RISC-V

Uvod

Što je RISC-V

- Open-source izvedba RISC arhitekture skupa naredaba (ISA – Instruction Set Architecture)

- VAŽNO:
  - RISC-V je SPECIFIKACIJA naredaba procesora
  - RISC-V nije stvarni procesor (komponenta)
  - RISC-V nije arhitektura procesora (ima puno različitih verzija koje implementiraju ISA)
- ISA namjerno napravljena da ne koristi ničije patente ni intelektualno vlasništvo kako bi bila potpuno slobodna za korištenje svima koji na temelju nje žele napraviti izvedbu procesora
- Zasnovana na nekim starim i široko poznatim načinima definiranja naredaba

Baze i proširenja

- Modularan dizajn koji se zasniva na nekoj izabranoj BAZI kojoj se mogu dodati PROŠIRENJA
- Samo neke baze i proširenja su finalizirana i potvrđena dok se na mnogima još radi i nisu dovršeni
- Neke BAZE:
  - RV32I Base ISA 32b Integer
  - RV64I Base ISA 64b Integer
- Neka PROŠIRENJA:
  - M Integer Multiplication and Division
  - A Atomic Instructions
  - F Single-Precision Floating-Point
  - D Double-Precision Floating-Point
  - Zicsr Control and Status Register (CSR)
  - Zifencei Instruction-Fetch Fence
  - G = IMAFDZicsr Zifencei, neki klasični “GPP ISA”

Izvedbe

- Izvedbe pojedinog procesora mogu se značajno razlikovati
  - Različit izbor baze
  - Različit izbor proširenja
  - Različita izvedba arhitekture puta podataka
  - Različita izvedba arhitekture pristupa memoriji
  - …

- Iz gornjeg se vrlo lako može zaključiti da je RISC-V još u svojim početcima i da je još puno nepoznanica prije nego će se moći reći da je to neki široko primjenjiv procesor opće namjene

- Trenutno RV koriste mnoge organizacije kao zamjenu za neke manje procesore ili kao pomoćne procesore u većim sustavima

FRISC

FER RISC

Kratka povijest FRISC-a

- FRISC (FER RISC) je naziv za arhitekturu RISC procesora projektiranih u okviru grupe RASIP na FERu i koristio se za istraživanje i u predmetu Arhitektura računala 1
- Prethodna generacija procesora bila je verzija FRISC-3 i zasnivala se na skupu naredaba koje smo mi sami definirali

- Nanovija verzija zasniva se na RISC-V ISA te je preskočena verzija FRISC-4 i prešlo se na verziju pod nazivom FRISC-V kako bi ime procesora na neki način bilo povezano sa nazivom ISA-e

FRISC-V

FRISC-V Primjer izvedbe procesora

- Do sada smo razmatrali arhitekturu postojećeg komercijalnog procesora
- Naš sljedeći cilj je (načelno) proći proces projektiranja novog procesora
- Stvarni postupak projektiranja te izvedba u FPGA/ASIC je nastavak AR1R u okviru drugih naših kolegija te završnih/diplomskih radova kao i istraživačkih projekata

FRISC-V

- U okviru istraživačkih aktivnosti FER Razvojnog centra za arhitekture i aplikacije računala visokih performanci (HPCAARC) već je oformljena grupa studenata koja radi na budućim proširenjima naredaba isto kao i na sklopovskoj implementaciji FRISC-V. Grupa će se proširivati novim zainteresiranim studentima (zainteresirani nas mogu osobno kontaktirati…)
- FER HPCAARC je član RISC-V International konzorcija
- FRISC-V u potpunosti podržava specifikacije prihvaćene od strane RISC-V konzorcija kako bi omogućili globalnu programsku kompatibilnost
- FRISC-V je u inicijalnoj verziji dizajniran da podržava RV-32I

