küçük harf arasındaki fark **20h**, yani yalnızca bit 5'tir: `SUB AL, 20h` ya da `AND AL, 0DFh` büyütür, `OR AL, 20h` küçültür.

`AND AL, 0DFh` yalnızca harflerde doğru çalışır. '1' (31h) üzerinde uygulanırsa sonuç 11h olur ve rakam bozulur; bu yüzden harf olup olmadığı önce denetlenir ('a' ≤ AL ≤ 'z').

Kod İzleme: Dizgiyi Büyük Harfe Çevirme

```
1  ORG 100h
2  LEA SI, s
3  don:
4  MOV AL, [SI]
5  CMP AL, 0          ; sonlandırıcı?
6  JE bitti
7  CMP AL, 'a'
8  JB sonraki
9  CMP AL, 'z'
10 JA sonraki
11 SUB AL, 20h        ; 'a' - 'A'
12 MOV [SI], AL
13 sonraki:
14 INC SI
15 JMP don
16 bitti:
17 RET
18 s DB 'Ab1c', 0
```



`LEA SI, s` : SI = **0119h**, dizginin ilk baytı. Dizgi 'Ab1c' ve sonlandırıcı 0.

Adım 1 / 6

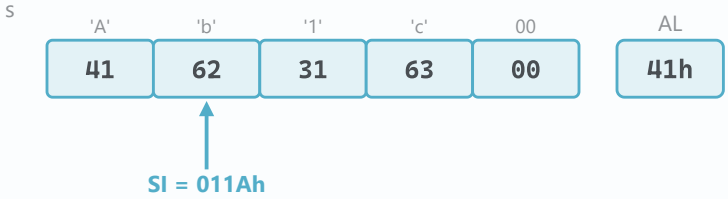
DEĞİŞKENLER

`SI = 0119h` `AL = 00h` `ZF = 0`

Dizgi yerinde değiştirildi: her bayt okunur, harfse 20h çıkarılıp aynı adrese yazılır. Uzunluk yerine sonlandırıcı (0) kullanıldı.

Kod İzleme: Dizgiyi Büyük Harfe Çevirme

```
1  ORG 100h
2  LEA SI, s
3  don:
▶  MOV AL, [SI]
▶  CMP AL, 0          ; sonlandırıcı?
▶  JE bitti
▶  CMP AL, 'a'
▶  JB sonraki
9  CMP AL, 'z'
10 JA sonraki
11 SUB AL, 20h        ; 'a' - 'A'
12 MOV [SI], AL
13 sonraki:
▶  INC SI
15 JMP don
16 bitti:
17 RET
18 s DB 'Ab1c', 0
```



'A' (41h): 0 değil; `CMP AL, 'a'` → 41h < 61h, `JB` atlar. Büyük harf olduğu için yazılmadı; SI ilerledi.

Adım 2 / 6

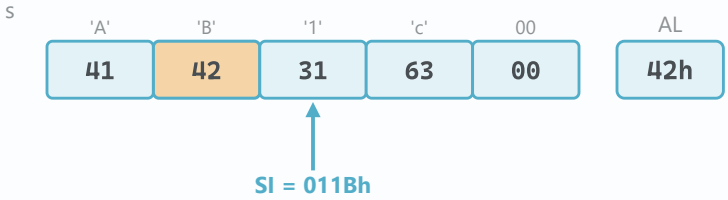
DEĞİŞKENLER

`SI = 011Ah` `AL = 41h` `ZF = 0`

Dizgi yerinde değiştirildi: her bayt okunur, harfse 20h çıkarılıp aynı adrese yazılır. Uzunluk yerine sonlandırıcı (0) kullanıldı.

Kod İzleme: Dizgiyi Büyük Harfe Çevirme

```
1  ORG 100h
2  LEA SI, s
3  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
13 sonraki:
   INC SI
   JMP don
16 bitti:
17 RET
18 s DB 'Ab1c', 0
```



'b' (62h): 'a' ≤ 62h ≤ 'z' → `SUB AL, 20h` = **42h** ('B'), `MOV [SI], AL` belleğe geri yazar.

Adım 3 / 6

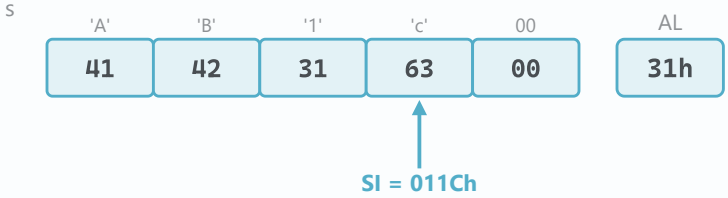
DEĞİŞKENLER

`SI = 011Bh` `AL = 42h` `ZF = 0`

Dizgi yerinde değiştirildi: her bayt okunur, harfse 20h çıkarılıp aynı adrese yazılır. Uzunluk yerine sonlandırıcı (0) kullanıldı.

Kod İzleme: Dizgiyi Büyük Harfe Çevirme

```
1  ORG 100h
2  LEA SI, s
3  don:
▶  MOV AL, [SI]
▶  CMP AL, 0          ; sonlandırıcı?
▶  JE bitti
▶  CMP AL, 'a'
▶  JB sonraki
9  CMP AL, 'z'
10 JA sonraki
11 SUB AL, 20h        ; 'a' - 'A'
12 MOV [SI], AL
13 sonraki:
▶  INC SI
▶  JMP don
16 bitti:
17 RET
18 s DB 'Ab1c', 0
```



'1' (31h): 'a'dan küçük, atlanır. Harf denetimi olmasaydı 31h − 20h = 11h ile rakam bozulurdu.

Adım 4 / 6

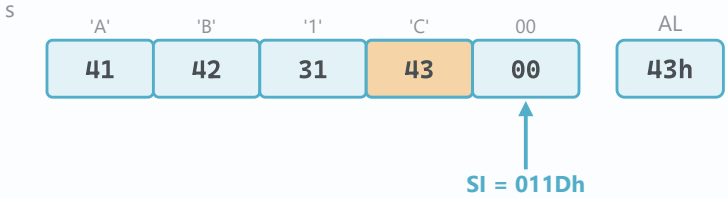
DEĞİŞKENLER

SI = 011Ch | AL = 31h | ZF = 0

Dizgi yerinde değiştirildi: her bayt okunur, harfse 20h çıkarılıp aynı adrese yazılır. Uzunluk yerine sonlandırıcı (0) kullanıldı.

Kod İzleme: Dizgiyi Büyük Harfe Çevirme

```
1  ORG 100h
2  LEA SI, s
3  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
13 sonraki:
   INC SI
   JMP don
16 bitti:
17 RET
18 s DB 'Ab1c', 0
```



'c' (63h) → **43h** ('C'). Dizgi artık 'AB1C'.

Adım 5 / 6

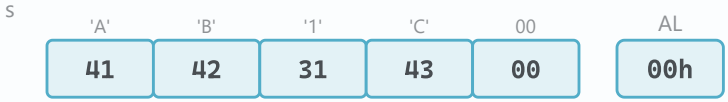
DEĞİŞKENLER

SI = 011Dh | AL = 43h | ZF = 0

Dizgi yerinde değiştirildi: her bayt okunur, harfse 20h çıkarılıp aynı adrese yazılır. Uzunluk yerine sonlandırıcı (0) kullanıldı.

Kod İzleme: Dizgiyi Büyük Harfe Çevirme

```
1  ORG 100h
2  LEA SI, s
3  don:
4  MOV AL, [SI]
5  CMP AL, 0          ; sonlandırıcı?
6  JE bitti
7  CMP AL, 'a'
8  JB sonraki
9  CMP AL, 'z'
10 JA sonraki
11 SUB AL, 20h        ; 'a' - 'A'
12 MOV [SI], AL
13 sonraki:
14 INC SI
15 JMP don
16 bitti:
17 RET
18 s DB 'Ab1c', 0
```



sonlandırıcı: ZF = 1

Okunan bayt 0: `CMP AL, 0` → ZF = 1, `JE bitti`. Döngü dizginin uzunluğunu bilmeden, sonlandırıcıda durdu.

Adım 6 / 6

DEĞİŞKENLER

`SI = 011Dh` `AL = 00h` `ZF = 1`

Dizgi yerinde değiştirildi: her bayt okunur, harfse 20h çıkarılıp aynı adrese yazılır. Uzunluk yerine sonlandırıcı (0) kullanıldı.

İki Boyutlu Diziler (Tablolar)



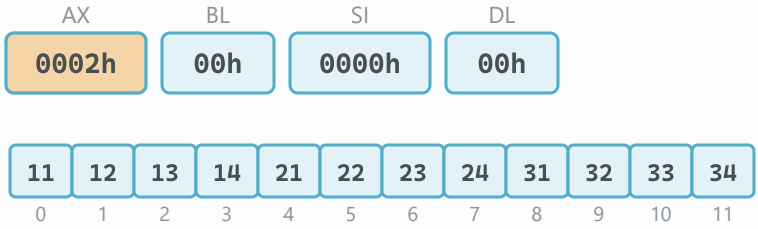

```
mat DB 3 DUP(4 DUP(0)) ; 3 × 4
tab DW 3 DUP(4 DUP(0)) ; 24 bayt
```

- Bellek tek boyutludur; tablo **satır öncelikli düzende** yerleşir: önce 0. satırın tüm sütunları, sonra 1. satır...
- Bağıl uzaklık = (satır × sütun sayısı + sütun) × eleman boyu.
- Sözcük tablosunda [1][2]: (1 × 4 + 2) × 2 = **12** bayt.
- İç içe DUP satır ve sütunu birlikte ayırır: 3 kez (4 kez 0).

Sütun sayısı 2'nin kuvvetiyse çarpma yerine kaydırma yeterlidir: satır × 4 için iki kez `SHL`. Satır sayısı dizinin yerleşimine girmez; yalnızca satır indisinin sınırını belirler.

Kod İzleme: mat[2][1] Elemanını Okuma

```
1  SUTSAY EQU 4
2  ORG 100h
▶  MOV AL, satir       ; AL = 2
4  MOV BL, SUTSAY      ; BL = 4
5  MUL BL              ; AX = 2 × 4
6  ADD AL, sutun       ; + 1
7  MOV SI, AX          ; SI = 9
8  MOV DL, mat[SI]     ; mat[2][1]
9  RET
10 mat DB 11h, 12h, 13h, 14h
11     DB 21h, 22h, 23h, 24h
12     DB 31h, 32h, 33h, 34h
13 satir DB 2
14 sutun DB 1
```



AL ← satir = **2**. Hedef eleman: mat[2][1]. Tablo 0112h adresinde, 3 satır × 4 sütun bayt.

Adım 1 / 7

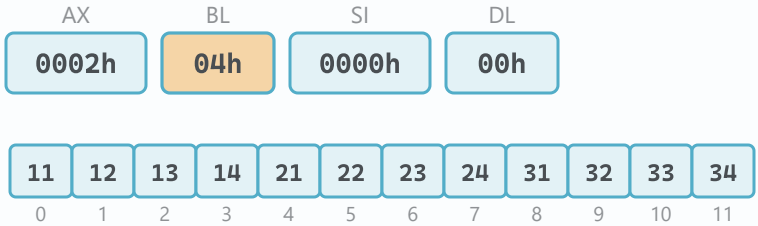
DEĞİŞKENLER

`AX = 0002h` `BL = 00h` `SI = 0000h` `DL = 00h`

Satır ve sütun çalışma anında değişkenlerden okundu; aynı kod her eleman için çalışır. Sözcük tablosunda sonuç ayrıca 2 ile çarpılırdı.

Kod İzleme: mat[2][1] Elemanını Okuma

```
1  SUTSAY EQU 4
2  ORG 100h
3  MOV AL, satir        ; AL = 2
4  MOV BL, SUTSAY       ; BL = 4
5  MUL BL               ; AX = 2 × 4
6  ADD AL, sutun        ; + 1
7  MOV SI, AX           ; SI = 9
8  MOV DL, mat[SI]      ; mat[2][1]
9  RET
10 mat DB 11h, 12h, 13h, 14h
11     DB 21h, 22h, 23h, 24h
12     DB 31h, 32h, 33h, 34h
13 satir DB 2
14 sutun DB 1
```



`SUTSAY EQU 4` komuta gömülür: BL = **04h** (sütun sayısı).

