Bölüm 1.3: Bilgisayarlar

Bilgisayar ve Programlamaya Giriş

Veri temsili | Metin | Saklanan program | CPU ve bellek | Soyutlama

Sercan KÜLCÜ · Giresun Üniversitesi

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü: https://sercankulcu.github.io/files/computer/slides/Bolum_01_03_Bilgisayarlar.html

GİRİŞ

Öğrenme Çıktıları

Sayı temsili

Sınırlı bit sayısının sonuçlarını (negatif sayılar, taşma, kayan nokta) açıklamak.

Metin temsili

ASCII, Unicode ve dizgi kavramlarını ayırt etmek.

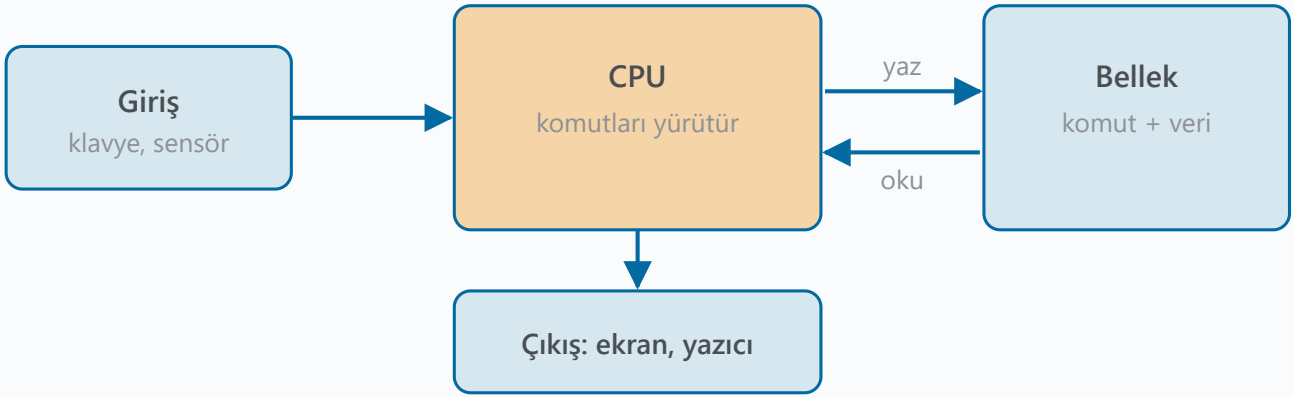
Program yürütme

Saklanan program ve getir–çöz–yürüt döngüsünü açıklamak.

Mimari

CPU, bellek, yazmaç ve veri yolu ilişkisini yorumlamak.

Bilgisayarın Temel Parçaları



CPU
Komutları yürütür.

Bellek
Veri ve komutları saklar.

Giriş / çıkış
Dış dünya ile veri alışverişi sağlar.

VERİ TEMSİLİ

Bilgiden Bitlere

Sayılar, harfler, görüntüler ve program komutları bellekte bit dizileri olarak saklanır. Bit dizisinin anlamını veri tipi ve kodlama belirler.




```
01000001 (65) + 00100000 (32) = 01100001 (97) = "a"
```

Taban Kavramı

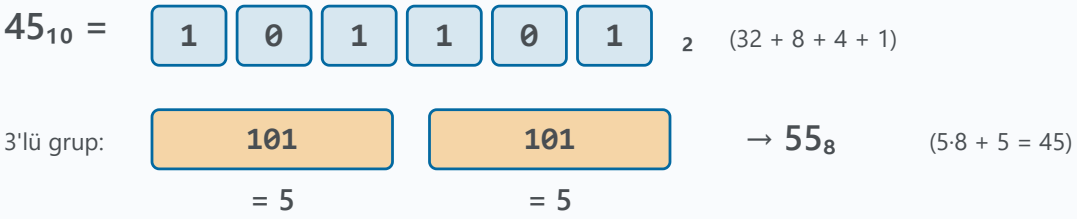
Sistem	Taban	Rakamlar	Örnek
İkili	2	0–1	1101_2
Sekizli	8	0–7	15_8
Onluk	10	0–9	13_{10}
Onaltılık	16	0–9, A–F	D_{16}

45_{10} = **1 0 1 1 0 1** $_2$ (32 + 8 + 4 + 1)

45_{10} = **101101_2**. Sekizli ve onaltılık tabanlar, uzun ikili sayıları kısa yazmanın yoludur.

Taban Kavramı

Sistem	Taban	Rakamlar	Örnek
İkili	2	0–1	1101_2
Sekizli	8	0–7	15_8
Onluk	10	0–9	13_{10}
Onaltılık	16	0–9, A–F	D_{16}



8 = 2^3 olduğundan bitler sağdan **üçerli** gruplanır: 101 101 → **55_8**.

Taban Kavramı

Sistem	Taban	Rakamlar	Örnek
İkili	2	0–1	1101_2
Sekizli	8	0–7	15_8
Onluk	10	0–9	13_{10}
Onaltılık	16	0–9, A–F	D_{16}

45_{10} = **1 0 1 1 0 1** $_2$ (32 + 8 + 4 + 1)

3'lü grup: **101** **101** → 55_8 (5·8 + 5 = 45)

= 5 = 5

4'lü grup: **0010** **1101** → $2D_{16}$ (2·16 + 13 = 45)

= 2 = 2 = 13 = D

16 = 2^4 olduğundan **dörderli** gruplanır: 0010 1101 → 2 ve 13 (D) → **$2D_{16}$**.

VERİ TEMSİLİ

Negatif Tam Sayılar

Modern bilgisayarlar negatif tam sayıları çoğunlukla **ikiye tümleyen** gösterimle saklar.

Tek sıfır

Pozitif ve negatif sıfır ayrımı oluşmaz.

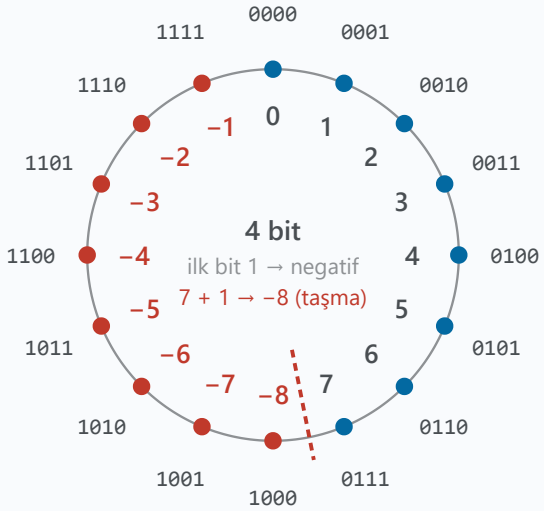
Aritmetik

Toplama devreleri pozitif ve negatif sayılarda aynı kuralları kullanabilir.

n bit aralığı

-2^{n-1} ile $2^{n-1}-1$ arasındadır.

```
8 bit: −128 … 127
```

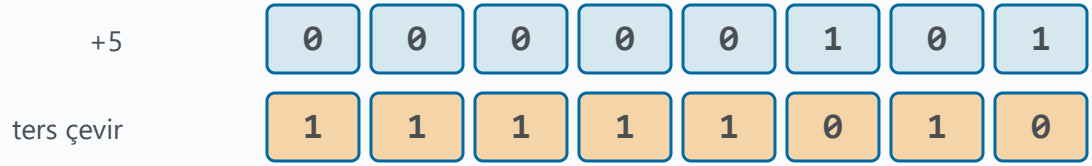


İkiye Tümleyen: −5 Nasıl Yazılır?



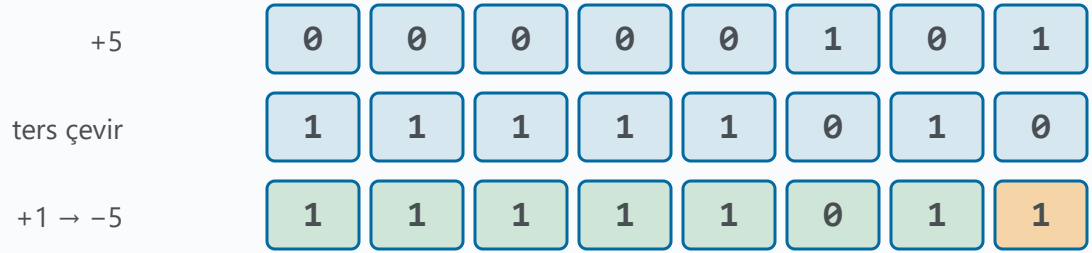
8 bitte **+5 = 00000101**. −5'i bulmak için iki adım yeterli.

İkiye Tümleyen: −5 Nasıl Yazılır?



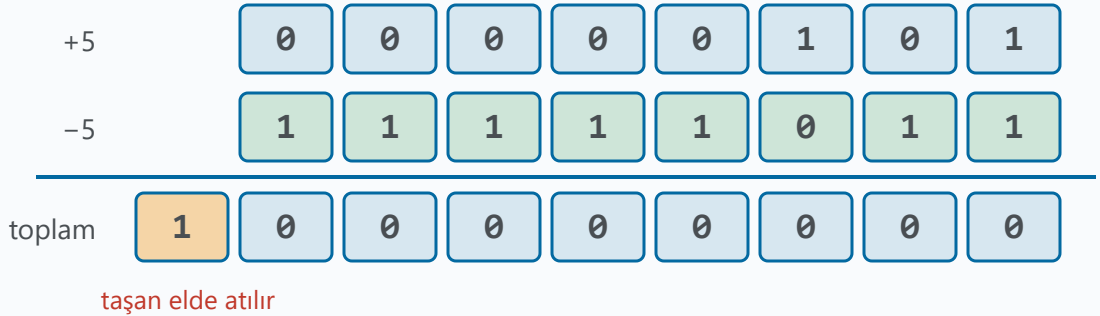
1. Tüm bitleri **ters çevir** (0 ↔ 1): 11111010. Buna birler tümleyeni denir.

İkiye Tümleyen: −5 Nasıl Yazılır?



2. Sonuca **1 ekle**: 11111010 + 1 = **11111011** = −5. En soldaki 1, sayının negatif olduğunu gösterir.

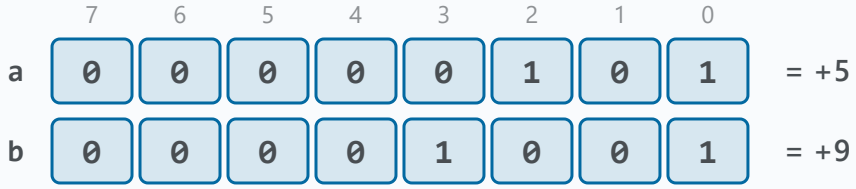
İkiye Tümleyen: −5 Nasıl Yazılır?



Kontrol: 00000101 + 11111011 = 100000000. 9. bit 8 bite sığmaz ve atılır → **0**. Toplama devresi negatifler için ayrıca bir şey yapmaz.

VERİ TEMSİLİ · ADIM ADIM

Çıkarma Aslında Toplamadır: 5 − 9



İşlemcide ayrı bir çıkarma devresi gerekmez: **a − b = a + (−b)**. Önce b'nin ikiye tümleyeni bulunur.

Adım 1 / 5

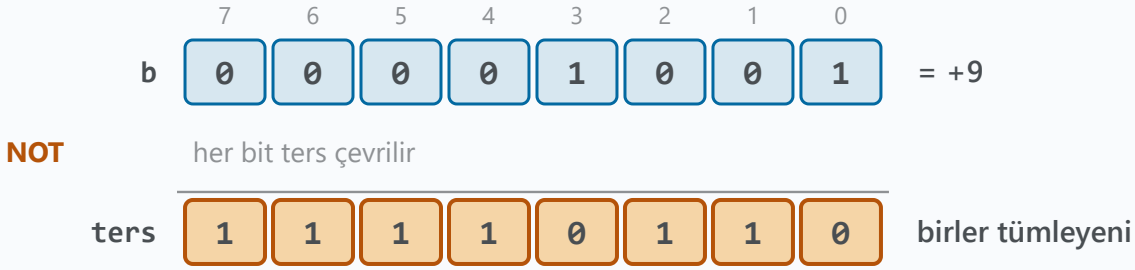
```
▶ a ← 00000101      ; +5
▶ b ← 00001001      ; +9
3 ters ← NOT b      ; bitleri çevir
4 eksi ← ters + 1   ; −9
5 s ← a + eksi      ; 5 + (−9)
6 yaz(s)            ; işaretli oku
```

DEĞİŞKENLER

`a = 00000101` `b = 00001001`

`ters = –` `eksi = –` `s = –`

Çıkarma Aslında Toplamadır: 5 − 9



Bütün bitler ters çevrilir (0 ↔ 1): **11110110**.

Adım 2 / 5

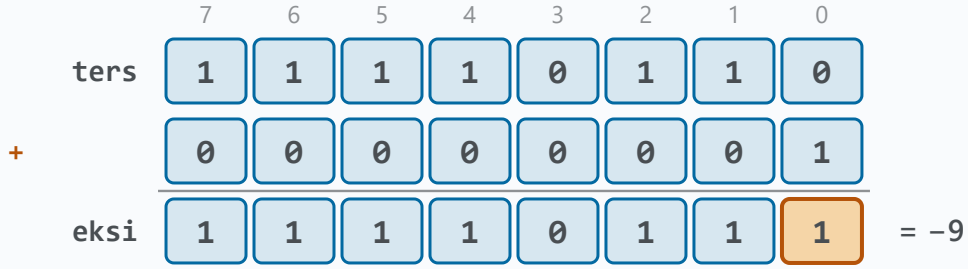
```
1 a ← 00000101      ; +5
2 b ← 00001001      ; +9
▶ ters ← NOT b      ; bitleri çevir
4 eksi ← ters + 1   ; −9
5 s ← a + eksi      ; 5 + (−9)
6 yaz(s)            ; işaretli oku
```

DEĞİŞKENLER

`a = 00000101` `b = 00001001`

`ters = 11110110` `eksi = –` `s = –`

Çıkarma Aslında Toplamadır: 5 − 9



1 eklenir: **11110111** = −9. (İşaretsiz okunursa 247 = 256 − 9.)

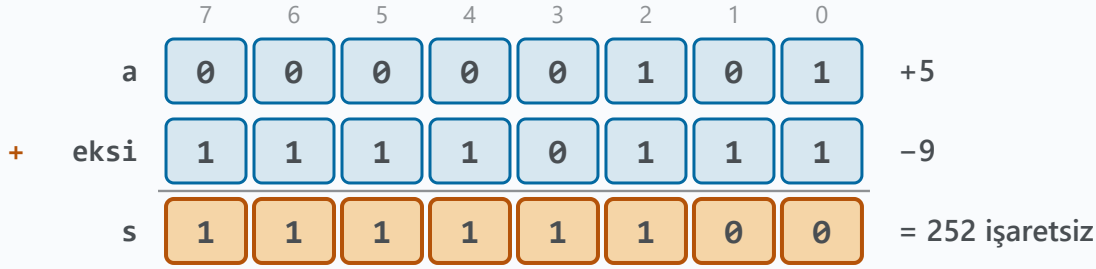
Adım 3 / 5

```
1 a ← 00000101      ; +5
2 b ← 00001001      ; +9
3 ters ← NOT b      ; bitleri çevir
▶ eksi ← ters + 1   ; -9
5 s ← a + eksi      ; 5 + (-9)
6 yaz(s)            ; işaretli oku
```

DEĞİŞKENLER

- a = 00000101
- b = 00001001
- ters = 11110110
- eksi = 11110111
- s = −

Çıkarma Aslında Toplamadır: 5 − 9



Sıradan ikili toplama: 5 + 247 = 252 = **11111100**. 256'ya ulaşmadığı için 9. bite elde çıkmaz.

Adım 4 / 5

```
1 a ← 00000101      ; +5
2 b ← 00001001      ; +9
3 ters ← NOT b      ; bitleri çevir
4 eksi ← ters + 1   ; −9
▶ s ← a + eksi      ; 5 + (−9)
6 yaz(s)            ; işaretli oku
```

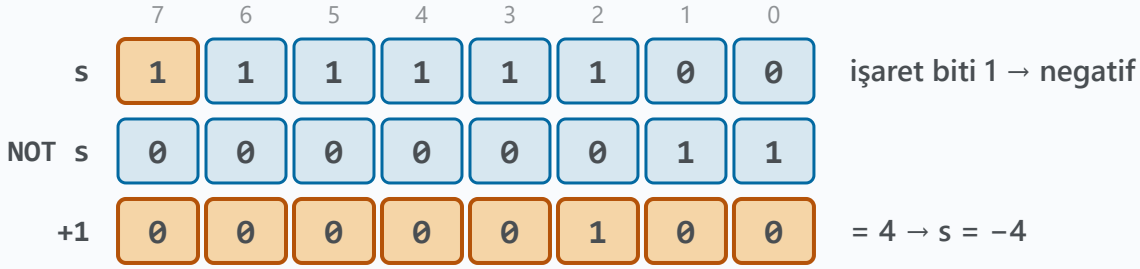
DEĞİŞKENLER

`a = 00000101` `b = 00001001`

`ters = 11110110` `eksi = 11110111`

`s = 11111100`

Çıkarma Aslında Toplamadır: 5 − 9



İşaretli okuma: en soldaki bit 1 olduğundan sayı negatiftir. Büyüklük için yine ters çevir + 1: 00000100 = 4 → **s = −4** ✓.

