



BİLGİSAYAR MİMARİSİ

2021-2022

2. HAFTA

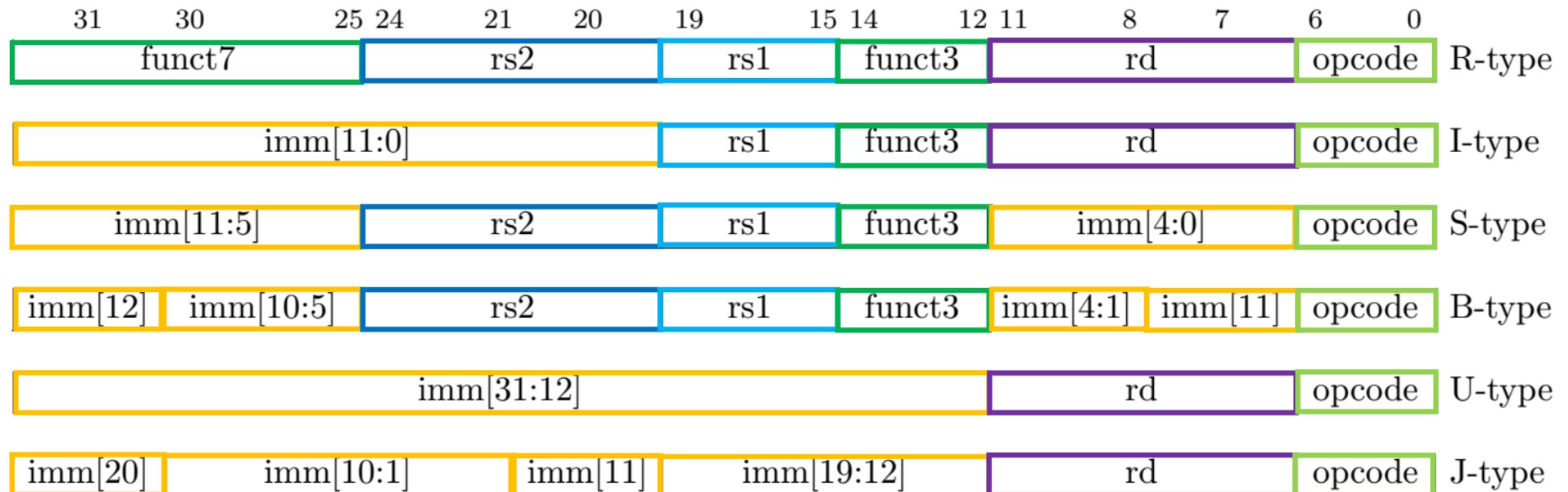
Dr. Öğr. Üyesi Ümit ŞENTÜRK

Okuma Listesi

Gerekli

- Computer Organization and Design: The Hardware Software Interface [RISC-V Edition] David A. Patterson, John L. Hennessy
 - 2.7-2. Bölüm sonuna kadar
- TOBB üniversitesi, Prof Dr. Oğuz ERGİN, Bilgisayar Mimarisi ve Organizasyonu dersi ders sunumları

RISC-V Buyruk Türleri



Örnekler

R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (add)	0000000	00011	00010	000	00001	0110011	add x1, x2, x3
sub (sub)	0100000	00011	00010	000	00001	0110011	sub x1, x2, x3

I-type Instructions	immediate	rs1	funct3	rd	opcode	Example
addi (add immediate)	001111101000	00010	000	00001	0010011	addi x1, x2, 1000
ld (load doubleword)	001111101000	00010	011	00001	0000011	ld x1, 1000 (x2)

S-type Instructions	immed -iate	rs2	rs1	funct3	immed -iate	opcode	Example
sd (store doubleword)	0011111	00001	00010	011	01000	0100011	sd x1, 1000(x2)

funct7	rs2	rs1	funct3	rd	opcode
32	9	10	000	11	51

1. subx9, x10, x11
2. addx11, x9, x10
3. subx11, x10, x9
4. subx11, x9, x10

Mantıksal İşlemler

Logical operations	C operators	Java operators	RISC-V instructions
Shift left	<<	<<	sll, slli
Shift right	>>	>>>	srl, srli
Shift right arithmetic	>>	>>	sra, srai
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	xori