Adım 2 / 7

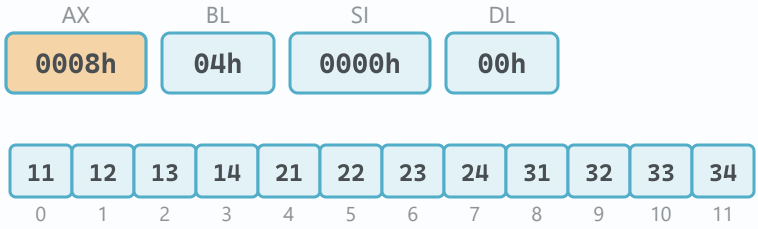
DEĞİŞKENLER

`AX = 0002h` `BL = 04h` `SI = 0000h` `DL = 00h`

Satır ve sütun çalışma anında değişkenlerden okundu; aynı kod her eleman için çalışır. Sözcük tablosunda sonuç ayrıca 2 ile çarpılırdı.

Kod İzleme: mat[2][1] Elemanını Okuma

```
1  SUTSAY EQU 4
2  ORG 100h
3  MOV AL, satir        ; AL = 2
4  MOV BL, SUTSAY       ; BL = 4
▶  MUL BL               ; AX = 2 × 4
6  ADD AL, sutun        ; + 1
7  MOV SI, AX           ; SI = 9
8  MOV DL, mat[SI]      ; mat[2][1]
9  RET
10 mat DB 11h, 12h, 13h, 14h
11     DB 21h, 22h, 23h, 24h
12     DB 31h, 32h, 33h, 34h
13 satir DB 2
14 sutun DB 1
```



`MUL BL` : AX = AL × BL = 2 × 4 = **0008h**. 2. satırın başı, tablonun 8. baytı.

Adım 3 / 7

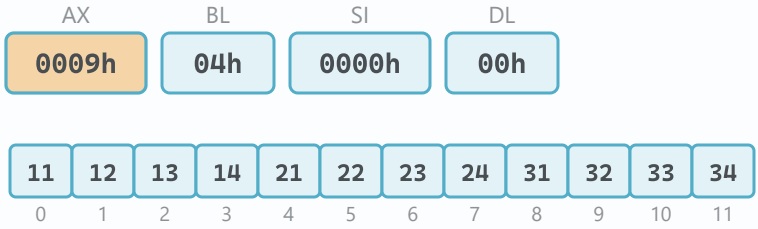
DEĞİŞKENLER

AX = 0008h | BL = 04h | SI = 0000h | DL = 00h

Satır ve sütun çalışma anında değişkenlerden okundu; aynı kod her eleman için çalışır. Sözcük tablosunda sonuç ayrıca 2 ile çarpılırdı.

Kod İzleme: mat[2][1] Elemanını Okuma

```
1  SUTSAY EQU 4
2  ORG 100h
3  MOV AL, satir        ; AL = 2
4  MOV BL, SUTSAY       ; BL = 4
5  MUL BL               ; AX = 2 × 4
▶  ADD AL, sutun        ; + 1
7  MOV SI, AX           ; SI = 9
8  MOV DL, mat[SI]      ; mat[2][1]
9  RET
10 mat DB 11h, 12h, 13h, 14h
11     DB 21h, 22h, 23h, 24h
12     DB 31h, 32h, 33h, 34h
13 satir DB 2
14 sutun DB 1
```



`ADD AL, sutun` : 8 + 1 = **9**. AH zaten 0 olduğu için AX = 0009h.

Adım 4 / 7

DEĞİŞKENLER

AX = **0009h** | BL = **04h** | SI = **0000h** | DL = **00h**

Satır ve sütun çalışma anında değişkenlerden okundu; aynı kod her eleman için çalışır. Sözcük tablosunda sonuç ayrıca 2 ile çarpılırdı.

Kod İzleme: mat[2][1] Elemanını Okuma

```
1  SUTSAY EQU 4
2  ORG 100h
3  MOV AL, satir       ; AL = 2
4  MOV BL, SUTSAY      ; BL = 4
5  MUL BL              ; AX = 2 × 4
6  ADD AL, sutun       ; + 1
▶  MOV SI, AX          ; SI = 9
8  MOV DL, mat[SI]     ; mat[2][1]
9  RET
10 mat DB 11h, 12h, 13h, 14h
11     DB 21h, 22h, 23h, 24h
12     DB 31h, 32h, 33h, 34h
13 satir DB 2
14 sutun DB 1
```



Uzaklık indis yazmacına alınır: SI = **0009h**. AL indis olarak kullanılamaz; adreslemede yalnız BX, SI, DI, BP.

Adım 5 / 7

DEĞİŞKENLER

AX = 0009h | BL = 04h | SI = 0009h | DL = 00h

Satır ve sütun çalışma anında değişkenlerden okundu; aynı kod her eleman için çalışır. Sözcük tablosunda sonuç ayrıca 2 ile çarpılırdı.

Kod İzleme: mat[2][1] Elemanını Okuma

```
1  SUTSAY EQU 4
2  ORG 100h
3  MOV AL, satir        ; AL = 2
4  MOV BL, SUTSAY       ; BL = 4
5  MUL BL               ; AX = 2 × 4
6  ADD AL, sutun        ; + 1
7  MOV SI, AX           ; SI = 9
▶  MOV DL, mat[SI]      ; mat[2][1]
9  RET
10 mat DB 11h, 12h, 13h, 14h
11     DB 21h, 22h, 23h, 24h
12     DB 31h, 32h, 33h, 34h
13 satir DB 2
14 sutun DB 1
```



`mat[SI]` = [0112h + 9]: DL = **32h**. 2. satır 1. sütun.

Adım 6 / 7

DEĞİŞKENLER

AX = **0009h** · BL = **04h** · SI = **0009h** · DL = **32h**

Satır ve sütun çalışma anında değişkenlerden okundu; aynı kod her eleman için çalışır. Sözcük tablosunda sonuç ayrıca 2 ile çarpılırdı.

Kod İzleme: mat[2][1] Elemanını Okuma

```
1  SUTSAY EQU 4
2  ORG 100h
3  MOV AL, satir       ; AL = 2
4  MOV BL, SUTSAY      ; BL = 4
5  MUL BL              ; AX = 2 × 4
6  ADD AL, sutun       ; + 1
7  MOV SI, AX          ; SI = 9
8  MOV DL, mat[SI]     ; mat[2][1]
▶  RET
10 mat DB 11h, 12h, 13h, 14h
11     DB 21h, 22h, 23h, 24h
12     DB 31h, 32h, 33h, 34h
13 satir DB 2
14 sutun DB 1
```



RET. Sabit adı SUTSAY seçildi: `SUTUN EQU 4` yazılsaydı assembler büyük-küçük harfe duyarsız olduğundan `sutun DB 1` ile aynı ad olur ve “iki kez tanımlı” hatası verirdi.

Adım 7 / 7

DEĞİŞKENLER

`AX = 0009h` `BL = 04h` `SI = 0009h` `DL = 32h`

Satır ve sütun çalışma anında değişkenlerden okundu; aynı kod her eleman için çalışır. Sözcük tablosunda sonuç ayrıca 2 ile çarpılırdı.

ADRESLER

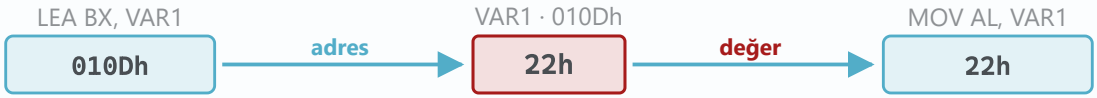
Değişkenin Adresini Alma

LEA (Load Effective Address)

Değişkenin **adresini** (bağıl konum) yazmaca yükler: `LEA BX, VAR1`. İndisli değişkenlerin adresini almak için de kullanılabilir: `LEA BX, a[SI]`.

OFFSET

Assembler işlecidir; değişkenin bağıl konum **değerini** derleme anında sağlar: `MOV BX, OFFSET VAR1`.

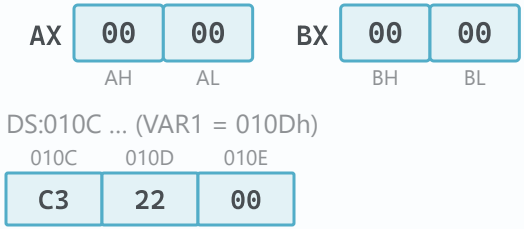


Değişkenin adresi, bir **prosedüre parametre geçirmek** için çok kullanışlıdır: prosedüre büyük bir dizinin kendisi değil, yalnızca 16 bitlik adresi verilir.

Komut	BX'e ne yüklenir?
`MOV BX, VAR1` (VAR1 DW)	VAR1'in **değeri** (içerik)
`LEA BX, VAR1` · `MOV BX, OFFSET VAR1`	VAR1'in **adresi** (bağıl konum)

Kod İzleme: LEA ile Değişken Güncelleme

```
1  ORG 100h
2  MOV AL, VAR1          ; değer
3  LEA BX, VAR1          ; adres
4  MOV BYTE PTR [BX], 44h
5  MOV AL, VAR1          ; yeni değer
6  RET
7  VAR1 DB 22h
8  END
```



Kod 0100h'den başlar; derleyici VAR1'i **010Dh**'ye yerleştirdi (12 bayt kod + RET).

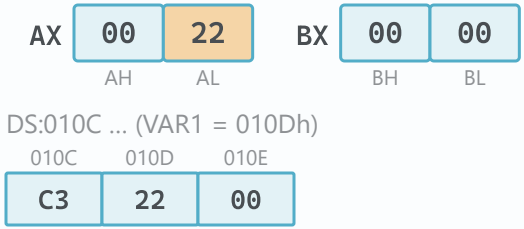
Adım 1 / 6

DEĞİŞKENLER

`AL = 00h` `BX = 0000h` `VAR1 = 22h`

Kod İzleme: LEA ile Değişken Güncelleme

```
1 ORG 100h
▶ MOV AL, VAR1          ; değer
3 LEA BX, VAR1          ; adres
4 MOV BYTE PTR [BX], 44h
5 MOV AL, VAR1          ; yeni değer
6 RET
7 VAR1 DB 22h
8 END
```



AL = VAR1'in **değeri** = 22h.

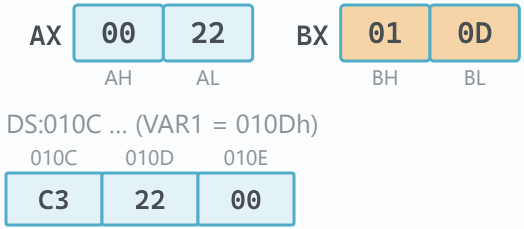
Adım 2 / 6

DEĞİŞKENLER

AL = **22h** | BX = **0000h** | VAR1 = **22h**

Kod İzleme: LEA ile Değişken Güncelleme

```
1 ORG 100h
2 MOV AL, VAR1          ; değer
▶ LEA BX, VAR1          ; adres
4 MOV BYTE PTR [BX], 44h
5 MOV AL, VAR1          ; yeni değer
6 RET
7 VAR1 DB 22h
8 END
```



BX = VAR1'in **adresi** = 010Dh. BX artık VAR1'i gösteren bir işaretçidir.

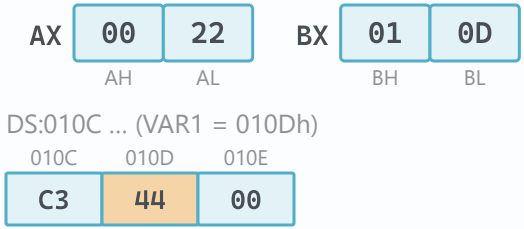
Adım 3 / 6

DEĞİŞKENLER

AL = **22h** | BX = **010Dh** | VAR1 = **22h**

Kod İzleme: LEA ile Değişken Güncelleme

```
1 ORG 100h
2 MOV AL, VAR1          ; değer
3 LEA BX, VAR1          ; adres
▶ MOV BYTE PTR [BX], 44h
5 MOV AL, VAR1          ; yeni değer
6 RET
7 VAR1 DB 22h
8 END
```



[BX] üzerinden VAR1 güncellenir: [010Dh] = **44h**.

