5c
Kontrolni ukazi

Kontrolni ukazi

- Kontrolni ukazi omogočajo spremembo vrstnega reda izvajanja ukazov
  - takim ukazom rečemo skoki
  - Zmožnost odločitev razlikuje računalnik od kalkulatorja!
- 2 vrsti skokov:
  - brezpogojni
    - vedno se izvede
    - omogoča preskok dela programa, pa tudi vrnitev nazaj
  - pogojni
    - izvede se, če je izpolnjen določen pogoj
    - omogoča pogojni preskok dela programa in končne zanke
- Kontrolni ukazi omogočajo vejitve in zanke
  - seveda pa tudi klice podprogramov ter poljubne skoke

- Skoki pri RISC-V:
  - Brezpogojni skok:
    - JAL (jump and link) – format J
    - JALR (jump and link register) – format I
  - Pogojni skoki (format B):
    - BEQ (branch if equal to), če rs1==rs2
    - BNE (branch if not equal zero), če rs1!=rs2
    - BLT, BLTU (branch if less than (unsigned)), če rs1 < rs2
    - BGE, BGEU (branch if greater or equal (unsigned)), če rs1 >= rs2
  - Pogojni skoki uporabljajo format B in *PC-relativno naslavljanje*
    - za bazni register je uporabljen PC

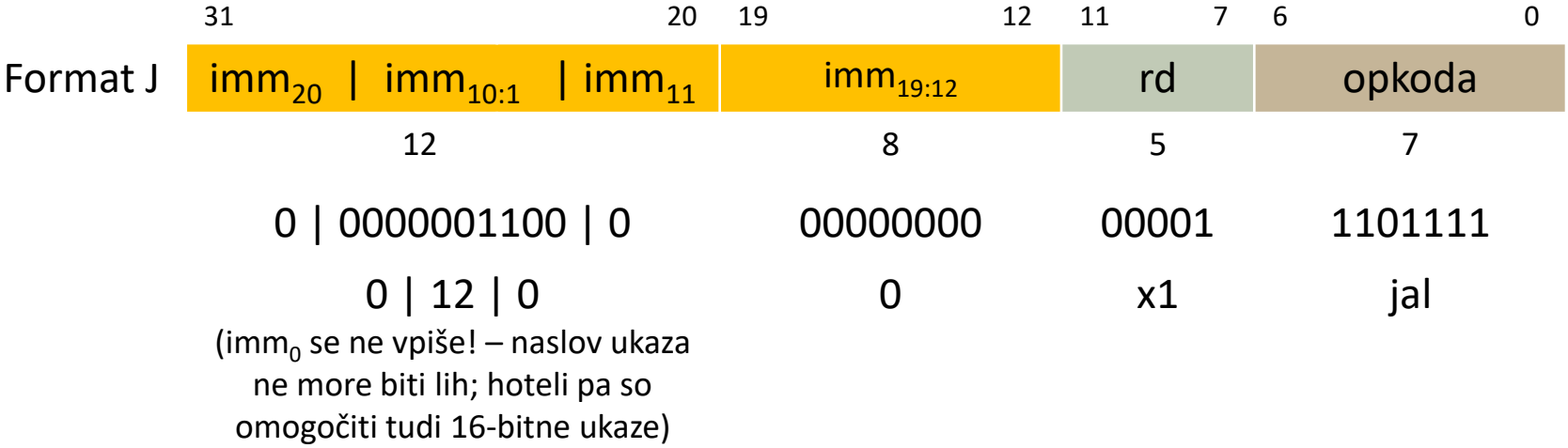
JAL (Jump and link):

jal rd, target # rd ← PC+4,
PC ← target = PC + se(2*imm20)

Zbirnik gornji ukaz prevede v `jal rd,imm20`

Pazi: target in imm ni ista stvar!

```
Npr.:  0x10          jal  x1, nekam   # x1 ← PC+4 (= 0x14 =20)
                      . . .           # PC ← nekam (= 0x28)
                      . . .           # imm20 = (0x28-0x10)/2 =
       0x28  nekam:   . . .           #       = (40-16)/2 = 12
```