- Za početak, proučiti ćemo RV32I ISA

RV32I

ISA

Registri

Ime	Alt.ime	Opis
x0	zero	hardwired zero
x1	ra	return address
x2	sp	stack pointer
x3	gp	global pointer
x4	tp	thread pointer
x5	t0	temporary / alternate link register
x6	t1	temporary
x7	t2	temporary
x8	s0/fp	saved register / frame pointer
x9	s1	saved register
x10	a0	function argument / return value
x11	a1	function argument / return value
x12	a2	function argument
x13	a3	function argument
x14	a4	function argument
x15	a5	function argument
x16	a6	function argument
x17	a7	function argument
x18	s2	saved register
x19	s3	saved register
x20	s4	saved register
x21	s5	saved register
x22	s6	saved register
x23	s7	saved register
x24	s8	saved register
x25	s9	saved register
x26	s10	saved register
x27	s11	saved register
x28	t3	temporary
x29	t4	temporary
x30	t5	temporary
x31	t6	temporary

Ime
pc

Registri

- 32 registra opće namjene

- r0 vrlo specifičan “registar” koji to u stvari nije nego predstavlja konstantu 0
  - Čitanje iz tog registra dati će 0
  - Pisanje u taj registar procesor će zanemariti

- Izveden je kako se nula kao jedna od najčešćih konstanti ne bi trebala kodirati kao broj, a poslužilo je i za situacije kad ne trebamo/želimo spremati rezultat

- pc – program counter je zaseban registar i ne spada u skup 32 registra opće namjene

Arhitektura

- Load-store arhitektura

- NEMA ZASTAVICA !!!

Osnovne AL naredbe

- AL naredbe sa 3 operanda
  - op1 = registar (rs1)
  - op2 može biti
    - registar (rs2) ili
    - neposredna vrijednost (imm/imm20)
  - rezultat = registar (rd)

Arithmetic		
	add	rd, rs1, rs2
	addi	rd, rs1, imm
	sub	rd, rs1, rs2
Logical		
	xor	rd, rs1, rs2
	xori	rd, rs1, imm
	or	rd, rs1, rs2
	ori	rd, rs1, imm
	and	rd, rs1, rs2
	andi	rd, rs1, imm
Shifts		
	sll	rd, rs1, rs2
	slli	rd, rs1, imm
	srl	rd, rs1, rs2
	srli	rd, rs1, imm
	sra	rd, rs1, rs2
	srai	rd, rs1, imm
Compare		
	slt	rd, rs1, rs2
	slti	rd, rs1, imm
	sltu	rd, rs1, rs2
	sltiu	rd, rs1, imm
Special arithmetic		
	lui	rd,imm20
	auipc	rd,imm20

AL naredbe FRISC-V

Naziv	*Asembler*	*Operandi*	*Operacija*
Aritmetičke naredbe			
add	add	rd, rs1, rs2	rd = rs1 + rs2
add immediate	addi	rd, rs1, imm	rd = rs1 + sign_ext(imm12)
subtract	sub	rd, rs1, rs2	rd = rs1 - rs2
Logičke naredbe			
bitwise AND	and	rd, rs1, rs2	rd = rs1 and rs2
bitwise AND immediate	andi	rd, rs1, imm	rd = rs1 and sign_ext(imm12)
bitwise OR	or	rd, rs1, rs2	rd = rs1 or rs2
bitwise OR immediate	ori	rd, rs1, imm	rd = rs1 or sign_ext(imm12)
bitwise exclusive OR	xor	rd, rs1, rs2	rd = rs1 xor rs2
bitwise XOR immediate	xori	rd, rs1, imm	rd = rs1 xor sign_ext(imm12)
Pomaci			
shift left logical	sll	rd, rs1, rs2	rd = rs1 << rs2[4:0]
shift left logical immediate	slli	rd, rs1, imm	rd = rs1 << imm[4:0]
shift right logical	srl	rd, rs1, rs2	rd = rs1 >> rs2[4:0]
shift right logical immediate	srli	rd, rs1, imm	rd = rs1 >> imm[4:0]
shift right arithmetic	sra	rd, rs1, rs2	rd = rs1 $>>_A$ rs2[4:0]
shift right arithmetic immediate	srai	rd, rs1, imm	rd = rs1 $>>_A$ imm[4:0]
Usporedbe			
set less than	slt	rd, rs1, rs2	if (rs1<rs2) rd=1 else rd=0
set less than immediate	slti	rd, rs1, imm	if (rs1<sign_ext(imm12)) rd=1 else rd=0
set less than unsigned	sltu	rd, rs1, rs2	if (rs1<rs2) rd=1 else rd=0
set less than immediate unsigned	sltiu	rd, rs1, imm	if (rs1<imm12) rd=1 else rd=0
Posebne AL naredbe			
load upper immediate	lui	rd,imm20	rd = [[imm20]000000000000]
add upper immediate to pc	auipc	rd,imm20	rd = pc + [[imm20]000000000000]