Adım 5 / 5

```
1 a ← 00000101      ; +5
2 b ← 00001001      ; +9
3 ters ← NOT b      ; bitleri çevir
4 eksi ← ters + 1   ; −9
5 s ← a + eksi      ; 5 + (−9)
▶ yaz(s)            ; işaretli oku
```

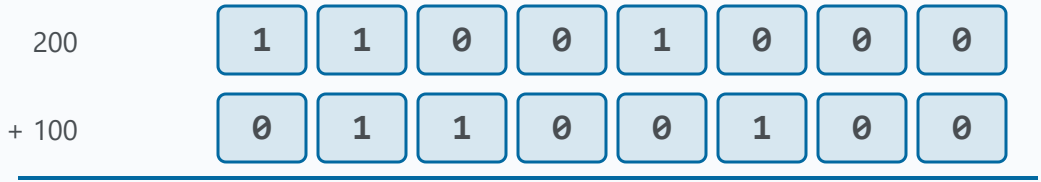
DEĞİŞKENLER

`a = 00000101` `b = 00001001`

`ters = 11110110` `eksi = 11111111`

`s = −4`

Taşma: 8 Bitte 200 + 100

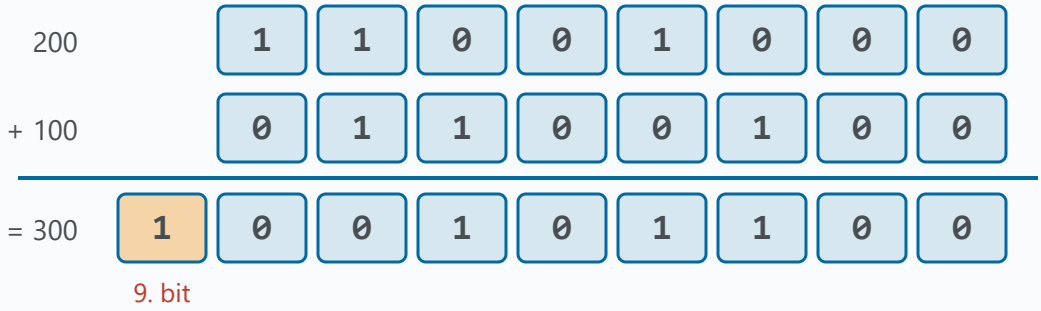


8 bit işaretsiz sayı en fazla **255** olabilir. 200 + 100 toplayalım.

Adım 1 / 3

İşaretli 8 bitte de 127 + 1 taşar ve −128 olur. Programlarda tip seçimi bu yüzden önemlidir.

Taşma: 8 Bitte 200 + 100



Gerçek toplam **300** = 100101100_2: sonuç **9 bit** gerektirir.

İşaretli 8 bitte de 127 + 1 taşar ve −128 olur. Programlarda tip seçimi bu yüzden önemlidir.

Taşma: 8 Bitte 200 + 100



Yazmaç yalnızca 8 bit tutar; 9. bit (256) kaybolur → sonuç **44**. Buna **taşma (overflow)** denir: 300 − 256 = 44.

İşaretli 8 bitte de 127 + 1 taşar ve −128 olur. Programlarda tip seçimi bu yüzden önemlidir.

İşaretli Taşma ve Elde Farkı

İşlem (8 bit)	İkili toplam	Elde	İşaretli taşma	Sonuç
5 + (−9)	`00000101 + 11110111 = 11111100`	0	yok	−4 ✓
(−1) + (−1)	`11111111 + 11111111 = 1 11111110`	1	yok	−2 ✓
100 + 50	`01100100 + 00110010 = 10010110`	0	**var**	−106 ✗
(−100) + (−50)	`10011100 + 11001110 = 1 01101010`	1	**var**	+106 ✗

Elde (9. bit)

İşaretsiz yorumda sonucun 8 bite sığmadığını gösterir. İşaretli toplamada elde atılır ve sonuç yine doğru olabilir: (−1) + (−1) = −2.

İşaretli taşma

Aynı işaretli iki sayının toplamı ters işaretli çıkarsa oluşur. Farklı işaretli iki sayının toplamı hiçbir zaman taşmaz.

Java'da `(byte) (100 + 50)` değeri −106'dır: tam sayı taşması hata vermez, sessizce yanlış sonuç üretir.

VERİ TEMSİLİ

Kayan Noktalı Sayılar

Kayan nokta

Çok büyük ve çok küçük değerleri bilimsel gösterime benzer biçimde saklar.

Sınırlı hassasiyet

Her ondalık değer sonlu bitlerle tam temsil edilemez.

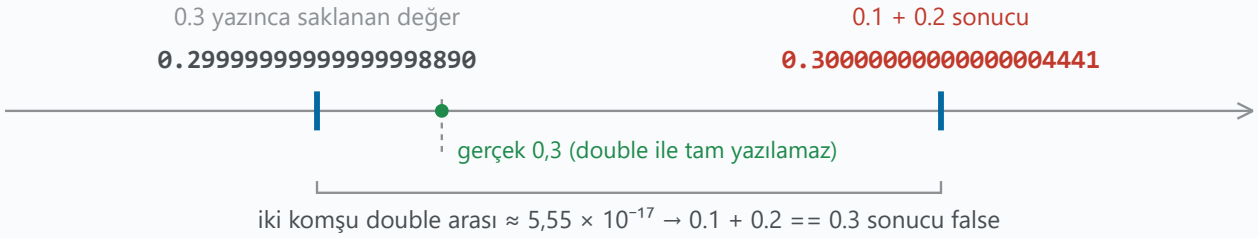
Yuvarlama

Hesaplama sonuçlarında küçük farklar oluşabilir.

Karşılaştırma

Kayan noktalı sonuçlar tolerans değeriyle karşılaştırılabilir.

```
0.1 + 0.2  →  0.30000000000000004   // double ile hesaplanınca
```



0,1 Neden Tam Saklanamaz?

Kesri sürekli 2 ile çarp; tam kısım bir sonraki bit

$0{,}1_{10} =$ **0,**

Ondalık kesir ikiliye nasıl çevrilir? Kesri 2 ile çarp, tam kısmı (0 ya da 1) bit olarak al, kalan kesirle devam et.

0,1 Neden Tam Saklanamaz?

Kesri sürekli 2 ile çarp; tam kısım bir sonraki bit

0,1 × 2 = 0,2 → bit 0

$0{,}1_{10}$ = **0,0**

0,1 × 2 = 0,2 → tam kısım **0**.

0,1 Neden Tam Saklanamaz?

Kesri sürekli 2 ile çarp; tam kısım bir sonraki bit

0,1 × 2 = 0,2 — bit 0

0,2 × 2 = 0,4 — bit 0

$0{,}1_{10}$ = `0,00`

0,2 × 2 = 0,4 → tam kısım **0**.

0,1 Neden Tam Saklanamaz?

Kesri sürekli 2 ile çarp; tam kısım bir sonraki bit

0,1 × 2 = 0,2	bit 0
0,2 × 2 = 0,4	bit 0
0,4 × 2 = 0,8	bit 0

$0{,}1_{10}$ = **0,000**

0,4 × 2 = 0,8 → tam kısım **0**.

0,1 Neden Tam Saklanamaz?

Kesri sürekli 2 ile çarp; tam kısım bir sonraki bit

0,1 × 2 = 0,2	bit 0
0,2 × 2 = 0,4	bit 0
0,4 × 2 = 0,8	bit 0
0,8 × 2 = 1,6	bit 1

$0{,}1_{10}$ = **`0,0001`**

0,8 × 2 = 1,6 → tam kısım **1**, kesir 1 çıkarılarak devam eder.

0,1 Neden Tam Saklanamaz?

Kesri sürekli 2 ile çarp; tam kısım bir sonraki bit

0,1 × 2 = 0,2	bit 0
0,2 × 2 = 0,4	bit 0
0,4 × 2 = 0,8	bit 0
0,8 × 2 = 1,6	bit 1
0,6 × 2 = 1,2	bit 1

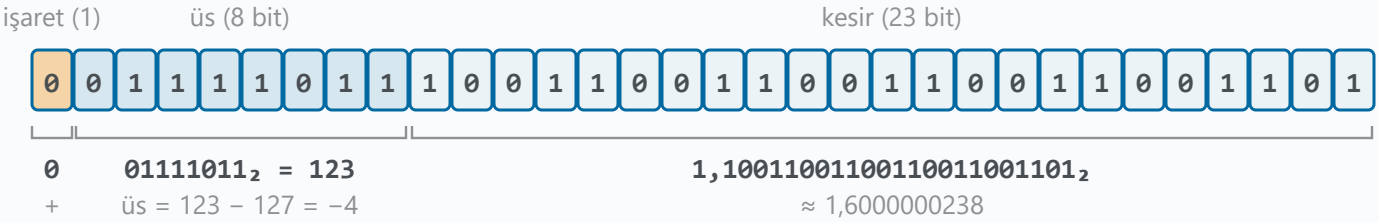
$0{,}1_{10}$ = **0,00011…**

0,2 tekrar geldi → “0011” sonsuza dek sürer

Sonlu bitle kesilir → küçük yuvarlama hatası

0,6 × 2 = 1,2 → tam kısım **1**, kesir 1 çıkarılarak devam eder. Kesir yine 0,2 oldu: açılım **döngüye girdi**, hiçbir zaman bitmez.

Kayan Noktalı Sayılar (IEEE 754)



değer = (−1)^işaret × 1,kesir × 2^(üs − 127)

0,1 (float) = 0x3DCCCCCD

≈ 1,6000000238 × 2^{-4} ≈ 0,10000000149

- **float** 32 bittir (1 + 8 + 23); **double** 64 bittir (1 + 11 + 52). Normal sayılarda baştaki 1 saklanmaz.
- 0,1 = 1,1001 1001 1001…$_2$ × 2^{-4}: kesir sonsuza dek tekrarlar, 23 bite sığmayınca yuvarlanır.
- Saklanan değer tam olarak 0,100000001490116119384765625'tir: 0,1'e çok yakın ama eşit değil.

VERİ TEMSİLİ · ADIM ADIM

−6,25 Sayısını `float` Olarak Kodlama

x = −6,25 → işaret biti 1



float 32 bittir: 1 işaret + 8 üs + 23 kesir. Sayı negatif olduğundan **işaret biti 1**; geri kalan işlemler 6,25 üzerinde yapılır.

Adım 1 / 6

```
▶ x ← -6,25
▶ işaret ← 1          ; negatif
3 ikili ← 110,01      ; 6 = 110, 0,25 = 0,01
4 norm ← 1,1001 × 2²  ; virgül 2 sola
5 üs ← 2 + 127 = 129  ; 10000001
6 kesir ← 1001 0000…  ; baştaki 1 yazılmaz
7 yaz(onaltılık(bitler))
```

DEĞİŞKENLER

`işaret = 1` `ikili = –` `üs = –` `kesir = –`

−6,25 Sayısını `float` Olarak Kodlama

$$6 = 110_2 \cdot 0{,}25 \times 2 = 0{,}5 \rightarrow 0 \cdot 0{,}5 \times 2 = 1{,}0 \rightarrow 1$$



Tam kısım 6 = 110_2; kesir 0,25, "2 ile çarp, tam kısmı al" yöntemiyle **$0{,}01_2$** olur ve kesir 0'a iner. 6,25 = **$110{,}01_2$**: 0,1'in aksine sonlu ve tam.

Adım 2 / 6

```
1 x ← −6,25
2 işaret ← 1          ; negatif
▶ ikili ← 110,01      ; 6 = 110, 0,25 = 0,01
4 norm ← 1,1001 × 2²  ; virgül 2 sola
5 üs ← 2 + 127 = 129  ; 10000001
6 kesir ← 1001 0000…  ; baştaki 1 yazılmaz
7 yaz(onaltılık(bitler))
```

DEĞİŞKENLER

`işaret = 1` `ikili = 110,01` `üs = –` `kesir = –`

VERİ TEMSİLİ · ADIM ADIM

−6,25 Sayısını `float` Olarak Kodlama

$$110{,}01_2 = 1{,}1001_2 \times 2^2$$



Bilimsel gösterime benzer biçimde virgül, solunda tek 1 kalacak kadar kaydırılır: 2 basamak sola → **1,1001 × 2²**.

Adım 3 / 6

```
1 x ← −6,25
2 işaret ← 1          ; negatif
3 ikili ← 110,01      ; 6 = 110, 0,25 = 0,01
▶ norm ← 1,1001 × 2²  ; virgül 2 sola
5 üs ← 2 + 127 = 129  ; 10000001
6 kesir ← 1001 0000…  ; baştaki 1 yazılmaz
7 yaz(onaltılık(bitler))
```

DEĞİŞKENLER

`işaret = 1` `ikili = 1,1001 × 2²` `üs = –` `kesir = –`

−6,25 Sayısını `float` Olarak Kodlama

üs alanı = 2 + 127 = 129 = 10000001_2



Üs alanına gerçek üs değil, **127 eklenmiş** hâli (sapma, bias) yazılır: 2 + 127 = 129. Böylece negatif üsler de işaretsiz saklanır.

Adım 4 / 6

```
1 x ← −6,25
2 işaret ← 1          ; negatif
3 ikili ← 110,01      ; 6 = 110, 0,25 = 0,01
4 norm ← 1,1001 × 2²  ; virgül 2 sola
▶ üs ← 2 + 127 = 129  ; 10000001
6 kesir ← 1001 0000…  ; baştaki 1 yazılmaz
7 yaz(onaltılık(bitler))
```

DEĞİŞKENLER

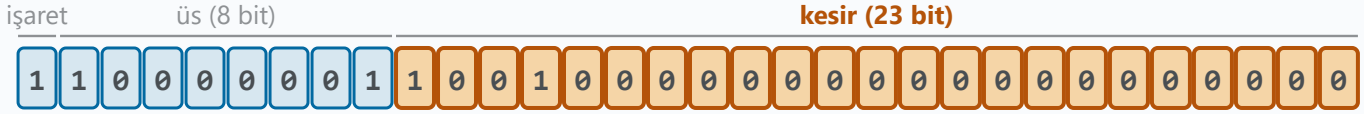
`işaret = 1` `ikili = 1,1001 × 2²`

`üs = 10000001` `kesir = —`

VERİ TEMSİLİ · ADIM ADIM

−6,25 Sayısını `float` Olarak Kodlama

kesir = 1,1001'in virgülden sonrası → 1001 ve 19 sıfır



Normal sayılarda baştaki 1 her zaman var olduğu için saklanmaz; yalnızca virgülden sonraki **1001** yazılır, kalan 19 bit 0 ile dolar.

Adım 5 / 6

```
1 x ← −6,25
2 işaret ← 1          ; negatif
3 ikili ← 110,01      ; 6 = 110, 0,25 = 0,01
4 norm ← 1,1001 × 2²  ; virgül 2 sola
5 üs ← 2 + 127 = 129  ; 10000001
▶ kesir ← 1001 0000…  ; baştaki 1 yazılmaz
7 yaz(onaltılık(bitler))
```

DEĞİŞKENLER

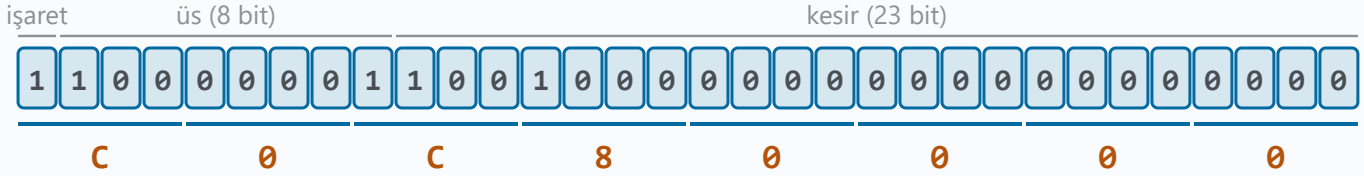
`işaret = 1` `ikili = 1,1001 × 2²`

`üs = 10000001` `kesir = 1001000…0`

VERİ TEMSİLİ · ADIM ADIM

−6,25 Sayısını `float` Olarak Kodlama

$$-6{,}25 = \mathrm{C0C80000}_{16}$$



32 bit dörderli gruplanır: 1100 0000 1100 1000 0000 … → **C0C80000**. Java'da `Float.floatToIntBits(-6.25f)` bu değeri verir.

Adım 6 / 6

```
1 x ← −6,25
2 işaret ← 1          ; negatif
3 ikili ← 110,01      ; 6 = 110, 0,25 = 0,01
4 norm ← 1,1001 × 2²  ; virgül 2 sola
5 üs ← 2 + 127 = 129  ; 10000001
6 kesir ← 1001 0000…  ; baştaki 1 yazılmaz
▶ yaz(onaltılık(bitler))
```

DEĞİŞKENLER

`işaret = 1` `ikili = 1,1001 × 2²`

`üs = 10000001` `kesir = 1001000…0`

METİN TEMSİLİ

ASCII

ASCII, temel Latin harflerini, rakamları, noktalama işaretlerini ve kontrol karakterlerini sayısal kodlarla eşleştirir.

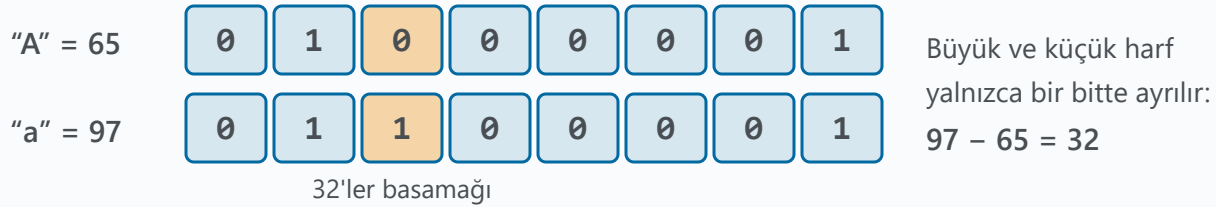
ASCII **7 bitlik** bir koddur: 128 karakter (0–127) tanımlar; bir bayt içinde saklandığında en yüksek bit 0'dır.