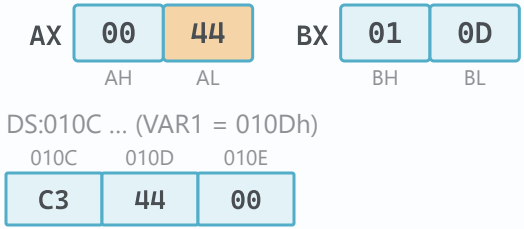
Adım 4 / 6

DEĞİŞKENLER

AL = **22h** | BX = **010Dh** | VAR1 = **44h**

Kod İzleme: LEA ile Değişken Güncelleme

```
1 ORG 100h
2 MOV AL, VAR1          ; değer
3 LEA BX, VAR1          ; adres
4 MOV BYTE PTR [BX], 44h
▶ MOV AL, VAR1          ; yeni değer
6 RET
7 VAR1 DB 22h
8 END
```



VAR1 tekrar okunur: AL = **44h**. Değişkene adı ile de işaretçi ile de erişilebilir.

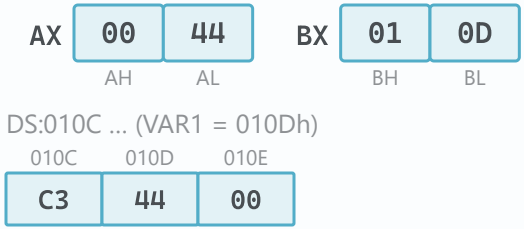
Adım 5 / 6

DEĞİŞKENLER

AL = **44h** | BX = **010Dh** | VAR1 = **44h**

Kod İzleme: LEA ile Değişken Güncelleme

```
1 ORG 100h
2 MOV AL, VAR1          ; değer
3 LEA BX, VAR1          ; adres
4 MOV BYTE PTR [BX], 44h
5 MOV AL, VAR1          ; yeni değer
▶ RET
7 VAR1 DB 22h
8 END
```



RET. `END` kaynak dosyanın sonunu bildiren direktiftir.

Adım 6 / 6

DEĞİŞKENLER

`AL = 44h` `BX = 010Dh` `VAR1 = 44h`

LEA ve OFFSET Aynı mı?

OFFSET ile aynı program

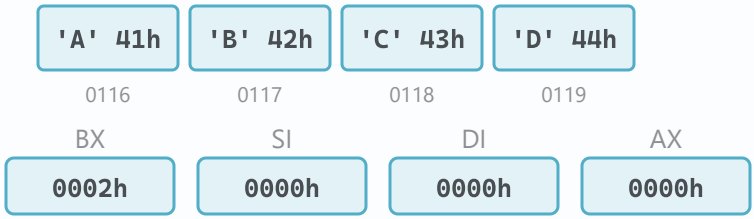
```
ORG 100h
MOV AL, VAR1
MOV BX, OFFSET VAR1
MOV BYTE PTR [BX], 44h
MOV AL, VAR1
RET
VAR1 DB 22h
END
```

- emu8086'da `LEA BX, VAR1` ve `MOV BX, OFFSET VAR1` **aynı makine koduna** derlenir: `BB 0D 01` (3 bayt, `MOV BX, 010Dh`). İki programda da VAR1 = 010Dh'dir.
- Gerçek 8086 assembler'ı MASM ise LEA'yı ayrı bir komut olarak kodlar: `8D 1E 0D 01` (4 bayt).
- Fark: `LEA BX, a[SI]` adresi **çalışma anında** hesaplar (a + SI); OFFSET yalnızca sabit adres verir.
- Köşeli parantez içinde yalnızca **BX, SI, DI, BP** kullanılabilir.

Kod İzleme: LEA a[BX] ile OFFSET a

```
1  ORG 100h
▶  MOV BX, 2
3  LEA SI, a[BX]       ; SI = a + BX
4  MOV DI, OFFSET a    ; sabit adres
5  ADD DI, BX          ; elle ekle
6  MOV AL, [SI]
7  INC BX
8  LEA SI, a[BX]       ; yeni adres
9  MOV AH, [SI]
10 RET
11 a DB 'ABCD'
```

a



BX = 2: indis. `a` dizisi 0116h'de başlıyor: 'A' 'B' 'C' 'D'.

Adım 1 / 8

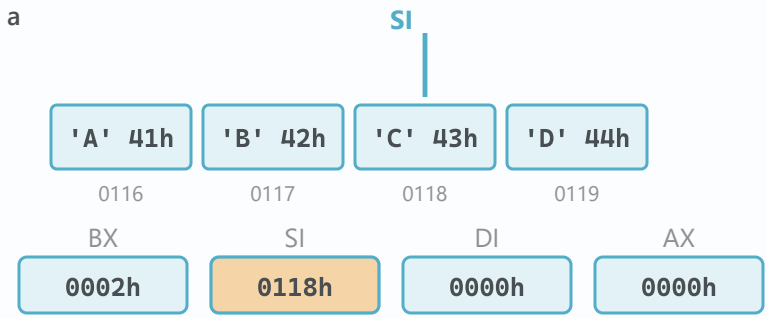
DEĞİŞKENLER

`BX = 0002h` `SI = 0000h` `DI = 0000h` `AX = 0000h`

LEA adresi yazmaçlarla çalışma anında hesaplar; OFFSET yalnızca derleme anında bilinen sabit adresi verir.

Kod İzleme: LEA a[BX] ile OFFSET a

```
1  ORG 100h
2  MOV BX, 2
▶  LEA SI, a[BX]       ; SI = a + BX
4  MOV DI, OFFSET a    ; sabit adres
5  ADD DI, BX          ; elle ekle
6  MOV AL, [SI]
7  INC BX
8  LEA SI, a[BX]       ; yeni adres
9  MOV AH, [SI]
10 RET
11 a DB 'ABCD'
```



`LEA SI, a[BX]` : adres **çalışma anında** hesaplanır: 0116h + 2 = **0118h**. Bellek okunmaz.

Adım 2 / 8

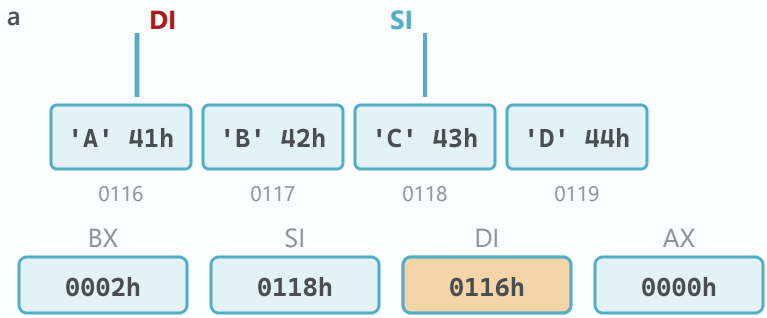
DEĞİŞKENLER

BX = 0002h | SI = 0118h | DI = 0000h | AX = 0000h

LEA adresi yazmaçlarla çalışma anında hesaplar; OFFSET yalnızca derleme anında bilinen sabit adresi verir.

Kod İzleme: LEA a[BX] ile OFFSET a

```
1  ORG 100h
2  MOV BX, 2
3  LEA SI, a[BX]      ; SI = a + BX
▶  MOV DI, OFFSET a   ; sabit adres
5  ADD DI, BX         ; elle ekle
6  MOV AL, [SI]
7  INC BX
8  LEA SI, a[BX]      ; yeni adres
9  MOV AH, [SI]
10 RET
11 a DB 'ABCD'
```



`OFFSET a` derleme anında bilinen sabittir: DI = **0116h**. BX'i katamaz.

Adım 3 / 8

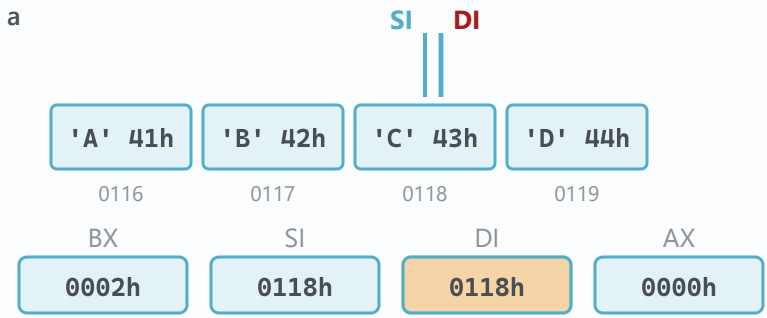
DEĞİŞKENLER

`BX = 0002h` `SI = 0118h` `DI = 0116h` `AX = 0000h`

LEA adresi yazmaçlarla çalışma anında hesaplar; OFFSET yalnızca derleme anında bilinen sabit adresi verir.

Kod İzleme: LEA a[BX] ile OFFSET a

```
1  ORG 100h
2  MOV BX, 2
3  LEA SI, a[BX]       ; SI = a + BX
4  MOV DI, OFFSET a    ; sabit adres
5  ADD DI, BX          ; elle ekle
6  MOV AL, [SI]
7  INC BX
8  LEA SI, a[BX]       ; yeni adres
9  MOV AH, [SI]
10 RET
11 a DB 'ABCD'
```



Aynı adres için OFFSET ile ayrıca `ADD` gerekir: DI = **0118h** = SI.

Adım 4 / 8

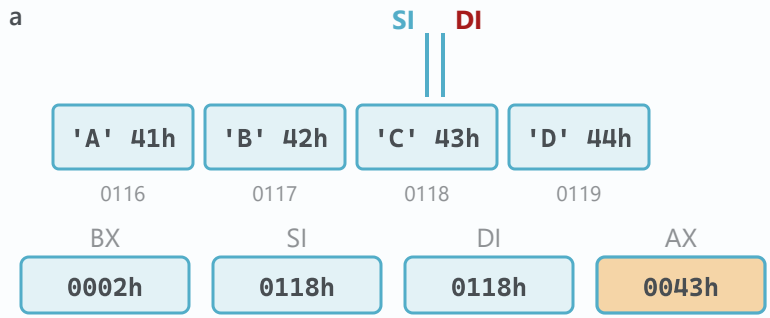
DEĞİŞKENLER

`BX = 0002h` `SI = 0118h` `DI = 0118h` `AX = 0000h`

LEA adresi yazmaçlarla çalışma anında hesaplar; OFFSET yalnızca derleme anında bilinen sabit adresi verir.

Kod İzleme: LEA a[BX] ile OFFSET a

```
1  ORG 100h
2  MOV BX, 2
3  LEA SI, a[BX]       ; SI = a + BX
4  MOV DI, OFFSET a    ; sabit adres
5  ADD DI, BX          ; elle ekle
6  MOV AL, [SI]
7  INC BX
8  LEA SI, a[BX]       ; yeni adres
9  MOV AH, [SI]
10 RET
11 a DB 'ABCD'
```



AL = [SI] = **43h** ('C').

Adım 5 / 8

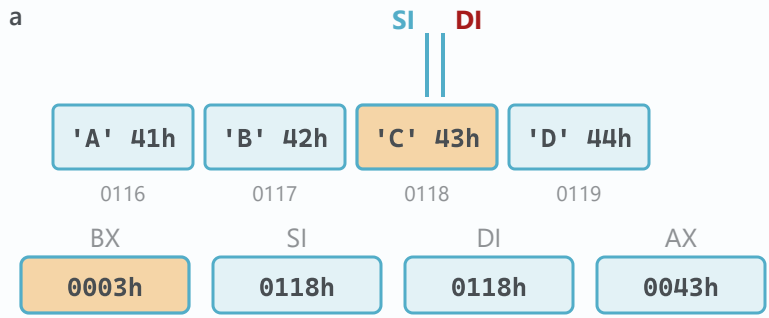
DEĞİŞKENLER

BX = 0002h | SI = 0118h | DI = 0118h | AX = 0043h

LEA adresi yazmaçlarla çalışma anında hesaplar; OFFSET yalnızca derleme anında bilinen sabit adresi verir.

Kod İzleme: LEA a[BX] ile OFFSET a

```
1  ORG 100h
2  MOV BX, 2
3  LEA SI, a[BX]       ; SI = a + BX
4  MOV DI, OFFSET a    ; sabit adres
5  ADD DI, BX          ; elle ekle
6  MOV AL, [SI]
▶  INC BX
8  LEA SI, a[BX]       ; yeni adres
9  MOV AH, [SI]
10 RET
11 a DB 'ABCD'
```



BX = 3.