Osnovne AL naredbe: arithmetic/logical

- **add,sub**
- **and, or, xor**

Naziv	*Asembler*	*Operandi*	*Operacija*
Aritmetičke naredbe			
add	add	rd, rs1, rs2	rd = rs1 + rs2
add immediate	addi	rd, rs1, imm	rd = rs1 + sign_ext(imm12)
subtract	sub	rd, rs1, rs2	rd = rs1 - rs2
Logičke naredbe			
bitwise AND	and	rd, rs1, rs2	rd = rs1 and rs2
bitwise AND immediate	andi	rd, rs1, imm	rd = rs1 and sign_ext(imm12)
bitwise OR	or	rd, rs1, rs2	rd = rs1 or rs2
bitwise OR immediate	ori	rd, rs1, imm	rd = rs1 or sign_ext(imm12)
bitwise exclusive OR	xor	rd, rs1, rs2	rd = rs1 xor rs2
bitwise XOR immediate	xori	rd, rs1, imm	rd = rs1 xor sign_ext(imm12)

- Neposredna vrijednost (imm) je broj širine 12 bitova predznačno proširen na 32 bita

Osnovne AL naredbe: pomaci

- **sll** shift left logical
- **srl** shift right logical
- **sra** shift right arithmetic

Pomaci			
shift left logical	sll	rd, rs1, rs2	rd = rs1 << rs2[4:0]
shift left logical immediate	slli	rd, rs1, imm	rd = rs1 << imm[4:0]
shift right logical	srl	rd, rs1, rs2	rd = rs1 >> rs2[4:0]
shift right logical immediate	srli	rd, rs1, imm	rd = rs1 >> imm[4:0]
shift right arithmetic	sra	rd, rs1, rs2	rd = rs1 $>>_A$ rs2[4:0]
shift right arithmetic immediate	srai	rd, rs1, imm	rd = rs1 $>>_A$ imm[4:0]

- Raspon pomaka može biti samo 0-31
  - rs2 – uzima se samo 5 nižih bitova
  - imm – neposredna vrijednost samo nižih 5 bitova

Osnovne AL naredbe: usporedbe

set less than

- **slt, slti** signed
- **sltu, sltiu** unsigned

Usporedbe			
set less than	slt	rd, rs1, rs2	if (rs1<rs2) rd=1 else rd=0
set less than immediate	slti	rd, rs1, imm	if (rs1<sign_ext(imm12)) rd=1 else rd=0
set less than unsigned	sltu	rd, rs1, rs2	if (rs1<rs2) rd=1 else rd=0
set less than immediate unsigned	sltiu	rd, rs1, imm	if (rs1<imm12) rd=1 else rd=0

- Ispituje se da li je prvi operand manji od drugog i ako je manji postavlja se rd u 1 a ako nije postavlja se rd u 0

Ako (rs1 < rs2/imm) rd=1

Inače rd=0

Primjeri nekih osnovnih operacija

Zbrojiti podatke u x1 i x2 i sumu spremiti u x3

```
add x3,x1,x2
```

Podatak u x1 uvećati za 32

```
addi x1,x1,32
```

Podatak u x1 smanjiti za 4 (ne postoji naredba subi !)

```
addi x1,x1,-4
```

Postaviti najniži bit (bit na poziciji 0) u registru x5

```
ori x5,x5, 1
```

Dvojni komplement broja u x3

```
xori x3,x3,-1
addi x3,x3,1
```

Primjeri nekih osnovnih operacija

Pomaknuti podatak u x1 aritmetički u desno za 2 bita (koristeći neposrednu vrijednost za broj bita pomaka

```
srai x3, x1, 2
```