0000 0000 0000 0000 0000 0000 0000 1001_{two} = 9_{ten}

0000 0000 0000 0000 0000 0000 1001 0000_{two} = 144_{ten}

Mantıksal İşlemler (Mantıksal Kaydırma)

```
slli x11, x19, 4 // reg x11 = reg x19 << 4 bits
```

func76	immediate	rs1	func3	rd	opcode
0	4	19	1	11	19

Hangi buyruk formatı ?

Sola kaydırmanın özel tarafı desimal sayı sisteminde sayıları sola i kaydırduğımızda sayı $\ast 10^i$ elde ederiz. İkili sayı sisteminde sola i kaydırduğımızda sayı $\ast 2^i$ olur. Sağa kaydırduğımızda da 2^i ye bölme işlemi gerçekleşir.

Yukarıdaki örnekte 4 bit kaydırma $2^4=16$ ile çarpım anlamına gelir.

Mantıksal İşlemler (Mantıksal VE)

```
and x9, x10, x11 // reg x9 = reg x10 & reg x11
```

0000 0000 0000 0000 0000 1101 1100 0000_{two}

0000 0000 0000 0000 0011 1100 0000 0000_{two}

0000 0000 0000 0000 0000 1100 0000 0000_{two}

VE (and) işleminde karşılıklı bitler 1-1 geldiğinde sonuç 1 olur.

Mantıksal İşlemler (Mantıksal VEYA)

```
or x9, x10, x11 // reg x9 = reg x10 | reg x11
```

0000 0000 0000 0000 0000 1101 1100 0000_{two}

0000 0000 0000 0000 0011 1100 0000 0000_{two}

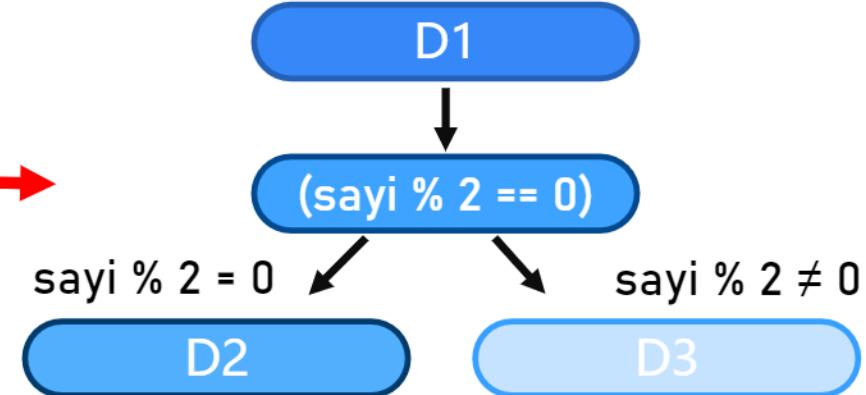
0000 0000 0000 0000 0011 1101 1100 0000_{two}

VEYA (or) işleminde bitlerden birisinin 1 olması sonucu 1 yapar.

Dallanma Buyrukları

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

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



Koşullu Dallanma

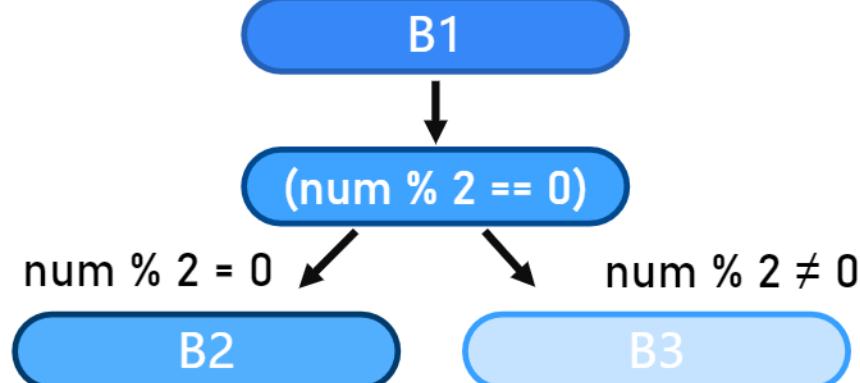
Dallanma Buyrukları

```
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;
}
```

