

Bilgisayar Mimarileri

**Bilgisayarların Dili
Komut Kümesi Mimarisi – 3**

(Instruction Set Architecture (ISA))

Okuma Listesi

Gerekli

- Computer Organization and Design: The Hardware Software Interface [RISC-V Edition] David A. Patterson, John L. Hennessy
 - 2. Bölüm (2.6, 2.7, 2.8, 2.10)

Önerilen

- Computer Organization and Design: The Hardware Software Interface [RISC-V Edition] David A. Patterson, John L. Hennessy
 - 2. ve 3. Bölümler

RISC-V operands

Name	Example	Comments
32 registers	x0 - x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{61} memory words	Memory[0], Memory[8], ..., Memory[18,446,744,073,709,551,608]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential doubleword accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

RISC-V Assembly Dili

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	<code>add x5, x6, x7</code>	$x5 = x6 + x7$	Three register operands; add
	Subtract	<code>sub x5, x6, x7</code>	$x5 = x6 - x7$	Three register operands; subtract
	Add immediate	<code>addi x5, x6, 20</code>	$x5 = x6 + 20$	Used to add constants

RISC-V Assembly Dili

Category	Instruction	Example	Meaning	Comments
Data transfer	Load doubleword	ld x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Doubleword from memory to register
	Store doubleword	sd x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Doubleword from register to memory
	Load word	lw x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Word from memory to register
	Load word, unsigned	lwu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Unsigned word from memory to register
	Store word	sw x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Word from register to memory
	Load halfword	lh x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Halfword from register to memory
	Load byte	lb x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Byte unsigned from memory to register
	Store byte	sb x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Byte from register to memory
	Load reserved	lr.d x5, (x6)	$x5 = \text{Memory}[x6]$	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	$\text{Memory}[x6] = x5; x7 = 0/1$	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	$x5 = 0x12345000$	Loads 20-bit constant shifted left 12 bits

RISC-V Assembly Dili

Category	Instruction	Example	Meaning	Comments
Logical	And	and x5, x6, x7	$x5 = x6 \& x7$	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x8	$x5 = x6 \mid x8$	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	$x5 = x6 \wedge x9$	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	$x5 = x6 \& 20$	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	$x5 = x6 \mid 20$	Bit-by-bit OR reg. with constant
	Exclusive or immediate	xori x5, x6, 20	$x5 = x6 \wedge 20$	Bit-by-bit XOR reg. with constant

RISC-V Assembly Dili

Category	Instruction	Example	Meaning	Comments
Shift	Shift left logical	sll x5, x6, x7	$x5 = x6 \ll x7$	Shift left by register
	Shift right logical	srl x5, x6, x7	$x5 = x6 \gg x7$	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	$x5 = x6 \gg x7$	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	$x5 = x6 \ll 3$	Shift left by immediate
	Shift right logical immediate	srlr x5, x6, 3	$x5 = x6 \gg 3$	Shift right by immediate
	Shift right arithmetic immediate	srair x5, x6, 3	$x5 = x6 \gg 3$	Arithmetic shift right by immediate

x30

01010101 01111111 11111111 00000110 R

Sabit değerin ilk 4 biti

01010101 01111111 11111111 00000001 S

RISC-V Assembly Dili

Category	Instruction	Example	Meaning	Comments
Conditional branch	Branch if equal	beq x5, x6, 100	if (x5 == x6) go to PC+100	PC-relative branch if registers equal
	Branch if not equal	bne x5, x6, 100	if (x5 != x6) go to PC+100	PC-relative branch if registers not equal
	Branch if less than	blt x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	bge x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	bltu x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	bgeu x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	jal x1, 100	x1 = PC+4; go to PC+100	PC-relative procedure call
	Jump and link register	jalr x1, 100(x5)	x1 = PC+4; go to x5+100	Procedure return; indirect call

Hafıza İşlemleri

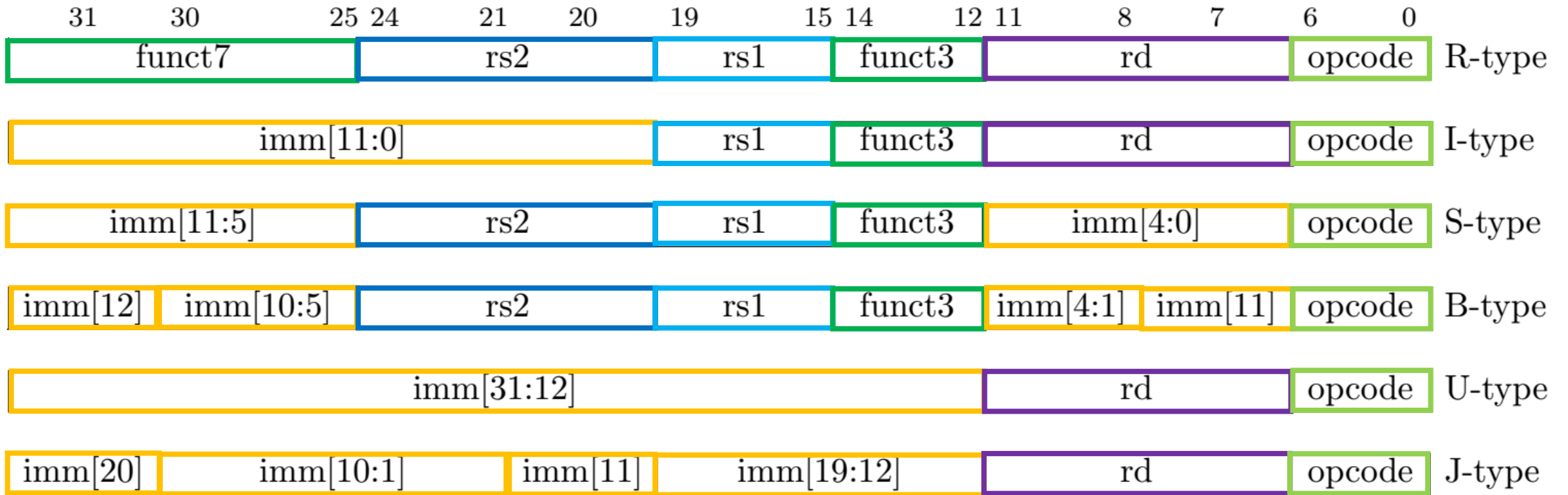
- RISC-V hafıza ve Registerlar arasında veri transferi yapan komutlar içerir.
- Bu tür komutlara **Veri Aktarım Komutları** (Data Transfer Instructions) denir.
- Verileri hafızadan bir registıra kopyalayan veri aktarım komutu yükleme (load) olarak adlandırılır.
- Bu komutun RISC-V adı **ld**'dir ve **load doubleword** anlamına gelir.
- $g \rightarrow x20$ $h \rightarrow x21$ A (Base Address) \rightarrow $x22$ (Base Register) $8 \rightarrow$ offset
- $g = h + A[8]$
- `ld x9, 8(x22)` // önce $A[8]$ 'i bir registıra transfer etmeliyiz
- `add x20, x21, x9`
- `sd x20, 0(x23)`