- Če pišemo takojšnji operand:

```
jal rd, imm21 ,
```

je imm21 dejanska razlika, imm20 pa je imm21/2

  - Npr., jal x5, 20 skoči za 5 ukazov naprej,
    - imm21 = 20, imm20 = 10

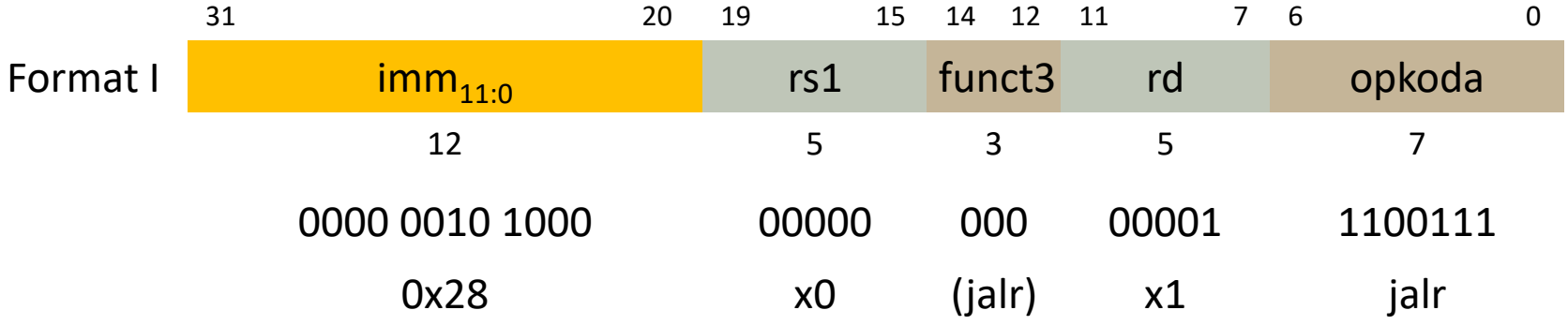
JALR (Jump and link register):

jalr rd, imm12(rs1)

rd ← PC+4,
PC ← target
= (rs1 + se(imm12)) & (-2)
zadnji bit je 0 (-2 = ..11110 = ~1)
(~ je 1'K)

rs1 omogoča skoke na zelo oddaljene procedure (naloži se ga predhodno z lui)

```
Npr.: 0x10          jalr  x1, nekam(x0)   # x1 ← PC+4 (= 0x14 =20)
                    . . .                 # PC ← nekam + x0 (= 0x28)
                    . . .                 #          imm12 = (0x28-0) = 40
      0x28  nekam:  . . .
```



Opomba: v simulatorju Ripes se ukaz JALR piše malo drugače!

jalr rd, rs, imm12

- Ukaz JAL lahko uporabimo tudi kot brezpogojni skok brez shranjevanja v rd (torej ‘linka’):
  - `jal x0, Oznaka`, kar dela psevdoukaz `j Oznaka`
- Ukaz JALR je koristen tudi kot ‘indirektni skok’
  - skočni naslov se da spreminjati s spreminjanjem vsebine registra
  - uporabno npr. pri stavku switch v jeziku C

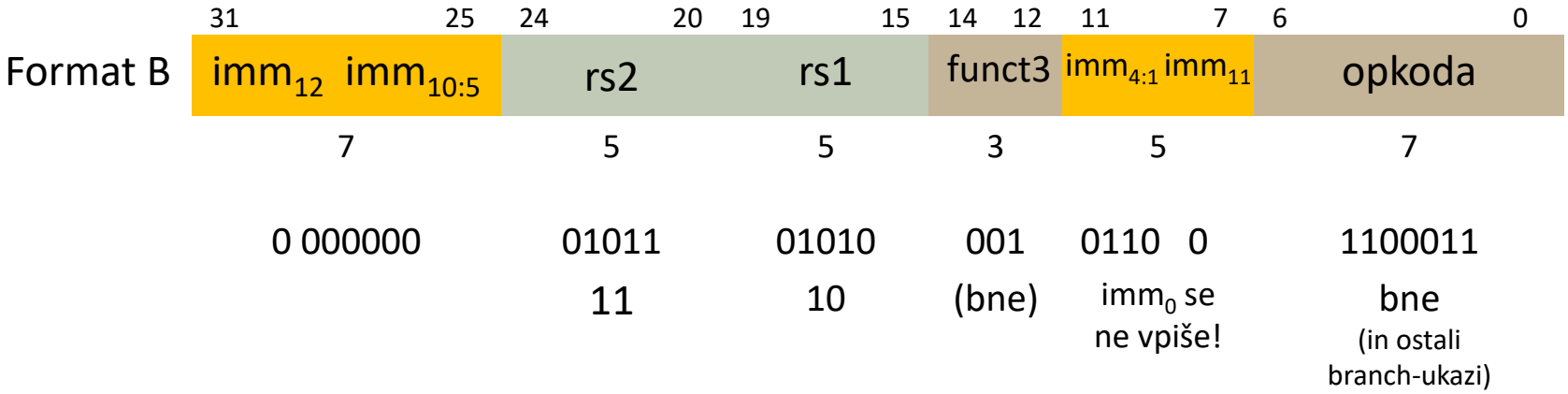
BNE (Branch if Not Equal to):