A
65
`01000001`

a
97
`01100001`

0
48
`00110000`

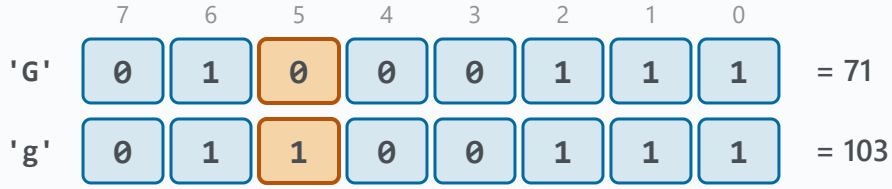
Satır sonu (LF)
Kontrol karakteri:
10
`00001010`



Büyük ve küçük harf yalnızca bir bitte ayrılır:
97 − 65 = 32

METİN TEMSİLİ · ADIM ADIM

Bit İşlemleri: Harfin Büyüklüğünü Değiştirme



"G" = 71, "g" = 103: fark 32 = 2^5. Büyük ve küçük harf yalnızca **5. bitte** (sağdan, 0'dan sayarak) ayrılır.

Adım 1 / 5

```
▶ c ← 'G'            ; 71 = 01000111
▶ m ← 00100000       ; maske: 5. bit (32)
3 k ← c OR m         ; biti 1 yap
4 b ← k AND (NOT m)  ; biti 0 yap
5 t ← b XOR m        ; biti ters çevir
6 s ← c << 1         ; sola kaydır
```

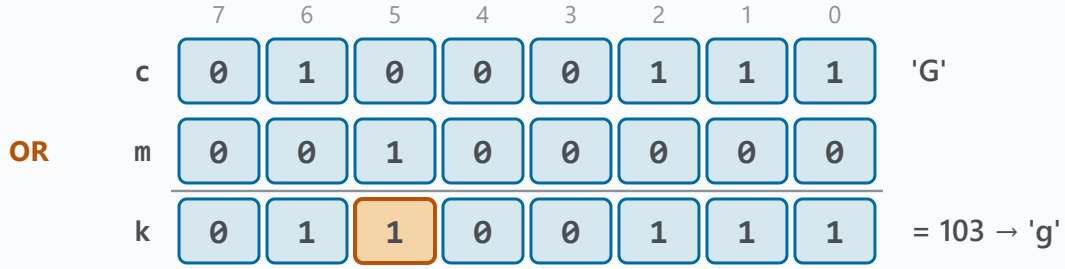
DEĞİŞKENLER

`c = 01000111` `m = 00100000` `k = –`

`b = –` `t = –` `s = –`

METİN TEMSİLİ · ADIM ADIM

Bit İşlemleri: Harfin Büyüklüğünü Değiştirme



OR: maskede 1 olan bit sonuçta 1 olur, öteki bitler aynen kalır. Büyük harf küçüğe döndü.

Adım 2 / 5

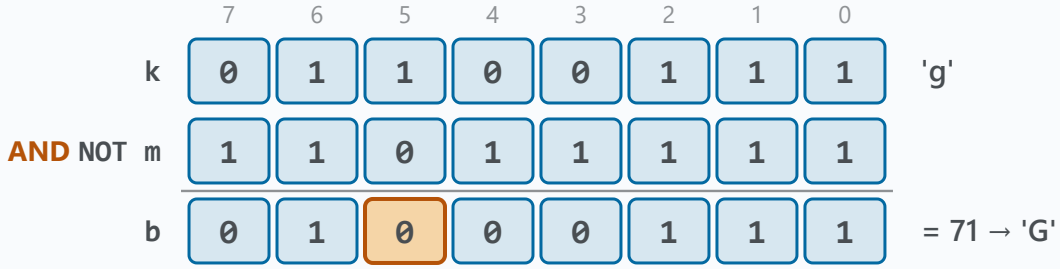
```
1 c ← 'G'            ; 71 = 01000111
2 m ← 00100000       ; maske: 5. bit (32)
▶ k ← c OR m         ; biti 1 yap
4 b ← k AND (NOT m)  ; biti 0 yap
5 t ← b XOR m        ; biti ters çevir
6 s ← c << 1         ; sola kaydır
```

DEĞİŞKENLER

`c = 01000111` `m = 00100000` `k = 01100111` `b = –` `t = –` `s = –`

METİN TEMSİLİ · ADIM ADIM

Bit İşlemleri: Harfin Büyüklüğünü Değiştirme



AND ters maskeyle: maskedeki bit 0'a çekilir, ötekiler korunur. Küçük harf yeniden büyük oldu.

Adım 3 / 5

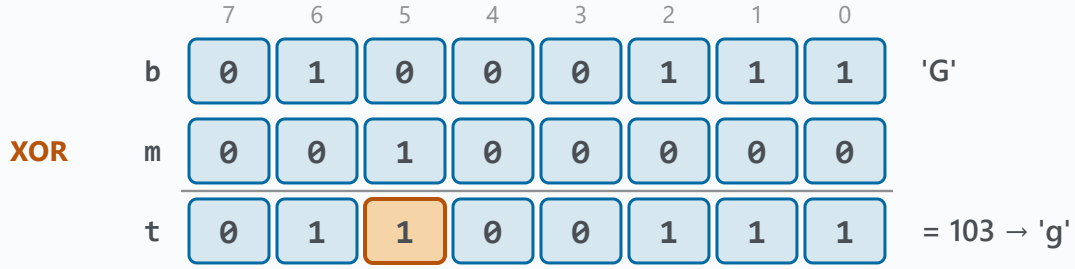
```
1 c ← 'G'            ; 71 = 01000111
2 m ← 00100000       ; maske: 5. bit (32)
3 k ← c OR m         ; biti 1 yap
▶ b ← k AND (NOT m)  ; biti 0 yap
5 t ← b XOR m        ; biti ters çevir
6 s ← c << 1         ; sola kaydır
```

DEĞİŞKENLER

c = 01000111 | m = 00100000 | k = 01100111 | b = 01000111 | t = – | s = –

METİN TEMSİLİ · ADIM ADIM

Bit İşlemleri: Harfin Büyüklüğünü Değiştirme



XOR: maskedeki bit ters çevrilir. Aynı maskeyle ikinci kez XOR, eski değeri geri verir.

Adım 4 / 5

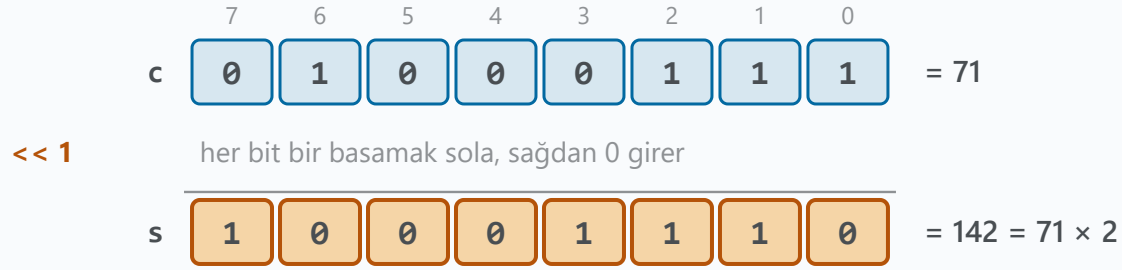
```
1 c ← 'G'            ; 71 = 01000111
2 m ← 00100000       ; maske: 5. bit (32)
3 k ← c OR m         ; biti 1 yap
4 b ← k AND (NOT m)  ; biti 0 yap
▶ t ← b XOR m        ; biti ters çevir
6 s ← c << 1         ; sola kaydır
```

DEĞİŞKENLER

- c = 01000111
- m = 00100000
- k = 01100111
- b = 01000111
- t = 01100111
- s = –

METİN TEMSİLİ · ADIM ADIM

Bit İşlemleri: Harfin Büyüklüğünü Değiştirme



Sola kaydırma: her bit bir basamak yükselir, sağdan 0 girer → değer 2 ile çarpılır. Sağa kaydırma (`>> 1`) 2'ye böler: 71 → 35.

Adım 5 / 5

```
1 c ← 'G'              ; 71 = 01000111
2 m ← 00100000         ; maske: 5. bit (32)
3 k ← c OR m           ; biti 1 yap
4 b ← k AND (NOT m)    ; biti 0 yap
5 t ← b XOR m          ; biti ters çevir
▶ s ← c << 1           ; sola kaydır
```

DEĞİŞKENLER

c = 01000111	m = 00100000
k = 01100111	b = 01000111
t = 01100111	s = 10001110

METİN TEMSİLİ

Unicode

Unicode, farklı yazı sistemlerindeki karakterlere benzersiz kod noktaları atar.

Kod noktası

Karakterin evrensel numarasıdır.

`A → U+0041`

UTF-8

Unicode karakterlerini değişken uzunlukta bayt dizileriyle kodlar.

Geniş kapsam

Türkçe karakterler, farklı alfabeler, matematik simgeleri ve emoji.

Dosya kodlaması

Aynı kod noktası farklı Unicode dönüşüm biçimleriyle saklanabilir.

karakter	kod noktası	UTF-8 baytları (onaltılık)	uzunluk
A	U+0041	41	1 bayt
ğ	U+011F	C4 9F	2 bayt
€	U+20AC	E2 82 AC	3 bayt
😀	U+1F600	F0 9F 98 80	4 bayt

METİN TEMSİLİ · ADIM ADIM

UTF-8 Kodlama: “ğ”

“ğ” → kod noktası U+011F = 287

Unicode, “ğ” harfine **U+011F** (onluk 287) kod noktasını atar. UTF-8 bu sayıyı baytlara nasıl yazar?

UTF-8 Kodlama: “ğ”

“ğ” → kod noktası U+011F = 287

11 bit: `0` `0` `1` `0` `0` `0` `1` `1` `1` `1` `1`

287 ikili yazılır: 00100011111. 7 bite sığmaz (ASCII değil) ama 11 bite sığar → **2 bayt** gerekir.

UTF-8 Kodlama: “ğ”

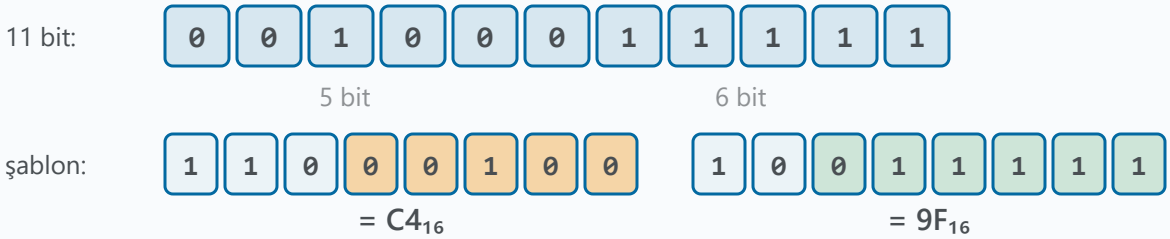
“ğ” → kod noktası U+011F = 287



2 baytlık UTF-8 şablonu **110xxxxx 10xxxxxx**: 11 bit, 5 + 6 olarak ikiye bölünür.

UTF-8 Kodlama: “ğ”

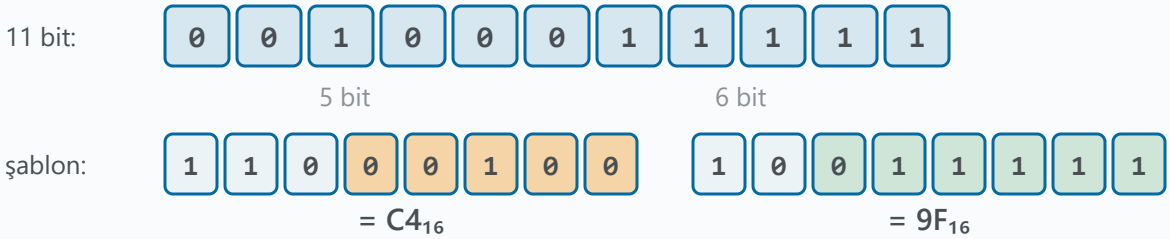
“ğ” → kod noktası U+011F = 287



Bitler şablona yerleşir: **11000100 10011111** = **C4 9F**. Baştaki 110 ve 10, baytın görevini söyler.

UTF-8 Kodlama: “ğ”

“ğ” → kod noktası U+011F = 287



Başka örnekler:

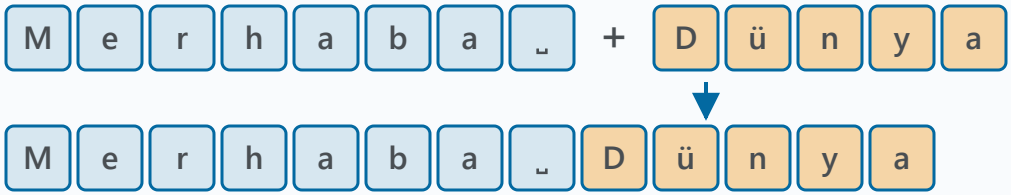
A	ğ	€	😜
41 (1 B)	C4 9F (2 B)	E2 82 AC (3 B)	F0 9F 98 80 (4 B)

ASCII karakterleri 1 bayttır; Türkçe harfler 2, € 3, emoji 4 bayt tutar. UTF-8 bu yüzden **değişken uzunluklu**dur.

METİN TEMSİLİ

Dizgiler

Dizgi (string), sıralı karakterlerden oluşan veri tipidir.



uzunluk: 13 karakter · UTF-8'de 14 bayt (“ü” 2 bayt)

Uzunluk
Karakter veya kod birimi sayısı.

Birleştirme
İki dizgiyi uç uca ekleme.

Arama
Bir karakteri veya alt dizgiyi bulma.

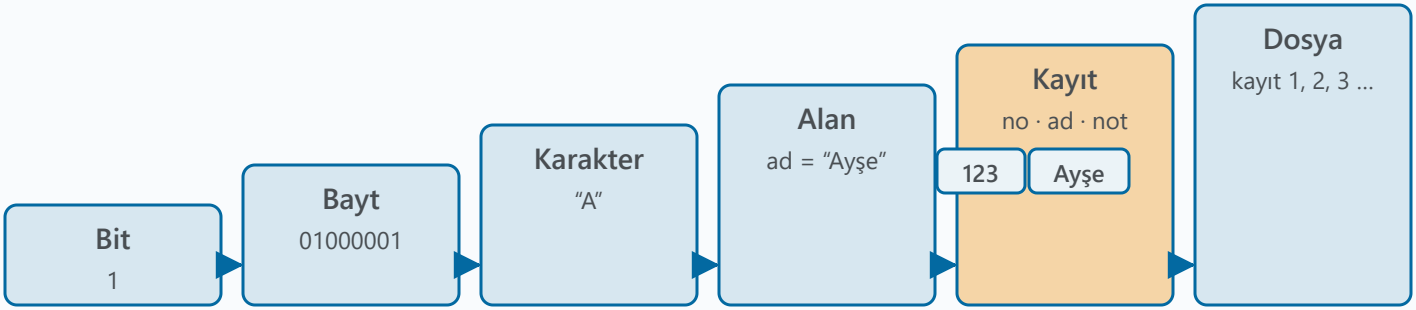
Karşılaştırma
İçerik ve sıralama ilişkisini denetleme.

Ekranda görünen karakter sayısı ile bellekteki bayt (kod birimi) sayısı her zaman aynı değildir.

VERİ TEMSİLİ

Yapılandırılmış Bilgi

Bitler daha büyük yapılar içinde düzenlenerek anlam kazanır.



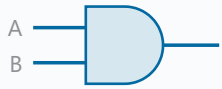
- Bir öğrenci kaydı numara, ad ve not alanlarından oluşabilir.
- Dosya biçimi, alanların nasıl sıralandığını ve yorumlandığını tanımlar.

Veri Temsilinde Sık Yapılan Hatalar

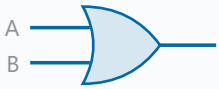
Sık yapılan hata	Doğrusu
Tabanı yazmadan okumak: “1101 bin yüz birdir”	1101_2 = 13. Taban alt simgeyle ya da `0b` / `0x` önekiyle belirtilir.
10_{16}'yı on sanmak	10_{16} = 16; onaltılıkta on, A ile yazılır.
İkiye tümleyende yalnız bitleri çevirmek	11111010 birler tümleyenidir (−5 değil); 1 eklenince 11111011 = −5.
Kayan noktalı sonuçları == ile karşılaştırmak	0,1 + 0,2 ≠ 0,3; |a − b| küçük bir toleranstan küçük mü diye bakılır.
Rakam karakteri ile sayıyı karıştırmak	'7' karakterinin kodu 55'tir; sayısal değeri '7' − '0' = 7.
Karakter sayısını bayt sayısı sanmak	UTF-8'de “çağ” 3 karakter, 5 bayttır (ç 2 + a 1 + ğ 2).
Elde ile işaretli taşmayı karıştırmak	Elde işaretsiz yorumun, taşma işaretli yorumun sorunudur; biri olurken öteki olmayabilir.