Sabit veya Anında İşlenenler (Constant or Immediate Operands)

- Çoğu zaman bir program bir işlemde bir sabit kullanır; örneğin, bir dizideki bir sonraki öğeyi işaret edecek şekilde bir dizini artırmak.
- Bir sabit değerli hızlı toplama komutuna **add immediate** veya **addi** denir. x22 yazmacına 4 eklemek için sadece şunu yazarız.
- $\text{addi } x22, x22, 4 // x22 = x22 + 4$
- Aynı komutla bir değişkene değer de atanabilir. X0 içinde her zaman 0 değeri vardır.
- $\text{addi } x22, x0, 4 // x22 = 4$

Tüm RISC-V Komut Tipleri

RISC-V mimarisinde toplamda 6 adet komut türü vardır.



Komut türlerde aynı işlevlerin aynı sıraya koyulduğu görülüyor.

Bunun amacı -ileride de bahsedileceği gibi- **daha sade devre tasarımıdır.**

MIPS Komut Tipleri Karşılaştırması

MIPS'de 3 tip komut vardır: R - I ve J

Register-register

	31	25	24	20	19	15	14	12	11	7	6	0												
RISC-V	funct7(7)					rs2(5)				rs1(5)				funct3(3)		rd(5)				opcode(7)				
	31	26	25	21	20	16	15	11	10	6	5	0												
MIPS	Op(6)					Rs1(5)				Rs2(5)				Rd(5)				Const(5)				Opx(6)		

R-tipi

Load

	31			20	19			15	14		12	11		7	6			0						
RISC-V	immediate(12)												rs1(5)			funct3(3)			rd(5)			opcode(7)		
	31		26	25			21	20			16	15						0						
MIPS	Op(6)						Rs1(5)				Rs2(5)				Const(16)									

I-tipi

Store

	31	25	24	20	19	15	14	12	11	7	6	0									
RISC-V	immediate(7)					rs2(5)				rs1(5)				funct3(3)		immediate(5)			opcode(7)		
	31	26	25	21	20	16	15						0								
MIPS	Op(6)					Rs1(5)				Rs2(5)				Const(16)							

S-tipi

Branch

	31	25	24	20	19	15	14	12	11	7	6	0													
RISC-V	immediate(7)					rs2(5)				rs1(5)				funct3(3)		immediate(5)			opcode(7)						
	31	26	25	21	20	16	15																		
MIPS	Op(6)					Rs1(5)					Opx/Rs2(5)					Const(16)									

B-tipi

B=A[1]

RISC-V R-tipi Komut Kodlaması

fonk7	kr2	kr1	fonk3	hr	işkodu
7 bit	5 bit	5 bit	3 bit	5 bit	7 bit

RISC-V Komut Alanları

İşkodu(OpCode): (İşlem Kodu) İşlemi ve komutun formatını belirten alan.

hr: Hedef registerın adresini tutan alandır.

fonk3: (İşlev Kodu) İşkoduna ek olarak fazladan işkodu alanı.

kr1: Birinci kaynak registerın adresini tutan alandır.

kr2: İkinci kaynak registerın adresini tutan alandır.

fonk7: Fazladan işkodu alanı.

İngilizce:

funct7	rs2	rs1	funct3	rd	opcode
--------	-----	-----	--------	----	--------

RISC-V R-tipi Komut Kodlaması

fonk7	kr2	kr1	fonk3	hr	işkodu
7 bit	5 bit	5 bit	3 bit	5 bit	7 bit

Farklı aritmetik buyrukların **işkodu alanı aynı fonk3 ve fonk7 alanları farklıdır:**

Komut	Format	fonk7	kr2	kr1	fonk3	hr	işkodu
topla	R-tipi	0000000	register	Register	000	register	0110011
çıkar	R-tipi	0100000	register	register	000	register	0110011

RISC-V R-tipi Komut Kodlaması

Bu kodlama her komut için uygun mu?

LD komutunu nasıl kodlayabiliriz?

$a=b+c$

Gibi bir işlem için kullanılır

fonk7	anlık	kr1	fonk3	hr	işkodu
7 bit	5 bit	5 bit	3 bit	5 bit	7 bit

`ld x1, 17(x2)` `// x1 = bellek[x2+17]`

`ld x1, 743(x2)` `// x1 = bellek[x2+743]`

5-bit alan büyük anlık değerleri kodlamak için yeterli değil.