`bne rs1,rs2,imm13` (PC-relativno)

Pozor: imm_0 se ne vpiše v strojni ukaz
imm12 = $imm_{12:1}$ = (ciljni naslov – PC (ukaza bne))/2
Npr., imm12 = 10 pomeni imm13=20 in skoči za 5 ukazov naprej
Če v ukazu zapišemo oznako (labelo), zbirnik izračuna imm12 ← (label – PC)/2

Npr.:

```
        bne  x10, x11, 20          # če x10 != x11, potem PC ← PC + 12,
        . . .                      #       sicer PC ← PC + 4
        . . .
lab1:   . . .
```



Vejitve

```
if ( pogoj)
      blok1
else
      blok2
```

- Če pogoj ni izpolnjen, je treba skočiti na blok2

- Ukaz beq izvaja pogojni skok

```
beq rs1,rs2,LABEL
```

  - Če rs1 == rs2, CPE skoči na naslov LABEL
- Podoben ukaz je

```
bne rs1,rs2,LABEL
```

  - Če rs1 != rs2, CPE skoči na naslov LABEL

- Primer:

```
if (c < 5)
        a = b + 1;
else
        a = 2;
```

Predpostavimo x1: c, x2: a, x3: b

- Več možnosti:

```
        addi  t0, x0, 5     # t0 = 5
        bge   x1, t0, Else  # if (c >= 5) goto Else;
        addi  x2, x3, 1     # a = b + 1;
        jal    x0, Ven      # goto Ven;
Else:   addi  x2, x0, 2     # a = 2;
Ven:    naslednji ukaz      # naslednji ukaz
```

```
        slti  t0, x1, 5     # t0 = (c < 5)?
        beq   t0, x0, Blk2  # if (t0==0) goto Else;
        addi  x2, x3, 1     # a = b + 1;
        jal   x0, Ven       # goto Ven;
Else:   addi  x2, x0, 2     # a = 2;
Ven:    naslednji ukaz      # Ven: naslednji ukaz

        ...
```

Zanke

```
while ( pogoj)
     Blok;
```

Pogosto je zanka WHILE take oblike:

```
i = I1;
while ( i < I2)
{
    ...
    i = i + K;
}
```

V takem primeru lahko uporabimo tudi zanko FOR:

```
for (i = I1; i < I2; i=i+K)
{
    ...
}
```

Primer 1

Jezik C	Zbirni jezik za RISC-V

`sum = 0;`
`i = 5;`
`while ( i > 2)`
`{`
`  sum = sum + i;`
`  i--;`
`}`
	`addi  x1, x0, 0  # sum`
	`addi  x2, x0, 2  # 2`
	`addi  x3, x0, 5  # i`
	`Loop:  bge  x2, x3, Ven  # if(2>=i) Ven`
	`add  x1, x1, x3`
	`addi  x3, x3, -1`
	`jal x0, Loop`
	`Ven:  ...`

`sum = 0;`
`for ( i=5; i>2; i--)`
`  sum = sum + i;`
	isto kot zgoraj

`sum = 0;`
`i = 5;`
`do {`
`  sum = sum + i;`
`  i--;`
`} while ( i > 2);`
	`addi  x1, x0, 0  # sum`
	`addi  x2, x0, 2  # 2`
	`addi  x3, x0, 5  # i`
	`Loop:  add  x1, x1, x3`
	`addi  x3, x3, -1`
	`blt  x2, x3, Loop`