Mantık Kapıları

A	B	AND	OR	XOR	NOT A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0



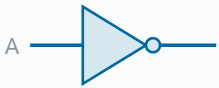
AND (VE)



OR (VEYA)



XOR (ÖZEL VEYA)

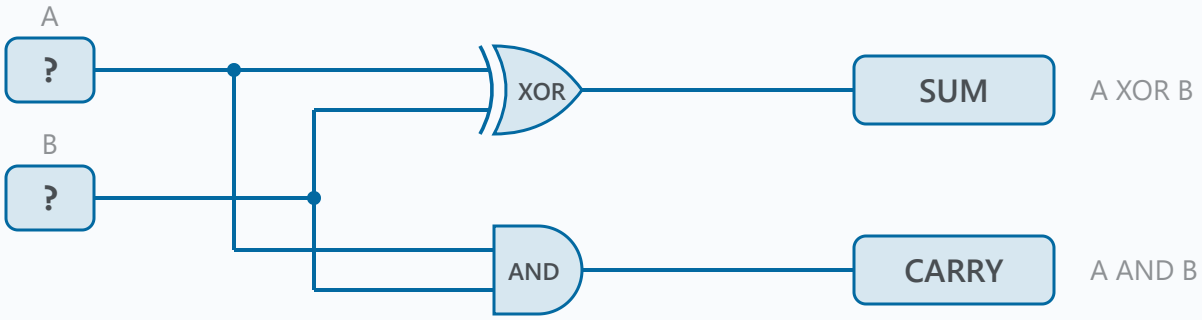


NOT (DEĞİL)

- **AND** iki girdi de 1 ise, **OR** girdilerden en az biri 1 ise 1 verir; **NOT** girdiyi tersine çevirir.
- **XOR** (özel veya): girdiler farklıysa 1. Toplayıcıda sonuç basamağını üretir.

Kapılar birkaç transistörden kurulur; işlemcideki tüm işlemler bu temel kapıların birleşiminden oluşur.

Yarım Toplayıcı: Kapılardan Toplamaya



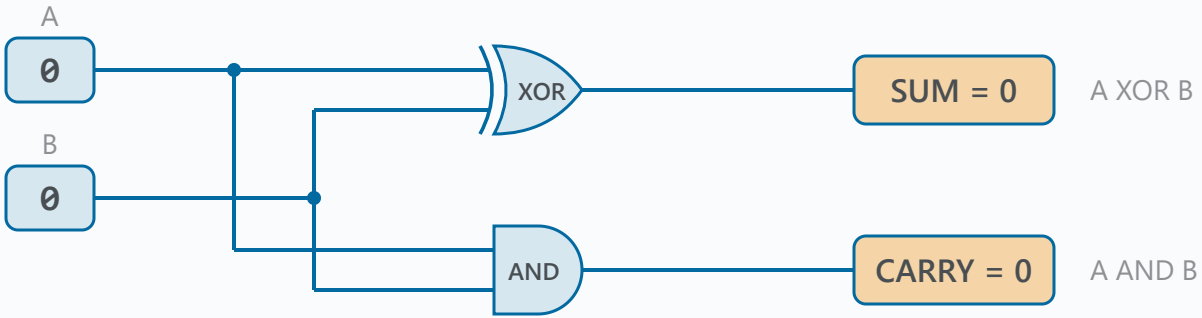
Sonuç iki bitle yazılır: CARRY SUM

Yarım toplayıcı iki tek bitlik sayıyı (A, B) toplar. **SUM = A XOR B** sonuç basamağını, **CARRY = A AND B** eldeyi verir. Yalnızca iki kapı yeterlidir.

Adım 1 / 4

Yarım toplayıcı gelen eldeyi almaz. Çok bitli toplama için üç girişli **tam toplayıcı** gerekir (iki yarım toplayıcı ve bir OR kapısı).

Yarım Toplayıcı: Kapılardan Toplamaya

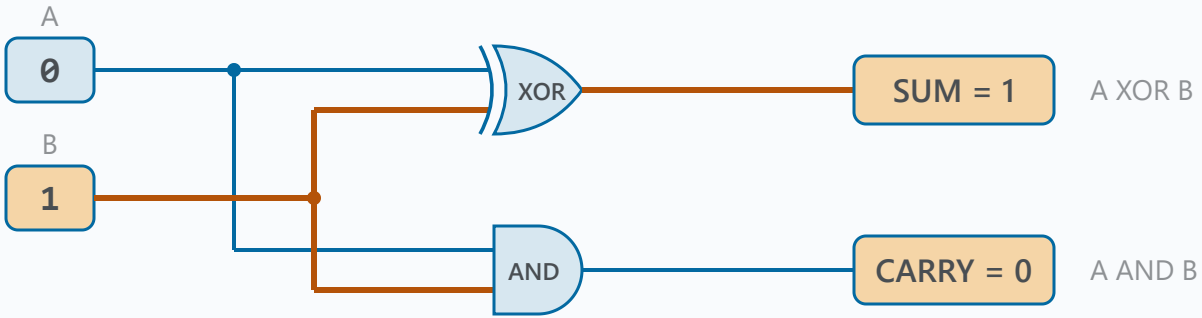


$0 + 0 = 00_2 = 0$
(CARRY, SUM) sırasıyla yazılır

0 + 0: XOR(0, 0) = 0, AND(0, 0) = 0 → **$00_2 = 0$**.

Yarım toplayıcı gelen eldeyi almaz. Çok bitli toplama için üç girişli **tam toplayıcı** gerekir (iki yarım toplayıcı ve bir OR kapısı).

Yarım Toplayıcı: Kapılardan Toplamaya



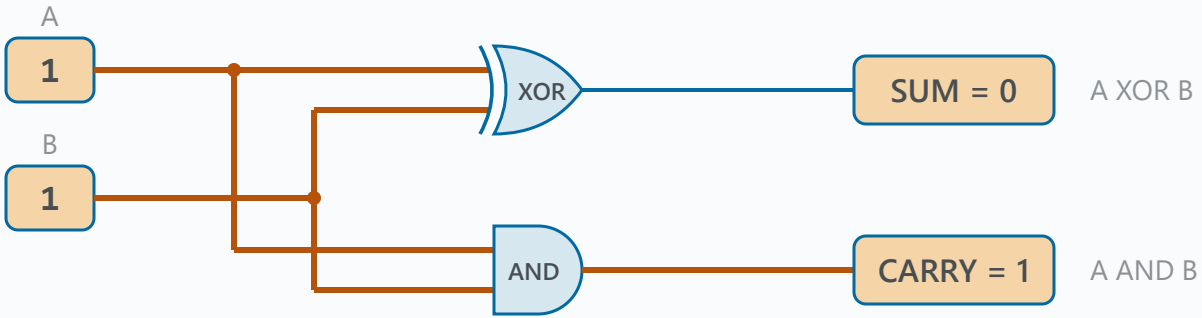
$0 + 1 = 01_2 = 1$
(CARRY, SUM) sırasıyla yazılır

0 + 1: XOR(0, 1) = 1, AND(0, 1) = 0 → **01_2 = 1**. Elde oluşmadı.

Adım 3 / 4

Yarım toplayıcı gelen eldeyi almaz. Çok bitli toplama için üç girişli **tam toplayıcı** gerekir (iki yarım toplayıcı ve bir OR kapısı).

Yarım Toplayıcı: Kapılardan Toplamaya



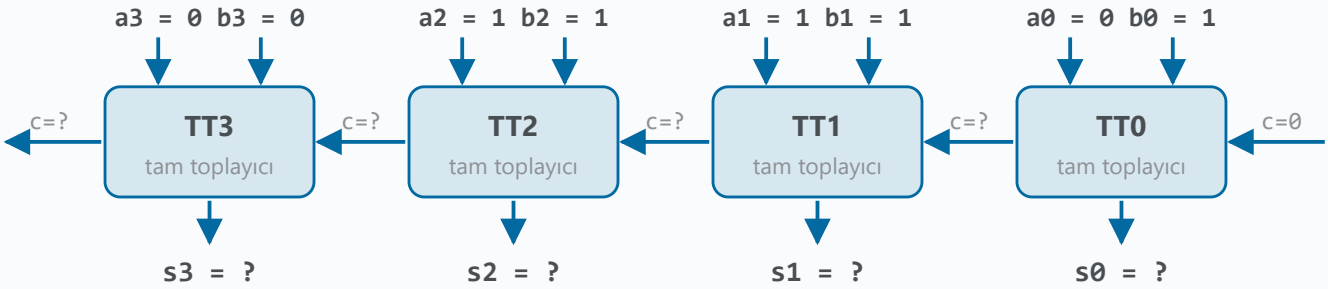
$1 + 1 = 10_2 = 2$

(CARRY, SUM) sırasıyla yazılır

1 + 1: XOR(1, 1) = 0 (basamak 0), AND(1, 1) = 1 (elde 1) → **10_2 = 2**. Onluk sistemdeki 9 + 1 = 10 gibi, elde bir üst basamağa gider.

Yarım toplayıcı gelen eldeyi almaz. Çok bitli toplama için üç girişli **tam toplayıcı** gerekir (iki yarım toplayıcı ve bir OR kapısı).

Tam Toplayıcı Zinciri: 0110 + 0111



Dört tam toplayıcı (TT) zincir hâlinde bağlıdır: her birinin elde çıkışı soldakinin elde girişidir. 6 + 7 hesaplanacak.

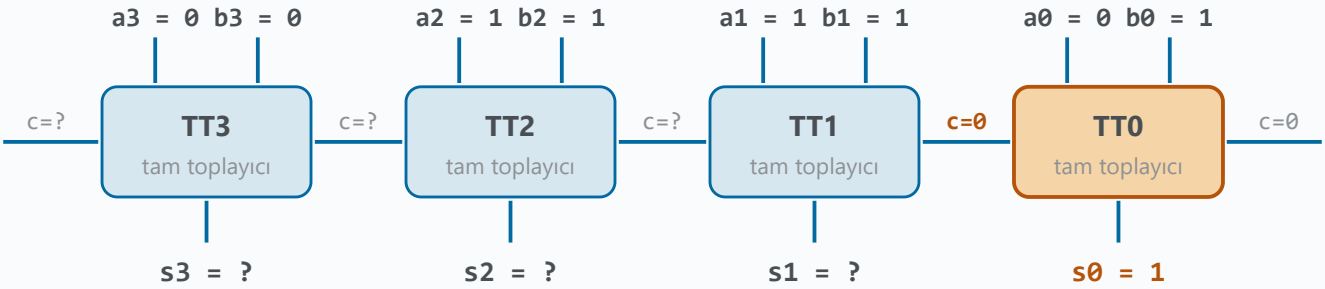
Adım 1 / 6

```
a ← 0110   ; 6
b ← 0111   ; 7
c ← 0      ; ilk elde
i = 0, 1, 2, 3 için: ; sağdan sola
  t ← a[i] XOR b[i]
  s[i] ← t XOR c
  c ← (a[i] AND b[i]) OR (t AND c)
yaz(c, s)
```

DEĞİŞKENLER

`i = –` `t = –` `c = 0` `s = ????`

Tam Toplayıcı Zinciri: 0110 + 0111



i = 0: a0 = 0, b0 = 1, c = 0 → t = 1, **s0 = 1**; iki giriş birden 1 olmadığı için elde **0**.

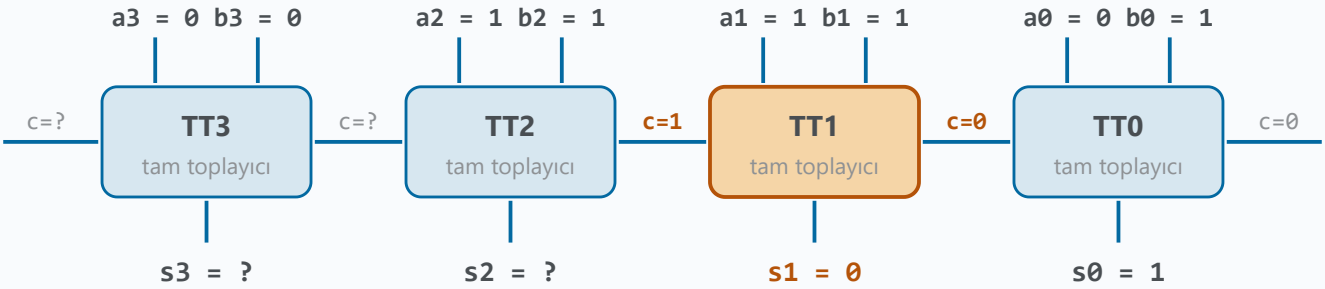
Adım 2 / 6

```
1 a ← 0110   ; 6
2 b ← 0111   ; 7
3 c ← 0      ; ilk elde
▶ i = 0, 1, 2, 3 için: ; sağdan sola
▶   t ← a[i] XOR b[i]
▶   s[i] ← t XOR c
▶   c ← (a[i] AND b[i]) OR (t AND c)
8 yaz(c, s)
```

DEĞİŞKENLER

`i = 0` `t = 1` `c = 0` `s = ???1`

Tam Toplayıcı Zinciri: 0110 + 0111



i = 1: 1 + 1 + 0 → t = 0, **s1 = 0**, a1 AND b1 = 1 olduğundan elde **1** (onluktaki 5 + 5 gibi).

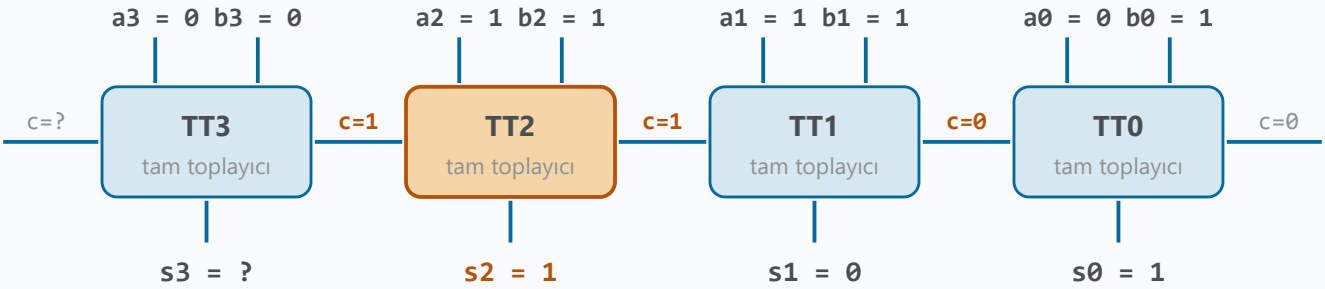
Adım 3 / 6

```
1 a ← 0110   ; 6
2 b ← 0111   ; 7
3 c ← 0      ; ilk elde
▶ i = 0, 1, 2, 3 için: ; sağdan sola
▶   t ← a[i] XOR b[i]
▶   s[i] ← t XOR c
▶   c ← (a[i] AND b[i]) OR (t AND c)
8 yaz(c, s)
```

DEĞİŞKENLER

`i = 1` `t = 0` `c = 1` `s = ??01`

Tam Toplayıcı Zinciri: 0110 + 0111



i = 2: 1 + 1 + 1 = 3 = 11_2 → **s2 = 1**, elde yine **1**.

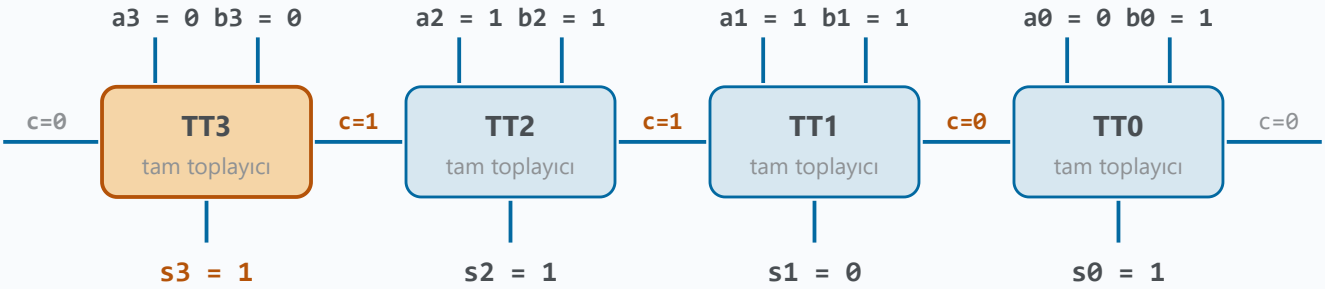
Adım 4 / 6

```
1 a ← 0110  ; 6
2 b ← 0111  ; 7
3 c ← 0     ; ilk elde
▶ i = 0, 1, 2, 3 için: ; sağdan sola
▶   t ← a[i] XOR b[i]
▶   s[i] ← t XOR c
▶   c ← (a[i] AND b[i]) OR (t AND c)
8 yaz(c, s)
```

DEĞİŞKENLER

`i = 2` `t = 0` `c = 1` `s = ?101`

Tam Toplayıcı Zinciri: 0110 + 0111



i = 3: 0 + 0 + 1 → **s3 = 1**, elde 0. Son elde 0 olduğu için sonuç 4 bite sığdı.

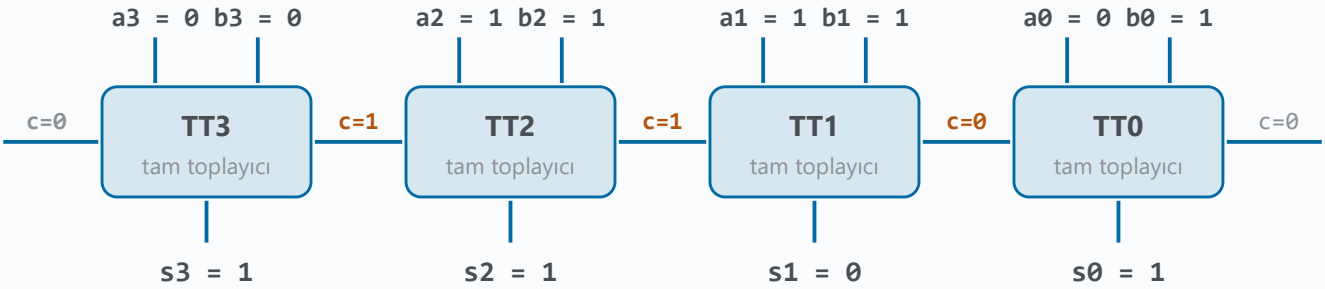
Adım 5 / 6

```
1 a ← 0110  ; 6
2 b ← 0111  ; 7
3 c ← 0     ; ilk elde
▶ i = 0, 1, 2, 3 için: ; sağdan sola
▶   t ← a[i] XOR b[i]
▶   s[i] ← t XOR c
▶   c ← (a[i] AND b[i]) OR (t AND c)
8 yaz(c, s)
```

DEĞİŞKENLER

`i = 3` `t = 0` `c = 0` `s = 1101`

Tam Toplayıcı Zinciri: 0110 + 0111



Sonuç **1101_2 = 13** = 6 + 7 ✓. Elde sağdan sola dalga gibi ilerler (**dalgalı elde toplayıcı**): her TT bir öncekini bekler, bu yüzden bit sayısı arttıkça gecikme büyür.