- Buyruk boyutlarının **sabit kalması** için **farklı komut tiplerine** ihtiyacımız var.

İyi tasarım bedel ödeme gerektirir.

RISC-V I-tipi Komut Kodlaması

Şu ana kadar gördüğümüz RISC-V kodlaması **R-tipi** (*register*) komutlar içindi.

I-tipi (*immediate*). Anlık değer, kaynak ve hedef register.

- Anlıkla topla (`addi`) ve yükle (`ld`) buyrukları.

anlık	kr1	fonk3	hr	işkodu
12 bit	5 bit	3 bit	5 bit	7 bit



12-bit 2'ye tümleyen değeri.
• Negatif ve pozitif sayılar.

`a=b+5`

`a=b[5]`

Gibi bir işlemler için kullanılır

RISC-V S-tipi Komut Kodlaması

Kaydet (sd) komutu için de yeni bir kodlamaya ihtiyacımız var.

- İki kaynak registerı, bir anlık değer.

S-tipi komut formatı:

anlık [11:5]	kr2	kr1	fonk3	anlık [4:0]	işkodu
7 bit	5 bit	5 bit	3 bit	5 bit	7 bit

Kaynak registerlarının tüm formatlarda aynı yerde bulunması için S-tipi komutlarda anlık iki parçaya ayrılmıştır.

- Basit donanım.

RISC-V Komut Kodlaması

Her komut yapısı özel bir işkodu değerine sahiptir.

Donanım işkodu değerine bakarak komut formatını anlayabilir.

Şu ana kadar gördüğümüz komutların formatları ve komut alanları:

Komut	Format	fonk7	kr2	kr1	fonk3	hr	işkodu
topla	R-tipi	0000000	Reg	Reg	000	Reg	0110011
çıkar	R-tipi	0100000	Reg	Reg	000	Reg	0110011

Komut	Format	Anlık	kr1	fonk3	hr	işkodu
anlıkla topla	I-tipi	sabit değer	Reg	000	Reg	0010011
yükle	I-tipi	sabit değer	Reg	011	Reg	0000011

Komut	Format	anlık	kr2	kr1	fonk3	anlık	işkodu
kaydet	S-tipi	adres	Reg	Reg	011	adres	0100011

C Kodundan RISC-V Makine Koduna

$A[30] = h + A[30] + 1;$

Derleyici Değişken Eşleştirmesi

A'nın başlangıç adresi \rightarrow x10

h \rightarrow x21

Komut	Format	fonk7	kr2	kr1	fonk3	hr	işkodu
topla	R-tipi	0000000	Reg	Reg	000	Reg	0110011
çıkar	R-tipi	0100000	Reg	Reg	000	Reg	0110011

Komut	Format	Anlık	kr1	fonk3	hr	işkodu
a-topla	I-tipi	sabit değer	Reg	000	Reg	0010011
yükle	I-tipi	sabit değer	Reg	011	Reg	0000011

Komut	Format	anlık	kr2	kr1	fonk3	anlık	işkodu
kaydet	S-tipi	adres	Reg	Reg	011	adres	0100011

C Kodundan RISC-V Makine Koduna

$A[30] = h + A[30] + 1;$

Derleyici Değişken Eşleştirmesi

A'nın başlangıç adresi \rightarrow x10

h \rightarrow x21

ld x9, 240(x10)

add x9, x21, x9

addi x9, x9, 1

sd x9, 240(x10)

(I) anlık					
(R) fonk7 (S) anlık[11:5]	(R) kr2	kr1	fonk3	hr (S) anlık[4:0]	işkodu
240		10	3	9	3
0	9	21	0	9	51
1		9	0	9	19
7	9	10	3	16	35

000011110000		01010	011	01001	0000011
0000000	01001	10101	000	01001	0110011
0000000000001		01001	000	01001	0010011
0000111	01001	01010	011	10000	0100011

İkilik Taban

Mantıksal İşlem Buyrukları

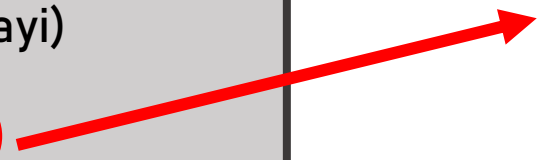
Mantıksal İşlemler	C / Java Dili Karşılığı	RISC-V buyrukları
Sola kaydır	<<	sll, slli
Sağa kaydır	>>	srl, srli
Aritmetik sağa kaydır	>>	sra, srai
VE (bit düzeyinde)	&	and, andi
VEYA (bit düzeyinde)		or, ori
DIŞLAYAN VEYA (bit düzeyinde)	^	xor, xori
DEĞİL (bit düzeyinde)	~	xori

DEĞİL işlemini gerçekleştirmenin bir yolu her biti 1 olan bir sayı (FFFF FFFF FFFF_{hex}) ile DIŞLAYAN VEYA işlemi yapılmasıdır.