B1

B2
B3



Koşullu Dallanma



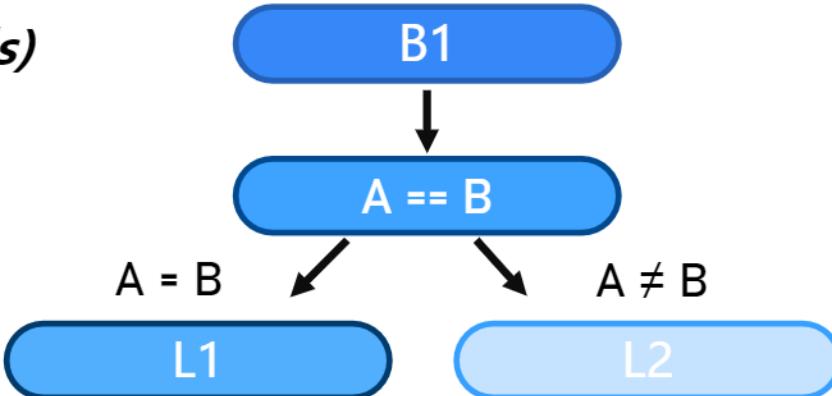
Koşulsuz Atlama

Koşullu Dallanma Buyrukları

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

Karar Verme Buyrukları

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

f,g,h,i,j verileri x19-x23 yazmaçlarında bulunsun

```
bne x22, x23, Else // go to Else if i ≠ j
```

```
add x19, x20, x21 // f = g + h (skipped if i ≠ j)
```

```
beq x0, x0, Exit // if 0 == 0, go to Exit
```

```
Else:sub x19, x20, x21 // f = g - h
```

```
Exit:
```

Döngüler

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

i ve k verileri x22-x24 , save[0] taban adresi--- x25 yazmaçlarında bulunsun.

```
Loop: slli x10, x22, 3 // Temp reg x10 = i * 8
add x10, x10, x25 // x10 = address of save[i]
ld x9, 0(x10) // Temp reg x9 = save[i]
bne x9, x24, Exit // go to Exit if save[i] ≠ k
addi x22, x22, 1 // i = i + 1
beq x0, x0, Loop // go to Loop
Exit:
```

Yazmaçlar

x9

x10

x22

x24

x25

Bellek

save[i]

save[0]

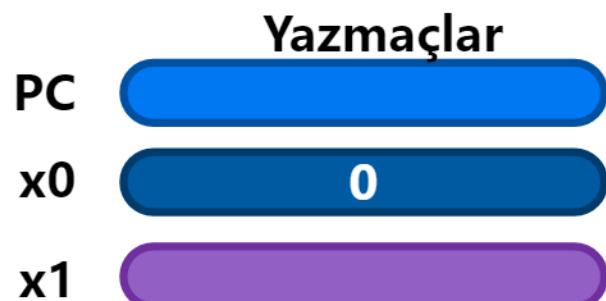
save[8]

save[16]