Adım 6 / 8

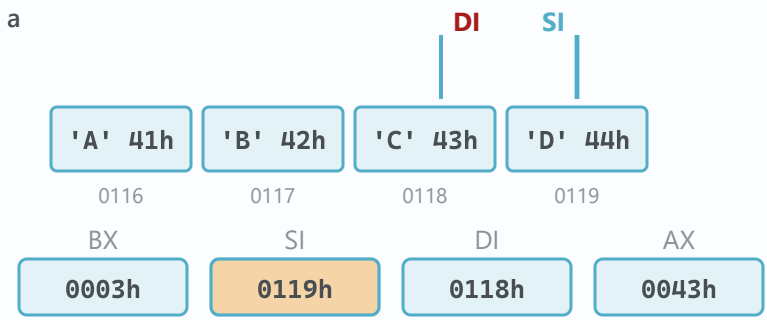
DEĞİŞKENLER

BX = 0003h | SI = 0118h | DI = 0118h | AX = 0043h

LEA adresi yazmaçlarla çalışma anında hesaplar; OFFSET yalnızca derleme anında bilinen sabit adresi verir.

Kod İzleme: LEA a[BX] ile OFFSET a

```
1  ORG 100h
2  MOV BX, 2
3  LEA SI, a[BX]      ; SI = a + BX
4  MOV DI, OFFSET a   ; sabit adres
5  ADD DI, BX         ; elle ekle
6  MOV AL, [SI]
7  INC BX
▶ LEA SI, a[BX]      ; yeni adres
9  MOV AH, [SI]
10 RET
11 a DB 'ABCD'
```



`LEA` güncel BX ile yeniden hesaplar: SI = **0119h**.

Adım 7 / 8

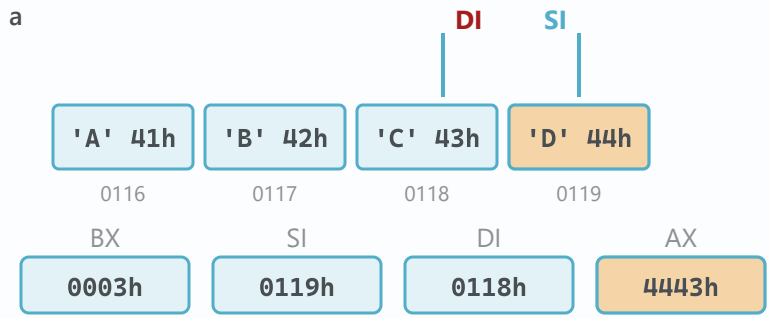
DEĞİŞKENLER

`BX = 0003h` `SI = 0119h` `DI = 0118h` `AX = 0043h`

LEA adresi yazmaçlarla çalışma anında hesaplar; OFFSET yalnızca derleme anında bilinen sabit adresi verir.

Kod İzleme: LEA a[BX] ile OFFSET a

```
1  ORG 100h
2  MOV BX, 2
3  LEA SI, a[BX]      ; SI = a + BX
4  MOV DI, OFFSET a   ; sabit adres
5  ADD DI, BX         ; elle ekle
6  MOV AL, [SI]
7  INC BX
8  LEA SI, a[BX]      ; yeni adres
▶  MOV AH, [SI]
10 RET
11 a DB 'ABCD'
```



AH = [SI] = **44h** ('D'). AX = 4443h.

Adım 8 / 8

DEĞİŞKENLER

BX = 0003h | SI = 0119h | DI = 0118h | AX = 4443h

LEA adresi yazmaçlarla çalışma anında hesaplar; OFFSET yalnızca derleme anında bilinen sabit adresi verir.

ADRESLER

İşaretçi Değişkenleri

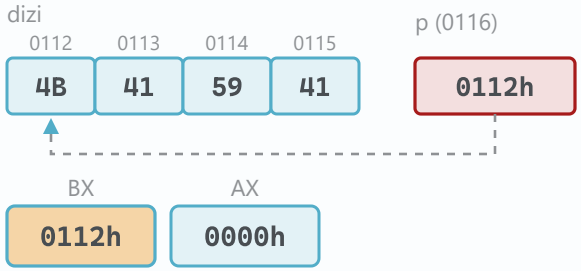
```
ORG 100h
MOV BX, p           ; 010Bh
LEA SI, p           ; 010Fh
MOV DI, OFFSET dizi
RET
dizi DB 'KAYA'      ; 010Bh
p    DW dizi        ; 010Fh, içi 010Bh
uzak DD dizi        ; bağıl konum + kesim
```

- DW'nin değeri bir değişken adı olursa o değişkenin **adresi** saklanır: p bir işaretçi değişkendir.
- İşaretçi bellekte durduğu için değiştirilebilir: `ADD p, 2` onu dizinin ilerisine kaydırır.
- Kullanmak için önce bir adres yazmacına alınır: `MOV BX, p` · `MOV AL, [BX]`.
- `DD` ile tanımlanan uzak işaretçi bağıl konum ve kesimi birlikte tutar; `LDS SI, uzak` ya da `LES DI, uzak` ile tek komutta yüklenir.

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

Kod İzleme: Bellekteki İşaretçi

```
1 ORG 100h
▶ MOV BX, p          ; p'nin içi
3 MOV AL, [BX]       ; dizi[0]
4 ADD p, 2           ; ilerlet
5 MOV BX, p
6 MOV AH, [BX]       ; dizi[2]
7 RET
8 dizi DB 'KAYA'
9 p    DW dizi       ; adres
```



`MOV BX, p` : p'nin **içeriği** okunur. p = `DW dizi` olduğundan BX = **0112h**, dizinin adresi.

Adım 1 / 6

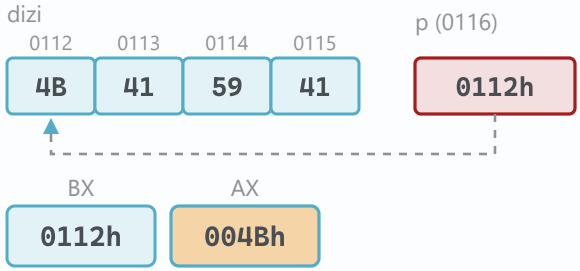
DEĞİŞKENLER

`BX = 0112h` `AX = 0000h` `p = 0112h`

İşaretçi değişkeni bir adresi bellekte saklar; yazmaçlar azken pek çok dizinin konumunu tutmanın yoludur. Her kullanımda önce BX, SI ya da DI'ye alınır.

Kod İzleme: Bellekteki İşaretçi

```
1 ORG 100h
2 MOV BX, p          ; p'nin içi
▶ MOV AL, [BX]       ; dizi[0]
4 ADD p, 2           ; ilerlet
5 MOV BX, p
6 MOV AH, [BX]       ; dizi[2]
7 RET
8 dizi DB 'KAYA'
9 p    DW dizi       ; adres
```



`MOV AL, [BX]` : işaretçinin gösterdiği bayt: AL = **4Bh** ('K').

Adım 2 / 6

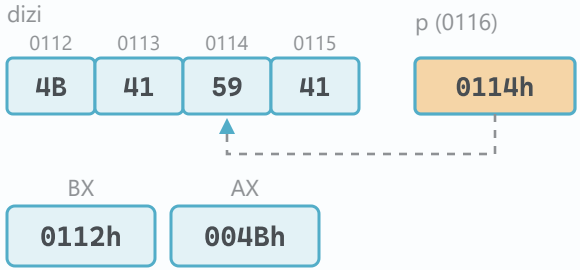
DEĞİŞKENLER

BX = **0112h** | AX = **004Bh** | p = **0112h**

İşaretçi değişkeni bir adresi bellekte saklar; yazmaçlar azken pek çok dizinin konumunu tutmanın yoludur. Her kullanımda önce BX, SI ya da DI'ye alınır.

Kod İzleme: Bellekteki İşaretçi

```
1 ORG 100h
2 MOV BX, p          ; p'nin içi
3 MOV AL, [BX]       ; dizi[0]
▶ ADD p, 2           ; ilerlet
5 MOV BX, p
6 MOV AH, [BX]       ; dizi[2]
7 RET
8 dizi DB 'KAYA'
9 p    DW dizi       ; adres
```



işaretçi kaydı

`ADD p, 2` : bellekteki işaretçi artık **0114h**'yi gösterir. BX değişmedi; eski adresi tutuyor.

Adım 3 / 6

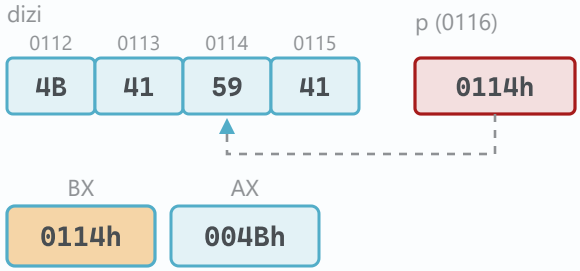
DEĞİŞKENLER

`BX = 0112h` `AX = 004Bh` `p = 0114h`

İşaretçi değişkeni bir adresi bellekte saklar; yazmaçlar azken pek çok dizinin konumunu tutmanın yoludur. Her kullanımda önce BX, SI ya da DI'ye alınır.

Kod İzleme: Bellekteki İşaretçi

```
1 ORG 100h
2 MOV BX, p          ; p'nin içi
3 MOV AL, [BX]       ; dizi[0]
4 ADD p, 2           ; ilerlet
▶ MOV BX, p
6 MOV AH, [BX]       ; dizi[2]
7 RET
8 dizi DB 'KAYA'
9 p    DW dizi       ; adres
```



İşaretçi yeniden yüklenir: BX = **0114h**.

Adım 4 / 6

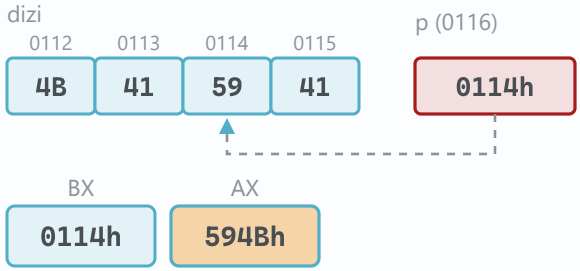
DEĞİŞKENLER

BX = 0114h | AX = 004Bh | p = 0114h

İşaretçi değişkeni bir adresi bellekte saklar; yazmaçlar azken pek çok dizinin konumunu tutmanın yoludur. Her kullanımda önce BX, SI ya da DI'ye alınır.

Kod İzleme: Bellekteki İşaretçi

```
1 ORG 100h
2 MOV BX, p          ; p'nin içi
3 MOV AL, [BX]       ; dizi[0]
4 ADD p, 2           ; ilerlet
5 MOV BX, p
▶ MOV AH, [BX]       ; dizi[2]
7 RET
8 dizi DB 'KAYA'
9 p    DW dizi       ; adres
```



`MOV AH, [BX]` : AH = **59h** ('Y'). AX = 594Bh.

Adım 5 / 6

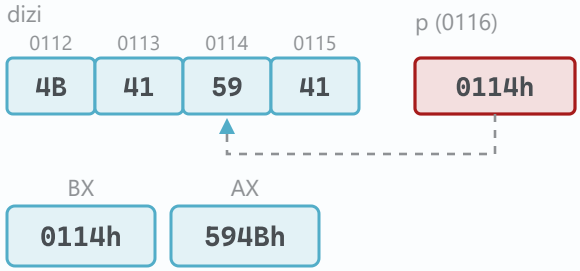
DEĞİŞKENLER

BX = **0114h** | AX = **594Bh** | p = **0114h**

İşaretçi değişkeni bir adresi bellekte saklar; yazmaçlar azken pek çok dizinin konumunu tutmanın yoludur. Her kullanımda önce BX, SI ya da DI'ye alınır.

Kod İzleme: Bellekteki İşaretçi

```
1 ORG 100h
2 MOV BX, p          ; p'nin içi
3 MOV AL, [BX]       ; dizi[0]
4 ADD p, 2           ; ilerlet
5 MOV BX, p
6 MOV AH, [BX]       ; dizi[2]
▶ RET
8 dizi DB 'KAYA'
9 p    DW dizi       ; adres
```



RET. Dizinin adresi kodda hiç yazılmadı; hangi elemanın okunacağına bellekteki işaretçi karar verdi.

Adım 6 / 6

DEĞİŞKENLER

BX = **0114h** AX = **594Bh** p = **0114h**

İşaretçi değişkeni bir adresi bellekte saklar; yazmaçlar azken pek çok dizinin konumunu tutmanın yoludur. Her kullanımda önce BX, SI ya da DI'ye alınır.

SABİTLER

Sabitler (EQU)

- Değişkenlere benzer, ancak yalnızca **derleme sırasında** var olurlar; bellekte yer kaplamazlar.
- Bir sabitin değeri tanımlandıktan sonra **değiştirilemez**.
- Programın farklı bölümlerinde aynı değerin kullanılmasını sağlar; değer tek yerden güncellenir.
- Tanım: `ad EQU <herhangi bir ifade>`
- Assembler her kullanımda adı değeriyle değiştirir: `MOV AX, k` → `MOV AX, 5` (`B8 05 00`).