Dallanma Komutları

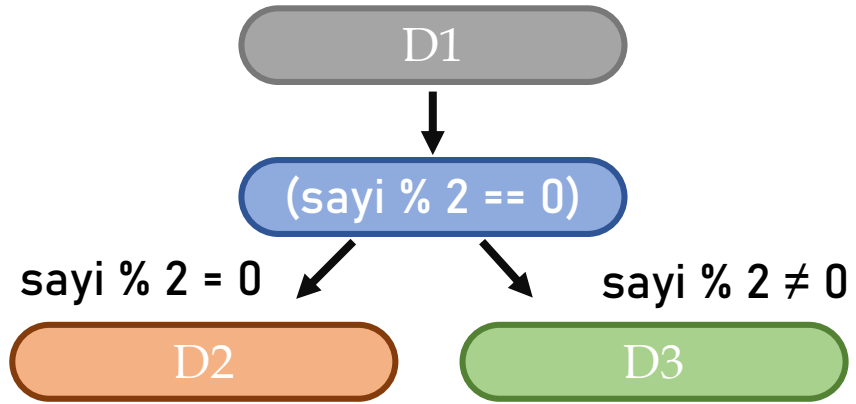
```
void tekMiCiftMi(int sayi)
{
    if (sayi % 2 == 0)
        // çiftte zıpla
        goto even;
    else
        // teke zıpla
        goto odd;

    odd:
        printf("%d sayisi tek", sayi);
    even:
        printf("%d sayisi cift", cift);
        // cift ise don
        return;
}
```



D2

D3

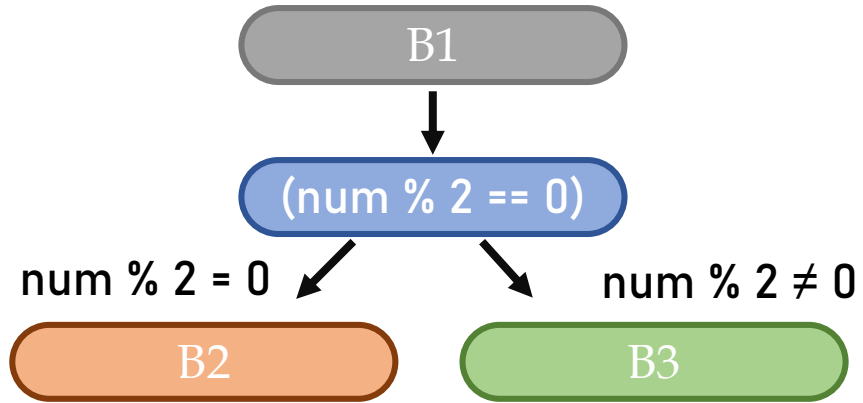


Koşullu Dallanma

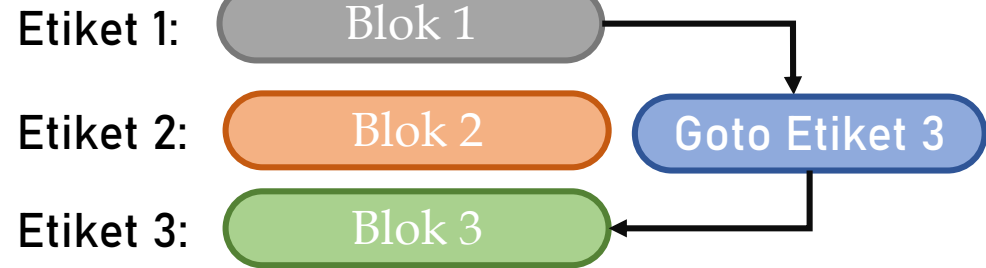
Dallanma Komutları

```
void tekMiCiftMi(int sayi)
{
    if (sayi % 2 == 0)          } B1
        // çift zıpla
        goto even;
    else
        // teke zıpla
        goto odd;

    odd:                        } B2
        printf("%d sayisi tek", sayi);
    even:                       } B3
        printf("%d sayisi cift", cift);
        // cift ise don
        return;
}
```



*Koşullu Dallanma
Branch*



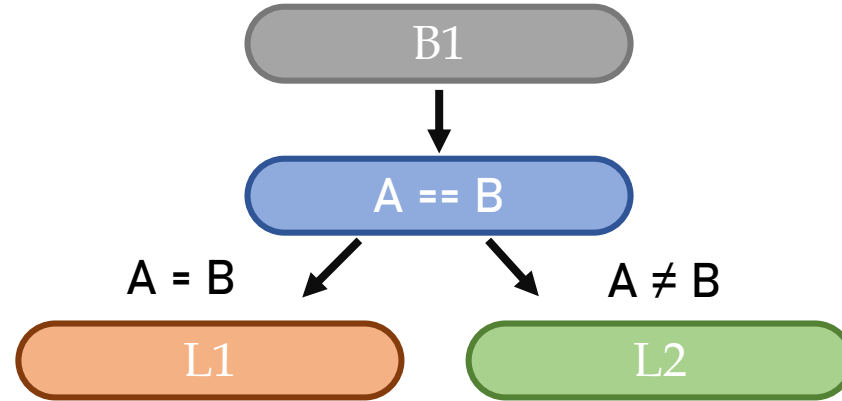
*Koşulsuz Atlama
(Jump)*

Koşullu Dallanma Komutları

eşit ise atla (Branch EQuals)

beq rs1, rs2, L1
bne rs1, rs2, L2

eşit değilse atla (Branch Not Equals)