- Zanka do-while je v zbirnem jeziku enostavnejša od while
  - Seveda pa while in do-while v splošnem nista ekvivalentna!
    - pri slednjem se blok prvič vedno izvede

Primer 2

```
                //int a[10] = {5, 5, 5, 8, 2};
                //int ax = 5;
                //int i = 0;
                while (a[i] == ax)
                   i++;

                (x1: i, x2: ax, x3: bazni naslov a, tj. naslov od a[0], &a[0])

Loop:           slli    x4, x1, 2       # 4*i
                add     x4, x4, x3      # a + 4*i
                lw      x5, 0(x4)       # v x4 je naslov a[i]
                bne     x5,x2,Exit      # (a[i]==ax)? Exit
                addi    x1, x1, 1
                jal     x0, Loop
Exit:           ...
```

- Nekateri procesorji imajo samo pogojne skoke tipa ‘branch if equal zero’ in ‘branch if not equal zero’ (npr. MIPS) – za druge primerjave je potrebno nastaviti nek register (npr. s SLT) na 1 oz. 0 in potem izvesti pogojni skok – a pri tem sta potrebna 2 ukaza
- Drugi (npr. ARM) imajo zastavice, ki povedo, ali je bil rezultat neke operacije Z (zero) ali N (negative), tudi, ali je prišlo do preliva (V - overflow). Pogojni skok nato pogleda vrednost teh zastavic.
  - To pa vnaša podatkovne odvisnosti, kar ni dobro za realizacijo cevovoda.

Primeri kontrolnih ukazov

	Primer ukaza	Ime ukaza	Opis
JAL	x9, 84(x8)	Jump and link	x9 ← PC + 4, PC ← x8 + 84 (če je rd==x0, je jal navaden brezpogojni skok)
JALR	x2, 84(x8)	Jump and link register	x9 ← PC + 4, PC ← x8 + 84
BEQ	x7, x8, 0x8C	Branch if EQual to	če x7 == x8, potem PC ← PC + 0x8C, sicer PC ← PC + 4
BNE	x7, x8, 0x8C	Branch if Not Equal to	če x7 != x8, potem PC ← PC + 0x8C, sicer PC ← PC + 4
BLT	x5, x6, 24	Branch if Less Than	če x5 < x6, potem PC ← PC + 24, sicer PC ← PC + 4
BGE	x5, x6, 24	Branch if Greater or Equal than	če x5 >= x6, potem PC ← PC + 24, sicer PC ← PC + 4
BLTU	x5, x6, 24	Branch if Less Than, Unsigned	če x5 < x6 **(nepredznačeno)**, potem PC ← PC + 24, sicer PC ← PC + 4
BGEU	x5, x6, 24	Branch if Greater or Equal than, Unsigned	če x5 >= x6 **(nepredznačeno)**, potem PC ← PC + 24, sicer PC ← PC + 4

Sistemski ukazi

CSR – Control and Status Register

- 12-bitni takojšnji operand določa enega od možnih 4096 registrov CSR

Format	Opkoda	funct3	Ukaz		Opis (ze ... zero extended)
I	1110011	001	CSRRW	Atomic Read/Write CSR	rd ← ze(CSR), razen če rd==x0. CSR ← rs1
I	1110011	010	CSRRS	Atomic Read and Set bits in CSR	rd ← ze(CSR). Biti CSR, ki imajo v rs1 (=maska) 1, se postavijo na 1, razen če rs1==x0
I	1110011	011	CSRRC	Atomic Read and Clear bits in CSR	rd ← ze(CSR). Biti CSR, ki imajo v rs1 (=maska) 1, se brišejo (0), razen če rs1==x0
I	1110011	101	CSRRWI	CSRRW immediate	rd ← ze(CSR), razen če rd==x0. CSR ← ze($uimm_{4:0}$) (v polju rs1)
I	1110011	110	CSRRSI	CSRRS imm.	rd ← ze(CSR). Biti CSR, ki imajo v rs1 (=maska) 1, se postavijo na 1, razen če $uimm_{4:0}$==0
I	1110011	111	CSRRCI	CSRRC imm.	rd ← ze(CSR). Biti CSR, ki imajo v rs1 (=maska) 1, se brišejo (0), razen če $uimm_{4:0}$==0