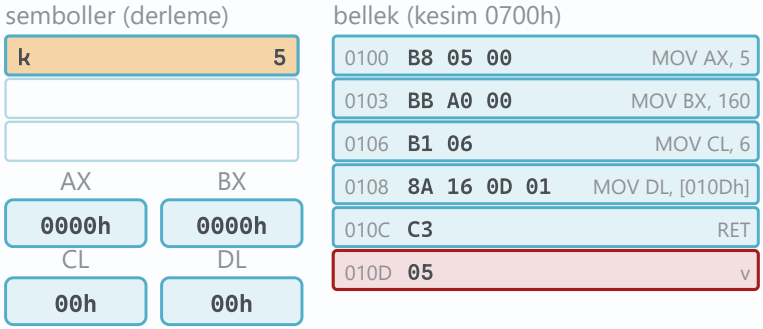
```
k       EQU 5
SATIR   EQU 80

MOV AX, k          ; AX = 5
MOV BX, SATIR*2    ; BX = 160
MOV CL, k+1        ; CL = 6
```

	`k EQU 5`	`k DB 5`
Bellek	kullanmaz	1 bayt
`MOV AL, k`	AL ← 5 (anlık)	AL ← [k] (bellekten)
Değiştirilebilir mi?	hayır	evet: `MOV k, 7`

Kod İzleme: EQU Komuta Gömülür

```
1  k      EQU 5
2  SATIR  EQU 80
3  ORG 100h
4  MOV AX, k          ; MOV AX, 5
5  MOV BX, SATIR*2    ; MOV BX, 160
6  MOV CL, k+1        ; MOV CL, 6
7  MOV DL, v          ; bellekten
8  RET
9  v DB 5             ; 1 bayt yer
```



`k EQU 5` : assembler sembol tablosuna **k = 5** yazar. Bellekte yer ayrılmaz, makine kodu üretilmez.

Adım 1 / 6

DEĞİŞKENLER

AX = 0000h | BX = 0000h | CL = 00h | DL = 00h

Makine kodları MASM/emu8086 kodlamasıdır: B8 iw (MOV AX, sabit), BB iw (MOV BX, sabit), B1 ib (MOV CL, sabit), 8A 16 adres (MOV DL, [adres]).

Kod İzleme: EQU Komuta Gömülür

```
1 k      EQU 5
2 SATIR EQU 80
3 ORG 100h
4 MOV AX, k          ; MOV AX, 5
5 MOV BX, SATIR*2    ; MOV BX, 160
6 MOV CL, k+1        ; MOV CL, 6
7 MOV DL, v          ; bellekten
8 RET
9 v DB 5             ; 1 bayt yer
```

semboller (derleme)

k	5
SATIR	80

AX	BX
0000h	0000h
CL	DL
00h	00h

bellek (kesim 0700h)

0100	B8 05 00	MOV AX, 5
0103	BB A0 00	MOV BX, 160
0106	B1 06	MOV CL, 6
0108	8A 16 0D 01	MOV DL, [010Dh]
010C	C3	RET
010D	05	v

`SATIR = 80` da yalnız derleme anında vardır. Program 0100h'den başlar; ilk bayt `MOV AX, k` 'nın kodudur.

Adım 2 / 6

DEĞİŞKENLER

AX = 0000h · BX = 0000h · CL = 00h · DL = 00h

Makine kodları MASM/emu8086 kodlamasıdır: B8 iw (MOV AX, sabit), BB iw (MOV BX, sabit), B1 ib (MOV CL, sabit), 8A 16 adres (MOV DL, [adres]).

Kod İzleme: EQU Komuta Gömülür

```
1 k      EQU 5
2 SATIR  EQU 80
3 ORG 100h
▶ MOV AX, k          ; MOV AX, 5
5 MOV BX, SATIR*2    ; MOV BX, 160
6 MOV CL, k+1        ; MOV CL, 6
7 MOV DL, v          ; bellekten
8 RET
9 v DB 5             ; 1 bayt yer
```

semboller (derleme)

k	5
SATIR	80
v	010Dh (adres)

AX	BX
0005h	0000h

CL	DL
00h	00h

bellek (kesim 0700h)

0100	B8 05 00	MOV AX, 5
0103	BB A0 00	MOV BX, 160
0106	B1 06	MOV CL, 6
0108	8A 16 0D 01	MOV DL, [010Dh]
010C	C3	RET
010D	05	v

`MOV AX, k` → `MOV AX, 5` (B8 05 00): 5 komutun içinde, anlık değer. AX = **0005h**.

Adım 3 / 6

DEĞİŞKENLER

AX = 0005h | BX = 0000h | CL = 00h | DL = 00h

Makine kodları MASM/emu8086 kodlamasıdır: B8 iw (MOV AX, sabit), BB iw (MOV BX, sabit), B1 ib (MOV CL, sabit), 8A 16 adres (MOV DL, [adres]).

Kod İzleme: EQU Komuta Gömülür

```
1 k      EQU 5
2 SATIR  EQU 80
3 ORG 100h
4 MOV AX, k          ; MOV AX, 5
5 MOV BX, SATIR*2    ; MOV BX, 160
6 MOV CL, k+1        ; MOV CL, 6
7 MOV DL, v          ; bellekten
8 RET
9 v DB 5             ; 1 bayt yer
```

semboller (derleme)

k	5
SATIR	80
v	010Dh (adres)

AX	BX
0005h	00A0h

CL	DL
00h	00h

bellek (kesim 0700h)

0100	B8 05 00	MOV AX, 5
0103	BB A0 00	MOV BX, 160
0106	B1 06	MOV CL, 6
0108	8A 16 0D 01	MOV DL, [010Dh]
010C	C3	RET
010D	05	v

`SATIR*2` derleme anında hesaplandı: 160 = 00A0h → **BX = 00A0h**.

Adım 4 / 6

DEĞİŞKENLER

AX = 0005h | BX = 00A0h | CL = 00h | DL = 00h

Makine kodları MASM/emu8086 kodlamasıdır: B8 iw (MOV AX, sabit), BB iw (MOV BX, sabit), B1 ib (MOV CL, sabit), 8A 16 adres (MOV DL, [adres]).

Kod İzleme: EQU Komuta Gömülür

```
1 k      EQU 5
2 SATIR  EQU 80
3 ORG 100h
4 MOV AX, k          ; MOV AX, 5
5 MOV BX, SATIR*2    ; MOV BX, 160
▶ MOV CL, k+1        ; MOV CL, 6
7 MOV DL, v          ; bellekten
8 RET
9 v DB 5             ; 1 bayt yer
```

semboller (derleme)

k	5
SATIR	80
v	010Dh (adres)

AX	BX
0005h	00A0h

CL	DL
06h	00h

bellek (kesim 0700h)

0100	B8 05 00	MOV AX, 5
0103	BB A0 00	MOV BX, 160
0106	B1 06	MOV CL, 6
0108	8A 16 0D 01	MOV DL, [010Dh]
010C	C3	RET
010D	05	v

`k+1` = 6 → CL = **06h**. Çalışma anında toplama yapılmadı.

Adım 5 / 6

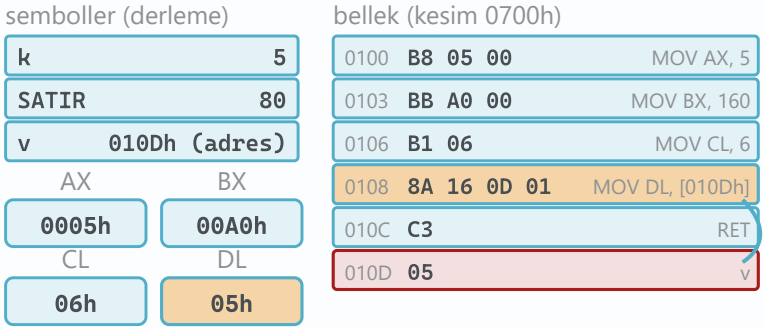
DEĞİŞKENLER

AX = 0005h | BX = 00A0h | CL = 06h | DL = 00h

Makine kodları MASM/emu8086 kodlamasıdır: B8 iw (MOV AX, sabit), BB iw (MOV BX, sabit), B1 ib (MOV CL, sabit), 8A 16 adres (MOV DL, [adres]).

Kod İzleme: EQU Komuta Gömülür

```
1 k      EQU 5
2 SATIR  EQU 80
3 ORG 100h
4 MOV AX, k          ; MOV AX, 5
5 MOV BX, SATIR*2    ; MOV BX, 160
6 MOV CL, k+1        ; MOV CL, 6
▶ MOV DL, v          ; bellekten
8 RET
9 v DB 5             ; 1 bayt yer
```



`v` bir değişken: komut adresini (010Dh) taşır, değer **bellekten** okunur: DL = 05h. v 1 bayt yer kaplar, k hiç kaplamaz.

Adım 6 / 6

DEĞİŞKENLER

AX = 0005h | BX = 00A0h | CL = 06h | DL = 05h

Makine kodları MASM/emu8086 kodlamasıdır: B8 iw (MOV AX, sabit), BB iw (MOV BX, sabit), B1 ib (MOV CL, sabit), 8A 16 adres (MOV DL, [adres]).

SABİTLER

Konum Sayacı ($) ve Boyut İşleçleri

- Assembler bir sonraki baytın yerleşeceği bağıl konumu bir sayaçta tutar: **konum sayacı**. Kaynakta `$` ile okunur.
- Bir dizinin hemen arkasında `$ - dizi`, dizinin bayt sayısıdır. Dizi büyüyüp küçülse de sabit kendiliğinden doğru kalır.
- EQU satırı dizinin **hemen ardından** gelmelidir; araya başka bir tanım girerse onun baytları da sayılır.
- MASM'daki `TYPE`, `LENGTHOF` ve `SIZEOF` işleçleri aynı bilgiyi değişkenin tanımından okur.

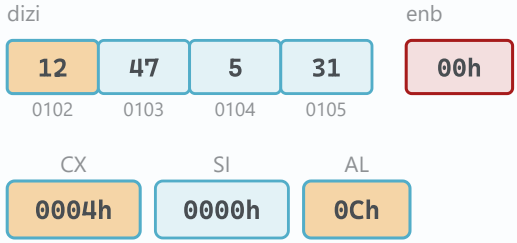
```
s DB 'Merhaba', 0
n EQU $ - s        ; 8
w DW 10 DUP(0)
```

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

Hepsi derleme anı sabitidir: `MOV CX, LENGTHOF w` komuta 10 gömer, bellekte yer kaplamaz. Sözcük dizisinde döngü sayacı LENGTHOF, bayt uzunluğu SIZEOF ile alınır.

Kod İzleme: $ ile Boyut, Döngüyle En Büyük Eleman

```
1  ORG 100h
2  JMP basla
3  dizi DB 12, 47, 5, 31
4  n EQU $ - dizi      ; 4
5  enb DB ?
6  basla:
7  MOV CX, n           ; CX = 4
8  MOV SI, 0
9  MOV AL, dizi[SI]    ; ilk aday
10 bak:
11 CMP AL, dizi[SI]
12 JAE gec             ; AL >= eleman
13 MOV AL, dizi[SI]    ; yeni aday
14 gec:
15 INC SI
16 LOOP bak
17 MOV enb, AL
18 RET
```



AL = 12

`JMP basla` veriyi atlar. `n EQU $ - dizi` = 4 derleme anında hesaplandı: CX = **0004h**. İlk eleman aday: AL = 12.

Adım 1 / 6

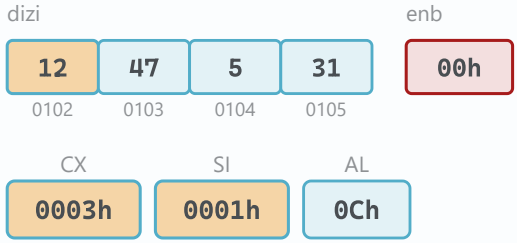
DEĞİŞKENLER

`CX = 0004h` `SI = 0000h` `AL = 0Ch` `CF = 0`

Döngü sayısı elle yazılmadı; diziye eleman eklenirse n kendiliğinden değişir. JAE işaretsiz karşılaştırır; işaretli dizilerde JGE kullanılır.