Koşulsuz Atlama

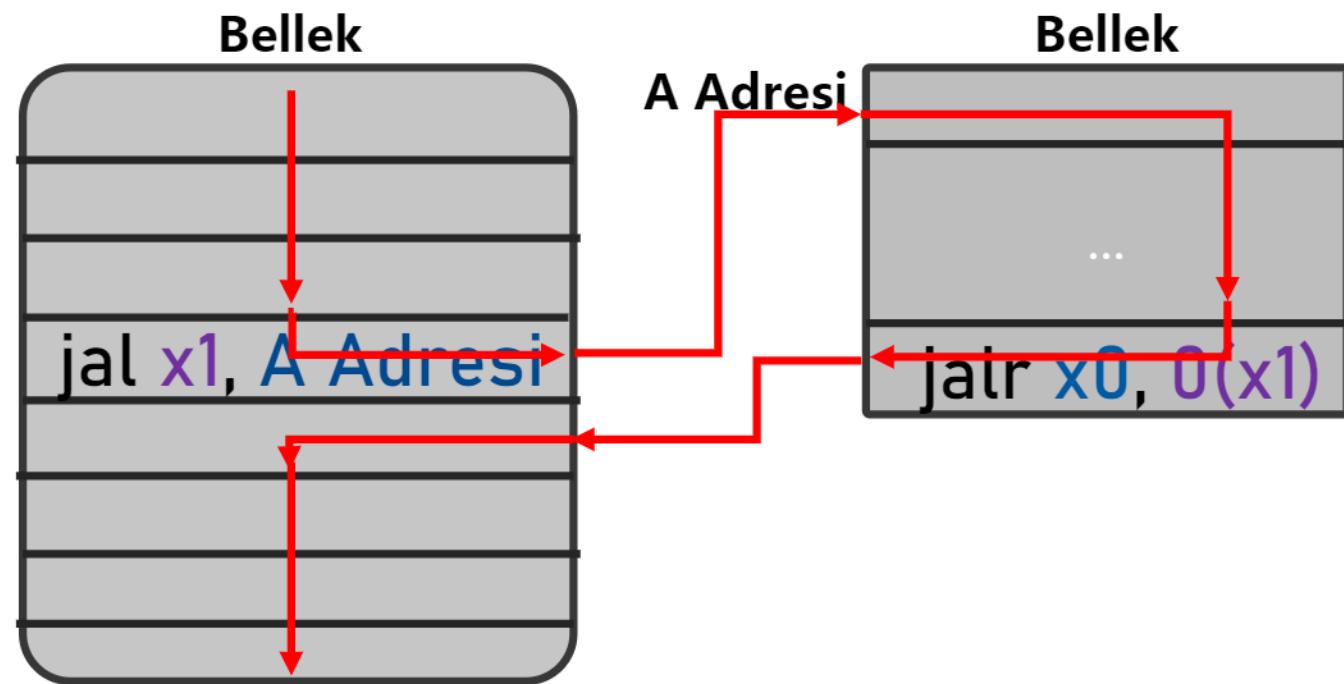
jal x1, A Adresi

jalr x0, 0(x1)