Döngü:

```
for (int i = 0; i < 20; i++) {  
    ...  
}  
for (int i = 30; i >= 20; i--) {  
    ...  
}
```

eşit veya büyükse atla

bge rs1, rs2, L1
blt rs1, rs2, L2

küçükse atla

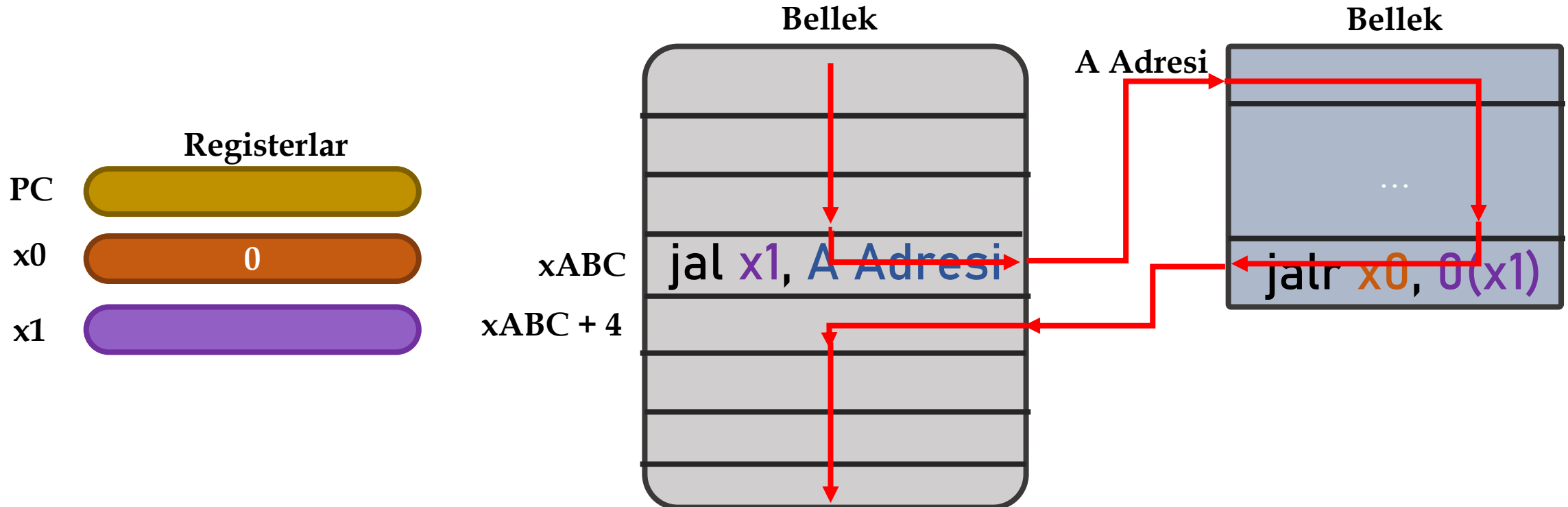
Koşulsuz Atlama

`jal x1, A Adresi`

Atla ve kaydet (*jump and link*) komutu, **A adresine** atlar ve dönüş adresini **x1'e** kaydeder.

`jalr x0, 0(x1)`

Registıra atla ve kaydet (*jump and link register*) komutu, **x1 Registırındaki adrese** atlar ve dönüş adresini **x0'a** kaydeder.



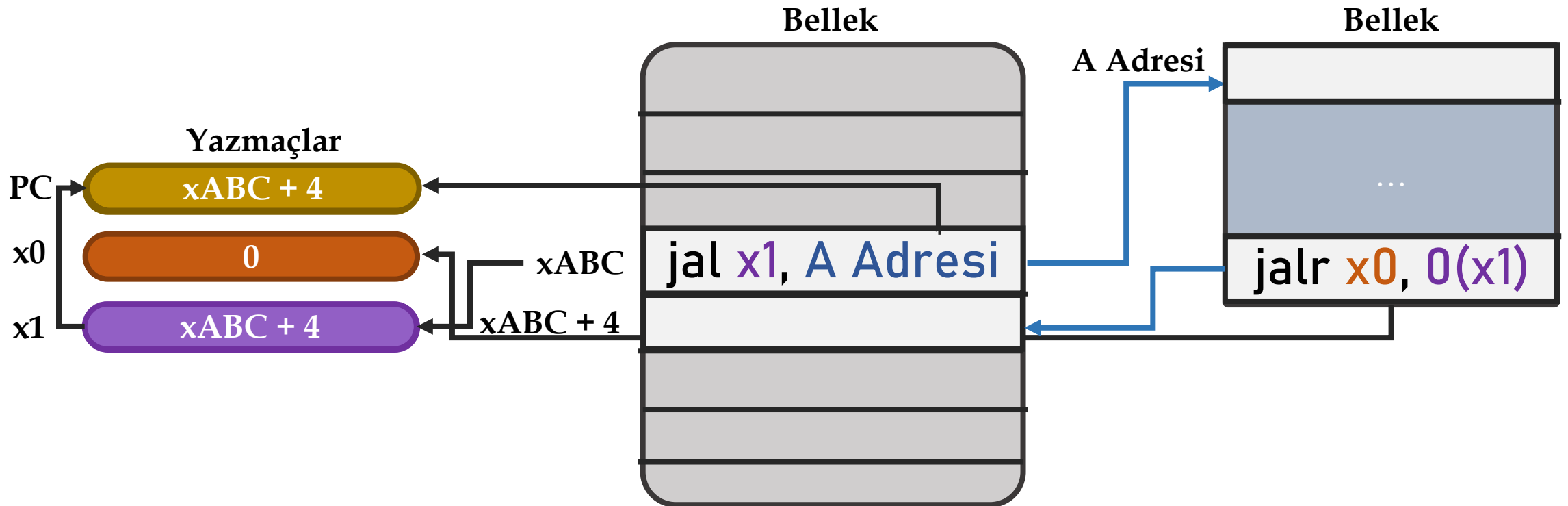
Koşulsuz Atlama

`jal x1, A Adresi`

Atla ve kaydet (*jump and link*) buyruğu, **A adresine** atlar ve dönüş adresini **x1'e** kaydeder.

`jalr x0, 0(x1)`

Yazmaca atla ve kaydet (*jump and link register*) buyruğu, **x1 yazmacındaki adrese** atlar ve dönüş adresini **x0'a** kaydeder.