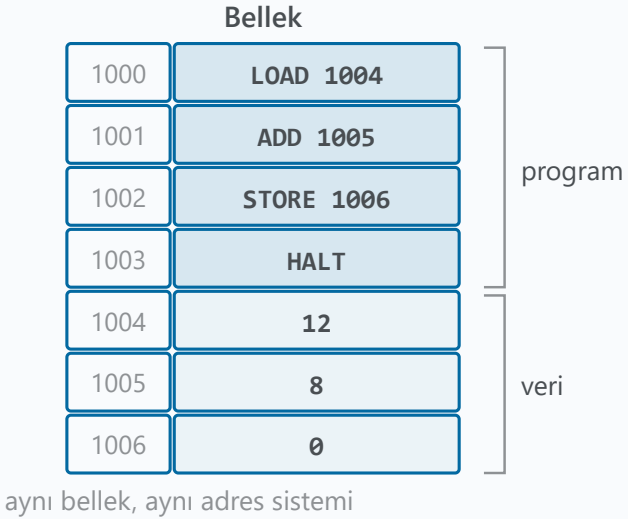
Adım 6 / 6

```
1 a ← 0110  ; 6
2 b ← 0111  ; 7
3 c ← 0     ; ilk elde
4 i = 0, 1, 2, 3 için: ; sağdan sola
5   t ← a[i] XOR b[i]
6   s[i] ← t XOR c
7   c ← (a[i] AND b[i]) OR (t AND c)
▶ yaz(c, s)
```

DEĞİŞKENLER

`i = –` `t = –` `c = 0` `s = 1101`

Depolanan Program Kavramı



Program komutları ve işlenecek veriler **aynı bellekte** sayısal kodlar olarak saklanır. İşlemci komutları sırayla okuyup yürütür.

Esneklik

Aynı donanım farklı programlarla farklı görevler yapar.

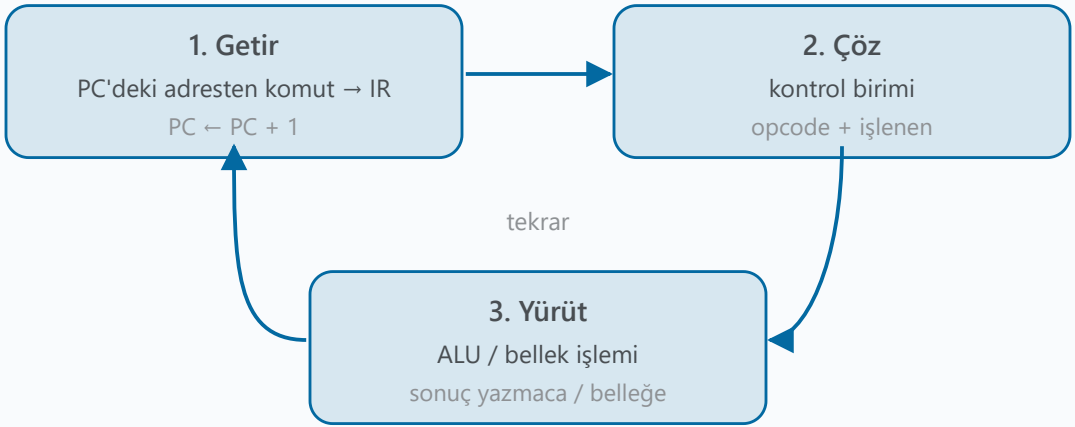
Güncelleme

Davranış donanımı değiştirmeden yazılımla değişebilir.

Genel amaçlılık

Tek sistem hesaplama, iletişim ve medya görevlerini yürütebilir.

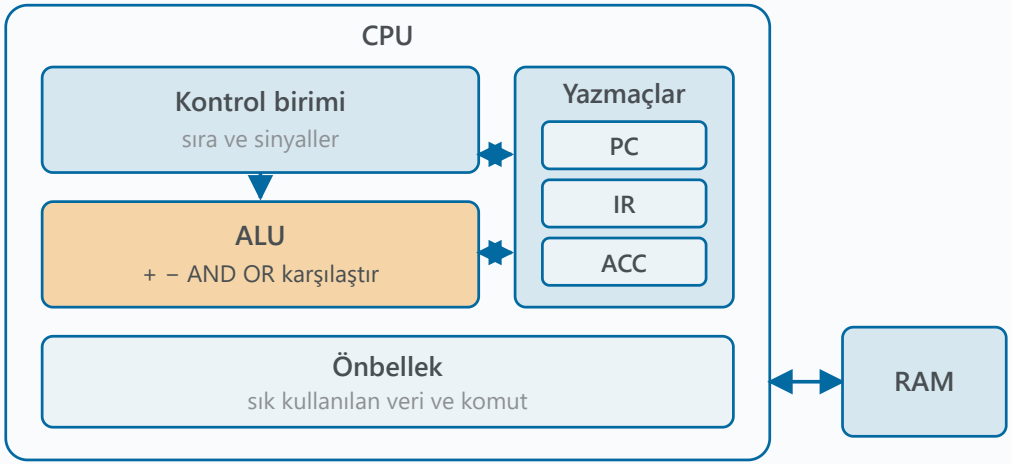
Getir–Çöz–Yürüt Döngüsü



- **Program sayacı (PC)** sıradaki komutun adresini, **komut yazmacı (IR)** getirilen komutu tutar.
- Kontrol birimi işlem kodunu (opcode) yorumlar.
- ALU veya başka bir birim işlemi gerçekleştirir.
- Komut ve veri çoğu zaman RAM yerine işlemcideki **önbellekten** gelir; yazmaç ve önbellek RAM'den çok daha hızlıdır.

BİLGİSAYAR MİMARİSİ

CPU'nun Temel Bölümleri



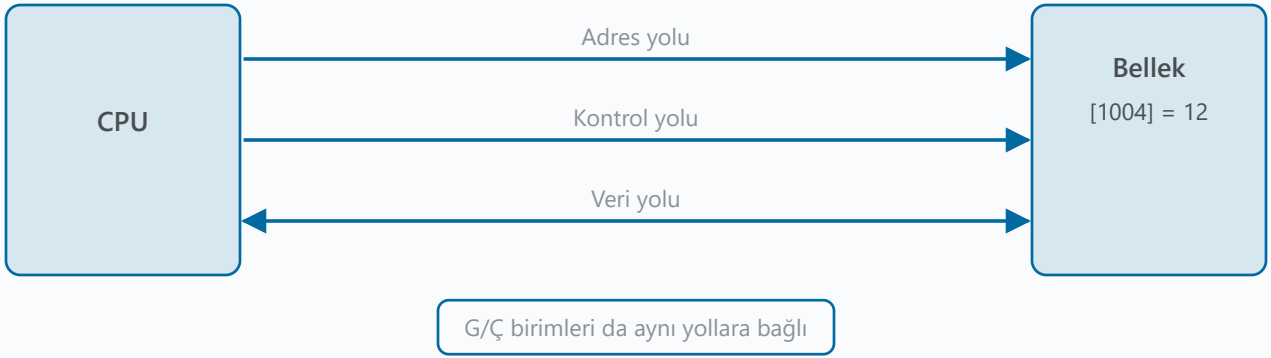
Kontrol birimi
Komutların sırasını ve veri hareketini yönetir.

ALU
Aritmetik ve mantıksal işlemleri yapar.

Yazmaçlar
İşlem sırasında kullanılan küçük ve hızlı depolama alanlarıdır.

Önbellek
Sık kullanılan veri ve komutları işlemciye yakın tutar.

Veri Yolları: Bellekten Okuma

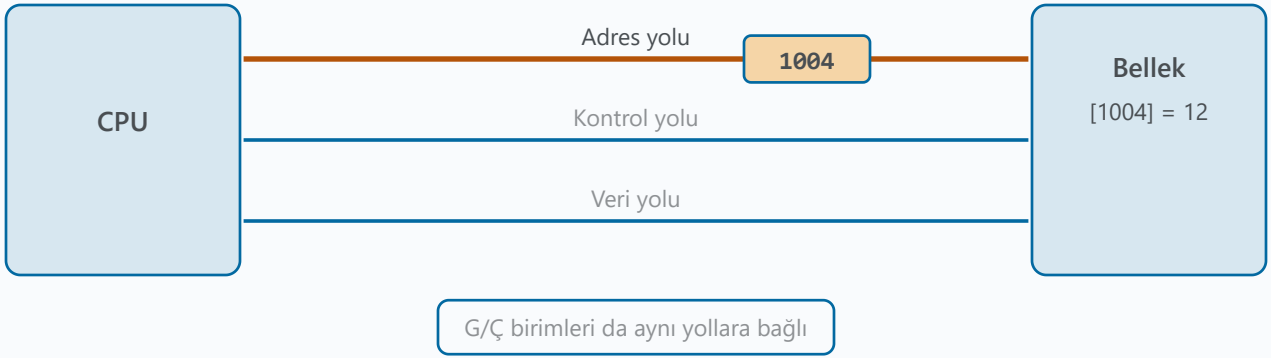


Üç yol CPU'yu bellek ve G/Ç birimlerine bağlar. Örnek: CPU, 1004 adresindeki değeri okuyacak.

Adım 1 / 4

Yol	Görev
Veri yolu	CPU, bellek ve aygıtlar arasında veriyi taşır.
Adres yolu	İşlem yapılacak bellek veya aygıt konumunu belirtir.
Kontrol yolu	Okuma, yazma ve zamanlama sinyallerini taşır.

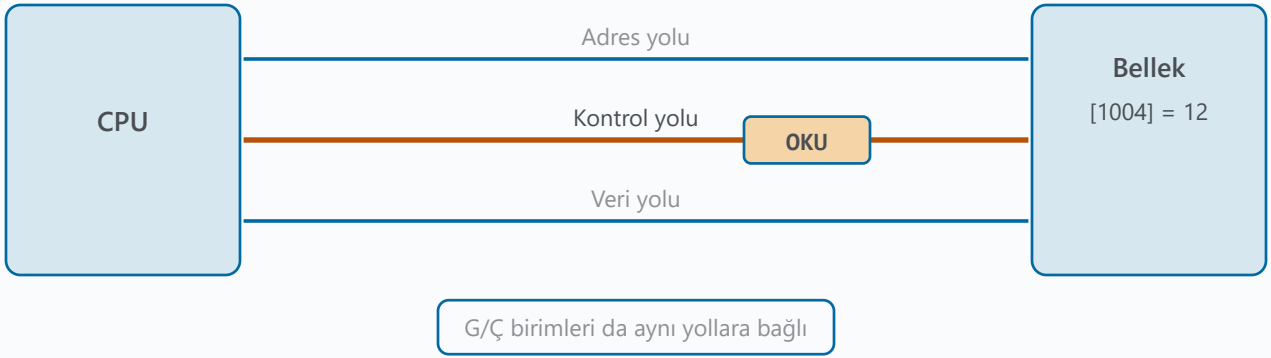
Veri Yolları: Bellekten Okuma



Adres yolu: CPU hangi hücreyi istediğini gönderir → **1004**. Tek yönlüdür (CPU'dan çıkar).

Yol	Görev
Veri yolu	CPU, bellek ve aygıtlar arasında veriyi taşır.
Adres yolu	İşlem yapılacak bellek veya aygıt konumunu belirtir.
Kontrol yolu	Okuma, yazma ve zamanlama sinyallerini taşır.

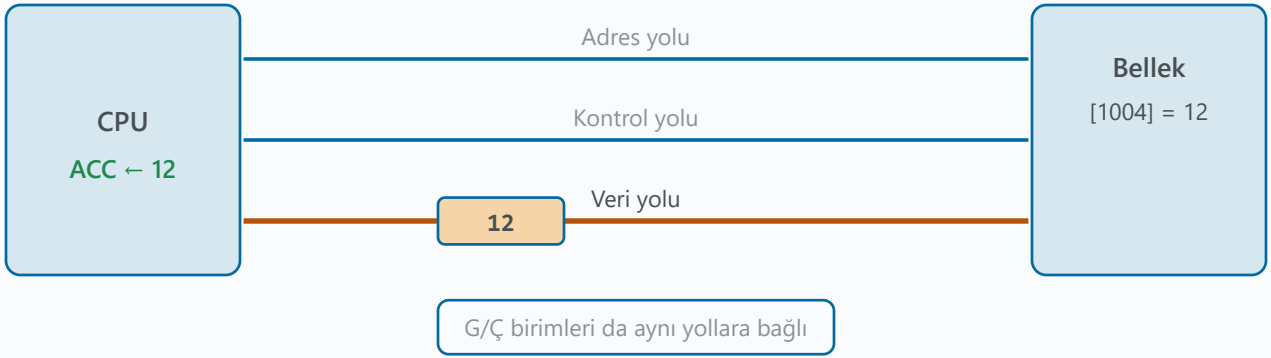
Veri Yolları: Bellekten Okuma



Kontrol yolu: CPU **OKU** sinyalini gönderir (yazma olsaydı YAZ). Zamanlama sinyalleri de bu yoldadır.

Yol	Görev
Veri yolu	CPU, bellek ve aygıtlar arasında veriyi taşır.
Adres yolu	İşlem yapılacak bellek veya aygıt konumunu belirtir.
Kontrol yolu	Okuma, yazma ve zamanlama sinyallerini taşır.

Veri Yolları: Bellekten Okuma



Veri yolu: bellek hücrenin içeriğini (**12**) CPU'ya gönderir. Veri yolu çift yönlüdür.

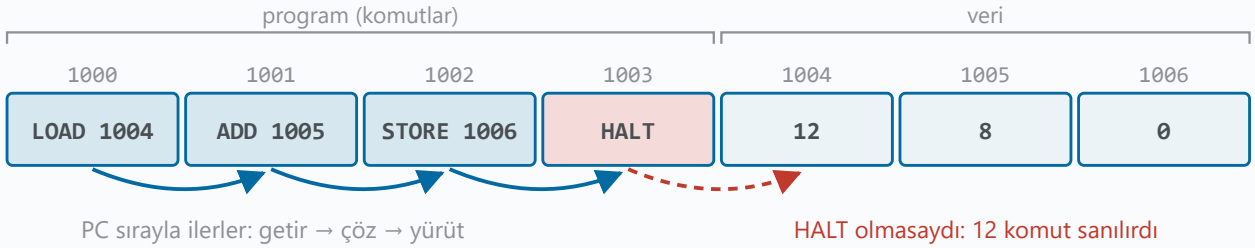
Adım 4 / 4

Yol	Görev
Veri yolu	CPU, bellek ve aygıtlar arasında veriyi taşır.
Adres yolu	İşlem yapılacak bellek veya aygıt konumunu belirtir.
Kontrol yolu	Okuma, yazma ve zamanlama sinyallerini taşır.

Bellek Adresleri

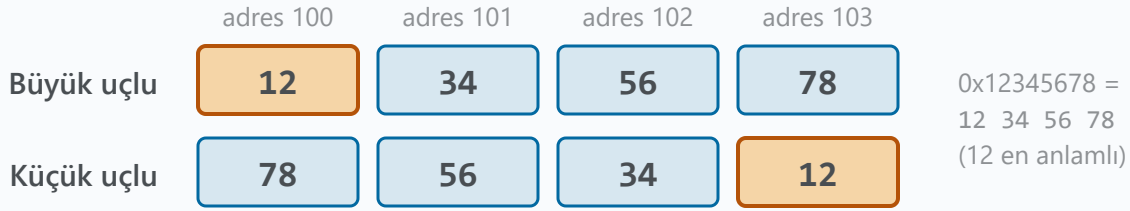
Adres	İçerik
1000	LOAD 1004
1001	ADD 1005
1002	STORE 1006
1003	HALT
1004	12
1005	8
1006	0

- Her bellek hücresinin bir adresi vardır.
- Komutlar adres kullanarak veriye ulaşır.
- Aynı bit dizisi komut veya veri olarak yorumlanabilir.
- **HALT** olmasaydı CPU 1004'teki 12'yi de komut sanıp çalıştırmaya çalışırdı.



Çok Baytlı Değerler ve Bayt Sırası

Bellek **bayt adreslidir**: her adres 1 bayt tutar. 32 bitlik bir `int` 4 ardışık adrese yayılır; baytların hangi sırayla yazılacağına **bayt sırası** denir.



Büyük uçlu

En anlamlı bayt en küçük adreste. Ağ protokolleri bu sırayı kullanır (ağ bayt sırası).