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

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



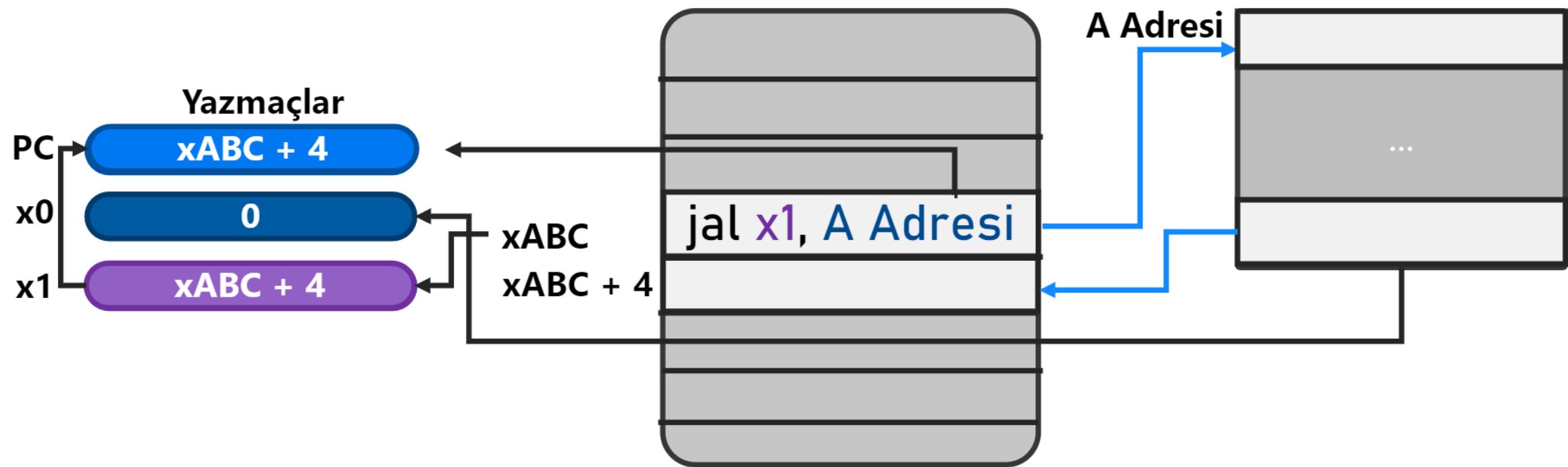
Koşulsuz Atlama

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 adres'e atlar ve dönüş adresini $x0'$ a kaydeder.



İşlevler (-ing. Functions) (Java'da Metot)

Kodu kolayca **tekrar kullanabilmek** ve kod yazarken **tek bir hedefe** odaklanmak için programcılar **işlev** (-ing. **function**) adı verilen yapılardan faydalananırlar.

```
int çağrıran_İşlev()
{
    int a = 5;
    int b = 8;
    çağrılan_İşlev(96);
    return a + b;
}

→ addi x8, x0, #5          // int a = 5
   addi x9, x0, #8          // int b = 8
   addi x10, x0, #96         // çağrılan_İşlev argümanı
   jal x1, çağrılan_İşlev  // çağrılan_İşlev(96)
   add x10, x8, x9           // return a + b
```

x10-x17 arasındaki yazmaçlar işlevlere **argüman** olarak verilen değerleri ve işlemlerin **döndüğü değerleri** barındırır.
x1 işlevin **döneceği adresi** barındırır.

Program Yığıtı

```
int çağrıran_İşlev()
{
    int a = 5;
    int b = 8;

    çağrılan_İşlev(96); →
    return a + b;
}
```

addi x8, x0, #5	// int a = 5
addi x9, x0, #8	// int b = 8
addi x10, x0, #96	// çağrılan_İşlev argümanı
jal x1, çağrılan_İşlev	// çağrılan_İşlev(96)
add x10, x8, x9	

Çağrılan işlev x8 ve x9 yazmaçlarını kullanırsa (üzerine yazarsa) ne olur?

Program yanlış çalışır.

→ İşlevlere ait **veri** program yığıtında (-ing. stack) saklanır.

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

→ **Kodun belleğe saçılmaması** işlevde ait yerel değerlerin **program yığıtına yazılması ile** gerçekleşir.

Yığıt bellekte tutulur. Yığıtin bellekteki adresi **yığıt işaretçisi** (-ing. stack pointer) ile belirtilir.

→ RISC-V' de yığıt işaretçisi **x2 yazmacında** tutulur.

Push: Yığıtin en üstüne veri eklemek.

Pop: Yığıttan en üstteki veriyi çıkarmak.

→ Yığıt "yukarıdan aşağıya" doğru genişler.

 → Push: Yığıt işaretçisini **azaltır**.

 → Pop: Yığıt işaretçisini **artırır**.