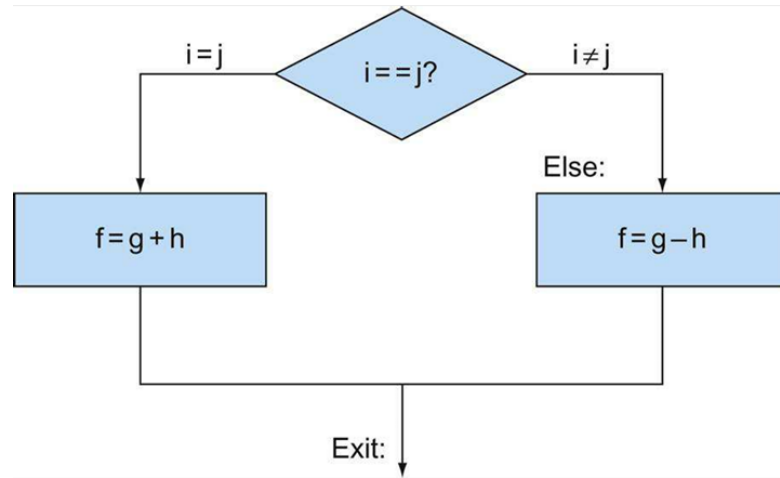


if-then-else Koşullu Dallanma Yapısının Derlenmesi

```
if (i == j) {  
    f = g + h;  
} else {  
    f = g - h;  
}
```

```
bne x22, x23, else // i=j ise else'git  
add x19, x20, x21 // f = g + h  
beq x0, x0, exit // if 0=0 ise exit'e git  
else:  
sub x19, x20, x21 // f = g - h  
exit:
```

f → x19
g → x20
h → x21
i → x22
j → x23



Döngüler: While.

```
while (dizi[i] == k)
    i += 1;
```

i → x22

k → x24

dizi → x25 (Base)

- İlk adım dizi[i]'yi geçici bir registıra yüklemektir. Ve dizi[i]'yi geçici bir registıra yükleyebilmemiz için önce dizi'nin base adresini bilmemiz gerekir.
- Adresi oluşturmak için i'yi dizi kaydının tabanına eklemekten önce, bayt adresleme sorunu nedeniyle i indeksini 8 ile çarpmamız gerekir.

```
Loop: slli x10, x22, 3 // Geçici register x10 = i * 8
```

- Döngünün sonunda bu komuta geri dönebilmemiz için ona «loop» gibi bir etiket eklememiz gerekir.
- dizi[i] adresini almak için x10'u ve dizi[] tabanını x25'e kaydetmemiz gerekir:

```
add x10, x10, x25 // x10 = dizi[i]'nin adresi
```

- Artık bu adresi dizi[i]'yi geçici bir registıra yüklemek için kullanabiliriz:

```
ld x9, 0(x10) // Geçici reg x9 = dizi[i]
```

- Bir sonraki komut döngü sorgusunu gerçekleştirir ve dizi[i] ≠ k ise exit etiketine atlar:

```
bne x9, x24, exit // eğer dizi[i] ≠ k ise exit etiketine git
```


Döngüler: While

Aşağıdaki komut i'ye 1 ekler:

```
addi x22, x22, 1 // i = i + 1
```

- Döngünün sonu, döngünün tepesindeki **while** sorgusuna (loop etiketine) geri döner. Hemen arkasına **exit** etiketini ekliyoruz.

```
beq x0, x0, Loop // loop etiketine atla  
exit:
```

- Tüm While döngüsü

```
while (dizi[i] == k)  
    i += 1;
```

i → x22

k → x24

dizi → x25 (Base)


Loop:

```
slli x10, x22, 3  
add x10, x10, x25  
ld x9, 0(x10)  
bne x9, x24, exit  
addi x22, x22, 1  
beq x0, x0, loop
```

exit:

Fonksiyonlar (-ing. Functions)

Kodu kolayca **tekrar kullanabilmek** ve kod yazarken **tek bir hedefe odaklanmak** için programcılar **fonksiyon** adı verilen yapılardan faydalanırlar.

<pre>int çağırılan_fonk() { int a = 5; int b = 8; çağırılan_fonk(96); return a + b; }</pre>		<pre>addi x8, x0, #5 // int a = 5 addi x9, x0, #8 // int b = 8 addi x10, x0, #96 // çağırılan_fonk parametresi jal x1, çağırılan_fonk // çağırılan_fonk(96) add x10, x8, x9 // return a + b</pre>
---	---	--

- **x10-x17** arasındaki yazmaçlar fonksiyonlara **parametre** olarak verilen değerleri ve işlemlerin **döndüğü değerleri (Return Values)** barındırır.
- **x1** fonksiyonunun **döneceği adresi** barındırır.

Program Yığını (Program Stack)

```
int çağırılan_fonk()
```

```
{
```

```
int a = 5;
```

```
int b = 8;
```

```
çağırılan_fonk(96);
```

```
return a + b;
```

```
}
```

```
addi x8, x0, #5
```

```
// int a = 5
```

```
addi x9, x0, #8
```

```
// int b = 8
```

```
addi x10, x0, #96
```

```
// çağırılan_fonk argümanı
```

```
jal x1, çağırılan_fonk
```

```
// çağırılan_fonk(96)
```

```
add x10, x8, x9
```

Çağırılan fonksiyon x8 ve x9 registırlarını kullanırsa (üzerine yazarsa) ne olur?

Program yanlış çalışır.

→ Fonksiyonlara ait **veri** program yığnında (-ing. stack) saklanır.

→ Yığıt: Öğelerden son gelenin ilk işlem göreceğ biçimde üst üste yığıldığı varsayılan veri yapısı.

→ Kodun belleğe saçılması fonksiyona ait yerel değişkenlerin program yığnına yazılması ile gerçekleşir.