Pomaknuti podatak u x1 aritmetički u desno za broj bita zadan u registru x2 (uočite da processor uzima samo najnižih 5 bitova pa pomiče za 2 !!!)

```
addi x2,x0,0x82
sra x4, x1, x2
```

Usporediti brojeve u a1 i a2. Ako je broj u a1 manji od broja u a2 u s0 staviti 1 a inače u s0 staviti 0

```
slt s0,a1,a2
```

Posebne AL naredbe

Posebne AL naredbe			
load upper immediate	lui	rd,imm20	rd = [[imm20]000000000000]
add upper immediate to pc	auipc	rd,imm20	rd = pc + [[imm20]000000000000]

- **lui** load upper immediate
  - Viših 20 bita registra rd puni sa imm20 a nižih 12bita nulama
  - rd = [[imm20]000000000000]
  - Služi za APSOLUTNO adresiranje ili za kreiranje proizvoljne 32bitne vrijednosti
- **auipc** add upper immediate to pc
  - rd = pc + [[imm20]000000000000]
  - Služi za RELATIVNO adresiranje u odnosu na PC
- Ove naredbe u većini slučajeva koristimo sa dodatnom naredbom koja definira nižih 12b (objasniti će se kasnije na primjerima)

Neposredna vrijednost u registar

Koristeći x0 i neposrednu vrijednost u alu naredbi[-2048,2047]

```
addi    x1,x0, 234
addi    x1,x0, -736
addi    x1,x0, -0x800
ori     x1,x0, 0x567
```

Koristeći lui + addi (može se kreirati bilo koji 32bitni broj). Koriste se funkcije asemblera %hi i %lo

```
lui x1,%hi(0x12345678)
addi x1,x1,%lo(0x12345678)
```

Ovo assembler pretvori u:

```
lui x1, 0x12345
addi x1,x1, 0x678
```

Load/Store naredbe

Prijenos podataka			
load	l[*b*|*bu*|*h*|*hu*|*w*]	rd, imm(rs1)	rd = MEM[rs1+sign_ext(imm)]
store	s[*b*|*h*|*w*]	rs2, imm(rs1)	MEM[rs1+sign_ext(imm)] = rs2

- Load
  - **lb,lh,lw** čitanje bajta (predzn.prošireno na 32 b), poluriječi (predzn.prošireno na 32 b), riječi
  - **lbu,lhu** čitanje bajta i poluriječi BEZ proširenja predznaka (unsigned)
- Store
  - **sb,sh,sw** spremanje bajta, poluriječi, riječi

- Izračun adrese
  - **imm(rs1)**
  - Adresa = rs1 + neposredna vrijednost imm (predznačno proširena)

Učitavanje iz memorije u registar

Učitavanjem iz memorije **sa "bliske" adrese** [-2048,2047]

```
lw      x1, labela_podatka(x0)
```

Učitavnjem iz memorije **sa bilo koje adrese apsolutno**

```
lui x1,%hi(labela_podatka)
addi x1,x1,%lo(labela_podatka)

lw x1, 0(x1)
```

Učitavnjem iz memorije **sa bilo koje adrese PC-relativno**

```
auipc x1,%pcrel_hi(labela_podatka)      ;auipc x1,0x0
addi x1,x0,%pcrel_lo(labela_podatka)    ;addi  x1,x1,0x10

lw x1, 0(x1)
```

Asemblerske funkcije %hi i %lo

Zašto su nam te dvije funkcije korisne kod kreiranja proizvoljnog 32-bitnog podatka sa lui,addi?