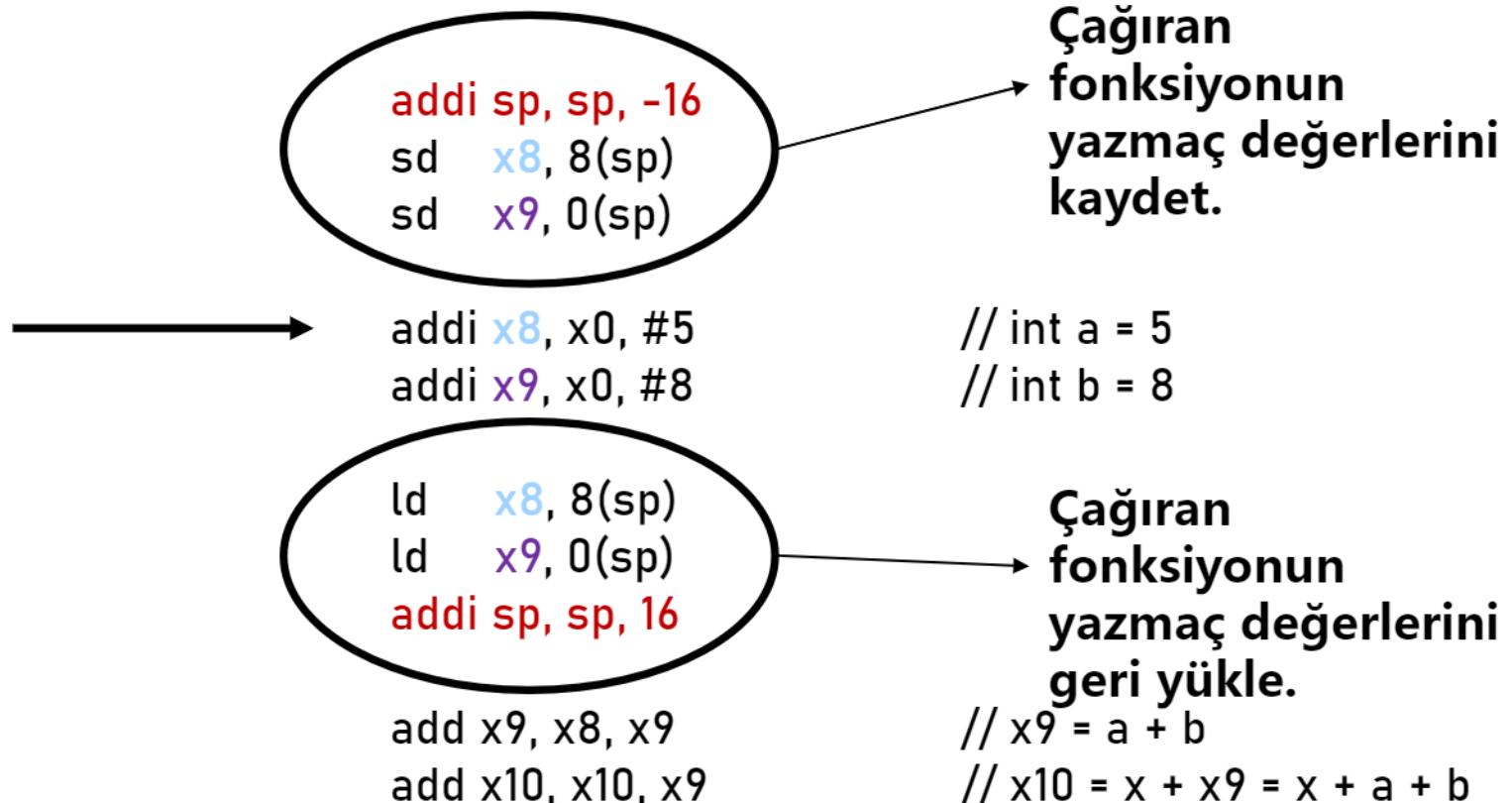
RISC-V' de **push** ve **pop** buyrukları yoktur.

→ Kaydet ve yükle buyrukları kullanılır.