Küçük uçlu

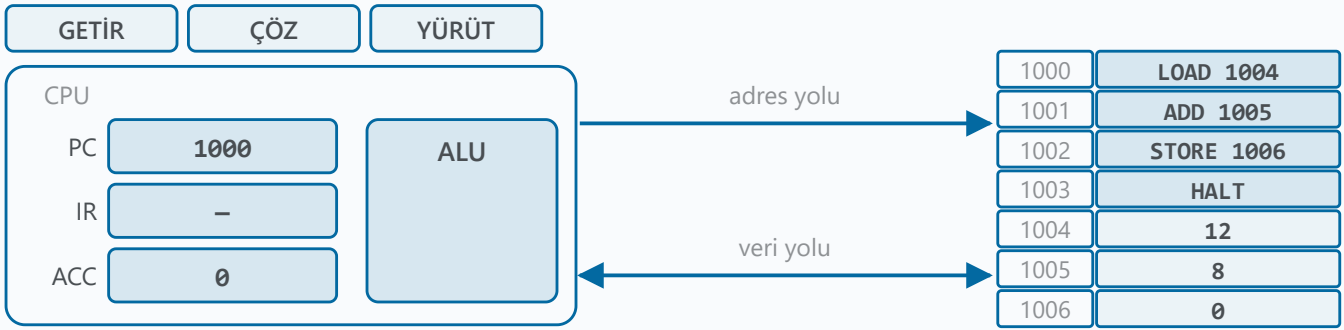
En az anlamlı bayt en küçük adreste. x86-64 ve yaygın ARM sistemleri bu sırayla çalışır.

Sorun

Ham ikili veri ters sıralı makinede okunursa 0x12345678, 0x78563412 olarak görülür.

Tek bayt (ör. ASCII karakter) için sıra sorunu yoktur; UTF-8 de bayt bayt okunduğu için bayt sırasından etkilenmez.

Programı Adım Adım Çalıştırma



Başlangıç: **PC = 1000** (ilk komutun adresi), ACC = 0. CPU döngüye başlar.

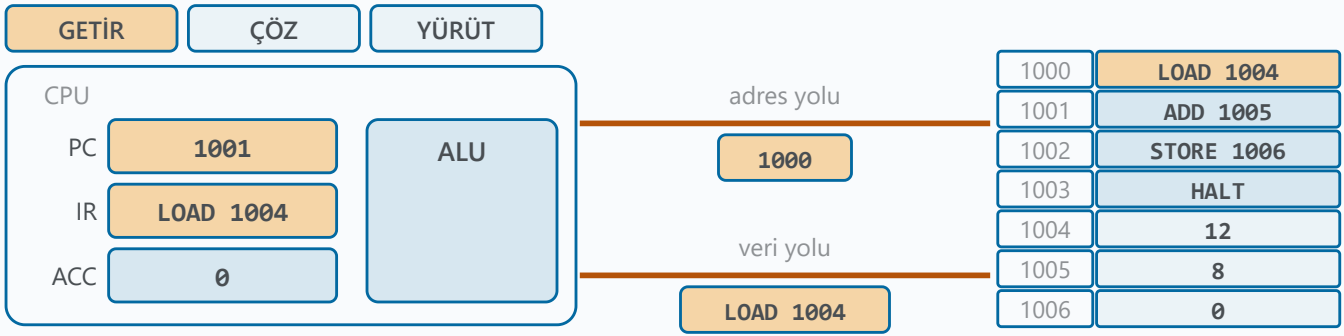
Adım 1 / 12

```
▶ 1000: LOAD  1004 ; ACC ← bellek[1004]
  1001: ADD   1005 ; ACC ← ACC + bellek[1005]
  1002: STORE 1006 ; bellek[1006] ← ACC
  1003: HALT       ; dur
  1004: 12         ; veri
  1005: 8          ; veri
  1006: 0          ; sonuç
```

DEĞİŞKENLER

PC = 1000 | IR = – | ACC = 0 | bellek[1006] = 0

Programı Adım Adım Çalıştırma



Getir: PC'deki 1000 adres yoluna çıkar, komut `LOAD 1004` veri yoluyla **IR**'ye gelir. PC bir artar → **1001**.

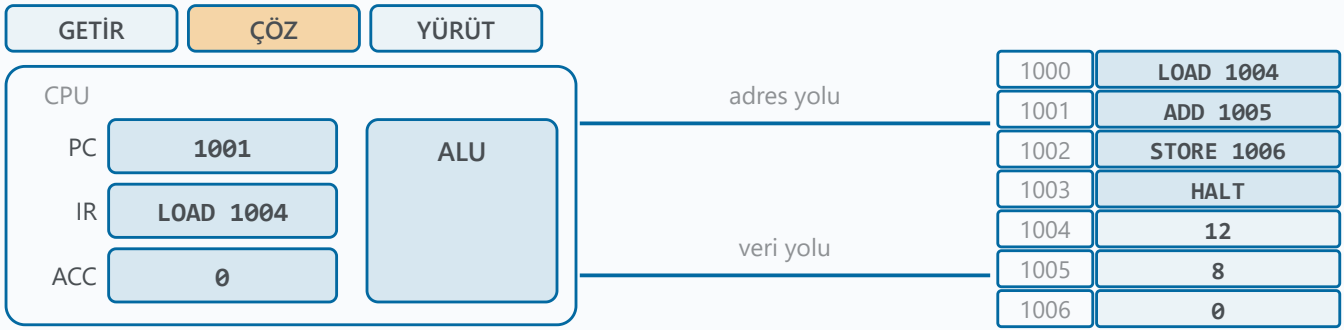
Adım 2 / 12

```
▶ 1000: LOAD  1004 ; ACC ← bellek[1004]
  1001: ADD   1005 ; ACC ← ACC + bellek[1005]
  1002: STORE 1006 ; bellek[1006] ← ACC
  1003: HALT       ; dur
  1004: 12         ; veri
  1005: 8          ; veri
  1006: 0          ; sonuç
```

DEĞİŞKENLER

`PC = 1001` `IR = LOAD 1004` `ACC = 0` `bellek[1006] = 0`

Programı Adım Adım Çalıştırma



Çöz: kontrol birimi IR'yi ayırır: opcode **LOAD**, adres **1004** → bellek[1004] okunup ACC'ye yüklenecek.

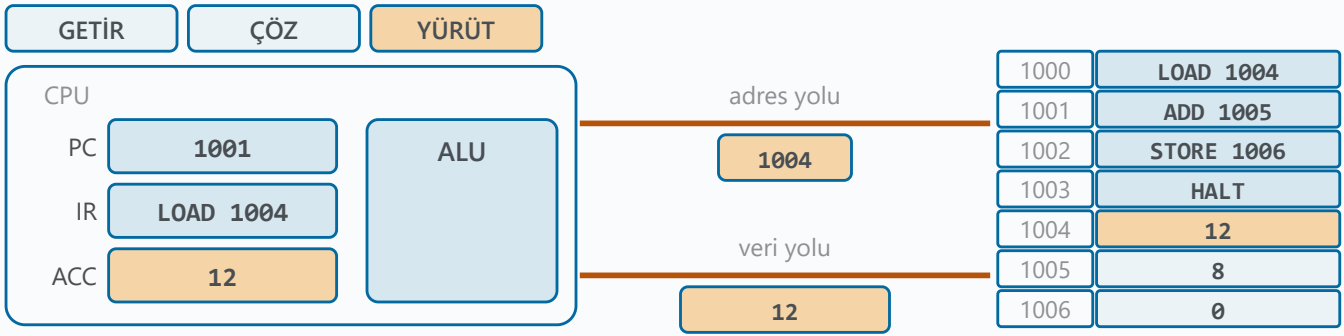
Adım 3 / 12

```
▶ 1000: LOAD  1004 ; ACC ← bellek[1004]
  1001: ADD   1005 ; ACC ← ACC + bellek[1005]
  1002: STORE 1006 ; bellek[1006] ← ACC
  1003: HALT       ; dur
  1004: 12         ; veri
  1005: 8          ; veri
  1006: 0          ; sonuç
```

DEĞİŞKENLER

`PC = 1001` `IR = LOAD 1004` `ACC = 0` `bellek[1006] = 0`

Programı Adım Adım Çalıştırma



Yürüt: 1004 adresi okunur, **12** ACC'ye yüklenir.

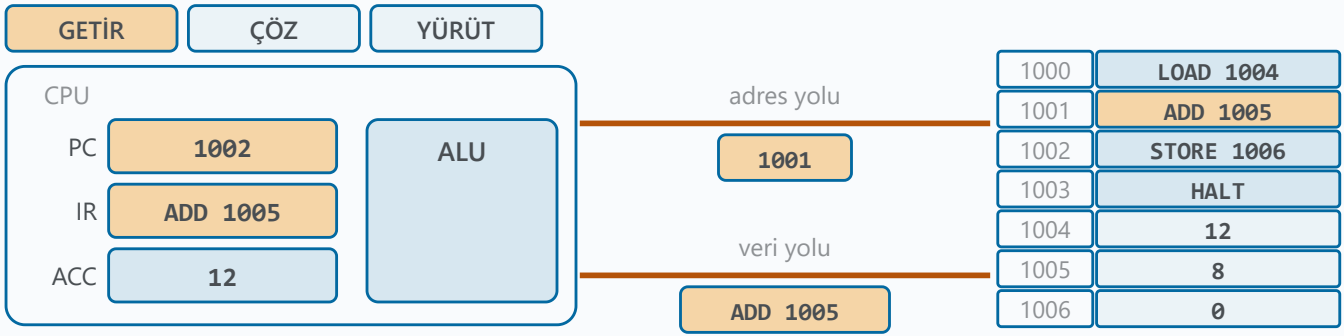
Adım 4 / 12

```
▶ 1000: LOAD  1004 ; ACC ← bellek[1004]
  1001: ADD   1005 ; ACC ← ACC + bellek[1005]
  1002: STORE 1006 ; bellek[1006] ← ACC
  1003: HALT       ; dur
▶ 1004: 12         ; veri
  1005: 8          ; veri
  1006: 0          ; sonuç
```

DEĞİŞKENLER

`PC = 1001` `IR = LOAD 1004` `ACC = 12` `bellek[1006] = 0`

Programı Adım Adım Çalıştırma



Getir: PC'deki 1001 adres yoluna çıkar, komut `ADD 1005` veri yoluyla **IR**'ye gelir. PC bir artar → **1002**.

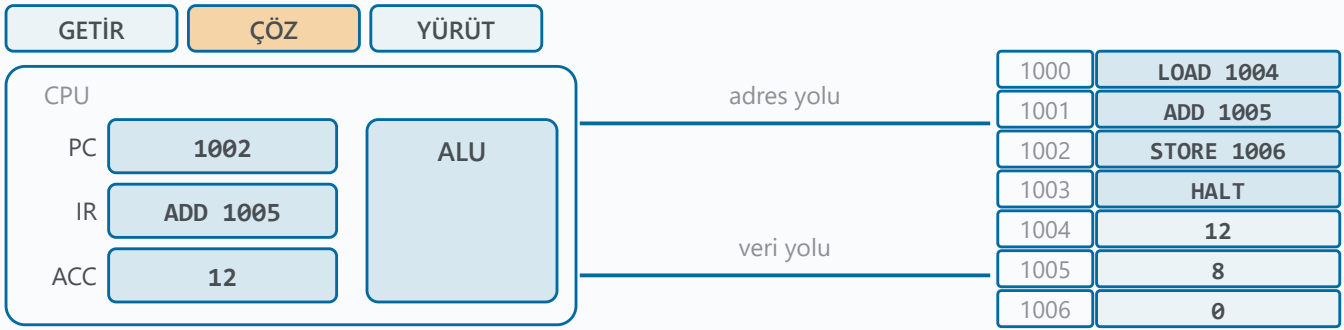
Adım 5 / 12

```
1 1000: LOAD  1004 ; ACC ← bellek[1004]
▶ 1001: ADD   1005 ; ACC ← ACC + bellek[1005]
3 1002: STORE 1006 ; bellek[1006] ← ACC
4 1003: HALT       ; dur
5 1004: 12         ; veri
6 1005: 8          ; veri
7 1006: 0          ; sonuç
```

DEĞİŞKENLER

`PC = 1002` `IR = ADD 1005` `ACC = 12` `bellek[1006] = 0`

Programı Adım Adım Çalıştırma



Çöz: kontrol birimi IR'yi ayırır: opcode **ADD**, adres **1005** → bellek[1005] ACC'ye eklenecek.

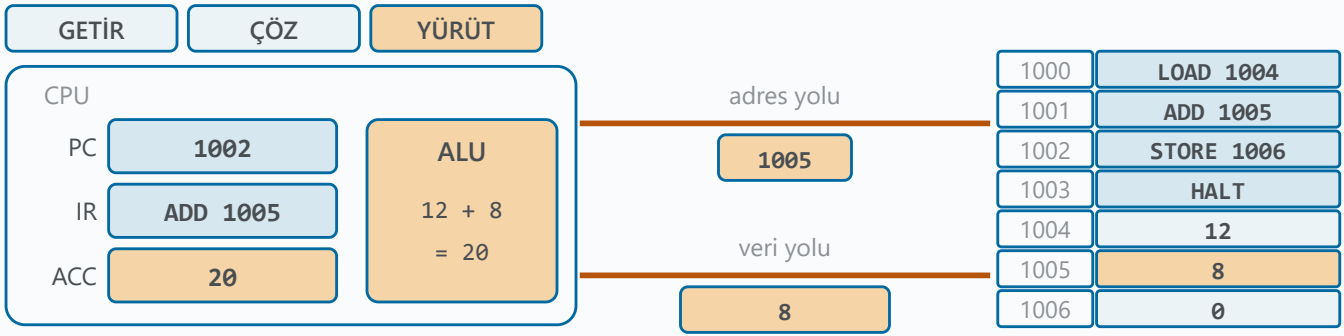
Adım 6 / 12

```
1 1000: LOAD  1004 ; ACC ← bellek[1004]
▶ 1001: ADD   1005 ; ACC ← ACC + bellek[1005]
3 1002: STORE 1006 ; bellek[1006] ← ACC
4 1003: HALT       ; dur
5 1004: 12         ; veri
6 1005: 8          ; veri
7 1006: 0          ; sonuç
```

DEĞİŞKENLER

`PC = 1002` `IR = ADD 1005` `ACC = 12` `bellek[1006] = 0`

Programı Adım Adım Çalıştırma



Yürüt: 1005 adresindeki 8 okunur, **ALU** 12 + 8 = **20** hesaplar, sonuç ACC'ye yazılır.

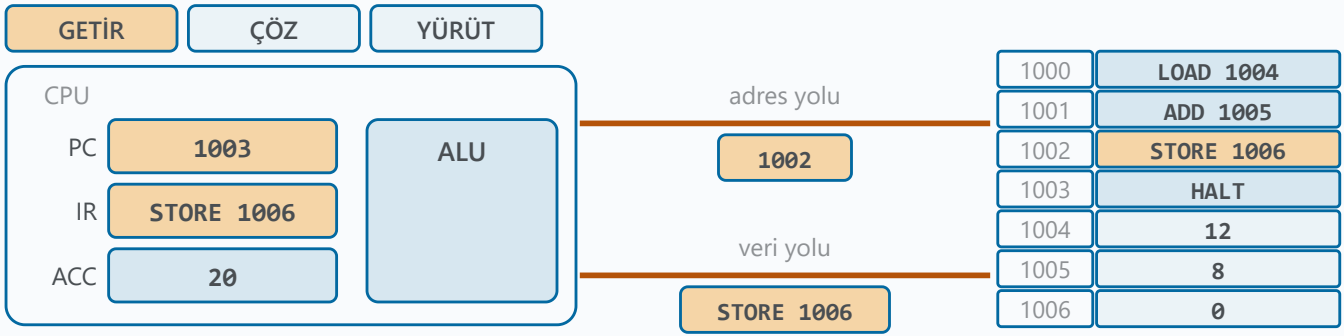
Adım 7 / 12

```
1 1000: LOAD  1004 ; ACC ← bellek[1004]
▶ 1001: ADD   1005 ; ACC ← ACC + bellek[1005]
3 1002: STORE 1006 ; bellek[1006] ← ACC
4 1003: HALT       ; dur
5 1004: 12         ; veri
▶ 1005: 8          ; veri
7 1006: 0          ; sonuç
```

DEĞİŞKENLER

PC = **1002** · IR = **ADD 1005** · ACC = **20** · bellek[1006] = **0**

Programı Adım Adım Çalıştırma



Getir: PC'deki 1002 adres yoluna çıkar, komut `STORE 1006` veri yoluyla **IR**'ye gelir. PC bir artar → **1003**.

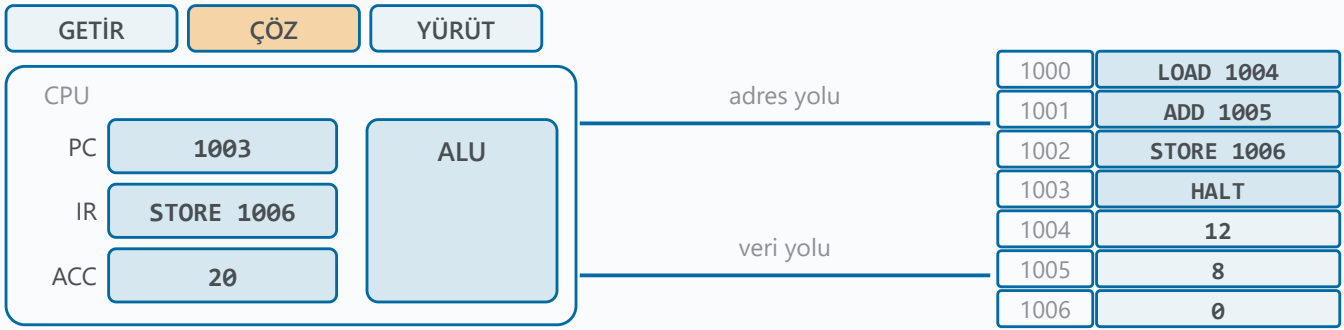
Adım 8 / 12

```
1 1000: LOAD  1004 ; ACC ← bellek[1004]
2 1001: ADD   1005 ; ACC ← ACC + bellek[1005]
▶ 1002: STORE 1006 ; bellek[1006] ← ACC
4 1003: HALT       ; dur
5 1004: 12         ; veri
6 1005: 8          ; veri
7 1006: 0          ; sonuç
```

DEĞİŞKENLER

`PC = 1003` `IR = STORE 1006` `ACC = 20` `bellek[1006] = 0`

Programı Adım Adım Çalıştırma



Çöz: kontrol birimi IR'yi ayırır: opcode **STORE**, adres **1006** → ACC bellek[1006]'e yazılacak.