- Ako je konst[11]=0 onda je sve jednostavno, npr želimo u x1 staviti 0x11111111:
  - u lui trebamo staviti dio 0x11111
  - U addi trebamo staviti 0x111

```
lui x1,0x11111
addi x1,x1,0x111
```

```
 11111000
+00000111
 11111111    -> KONSTANTA OK
```

Asemblerske funkcije %hi i %lo

Ali ako je konst[11]=1 (npr za 0x88888888):

- Ako u lui stavimo 0x88888
- Ako u addi stavimo 0x888

```
 88888000
+FFFFF888    -> bit [11]=1 predznačno proširenje
 88887888    -> KRIVA KONSTANTA!!!
```

Zato, ako je konst[11]=1 vrijednost koja se stavlja u lui mora biti za 1 veća kako bi se komenpenziralo predznačno proširenje kod addi.

```
 88889000
+FFFFF888    -> bit [11]=1 predznačno proširenje
 88888888    -> KONSTANTA OK
```

Asemblerske funkcije %hi i %lo

Da programer ne treba razmišljati o tome kad treba vrijednost u lui uvećati za 1 a kad ne, implementirane su dvije funkcije %hi i %lo koje to rade automatski

```
lui x1,%hi(0x11111111)          -> lui x1,0x11111
addi x1,x1,%lo(0x11111111)      -> addi x1,x1, 0x111

lui x1,%hi(0x88888888)          -> lui x1,0x88889
addi x1,x1,%lo(0x88888888)      -> addi x1,x1, 0x888
```

Asemblerske funkcije %pcrel_hi, %pcrel_lo

- Služe za jednostavniji izračun relativnog pomaka u odnosu na PC
- Isto kao %hi i %lo daju nam gornjih 20 i donjih 12 bita kako bi se formirala 32bitna vrijednost ali za izračun uzimaju ODMAK labele od trenutne vrijednosti PC

```
 org 0x100
        auipc x1,%pcrel_hi(lab)   -> auipc x1,0
        addi x1,x1,%pcrel_lo(lab) -> addi x1,x1,0x50
        lw a0, 0(x1)                       -> lw a0, 0(x1)
        ...

 org 0x150
lab dw 0x12345678
```

Asembler će izračunati da je labela na 0x150 a trenutna naredba na PC=0x100 pa će %pcrel_hi() u naredbu kodirati gornih 20 bita **razlike** 0x00000050

A u sljedećoj naredbi će %pcrel_lo() kodirati donjih 12 bita **razlike** 0x00000050

NAPOMENA (samo za objašnjenje ponašanja SSPARCSS-a, ne treba učiti):

u SSPARCSSu su ove f-je izvedene tako da:

- i kod %pcrel_lo i kod %pcrel_hi je parametar ciljna labela
- Funkcija %pcrel_hi će pored viših 20 bita razlike dodati I vrijednost adrese naredbe koja poziva %pcrel_lo kako bi SSPARCSS mogao
- naredba sa %pcrel_lo mora slijediti ODMAH nakon naredbe s %pcrel_hi

Ovo je nešto drugačije nego u std RV asembleru.

Primjer 64b zbrajanje i oduzimanje

- U memoriji se na lokacijama al, ah, bl i bh nalaze dva 64b broja. Izračunati njihovu sumu i preljev od 64b zbrajanja i rezultat spremiti na lokacije rl,rh i preljev.
- Na isti način rješiti i 64 bitno oduzimanje

- Kod Arm-a:

```
ADD     r4, r0,r2
ADC     r5, r1,r3
```

- RISC-V : NEMA ZASTAVICE !!! ???

Primjer 64b zbrajanje


```
        lw a4, al(x0)
        lw a5, ah(x0)
        lw a6, bl(x0)
        lw a7, bh(x0)

        add     t0, a4, a6      ; zbroji niže riječi
        sltu    t6, t0, a4      ; izračunaj preljev iz nižeg zbrajanja CL
        add     t1, a5, a7      ; zbroji više riječi
        sltu    t2, t1, a5      ; izračunaj preljev iz višeg zbrajanja CH_1
        add     t4, t1, t6      ; dodaj CL u višu sumu
        sltu    t3, t4, t1      ; izračunaj CH_2
        add     t6, t2, t3      ; kombiniraj CH_1 i CH_2 da se dobije CH

        sw t0, rl(x0)
        sw t4, rh(x0)
        sw t6, preljev(x0)
        halt

al      dw 0x80000001
ah      dw 0xFFFFFFF0
bl      dw 0x80000002
bh      dw 0xF
rl      dw 0
rh      dw 0
preljev dw 0
```