Kod İzleme: $ ile Boyut, Döngüyle En Büyük Eleman

```
1  ORG 100h
2  JMP basla
3  dizi DB 12, 47, 5, 31
4  n EQU $ - dizi      ; 4
5  enb DB ?
6  basla:
7  MOV CX, n           ; CX = 4
8  MOV SI, 0
9  MOV AL, dizi[SI]    ; ilk aday
10 bak:
▶ CMP AL, dizi[SI]
▶ JAE gec              ; AL >= eleman
13 MOV AL, dizi[SI]    ; yeni aday
14 gec:
▶ INC SI
▶ LOOP bak
17 MOV enb, AL
18 RET
```



1. tur: 12 ile 12 karşılaştırılır; `JAE` atlar (eşit). SI = 1, LOOP: CX = 3.

Adım 2 / 6

DEĞİŞKENLER

`CX = 0003h` `SI = 0001h` `AL = 0Ch` `CF = 0`

Döngü sayısı elle yazılmadı; diziye eleman eklenirse n kendiliğinden değişir. JAE işaretsiz karşılaştırır; işaretli dizilerde JGE kullanılır.

Kod İzleme: $ ile Boyut, Döngüyle En Büyük Eleman

```
1  ORG 100h
2  JMP basla
3  dizi DB 12, 47, 5, 31
4  n EQU $ - dizi      ; 4
5  enb DB ?
6  basla:
7  MOV CX, n           ; CX = 4
8  MOV SI, 0
9  MOV AL, dizi[SI]    ; ilk aday
10 bak:
▶  CMP AL, dizi[SI]
▶  JAE gec             ; AL >= eleman
▶  MOV AL, dizi[SI]    ; yeni aday
14 gec:
▶  INC SI
▶  LOOP bak
17 MOV enb, AL
18 RET
```



2. tur: 12 < 47 → CF = 1, `JAE` atlamaz; AL = **47** (2Fh). SI = 2, CX = 2.

Adım 3 / 6

DEĞİŞKENLER

`CX = 0002h` `SI = 0002h` `AL = 2Fh` `CF = 1`

Döngü sayısı elle yazılmadı; diziye eleman eklenirse n kendiliğinden değişir. JAE işaretsiz karşılaştırır; işaretli dizilerde JGE kullanılır.

Kod İzleme: $ ile Boyut, Döngüyle En Büyük Eleman

```
1  ORG 100h
2  JMP basla
3  dizi DB 12, 47, 5, 31
4  n EQU $ - dizi      ; 4
5  enb DB ?
6  basla:
7  MOV CX, n           ; CX = 4
8  MOV SI, 0
9  MOV AL, dizi[SI]    ; ilk aday
10 bak:
▶  CMP AL, dizi[SI]
▶  JAE gec             ; AL >= eleman
13 MOV AL, dizi[SI]    ; yeni aday
14 gec:
▶  INC SI
▶  LOOP bak
17 MOV enb, AL
18 RET
```



AL = 47

3. tur: 47 ≥ 5 → atlanır, AL değişmez. SI = 3, CX = 1.

Adım 4 / 6

DEĞİŞKENLER

`CX = 0001h` `SI = 0003h` `AL = 2Fh` `CF = 0`

Döngü sayısı elle yazılmadı; diziye eleman eklenirse n kendiliğinden değişir. JAE işaretsiz karşılaştırır; işaretli dizilerde JGE kullanılır.

Kod İzleme: $ ile Boyut, Döngüyle En Büyük Eleman

```
1  ORG 100h
2  JMP basla
3  dizi DB 12, 47, 5, 31
4  n EQU $ - dizi      ; 4
5  enb DB ?
6  basla:
7  MOV CX, n           ; CX = 4
8  MOV SI, 0
9  MOV AL, dizi[SI]    ; ilk aday
10 bak:
▶  CMP AL, dizi[SI]
▶  JAE gec             ; AL >= eleman
13 MOV AL, dizi[SI]    ; yeni aday
14 gec:
▶  INC SI
▶  LOOP bak
17 MOV enb, AL
18 RET
```



4. tur: 47 ≥ 31 → atlanır. LOOP: CX = 0, döngü biter.

Adım 5 / 6

DEĞİŞKENLER

CX = 0000h | SI = 0004h | AL = 2Fh | CF = 0

Döngü sayısı elle yazılmadı; diziye eleman eklenirse n kendiliğinden değişir. JAE işaretsiz karşılaştırır; işaretli dizilerde JGE kullanılır.

Kod İzleme: $ ile Boyut, Döngüyle En Büyük Eleman

```
1  ORG 100h
2  JMP basla
3  dizi DB 12, 47, 5, 31
4  n EQU $ - dizi      ; 4
5  enb DB ?
6  basla:
7  MOV CX, n           ; CX = 4
8  MOV SI, 0
9  MOV AL, dizi[SI]    ; ilk aday
10 bak:
11 CMP AL, dizi[SI]
12 JAE gec             ; AL >= eleman
13 MOV AL, dizi[SI]    ; yeni aday
14 gec:
15 INC SI
16 LOOP bak
▶  MOV enb, AL
▶  RET
```



enb = 47

AL = 47

`MOV enb, AL` : en büyük eleman **2Fh** = 47 belleğe yazıldı; RET.

Adım 6 / 6

DEĞİŞKENLER

`CX = 0000h` `SI = 0004h` `AL = 2Fh` `CF = 0`

Döngü sayısı elle yazılmadı; diziye eleman eklenirse n kendiliğinden değişir. JAE işaretsiz karşılaştırır; işaretli dizilerde JGE kullanılır.

Kod İzleme: Değişkenler Arasında Veri Taşıma

```
▶ jmp start
2 value1 DW 10     ; 2 baytlık değişken
3 value2 DW 0      ; ikinci değişken
4 start:
5 MOV AX, value1   ; AX ← value1
6 MOV value2, AX   ; value2 ← AX
7 end start
```



`jmp start` CPU'yu verilerin **üzerinden atlatır**; aksi hâlde veri baytları komut gibi yürütülürdü.

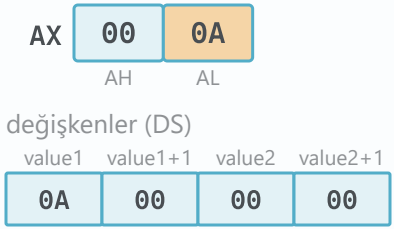
Adım 1 / 4

DEĞİŞKENLER

`AX = 0000h` `value1 = 000Ah` `value2 = 0000h`

Kod İzleme: Değişkenler Arasında Veri Taşıma

```
1 jmp start
2 value1 DW 10     ; 2 baytlık değişken
3 value2 DW 0      ; ikinci değişken
4 start:
▶ MOV AX, value1   ; AX ← value1
6 MOV value2, AX   ; value2 ← AX
7 end start
```



value1 → AX = **000Ah**. Bellekten belleğe doğrudan MOV olmadığı için ara yazmaç gerekir.

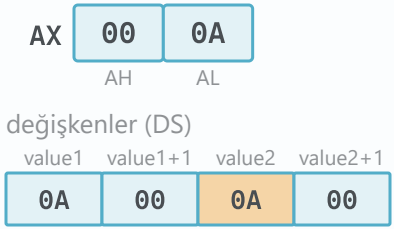
Adım 2 / 4

DEĞİŞKENLER

`AX = 000Ah` `value1 = 000Ah` `value2 = 0000h`

Kod İzleme: Değişkenler Arasında Veri Taşıma

```
1 jmp start
2 value1 DW 10     ; 2 baytlık değişken
3 value2 DW 0      ; ikinci değişken
4 start:
5 MOV AX, value1   ; AX ← value1
▶ MOV value2, AX   ; value2 ← AX
7 end start
```



AX → value2: bellekte **0A 00**. Artık value1 = value2 = 10.

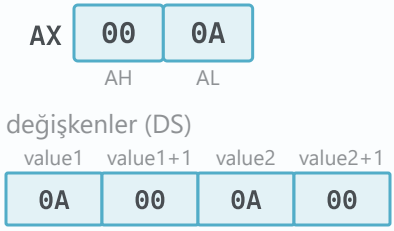
Adım 3 / 4

DEĞİŞKENLER

`AX = 000Ah` `value1 = 000Ah` `value2 = 000Ah`

Kod İzleme: Değişkenler Arasında Veri Taşıma

```
1 jmp start
2 value1 DW 10     ; 2 baytlık değişken
3 value2 DW 0      ; ikinci değişken
4 start:
5 MOV AX, value1   ; AX ← value1
6 MOV value2, AX   ; value2 ← AX
  end start
```



`end start` : kaynağın sonu; **start** programın giriş noktasıdır.

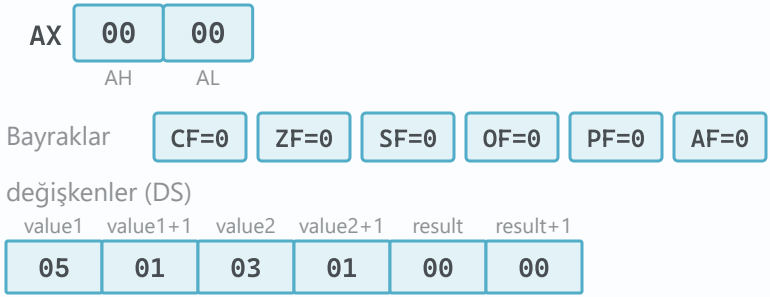
Adım 4 / 4

DEĞİŞKENLER

`AX = 000Ah` `value1 = 000Ah` `value2 = 000Ah`

Kod İzleme: Değişkenleri Toplama

```
▶ jmp start
2 value1 DW 105h  ; ilk değer 105h
3 value2 DW 103h  ; ilk değer 103h
4 result DW ?     ; belirsiz
5 start:
6 MOV AX, value1  ; AX ← value1
7 ADD AX, value2  ; AX ← AX + value2
8 MOV result, AX  ; result ← AX
9 end start
```



Başlangıç: result `?` ile tanımlı; emu8086 belleğe 00 00 koyar, ama değerine güvenilmez.

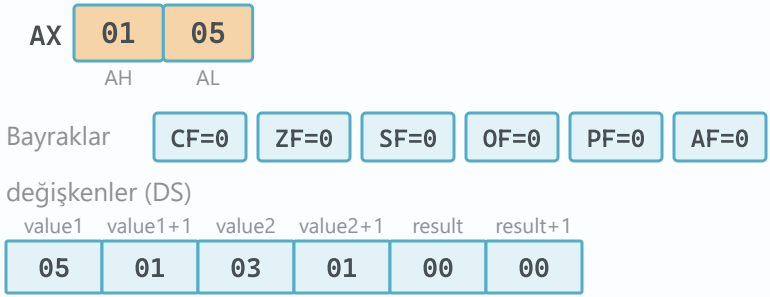
Adım 1 / 5

DEĞİŞKENLER

`AX = 0000h` `CF = 0` `ZF = 0` `SF = 0` `OF = 0` `PF = 0` `AF = 0` `value1 = 0105h` `value2 = 0103h` `result = 0000h`

Kod İzleme: Değişkenleri Toplama

```
1  jmp start
2  value1 DW 105h   ; ilk değer 105h
3  value2 DW 103h   ; ilk değer 103h
4  result DW ?      ; belirsiz
5  start:
▶  MOV AX, value1   ; AX ← value1
7  ADD AX, value2   ; AX ← AX + value2
8  MOV result, AX   ; result ← AX
9  end start
```



AX = value1 = **0105h**.

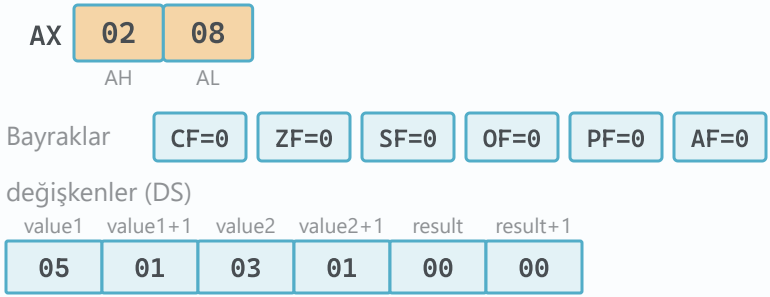
Adım 2 / 5

DEĞİŞKENLER

`AX = 0105h` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`
`PF = 0` `AF = 0` `value1 = 0105h` `value2 = 0103h`
`result = 0000h`

Kod İzleme: Değişkenleri Toplama

```
1 jmp start
2 value1 DW 105h  ; ilk değer 105h
3 value2 DW 103h  ; ilk değer 103h
4 result DW ?     ; belirsiz
5 start:
6 MOV AX, value1  ; AX ← value1
▶ ADD AX, value2  ; AX ← AX + value2
8 MOV result, AX  ; result ← AX
9 end start
```



ADD bellek işlenenini doğrudan kullanabilir: 0105h + 0103h = **0208h**. Bayraklar: CF = 0, ZF = 0, PF = 0 (08h'de bir adet 1).