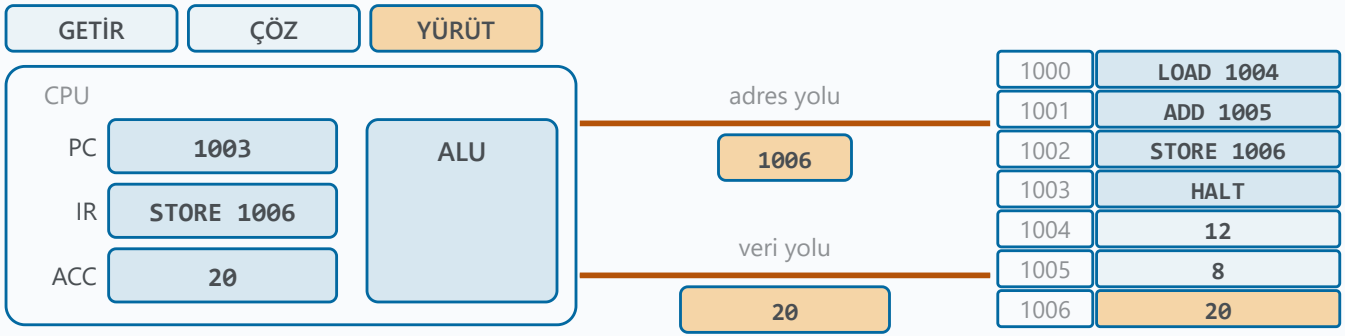
Adım 9 / 12

```
1 1000: LOAD  1004 ; ACC ← bellek[1004]
2 1001: ADD   1005 ; ACC ← ACC + bellek[1005]
▶ 1002: STORE 1006 ; bellek[1006] ← ACC
4 1003: HALT       ; dur
5 1004: 12         ; veri
6 1005: 8          ; veri
7 1006: 0          ; sonuç
```

DEĞİŞKENLER

`PC = 1003` `IR = STORE 1006` `ACC = 20` `bellek[1006] = 0`

Programı Adım Adım Çalıştırma



Yürüt: ACC'deki **20** veri yoluyla 1006 adresine **yazılır**.

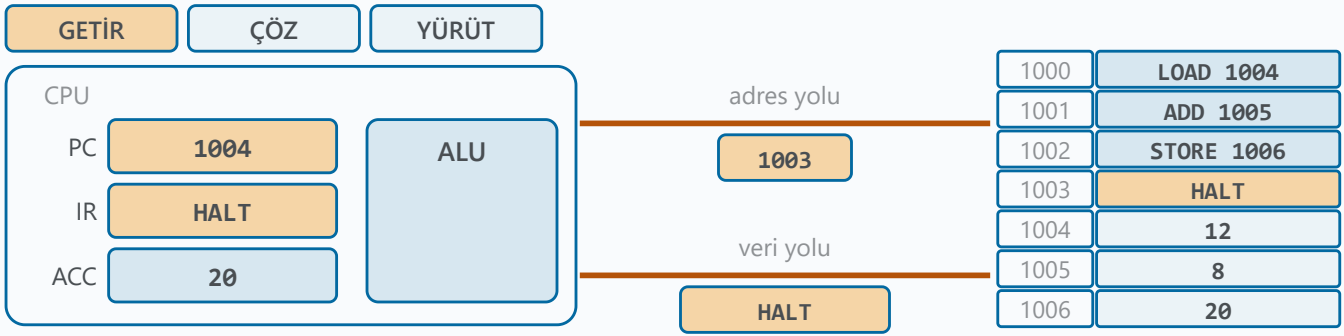
Adım 10 / 12

```
1 1000: LOAD  1004 ; ACC ← bellek[1004]
2 1001: ADD   1005 ; ACC ← ACC + bellek[1005]
▶ 1002: STORE 1006 ; bellek[1006] ← ACC
4 1003: HALT       ; dur
5 1004: 12         ; veri
6 1005: 8          ; veri
▶ 1006: 0          ; sonuç
```

DEĞİŞKENLER

`PC = 1003` `IR = STORE 1006` `ACC = 20` `bellek[1006] = 20`

Programı Adım Adım Çalıştırma



Getir: PC'deki 1003 adres yoluna çıkar, komut `HALT` veri yoluyla **IR**'ye gelir. PC bir artar → **1004**.

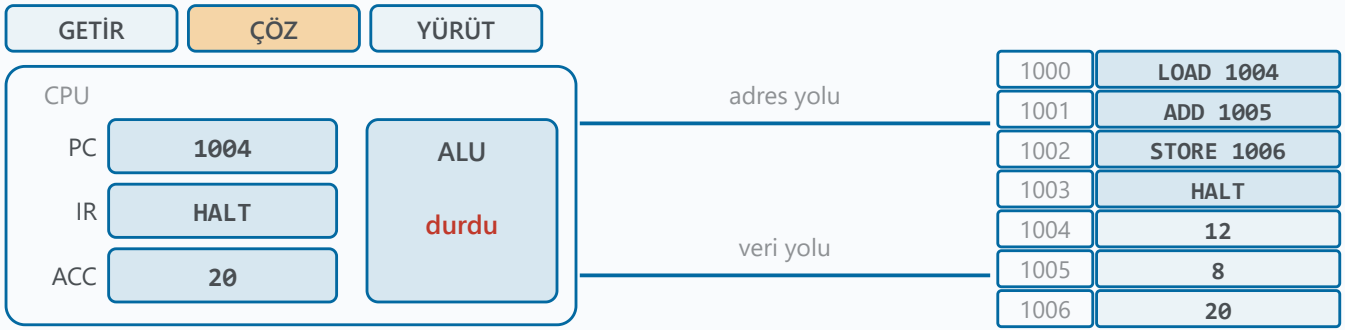
Adım 11 / 12

```
1 1000: LOAD  1004 ; ACC ← bellek[1004]
2 1001: ADD   1005 ; ACC ← ACC + bellek[1005]
3 1002: STORE 1006 ; bellek[1006] ← ACC
▶ 1003: HALT       ; dur
5 1004: 12         ; veri
6 1005: 8          ; veri
7 1006: 0          ; sonuç
```

DEĞİŞKENLER

`PC = 1004` `IR = HALT` `ACC = 20` `bellek[1006] = 20`

Programı Adım Adım Çalıştırma



Çöz: opcode HALT → CPU durur. Sonuç bellek[1006] = **20**. 4 komut, 4 getir–çöz–yürüt turu.

Adım 12 / 12

```
1 1000: LOAD  1004 ; ACC ← bellek[1004]
2 1001: ADD   1005 ; ACC ← ACC + bellek[1005]
3 1002: STORE 1006 ; bellek[1006] ← ACC
▶ 1003: HALT       ; dur
5 1004: 12         ; veri
6 1005: 8          ; veri
7 1006: 0          ; sonuç
```

DEĞİŞKENLER

`PC = 1004` `IR = HALT` `ACC = 20` `bellek[1006] = 20`

Dallanma: Döngüde PC Nasıl Değişir?

```
1  1000: LOAD  1010  ; ACC ← sayaç
2  1001: JUMPZ 1008  ; ACC = 0 ise atla
3  1002: ADD   1011  ; ACC ← ACC + toplam
4  1003: STORE 1011  ; toplam ← ACC
5  1004: LOAD  1010
6  1005: SUB   1012  ; ACC ← ACC − 1
7  1006: STORE 1010  ; sayaç ← ACC
8  1007: JUMP  1000  ; PC ← 1000
9  1008: HALT
10 1010: 2   1011: 0   1012: 1
```



sayaç 2'den geriye sayar; döngü bitince toplam = 2 + 1 olacak.

Adım 1 / 8

DEĞİŞKENLER

`PC = 1000` `ACC = 0` `sayaç = 2` `toplam = 0` `komut = 0`

Dallanma: Döngüde PC Nasıl Değişir?

```
▶ 1000: LOAD  1010  ; ACC ← sayaç
▶ 1001: JUMPZ 1008  ; ACC = 0 ise atla
3 1002: ADD   1011  ; ACC ← ACC + toplam
4 1003: STORE 1011  ; toplam ← ACC
5 1004: LOAD  1010
6 1005: SUB   1012  ; ACC ← ACC − 1
7 1006: STORE 1010  ; sayaç ← ACC
8 1007: JUMP  1000  ; PC ← 1000
9 1008: HALT
10 1010: 2   1011: 0   1012: 1
```



LOAD → ACC = 2. **JUMPZ**: ACC ≠ 0, atlama yok; PC sıradaki 1002'de kalır.

Adım 2 / 8

DEĞİŞKENLER

`PC = 1002` `ACC = 2` `sayaç = 2` `toplam = 0` `komut = 2`

Dallanma: Döngüde PC Nasıl Değişir?

```
1  1000: LOAD   1010  ; ACC ← sayaç
2  1001: JUMPZ  1008  ; ACC = 0 ise atla
▶  1002: ADD    1011  ; ACC ← ACC + toplam
▶  1003: STORE  1011  ; toplam ← ACC
5  1004: LOAD   1010
6  1005: SUB    1012  ; ACC ← ACC − 1
7  1006: STORE  1010  ; sayaç ← ACC
8  1007: JUMP   1000  ; PC ← 1000
9  1008: HALT
10 1010: 2   1011: 0   1012: 1
```



ADD: 2 + 0 = 2; STORE → toplam = **2**.

Adım 3 / 8

DEĞİŞKENLER

`PC = 1004` `ACC = 2` `sayaç = 2` `toplam = 2` `komut = 4`

Dallanma: Döngüde PC Nasıl Değişir?

```
1  1000: LOAD  1010  ; ACC ← sayaç
2  1001: JUMPZ 1008  ; ACC = 0 ise atla
3  1002: ADD   1011  ; ACC ← ACC + toplam
4  1003: STORE 1011  ; toplam ← ACC
▶  1004: LOAD  1010
▶  1005: SUB   1012  ; ACC ← ACC – 1
▶  1006: STORE 1010  ; sayaç ← ACC
8  1007: JUMP  1000  ; PC ← 1000
9  1008: HALT
10 1010: 2   1011: 0   1012: 1
```



LOAD, SUB, STORE: sayaç 2 − 1 = **1**.

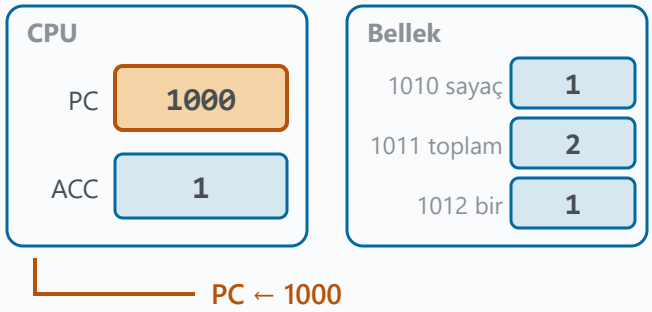
Adım 4 / 8

DEĞİŞKENLER

`PC = 1007` `ACC = 1` `sayaç = 1` `toplam = 2` `komut = 7`

Dallanma: Döngüde PC Nasıl Değişir?

```
1  1000: LOAD  1010  ; ACC ← sayaç
2  1001: JUMPZ 1008  ; ACC = 0 ise atla
3  1002: ADD   1011  ; ACC ← ACC + toplam
4  1003: STORE 1011  ; toplam ← ACC
5  1004: LOAD  1010
6  1005: SUB   1012  ; ACC ← ACC − 1
7  1006: STORE 1010  ; sayaç ← ACC
▶  1007: JUMP  1000  ; PC ← 1000
9  1008: HALT
10 1010: 2   1011: 0   1012: 1
```



JUMP 1000: getirmede 1008 olan PC'nin üzerine 1000 yazılır. Döngü başa döndü.

Adım 5 / 8

DEĞİŞKENLER

`PC = 1000` `ACC = 1` `sayaç = 1` `toplam = 2` `komut = 8`

Dallanma: Döngüde PC Nasıl Değişir?

```
1000: LOAD  1010  ; ACC ← sayaç
1001: JUMPZ 1008  ; ACC = 0 ise atla
1002: ADD   1011  ; ACC ← ACC + toplam
1003: STORE 1011  ; toplam ← ACC
1004: LOAD  1010
1005: SUB   1012  ; ACC ← ACC − 1
1006: STORE 1010  ; sayaç ← ACC
1007: JUMP  1000  ; PC ← 1000
1008: HALT
1010: 2   1011: 0   1012: 1
```



İkinci tur aynı 8 komutu yürütür: toplam = 1 + 2 = **3**, sayaç = **0**, yine JUMP.

Adım 6 / 8

DEĞİŞKENLER

`PC = 1000` `ACC = 0` `sayaç = 0` `toplam = 3` `komut = 16`

Dallanma: Döngüde PC Nasıl Değişir?

```
▶ 1000: LOAD  1010  ; ACC ← sayaç
▶ 1001: JUMPZ 1008  ; ACC = 0 ise atla
  1002: ADD   1011  ; ACC ← ACC + toplam
  1003: STORE 1011  ; toplam ← ACC
  1004: LOAD  1010
  1005: SUB   1012  ; ACC ← ACC − 1
  1006: STORE 1010  ; sayaç ← ACC
  1007: JUMP  1000  ; PC ← 1000
  1008: HALT
  1010: 2   1011: 0   1012: 1
```



LOAD → ACC = 0. **JUMPZ** bu kez atlar: PC ← 1008, döngü gövdesi atlanır.

Adım 7 / 8

DEĞİŞKENLER

`PC = 1008` `ACC = 0` `sayaç = 0` `toplam = 3` `komut = 18`

Dallanma: Döngüde PC Nasıl Değişir?

```
1  1000: LOAD   1010  ; ACC ← sayaç
2  1001: JUMPZ  1008  ; ACC = 0 ise atla
3  1002: ADD    1011  ; ACC ← ACC + toplam
4  1003: STORE  1011  ; toplam ← ACC
5  1004: LOAD   1010
6  1005: SUB    1012  ; ACC ← ACC − 1
7  1006: STORE  1010  ; sayaç ← ACC
8  1007: JUMP   1000  ; PC ← 1000
▶  1008: HALT
10 1010: 2   1011: 0   1012: 1
```



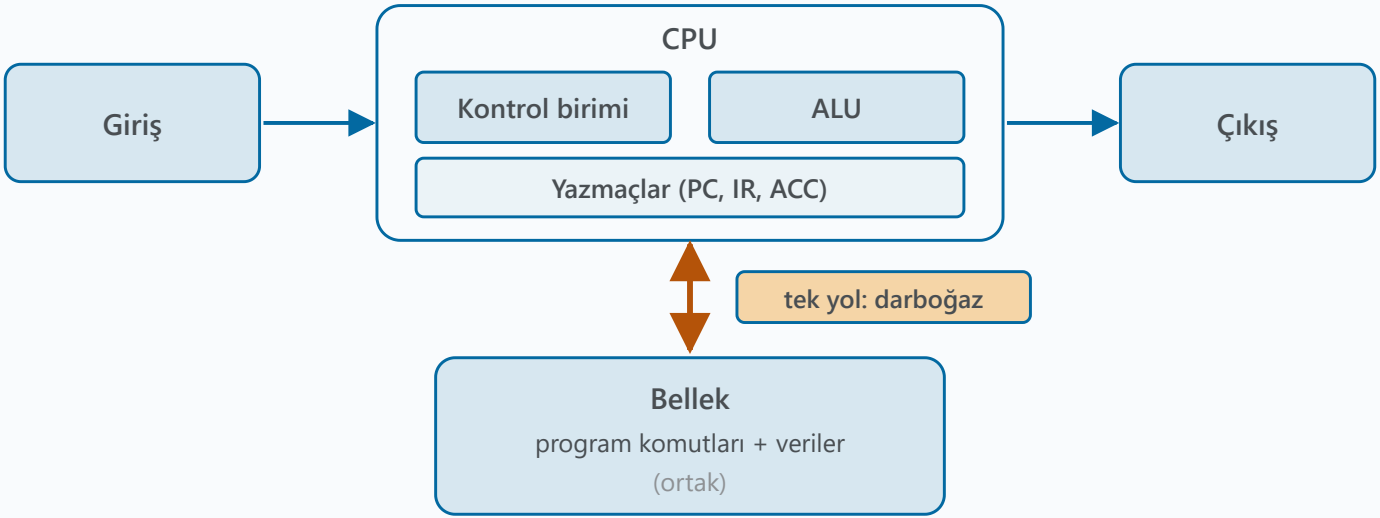
HALT: CPU durur. **19 komut** getirildi (8 + 8 + 3). Java'daki `while` döngüsü bu tür koşullu ve koşulsuz atlamalara dönüşür.

Adım 8 / 8

DEĞİŞKENLER

`PC = 1009` `ACC = 0` `sayaç = 0` `toplam = 3` `komut = 19`

von Neumann Mimarisi



- Program komutları ve veriler ortak bellekte tutulur.
- CPU komutları sırayla işler; dallanma komutları bu sırayı değiştirebilir.
- Komut ve veri aynı yoldan geldiği için CPU–bellek aktarımı performansı sınırlar (**von Neumann darboğazı**).