Primjer 64b odzimanje


```
        lw a4, al(x0)
        lw a5, ah(x0)
        lw a6, bl(x0)
        lw a7, bh(x0)

        sub     t0, a4, a6      ; oduzmi niže riječi
        sltu    t6, a4, t0      ; izračunaj posudbu iz nižeg oduzimanja Pos_L: t0>a4 =1
        sub     t1, a5, a7      ; oduzmi više riječi
        sltu    t2, a5, t1      ; izračunaj posudbu iz višeg oduzimanja Pos_H1
        sub     t4, t1, t6      ; oduzmi Posl_L
        sltu    t3, t1, t4      ; izračunaj posudbu iz višeg oduzimanja Pos_H2
        add     t6, t2, t3      ; kombiniraj Pos_H1 i Pos_H2 da se dobije Pos

        sw t0, rl(x0)
        sw t4, rh(x0)
        sw t6, posudba(x0)
        halt

al      dw 1
ah      dw 0x2
bl      dw 1
bh      dw 0x2
rl      dw 0
rh      dw 0
posudba dw 0
```

FRISC-V Pseudonaredba halt

- U prethodnim preimjerima vidjeli smo da na kraj naših programa stavljamo naredbu halt
- Naredba **halt** ne postoji kod Risc-V procesora.
- Ovo je pseudonaredba koju smo implementirali u SSPARCSS simulatoru da bi označili **kraj simulacije**
- Naredba će se prevesti u 0x00000000. Kad simulator prepozna taj kôd naredbe on će prekinuti simulaciju
- Kod procesora Arm koristili smo stvarnu naredbu procesora swi

Upravljačke naredbe:

Upravljačke naredbe			
jump and link	jal	rd, label	rd = PC+4; PC = label
jump and link register	jalr	rd, imm(rs1)	rd = PC+4; PC =(rs1 + sign_ext(imm12)) & 0xFFFFFFFE
branch	b[*eq*|*ne*|*lt*|*ge*|*ltu*|*geu*]	rs1, rs2, label	if (rs1 *condition* rs2) PC=label

- Dva tipa upravljačih naredaba
  - **jal, jalr** jump and link (register)
    - Bezuvjetni skok
  - **b** branch
    - Uvjetna grananja

Upravljačke naredbe: bezuvjetni skok

Upravljačke naredbe			
jump and link	jal	rd, label	rd = PC+4; PC = label
jump and link register	jalr	rd, imm(rs1)	rd = PC+4; PC =(rs1 + sign_ext(imm12)) & 0xFFFFFFFE
branch	b[*eq*|*ne*|*lt*|*ge*|*ltu*|*geu*]	rs1, rs2, label	if (rs1 *condition* rs2) PC=label

- **jal, jalr** naredbe služe za skokove
  - u funkcije
  - obične skokove

- **jal** sprema PC+4 (adresu sljedeće naredbe, povratnu adresu) u zadani registar rd i nakon toga izvodi skok

- Ako za rd stavimo r0 procesor neće spremiti povratnu adresu te će ovo postati obična naredba bezuvjetnog skoka

Upravljačke naredbe: bezuvjetni skok

Upravljačke naredbe			
jump and link	jal	rd, label	rd = PC+4; PC = label
jump and link register	jalr	rd, imm(rs1)	rd = PC+4; PC =(rs1 + sign_ext(imm12)) & 0xFFFFFFFE
branch	b[*eq*|*ne*|*lt*|*ge*|*ltu*|*geu*]	rs1, rs2, label	if (rs1 *condition* rs2) PC=label

jal i jalr se razlikuju po načinu kako definiramo adresu skoka:

jal rd, label

- relativan skok u odnosu na PC
- Stvarni oblik naredbe je jal rd, offset20:
    - rd = PC +4
    - PC=PC + sign_ext(offset20[20:1],0[0])
- Radi jednostavnijeg pisanja ovaj odmak računa assembler pa ćemo mi koristiti oblik pisanja jal rd, label