Program Yığının

Yığın bellekte tutulur. Yığının bellekteki adresi **yığın işaretçisi** (-ing. stack pointer) ile belirtilir.

→ RISC-V’de yığın işaretçisi **x2 yazmacında** tutulur.

Push: Yığının en üstüne veri eklemek.

Pop: Yığından en üstteki veriyi çıkarmak.

→ Yığın “yukarıdan aşağıya” doğru genişler.

→ Push: Yığın işaretçisini **azaltır**.

→ Pop: Yığın işaretçisini **artırır**.

RISC-V’de **push** ve **pop** komutları yoktur.

→ Kaydet ve yükle komutları kullanılır.

Program Yığını

```
int çağrılan_fonk(int x)
{
    int a = 5;
    int b = 8;

    return a + b + x;
}
```

$sp \rightarrow x2$

addi sp, sp, -16
sd **x8**, 8(sp)
sd **x9**, 0(sp)

Çağırılan fonksiyonun
register değerlerini
yığına kaydet.

addi **x8**, x0, #5
addi **x9**, x0, #8

// int a = 5
// int b = 8

ld **x8**, 8(sp)
ld **x9**, 0(sp)
addi sp, sp, 16

Çağırılan fonksiyonun
register değerlerini
yığından geri yükle.

add x9, x8, x9
add x10, x10, x9

// x9 = a + b
// x10 = x + x9 = x + a + b

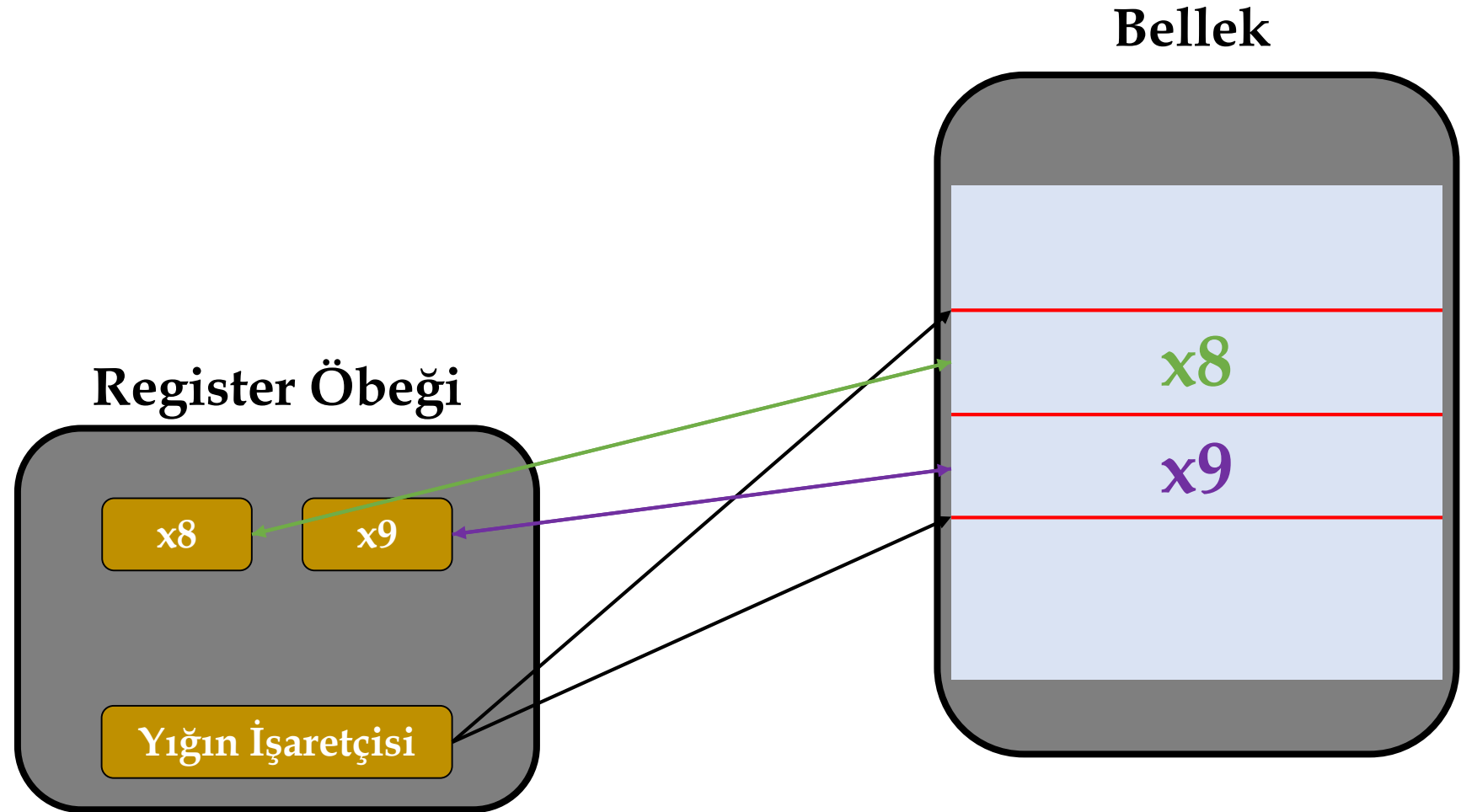
Program Yığının

`addi sp, sp, -16`
`sd x8, 8(sp)`
`sd x9, 0(sp)`

`addi x8, x0, #5`
`addi x9, x0, #8`

`ld x8, 8(sp)`
`ld x9, 0(sp)`
`addi sp, sp, 16`

`add x9, x8, x9`
`add x10, x10, x9`



RISC-V Adresleme Kipleri

Anlık adresleme (Immediate addressing)



→ İşlenen komutun içinde.