- Sistemski ukazi shranjujejo podani register CSR v podani splošnonamenski register rd, v CSR pa naložijo novo vrednost (nove bite)
  - pogosto pa ne potrebujemo obeh ‘storitev’, ampak le eno
- Psevdoukazi za enostavnejše primere:

Psevdoukaz	Ukaz	Opis
CSRR rd, csr	CSRRS rd, csr, x0	samo branje CSR
CSRW csr, rs1	CSRRW x0, csr, rs1	samo pisanje CSR
CSRWI csr, uimm	CSRRWI x0, csr, uimm	samo pisanje CSR iz immed.
CSRS csr, rs1	CSRRS x0, csr, rs1	nastavljanje (set) bitov v CSR, kadar stare vrednosti ne rabimo
CSRC csr, rs1	CSRRC x0, csr, rs1	brisanje (clear) bitov v CSR, kadar stare vrednosti ne rabimo
CSRSI csr, uimm	CSRRSI x0, csr, imm	nastavljanje (set) bitov v CSR iz imm., kadar stare vrednosti ne rabimo
CSRCI csr, uimm	CSRRCI x0, csr, imm	brisanje (clear) bitov v CSR iz imm., kadar stare vrednosti ne rabimo

Preostali sistemski ukazi

- FENCE
  - pomnilniška pregrada zagotavlja, da se pred pomnilniškim dostopom dokončajo vsi morebitni prejšnji dostopi
  - to je pomembno predvsem v kontekstu večnitenja in spremenjenega vrstnega reda izvajanja ukazov (out-of-order)
- FENCE.I
  - pregrada za ukaze zagotavlja, da se pred branjem ukaza izvedejo vsa morebitna prejšnja pisanja
- ECALL (environment call)
  - implementacija sistemskih klicev
  - sistemski klici omogočajo uporabniku, do dobi usluge od jedra OS (privilegiran način delovanja), tipično dostop do HW (pomnilnik, disk, terminal, ...)
- EBREAK
  - med izvajanjem programa vrne kontrolo razhroščevalniku

Kaj mi bo zbirni jezik?

Za prevedbo iz višjenivojskega ali ‘srednjenivojskega’ jezika (C) v zbirni jezik poskrbi prevajalnik

- npr.: gcc, clang, lcc, IAR, Visual C, Watcom, ...
- danes so prevajalniki že zelo dobri

Kljub temu pa je včasih potrebno napisati kako zbirniško kodo – v takem primeru ni potrebno prevajati konstruktov višjega jezika v zbirni jezik

- Torej, ni treba začeti z višjenivojsko kodo in jo prevajati, temveč lahko neposredno pišemo v zbirnem jeziku, saj lahko kaj naredimo tudi bolj učinkovito

Primeri uporabe programiranja v zbirnem jeziku

- **Zagonski programi** - nizkonivojska koda v bralnem oz. bliskovnem pomnilniku za inicializacijo in testiranje strojne opreme pred zagonom operacijskega sistema, npr. BIOS
- **Deli jedra OS**, sistemski klici za določeno arhitekturo
- Nekateri jeziki in prevajalniki omogočajo vključevanje delov zbirniške kode (**inline** assembly), npr. za specifično CPE
- **Disassembly** – koda v zbirnem jeziku, ki jo je ustvaril prevajalnik ob prevajanju iz višjega jezika – lahko se uporabi za razhroščevanje in/ali optimizacijo
- V zgodnjih računalnikih je bilo možno v zbirnem jeziku napisati **bolj učinkovito** kodo.
- **Vzvratno inženirstvo** (Reverse engineering) - strojne kode ni težko prevesti (disassembler) v zbirni jezik. Na ta način je *v principu* možno rekonstruirati izvorno kodo.

- Zanimiva uporaba zbirnika je tudi preverjanje, ali je indeks polja znotraj obsega (bounds check)
  - Če želimo preveriti na čimkrajši način, ali je neka spremenljivka x v obsegu $0 \leq x < y$, lahko predznačeno število obravnavamo kot nepredznačeno.
  - Negativna števila v 2'K izgledajo kot velika števila v nepredznačenem formatu!
  - Tako nam nepredznačena primerjava x < y preverja tako tudi, če je x < 0
  - Npr.: če x20 >= x11 ali x20 < 0, potem skoči na oznako IndexOutOfBounds :

```
bgeu x20, x11, IndexOutOfBounds
```