Upravljačke naredbe: bezuvjetni skok

jal i jalr se razlikuju po načinu kako definiramo adresu skoka:

jalr rd, imm(rs1)

- Skok na adresu zadanu registrom rs1 i neposrednim odmakom imm
  - rd = PC +4
  - PC = (rs1 + sign_ext(imm12)) & 0xFFFF FFFE
  - imm je širine 12b
  - Prije kopiranja u PC, bit 0 adrese procesor automatski postavlja u 0

Upravljačke naredbe: uvjetno grananje

Upravljačke naredbe			
jump and link	jal	rd, label	rd = PC+4; PC = label
jump and link register	jalr	rd, imm(rs1)	rd = PC+4; PC =(rs1 + sign_ext(imm12)) & 0xFFFFFFFE
branch	b[*eq*|*ne*|*lt*|*ge*|*ltu*|*geu*]	rs1, rs2, label	if (rs1 *condition* rs2) PC=label

- **b** branch

- Procesor usporedi vrijednosti u registrima rs1 i rs2 i ako je zadani uvjet zadovoljen skače na zadanu adresu
- beq branch if equal (rs1=rs2)
- bne branch if not equal (rs1≠rs2)
- blt branch if less than signed ($rs1 <_s rs2$)
- bge branch if greater than or equal signed ($rs1 \geq_s rs2$)
- bltu branch if less than unsigned ($rs1 <_u rs2$)
- bgeu branch if greater or equal unsigned ($rs1 \geq_u rs2$)

Upravljačke naredbe: uvjetno grananje

- relativan skok u odnosu na PC
- Stvarni oblik naredbe je b*xx* rs1,rs2, offset12:
    - PC=PC + sign_ext(offset12[12:1],0[0])
- Radi jednostavnijeg pisanja ovaj odmak računa assembler pa ćemo mi koristiti oblik pisanja b.. rs1,rs2,label
- Uvjete koji nisu definirani možemo dobiti zamjenom registara:
    - *Naredba* ***bgt*** x3,x2, labela ***ne postoji*** => blt x2, x3, labela
    - *Naredba* ***ble*** x3,x2, labela ***ne postoji*** => bge x2, x3, labela
    - *Naredba* ***bgtu*** x3,x2, labela ***ne postoji*** => bltu x2, x3, labela
    - *Naredba* ***bleu*** x3,x2, labela ***ne postoji*** => bgeu x2, x3, labela

Primjer

- Kopiranje većih podataka u blokovima
- Napisati program koji čita i uspoređuje podatke iz dva bloka pohranjena na adresama 0x6000 i 0x8000. Veći od dva podatka treba zapisati u odredišni blok na adresi 0x9000. Blokovi sadrže 5 podataka zapisanih u obliku 32-bitnog zapisa 2'k.

```
        lui     x1, %hi(IZV_A)          ; učitavanje adrese prvog
        addi    x1, x1, %lo(IZV_A)      ; izvorišta u x1
        lui     x2, %hi(IZV_B)          ; učitavanje adrese drugog
        addi    x2, x2, %lo(IZV_B)      ; izvorišta u x2
        lui     x3, %hi(ODR)            ; učitavanje adrese
        addi    x3, x3, %lo(ODR)        ; odredišta u x3
        addi    x4, x0, 5               ; upisivanje broja podataka u x4
petlja  lw      x5, 0(x1)               ; učitavanje podatka prvog bloka
        lw      x6, 0(x2)               ; učitavanje podatka drugog bloka
        addi    x1, x1, 4               ; pomicanje adrese prvog i
        addi    x2, x2, 4               ; drugog bloka
        blt     x5, x6, veci6           ; ako x6>=x5 skoci na veci6

veci5   sw      x5, 0(x3)
        addi    x4, x4, -1              ; smanjenje brojača podataka
        addi    x3, x3, 4               ; pomicanje adrese trećeg
        blt     x0, x4, petlja
        jal     x0, kraj

veci6   sw      x6, 0(x3)
        addi    x4, x4, -1              ; smanjenje brojača podataka
        addi    x3, x3, 4               ; pomicanje adrese trećeg
        blt     x0, x4, petlja
kraj    halt

        ORG    0x6000
IZV_A   DW   0x14, -4, -189, 11, 235667
        ORG    0x8000
IZV_B   DW   0x15, -4, 0x19, -0x100, -76
        ORG    0x9000
ODR     DS   20
```