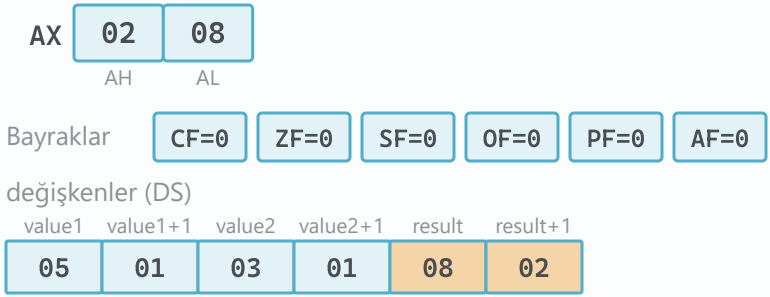
Adım 3 / 5

DEĞİŞKENLER

`AX = 0208h` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`
`PF = 0` `AF = 0` `value1 = 0105h` `value2 = 0103h`
`result = 0000h`

Kod İzleme: Değişkenleri Toplama

```
1 jmp start
2 value1 DW 105h  ; ilk değer 105h
3 value2 DW 103h  ; ilk değer 103h
4 result DW ?     ; belirsiz
5 start:
6 MOV AX, value1  ; AX ← value1
7 ADD AX, value2  ; AX ← AX + value2
▶ MOV result, AX  ; result ← AX
9 end start
```



result ← AX: bellekte **08 02** (little-endian).

Adım 4 / 5

DEĞİŞKENLER

`AX = 0208h` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`

`PF = 0` `AF = 0` `value1 = 0105h` `value2 = 0103h`

`result = 0208h`

Kod İzleme: Değişkenleri Toplama

```
1 jmp start
2 value1 DW 105h  ; ilk değer 105h
3 value2 DW 103h  ; ilk değer 103h
4 result DW ?     ; belirsiz
5 start:
6 MOV AX, value1  ; AX ← value1
7 ADD AX, value2  ; AX ← AX + value2
8 MOV result, AX  ; result ← AX
▶ end start
```

AX: AH = 02, AL = 08

Bayraklar: CF=0 ZF=0 SF=0 OF=0 PF=0 AF=0

değişkenler (DS)

value1	value1+1	value2	value2+1	result	result+1
05	01	03	01	08	02

Sonuç: result = 0208h = 520 (261 + 259).

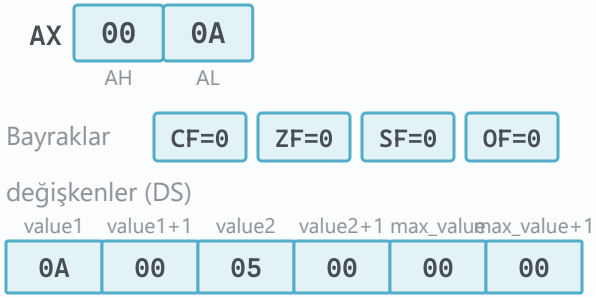
Adım 5 / 5

DEĞİŞKENLER

AX = 0208h | CF = 0 | ZF = 0 | SF = 0 | OF = 0 | PF = 0 | AF = 0 | value1 = 0105h | value2 = 0103h | result = 0208h

Kod İzleme: Büyük Olanı Bul (10, 5)

```
1  ORG 100h
▶ MOV AX, value1      ; AX ← value1
3  CMP AX, value2     ; AX – value2
4  JGE greater_or_equal
5  MOV AX, value2     ; küçükse value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX  ; sonucu sakla
11 RET
12 value1 DW 10
13 value2 DW 5
14 max_value DW ?
```



AX = value1 = **000Ah**.

Adım 1 / 6

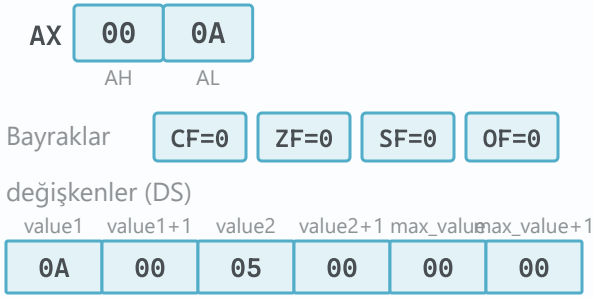
DEĞİŞKENLER

`AX = 000Ah` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`

`value1 = 000Ah` `value2 = 0005h` `max_value = 0000h`

Kod İzleme: Büyük Olanı Bul (10, 5)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX – value2
4  JGE greater_or_equal
5  MOV AX, value2      ; küçükse value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; sonucu sakla
11 RET
12 value1 DW 10
13 value2 DW 5
14 max_value DW ?
```



CMP çıkarma yapar ama sonucu **saklamaz**, yalnız bayrakları ayarlar: 10 − 5.

Adım 2 / 6

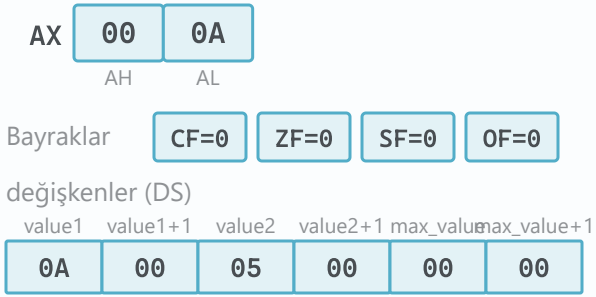
DEĞİŞKENLER

`AX = 000Ah` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`

`value1 = 000Ah` `value2 = 0005h` `max_value = 0000h`

Kod İzleme: Büyük Olanı Bul (10, 5)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX − value2
▶ 4  JGE greater_or_equal
5  MOV AX, value2      ; küçükse value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; sonucu sakla
11 RET
12 value1 DW 10
13 value2 DW 5
14 max_value DW ?
```



JGE (işaretli ≥): SF = OF olduğundan **atlanır** → greater_or_equal.

Adım 3 / 6

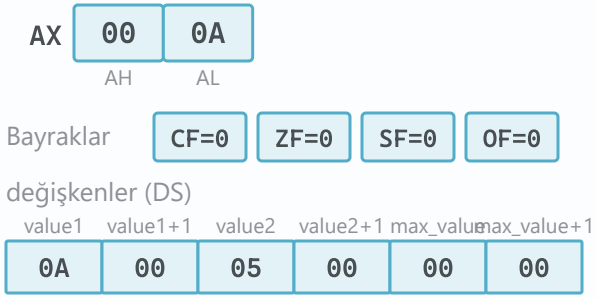
DEĞİŞKENLER

`AX = 000Ah` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`

`value1 = 000Ah` `value2 = 0005h` `max_value = 0000h`

Kod İzleme: Büyük Olanı Bul (10, 5)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX – value2
4  JGE greater_or_equal
5  MOV AX, value2      ; küçükse value2
6  JMP store_max
7  greater_or_equal:
▶  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; sonucu sakla
11 RET
12 value1 DW 10
13 value2 DW 5
14 max_value DW ?
```



AX = value1 (zaten öyleydi; bu satır gereksiz ama zararsız).

Adım 4 / 6

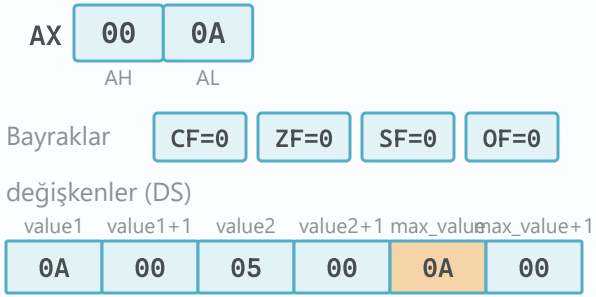
DEĞİŞKENLER

`AX = 000Ah` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`

`value1 = 000Ah` `value2 = 0005h` `max_value = 0000h`

Kod İzleme: Büyük Olanı Bul (10, 5)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX – value2
4  JGE greater_or_equal
5  MOV AX, value2      ; küçükse value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
▶  MOV max_value, AX   ; sonucu sakla
11 RET
12 value1 DW 10
13 value2 DW 5
14 max_value DW ?
```



max_value ← AX = **000Ah**.

Adım 5 / 6

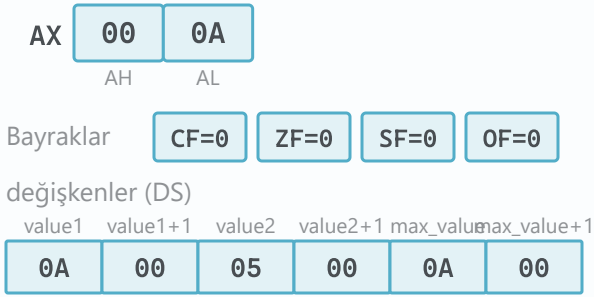
DEĞİŞKENLER

AX = **000Ah** | CF = **0** | ZF = **0** | SF = **0** | OF = **0**

value1 = **000Ah** | value2 = **0005h** | max_value = **000Ah**

Kod İzleme: Büyük Olanı Bul (10, 5)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX – value2
4  JGE greater_or_equal
5  MOV AX, value2      ; küçükse value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; sonucu sakla
▶  RET
12 value1 DW 10
13 value2 DW 5
14 max_value DW ?
```



RET: DOS'a dönüş. RET olmasaydı CPU value1'in baytlarını komut sanıp yürütürdü.

Adım 6 / 6

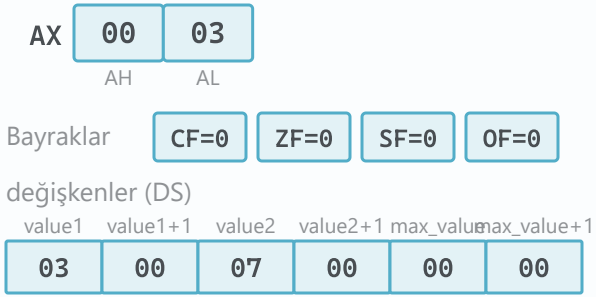
DEĞİŞKENLER

`AX = 000Ah` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`

`value1 = 000Ah` `value2 = 0005h` `max_value = 000Ah`

Kod İzleme: Büyük Olanı Bul (3, 7)

```
1  ORG 100h
▶  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX – value2
4  JGE greater_or_equal
5  MOV AX, value2      ; küçükse value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; sonucu sakla
11 RET
12 value1 DW 3
13 value2 DW 7
14 max_value DW ?
```



AX = value1 = **0003h**.

Adım 1 / 7

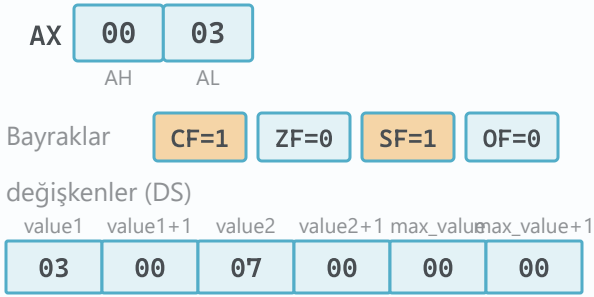
DEĞİŞKENLER

`AX = 0003h` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`

`value1 = 0003h` `value2 = 0007h` `max_value = 0000h`

Kod İzleme: Büyük Olanı Bul (3, 7)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX – value2
4  JGE greater_or_equal
5  MOV AX, value2      ; küçükse value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; sonucu sakla
11 RET
12 value1 DW 3
13 value2 DW 7
14 max_value DW ?
```



CMP çıkarma yapar ama sonucu **saklamaz**, yalnız bayrakları ayarlar: 3 − 7.