BİLGİSAYAR MİMARİSİ

von Neumann ve Harvard Mimarisi



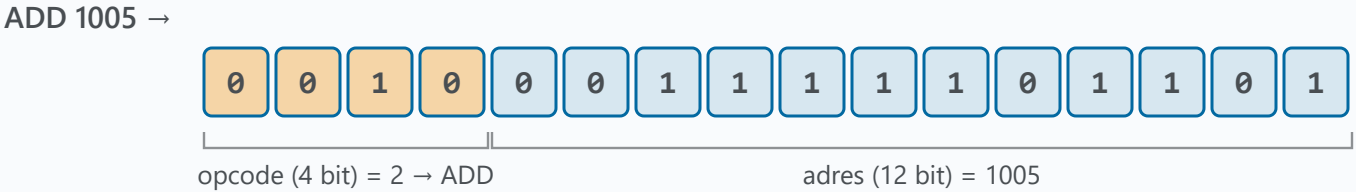
Özellik	von Neumann	Harvard
Bellek	Komut ve veri ortak bellekte	Komut ve veri ayrı belleklerde
Yol	Tek yol: komut ve veri sırayla taşınır	İki yol: komut ve veri aynı anda okunur
Esneklik	Program, veri gibi yüklenip değiştirilebilir	Program belleği çoğunlukla sabittir (flash)
Yaygın kullanım	Genel amaçlı bilgisayarlar	Mikrodenetleyiciler, sinyal işlemciler

Günümüz işlemcileri ikisini birleştirir: ana bellek ortaktır, çekirdeğe en yakın L1 önbellek ise komut ve veri için ayrıdır (değiştirilmiş Harvard).

BİLGİSAYAR MİMARİSİ

Komutların Kodlanması

Makine komutu genellikle yapılacak işlemi ve kullanılacak değer veya adresleri belirtir.



Örnek (varsayımsal) kodlar: HALT = 0000 · LOAD = 0001 · ADD = 0010 · STORE = 0011

Opcode (işlem kodu)

İşlemin türünü tanımlar.

`ADD`

İşlenen (operand)

İşlemde kullanılacak yazmaç, değer veya adres.

`R1, R2`

Komut biçimi

Bit alanlarının sayısını ve yerini mimari belirler.

Komut kümesi

İşlemcinin doğrudan anlayabildiği komutların tamamı.

Komut Kümesi Tasarımı: RISC ve CISC

Özellik	RISC	CISC
Komut kümesi	Az sayıda, basit komut	Çok sayıda, karmaşık komut
Komut uzunluğu	Sabit (ör. 32 bit)	Değişken (x86-64'te 1–15 bayt)
Bellek erişimi	Yalnızca LOAD / STORE komutları	İşlem komutları doğrudan bellek işleneni alabilir
Örnek	ARM, RISC-V	x86-64

RISC: bellekteki `a = a + b`

```
LOAD  R1, a
LOAD  R2, b
ADD   R1, R1, R2
STORE R1, a
```

CISC (x86-64): aynı iş

```
MOV EAX, [b]
ADD [a], EAX
```

4 komut yerine 2 komut; ama ikinci komut içeride okuma, toplama ve yazma adımlarına bölünür.

Mimari Sınırlar

Sonlu bellek
Aynı anda saklanabilecek veri miktarı sınırlıdır.

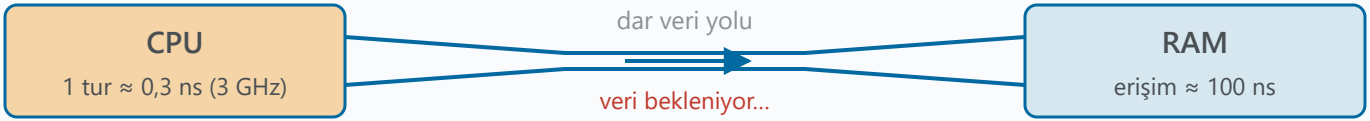
Sonlu hız
Her komut zaman ve enerji tüketir.

Temsil hataları
Sınırlı bit sayısı taşma ve yuvarlama üretebilir.

Veri darboğazı
CPU, bellekteki veriyi beklerken boş kalabilir.

Paralellik sınırı
Her problem eşit ölçüde paralelleştirilemez.

Güvenlik
Program ve veri aynı sistem kaynaklarını paylaştığı için yalıtım gerekir.



100 ns ÷ 0,3 ns ≈ 300 tur: önbellek bu bekleyişi azaltmak için vardır

BİLGİSAYAR MİMARİSİ · ÖRNEK

Performans: Komut Sayısı, CPI ve Saat Hızı

CPU süresi = komut sayısı × CPI ÷ saat hızı

CPI: komut başına ortalama saat turu sayısı

Program: 2×10^9 komut	Saat hızı	CPI	Süre
İşlemci A	3 GHz	1,5	$2 \times 10^9 \times 1{,}5 \div (3 \times 10^9)$ = **1,0 sn**
İşlemci B	4 GHz	2,5	$2 \times 10^9 \times 2{,}5 \div (4 \times 10^9)$ = **1,25 sn**
A, daha iyi derleyiciyle ($1{,}6 \times 10^9$ komut)	3 GHz	1,5	$1{,}6 \times 10^9 \times 1{,}5 \div (3 \times 10^9)$ = **0,8 sn**

Komut sayısı

Algoritma, derleyici ve komut kümesine bağlıdır.

CPI

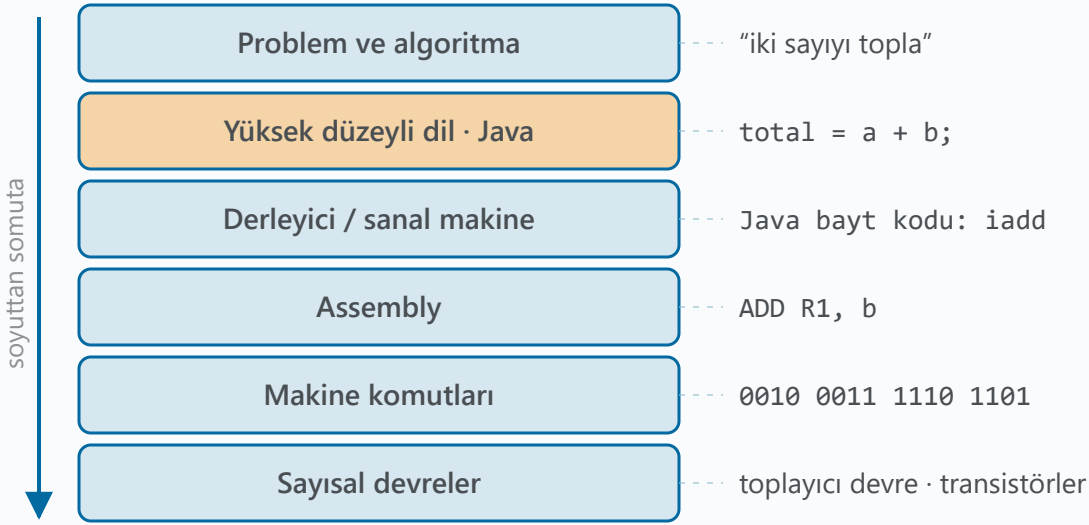
İç tasarıma ve bellek beklemelerine (önbellek ıskaları) bağlıdır.

Saat hızı

Üretim teknolojisi, güç ve ısı sınırlarına bağlıdır.

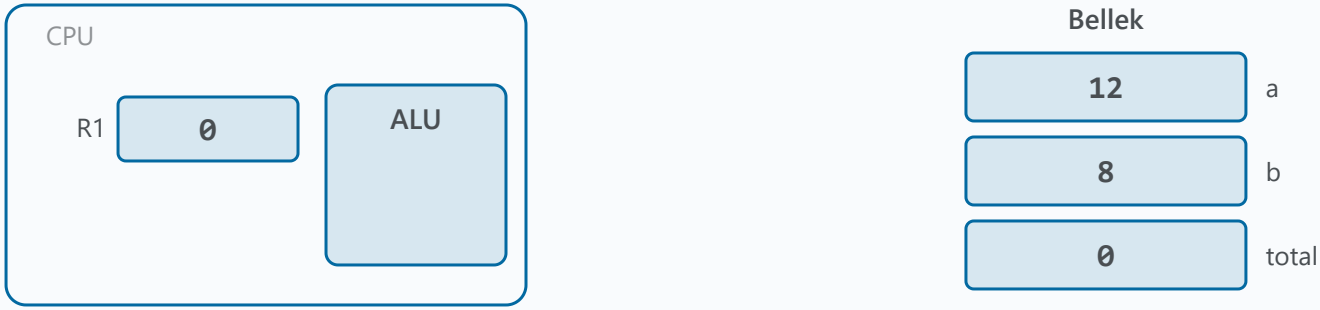
B'nin saat hızı daha yüksek olduğu hâlde A 1,25 kat hızlıdır: tek bir sayı performansı belirlemez.

Programlama Soyutlama Düzeyleri



Her düzey bir alttakinin ayrıntısını gizler. Yüksek düzeyli dil okunabilir ve taşınabilirdir; assembly ve onun ikili karşılığı olan makine komutları belirli bir işlemci mimarisine özgüdür. Örnek satırlar gösterim amaçlıdır.

Kod İzleme: `total = a + b`



Tek Java satırı işlemcide **üç** komuta dönüşür. Bellekte a = 12, b = 8 var.

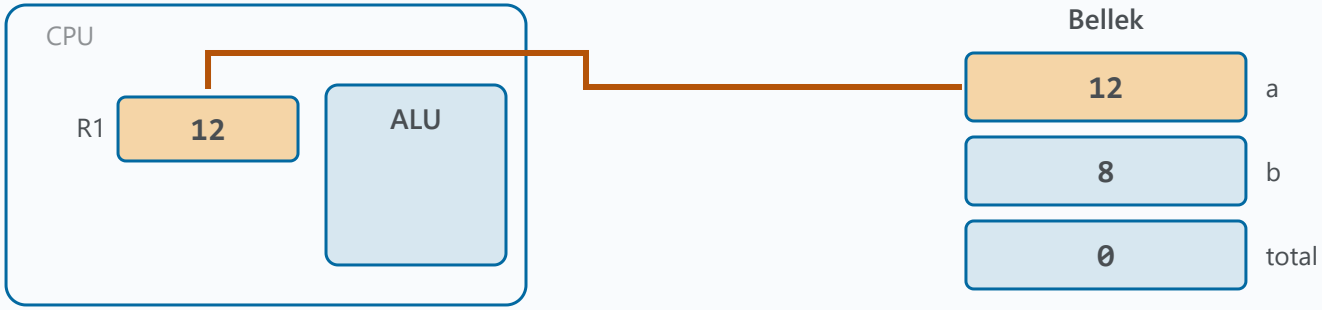
Adım 1 / 4

```
▶ // Java: int total = a + b;
2 LOAD  R1, a       ; R1 ← a
3 ADD   R1, b       ; R1 ← R1 + b
4 STORE R1, total   ; total ← R1
```

DEĞİŞKENLER

`R1 = 0` `a = 12` `b = 8` `total = 0`

Kod İzleme: `total = a + b`



LOAD: a'nın değeri bellekten yazmaca taşınır → R1 = **12**. ALU yalnızca yazmaçlarla çalışır.

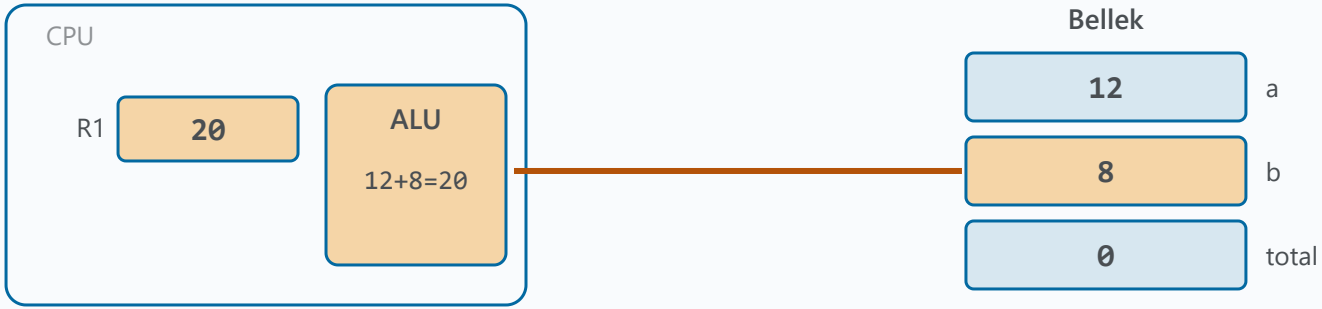
Adım 2 / 4

```
1 // Java: int total = a + b;
▶ LOAD  R1, a       ; R1 ← a
3 ADD   R1, b       ; R1 ← R1 + b
4 STORE R1, total   ; total ← R1
```

DEĞİŞKENLER

`R1 = 12` `a = 12` `b = 8` `total = 0`

Kod İzleme: `total = a + b`



ADD: b okunur, ALU 12 + 8 = **20** hesaplar, sonuç R1'e yazılır.

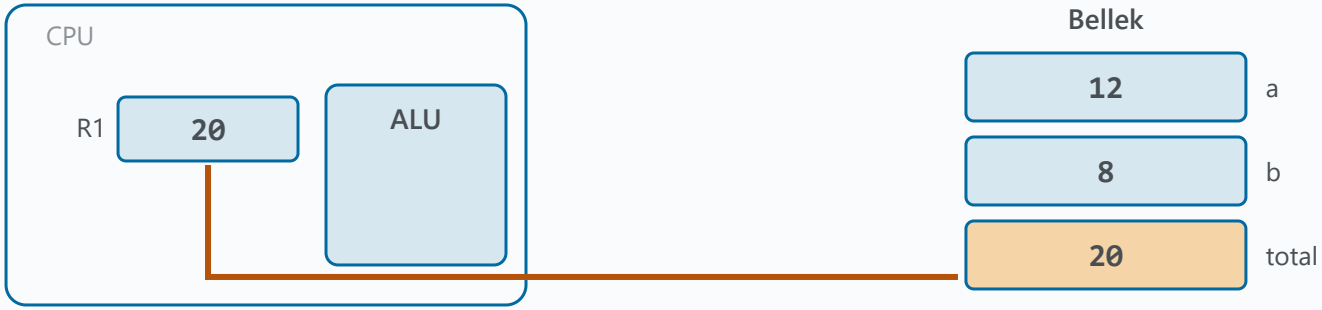
Adım 3 / 4

```
1 // Java: int total = a + b;
2 LOAD  R1, a       ; R1 ← a
▶ ADD   R1, b       ; R1 ← R1 + b
4 STORE R1, total   ; total ← R1
```

DEĞİŞKENLER

`R1 = 20` `a = 12` `b = 8` `total = 0`

Kod İzleme: `total = a + b`



STORE: R1'deki 20 belleğe, total değişkeninin yerine yazılır. Java satırı tamamlandı: total = **20**.

Adım 4 / 4

```
1 // Java: int total = a + b;
2 LOAD  R1, a       ; R1 ← a
3 ADD   R1, b       ; R1 ← R1 + b
▶ STORE R1, total   ; total ← R1
```

DEĞİŞKENLER

`R1 = 20` `a = 12` `b = 8` `total = 20`

ÖZET

Bölüm Özeti

- Bilgisayar bütün verileri bit dizileriyle temsil eder.
- İkili gösterim tam sayılar için kesin, kayan nokta bazı ondalıklar için yaklaşık olabilir.
- ASCII ve Unicode karakterleri sayısal kodlarla eşleştirir.
- Saklanan program yaklaşımında komutlar ve veriler bellekte tutulur.
- Mantık kapıları toplayıcı gibi devrelerin yapı taşıdır; ikili toplama tam toplayıcı zinciriyle yapılır.
- CPU, komutları getir–çöz–yürüt döngüsünde yürütür.
- Yüksek düzeyli diller donanım ayrıntılarını soyutlar.

Kontrol Soruları

1. 1101_2 sayısı onluk sistemde kaçtır?
2. 8 bit işaretli tam sayı aralığı nedir?
3. ASCII ile Unicode arasındaki temel fark nedir?
4. Program komutlarının bellekte saklanması hangi avantajı sağlar?
5. Veri, adres ve kontrol yollarının görevleri nelerdir?
6. Bir assembly komutu ile makine komutu arasındaki ilişki nedir?

Alıştırmalar

1. 8 bit ikiye tümleyen gösterimde 11111110 hangi sayıdır?
2. $3F_{16}$ sayısını ikili ve onluk sisteme çevirin.
3. "ç" (U+00E7) karakterini UTF-8 ile kodlayın; kaç bayt tutar?
4. 8 bit işaretsiz yazmaçta 250 + 10 işleminin sonucu ne olur? Neden?
5. "Programı Adım Adım Çalıştırma" örneğinde 1004 = 7 ve 1005 = 5 olsaydı program bitince 1006'da ne olurdu? Kaç komut getirilirdi?
6. `int fark = a - b;` satırı için LOAD / SUB / STORE ile assembly karşılığını yazın.

▶ **Cevapları göster**

Kaynaklar

- Patterson ve Hennessy, Computer Organization and Design.
- Bryant ve O'Hallaron, Computer Systems: A Programmer's Perspective.
- Unicode Consortium, The Unicode Standard.
- IEEE 754, Floating-Point Arithmetic.

Son

Bilgisayarlar

Sonraki sunum: İnternet · Çalışma soruları · Simülatörler: Sayı Tabanı Dönüştürücü, Mantık Kapıları, Basit CPU: Getir–Çöz–Yürüt · Terim sözlüğü