Program Yığıtı

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

    return a + b + x;
}
```



Program Yığıtı

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

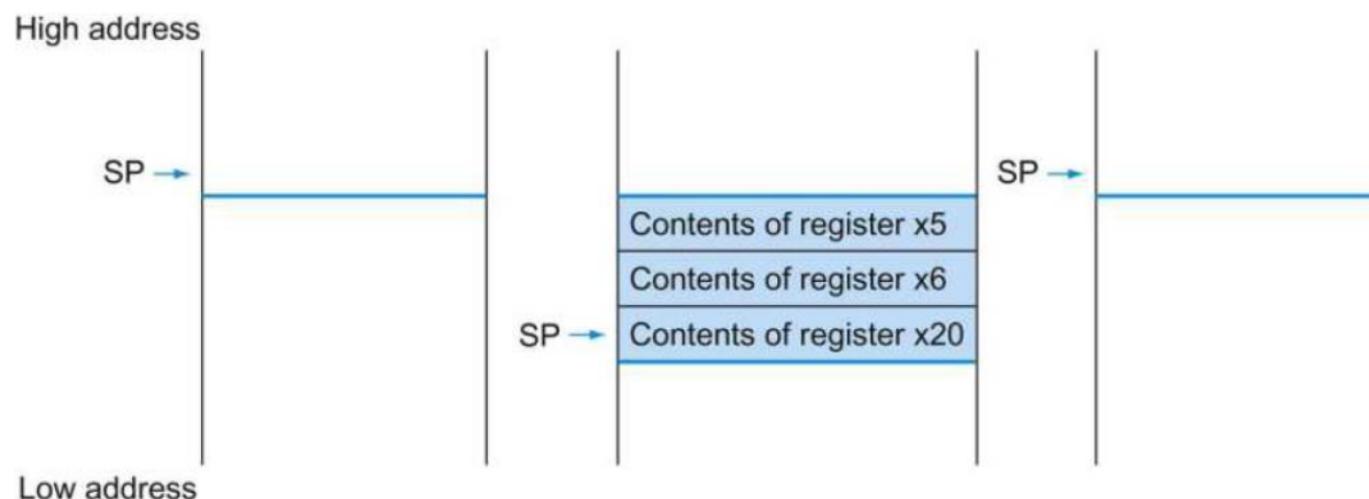


Metot Çağırma

```
long long int leaf_example (long long int g, long long int  
h, long long int i, long long int j)  
{  
    long long int f;  
    f = (g + h) - (i + j);  
    return f;  
}  
  
addi sp, sp, -24 // adjust stack to make room for 3 items  
sdx5, 16(sp) // save register x5 for use afterwards  
sdx6, 8(sp) // save register x6 for use afterwards  
sdx20, 0(sp) // save register x20 for use afterwards
```

g, h, i, j ve f-----x10, x11, x12,
x13 ve x20

x5, x6 geçici kaydediciler ve x20
için bellekten yer ayrılıyor.



Metot Çağırma

```
long long int leaf_example (long long int g, long long int
h, long long int i, long long int j)
{
    long long int f;
    f = (g + h) -(i + j);
    return f;
}
```

```
add x5, x10, x11// register x5 contains g + h
add x6, x12, x13// register x6 contains i + j
sub x20, x5, x6// f = x5 - x6, which is (g + h)-(i + j)
```

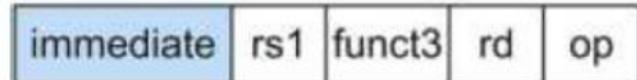
```
addi x10, x20, 0 // returns f (x10 = x20 + 0)
```

```
ld x20, 0(sp) // restore register x20 for caller
ld x6, 8(sp) // restore register x6 for caller
ld x5, 16(sp) // restore register x5 for caller
addi sp, sp, 24 // adjust stack to delete 3 items
```

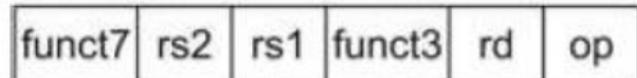
```
jalr x0, 0(x1) // branch back to calling routine
```

Adresleme Modları

1. Immediate addressing



2. Register addressing



Adresleme Modları

3. Base addressing



4. PC-relative addressing