Adım 2 / 7

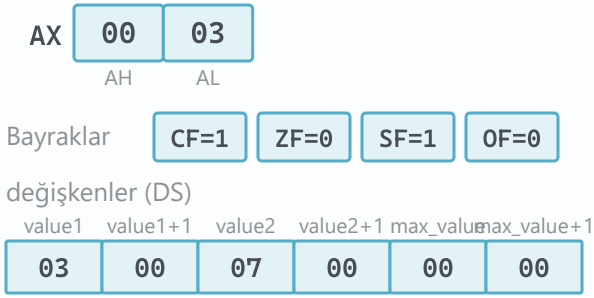
DEĞİŞKENLER

AX = 0003h | CF = 1 | ZF = 0 | SF = 1 | OF = 0

value1 = 0003h | value2 = 0007h | max_value = 0000h

Kod İzleme: Büyük Olanı Bul (3, 7)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX – value2
▶  JGE greater_or_equal
5  MOV AX, value2      ; küçükse value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; sonucu sakla
11 RET
12 value1 DW 3
13 value2 DW 7
14 max_value DW ?
```



SF ≠ OF → koşul sağlanmaz, **atlanmaz**; sıradaki komuta geçilir.

Adım 3 / 7

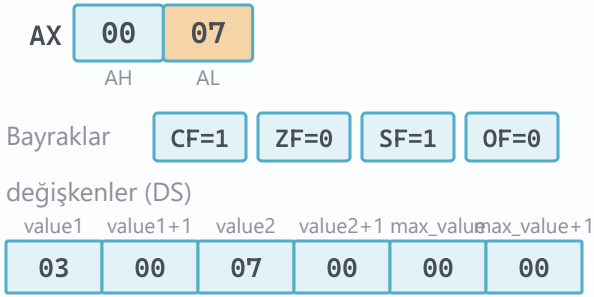
DEĞİŞKENLER

`AX = 0003h` `CF = 1` `ZF = 0` `SF = 1` `OF = 0`

`value1 = 0003h` `value2 = 0007h` `max_value = 0000h`

Kod İzleme: Büyük Olanı Bul (3, 7)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX − value2
4  JGE greater_or_equal
▶  MOV AX, value2      ; küçükse value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; sonucu sakla
11 RET
12 value1 DW 3
13 value2 DW 7
14 max_value DW ?
```



AX = value2 = **0007h** (büyük olan).

Adım 4 / 7

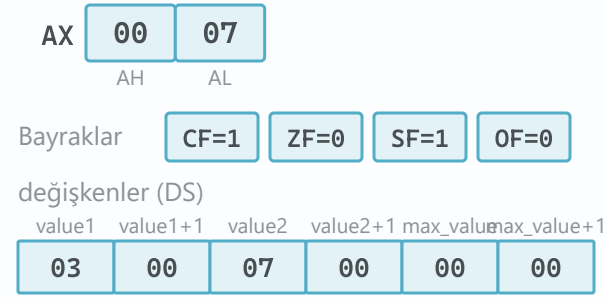
DEĞİŞKENLER

`AX = 0007h` `CF = 1` `ZF = 0` `SF = 1` `OF = 0`

`value1 = 0003h` `value2 = 0007h` `max_value = 0000h`

Kod İzleme: Büyük Olanı Bul (3, 7)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX – value2
4  JGE greater_or_equal
5  MOV AX, value2      ; küçükse value2
▶  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; sonucu sakla
11 RET
12 value1 DW 3
13 value2 DW 7
14 max_value DW ?
```



JMP koşulsuz atlar → store_max.

Adım 5 / 7

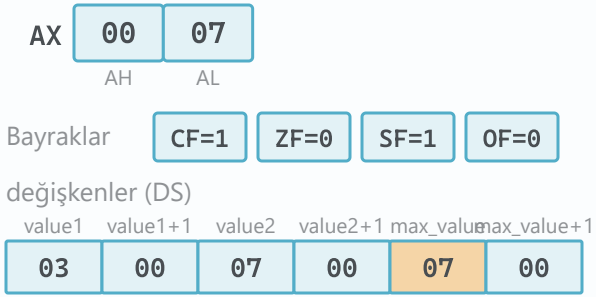
DEĞİŞKENLER

AX = 0007h | CF = 1 | ZF = 0 | SF = 1 | OF = 0

value1 = 0003h | value2 = 0007h | max_value = 0000h

Kod İzleme: Büyük Olanı Bul (3, 7)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX – value2
4  JGE greater_or_equal
5  MOV AX, value2      ; küçükse value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
▶  MOV max_value, AX   ; sonucu sakla
11 RET
12 value1 DW 3
13 value2 DW 7
14 max_value DW ?
```



max_value ← AX = **0007h**.

Adım 6 / 7

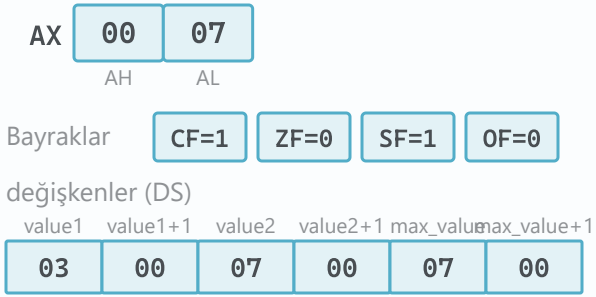
DEĞİŞKENLER

AX = **0007h** | CF = **1** | ZF = **0** | SF = **1** | OF = **0**

value1 = **0003h** | value2 = **0007h** | max_value = **0007h**

Kod İzleme: Büyük Olanı Bul (3, 7)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX – value2
4  JGE greater_or_equal
5  MOV AX, value2      ; küçükse value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; sonucu sakla
▶  RET
12 value1 DW 3
13 value2 DW 7
14 max_value DW ?
```



RET: DOS'a dönüş. RET olmasaydı CPU value1'in baytlarını komut sanıp yürütürdü.

Adım 7 / 7

DEĞİŞKENLER

AX = 0007h | CF = 1 | ZF = 0 | SF = 1 | OF = 0

value1 = 0003h | value2 = 0007h | max_value = 0007h

ÖRNEK PROGRAMLAR

Karşılaştırma Notları

CMP

`CMP a, b` = `a − b` hesaplanır, sonuç atılır; yalnız bayraklar (CF, ZF, SF, OF, PF, AF) güncellenir.

İşaretli / işaretsiz

`JGE`, `JG`, `JL`: işaretli (SF, OF). `JAE`, `JA`, `JB`: işaretsiz (CF). value1 = −1 (FFFFh), value2 = 5 için JGE atlamaz, JAE atlar.

Neden RET?

Veri tanımları kodun hemen ardından gelir; `RET` olmasaydı CPU `0A 00 05 00…` baytlarını komut olarak yürütürdü. Çözüm: `RET` (ya da başta `jmp start`).

İyileştirme

greater_or_equal altındaki `MOV AX, value1` gereksizdir; AX zaten value1'dir. Etiket doğrudan store_max'e atlayabilir.

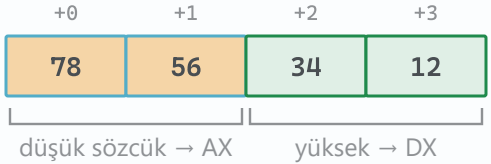


ÖRNEK PROGRAMLAR

Çift Sözcük Değişken

```
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]
```

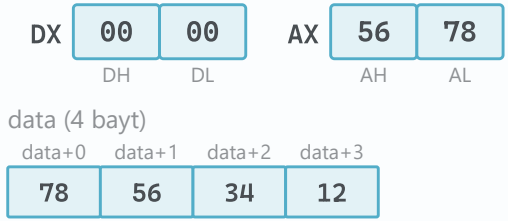
Dört tanımın da bellekteki karşılığı **aynıdır**:



- `DD` 32 bit = 4 bayt; en düşük bayt en düşük adreste.
- 8086'nın 32 bitlik yazmacı yoktur: değer **DX:AX** çiftinde tutulur (DX yüksek, AX düşük sözcük).

Kod İzleme: DX:AX'e 32 Bitlik Yükleme

```
1 ▶ mov ax, word ptr data       ; data+0
2   mov dx, word ptr [data+2]   ; data+2
3   ret
4   data dd 12345678h
```



data'nın ilk iki baytı (78 56) → AX = **5678h**. `word ptr` : data DD olduğu için boyut açıkça belirtilir.

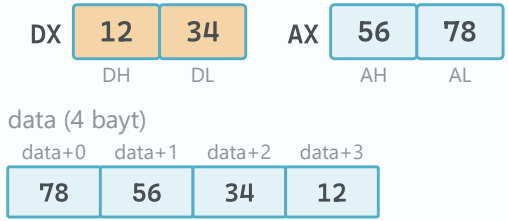
Adım 1 / 2

DEĞİŞKENLER

`DX = 0000h` `AX = 5678h`

Kod İzleme: DX:AX'e 32 Bitlik Yükleme

```
1 mov ax, word ptr data       ; data+0
▶ mov dx, word ptr [data+2]   ; data+2
3 ret
4 data dd 12345678h
```



data+2'deki iki bayt (34 12) → DX = **1234h**. DX:AX = **1234:5678h** = 12345678h.

Adım 2 / 2

DEĞİŞKENLER

DX = **1234h** AX = **5678h**

ÖRNEK PROGRAMLAR

Değişkenlerle İlgili Sık Hatalar

Değer ile adresi karıştırmak

`MOV SI, dizi` ilk sözcüğü yükler; adres için `LEA SI, dizi`.

Bellekten belleğe taşımak

`MOV v1, v2` yoktur; değer bir yazmaçtan geçer.

Sözcük dizisinde indis

3. eleman `w[4]`'tedir: indis bayt cinsindendir.

? değerine güvenmek

`toplam DW ?` sıfır olmayabilir; önce `MOV toplam, 0`.

Soneki unutmak

`DW 10` onluk 10'dur (000Ah); onaltılık için `10h`.

Ad çakışması

`SUTUN EQU 4` ile `sutun DB 1` aynı addır: "iki kez tanımlı".

Veriyi kodun yoluna koymak

RET ya da JMP yoksa veri baytları komut sanılır.

Sonlandırıcıyı unutmak

0 ya da '$' yoksa yazdırma arkadaki baytlarla sürer.

ÖRNEK PROGRAMLAR

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

Adres: `LEA BX, x` / `MOV BX, OFFSET x`. Değer: `MOV AL, x`. Boyut değiştirme: `word ptr`, `byte ptr`.

ÖZET

Özet

Değişken
Bir bellek konumuna verilen ad; derleyici adı bağıl konumla değiştirir.

DB / DW / DD
1, 2, 4 bayt; little-endian; `?` ilk değersiz.

ORG 100h
.COM kodu 0100h'den başlar; ilk 256 bayt PSP. Veriler RET'ten sonra ya da jmp ile atlanarak.

Diziler
Ardışık baytlar; `a[i]` bayt uzaklığıdır; DUP ile tekrar; DW dizisinde indis 2 × i.

LEA / OFFSET
Adres almak için; LEA çalışma anında indisli adres de hesaplar.

EQU
Derleme anı sabiti; bellekte yer kaplamaz, değiştirilemez.

ÖZET

Alıştırmalar

1. `x DB 5, 6` · `y DW 0ABCDh` · `z DB 'OK'` tanımları 0200h'den başlıyorsa her baytın adresi ve değeri?
2. `tablo DW 10 DUP(?)` kaç bayt kaplar? 7. elemanın (indis 6) bağıl uzaklığı nedir?
3. Önceki sorudaki `y` için `MOV AL, BYTE PTR y` ve `MOV AL, BYTE PTR y+1` sonrası AL?
4. `LEA SI, a[BX]` ile `MOV SI, OFFSET a` arasındaki fark nedir?
5. İki DW değişkenin toplamını üçüncüsüne yazan, toplamda taşma olursa (CF = 1) `tasma DB 1` yapan kodu yazın.
6. `dd 0FFFFFFFFh` değerine 1 eklemek için DX:AX'te hangi komut çifti gerekir?

▶ **Cevaplar**

Kaynaklar

- emu8086 belgeleri, *8086 assembler tutorial for beginners*, bölüm 3: variables (DB, DW, DUP, LEA, OFFSET, EQU).
- Barry B. Brey, *The Intel Microprocessors*, 8. baskı, Pearson.
- Kip R. Irvine, *Assembly Language for x86 Processors*, Pearson (veri tanımlama direktifleri).
- Microsoft MASM 6.1 Programmer's Guide: DB/DW/DD, DUP, EQU, OFFSET.

Son

Bölüm 4.2: Değişkenler

Sonraki bölüm: Bölüm 5.1: Kesmeler · Terim sözlüğü