Register adresleme (Register addressing)



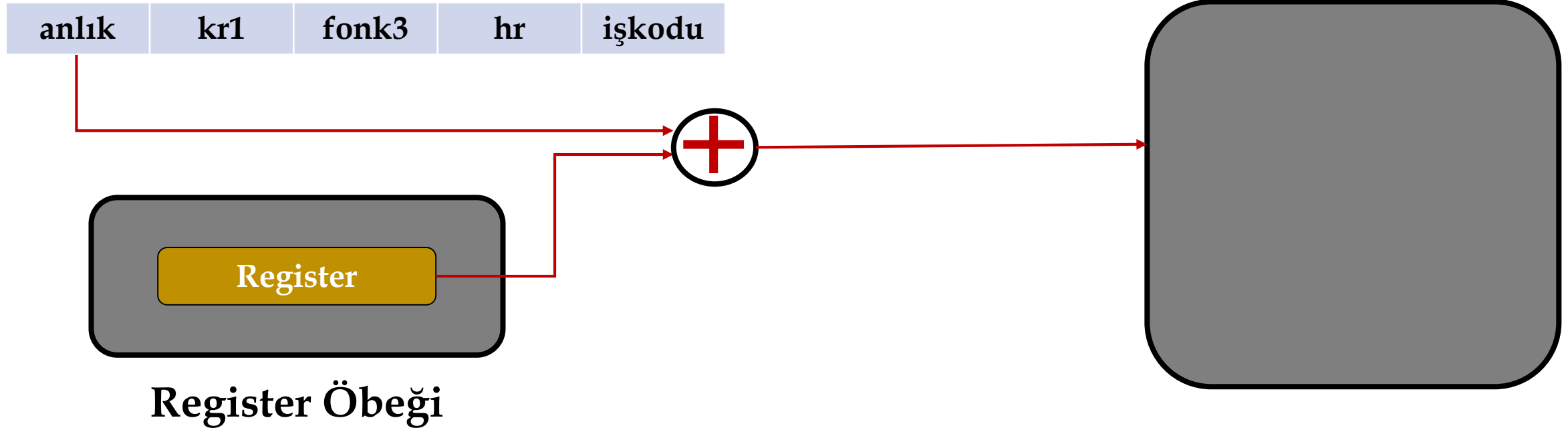
→ İşlenen Register öbeğinde.

Register Öbeği



RISC-V Adresleme Kipleri

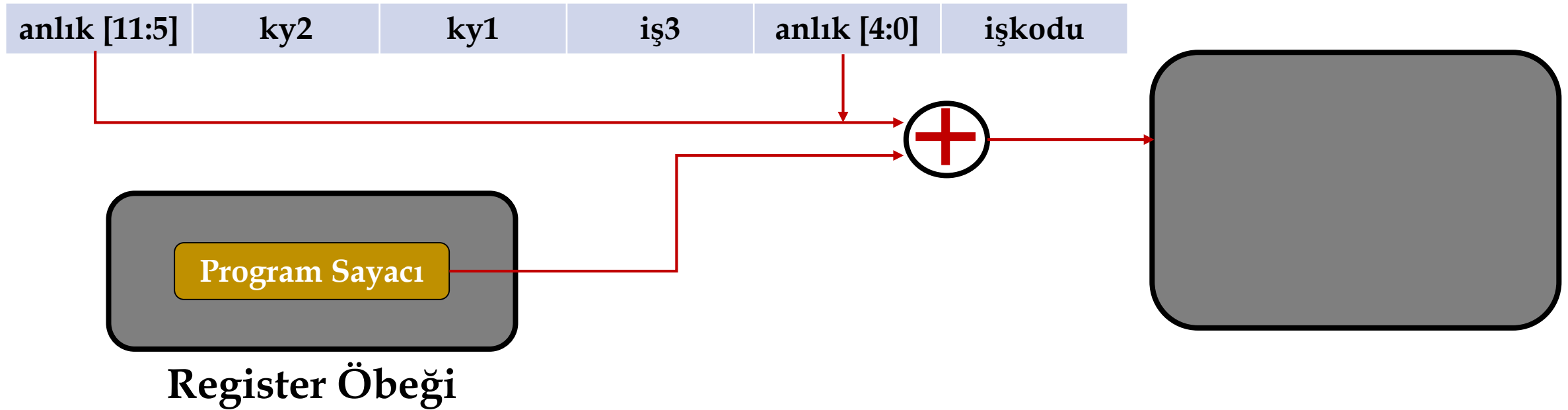
Eklemeli adresleme (Base addressing)



→ İşlenen bellekte, bir registıra anlık değerin **eklenmesi** ile adresleniyor.

RISC-V Adresleme Kipleri

Göreceli Adresleme (PC-relative addressing)



→ **Dallanma adresi** program sayacı ve anlık değerin toplamı olarak hesaplanır.

Özet

- Bilgisayarların Dili: Komut Kümesi Mimarisi
- Von Neumann ve Harvard Mimarileri
- RISC ve CISC Mimariler
- Sabit ve Değişken Boyutlu Komutlar
- RISC-V İşlemleri
 - Aritmetik İşlemler
 - Bellek İşlemleri
- RISC-V İşlenenleri
 - RISC-V Registrları
 - Anlık Değerler
- RISC-V Bellek Adreslemesi
 - Bayt adresleme
 - Adres hizalaması
- RISC-V Komutları
 - RISC-V Komut Kodlaması
 - RISC-V Komut Formatları
- RISC-V Dallanma Komutları
- RISC-V Mantıksal İşlem Komutları
- İşlevler ve Program Yığıtı
- RISC-V Adresleme Kipleri