Primjer

- Sličnost podataka
- Napisati program koji zbraja brojeve zapisane u registrima t0 i t1 i sprema ih u registar t2. Nakon toga treba usporediti t0 i t1 s registrom t2 da bi se odredilo koji od njih je sličniji broju t2. Sličniji broj je onaj koji ima više istih bitova na istim pozicijama i njega treba spremiti u registar t5. Ako imaju jednak broj istih bitova onda treba proizvoljno odabrati jedan od njih.

- Promatrajmo zbog jednostavnosti samo pet bitova: t0=01011, t1=01100 -> t2= 10111.
- U ovom slučaju sličniji je t0 jer ima dva ista bita xxx11, dok t1 ima samo jedan isti bit xx1xx.

```
        addi    t0, x0, 0b01011
        addi    t1, x0, 0b01100

        add     t2, t0, t1
        addi    t3, x0, 32      ;brojac
        xor     t4, t0, t2      ;0 -> isti
        xori    t4, t4, -1      ;1 -> isti
        addi    a0, x0, 0       ;brojac podudaranja t0


loop1   beq     t3, x0, break1
        andi    t5, t4, 1
        add     a0, a0, t5
        srli    t4, t4, 1
        addi    t3, t3, -1
        jal     x0, loop1
```

```
break1  addi    t3, x0, 32
        xor     t4, t1, t2
        xori    t4, t4, -1
        addi    a1, x0, 0       ;brojac podudaranja a1

loop2   beq     t3, x0, break2
        andi    t5, t4, 1
        add     a1, a1, t5
        srli    t4, t4, 1
        addi    t3, t3, -1
        jal     x0, loop2

break2  blt     a0, a1, manji

        addi    t5, t0, 0
        jal     x0, kraj

manji   addi    t5, t1, 0

kraj    halt
```

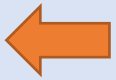
Primjer

- Zrcalni podatak
- Za 32-bitni podatak zapisan na adresi 100_{16} izračunati zrcalni podatak i spremiti ga umjesto originalnog podatka. Zrcalni podatak ima obrnut redoslijed bitova, odnosno ima zamijenjene bitove s nižih i viših pozicija. Na primjer, za 4-bitni podatak xyzq, njegov zrcalni podatak je qzyx (gdje oznake x, y, z i q predstavljaju pojedine bitove).

```
        lw      t1, POD(t0)
        addi    t2, x0, 32

loop    beq     t2, x0, break
        slli    t4, t4, 1
        andi    t3, t1, 1
        add     t4, t4, t3
        srli    t1, t1, 1
        addi    t2, t2, -1
        jal     x0, loop

break   sw      t4, POD(t0)
        halt

        org    0x100
POD     dw   0x1000000F
```

Funkcije

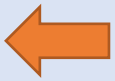

```
        jal     ra, func
```



Poziv funkcije

Povratna adresa u ra
(alternativni naziv za x1)

```
func    ...

        jalr    x0, 0(ra)
```



Povratak iz funkcije

Stog

INICIJALIZACIJA SP može se napraviti na razne načine

```
addi sp, x0, 0x100      ; verzija 1, sp=0x100
```

ili

```
lui sp, %hi(0x10000)    ; verzija 2, sp=0x10000
addi sp, sp, %lo(0x10000)
```

a može i na bilo koji drugi način….

```
addi    sp, x0, 0x100   ; verzija 3, sp=0x10000
slli    sp, sp, 8
```

Stog

- RISC-V ISA nije predvidjela posebne naredbe za čitanje/spremanje podataka na stog već se koristi jednostavna kombinacija dvije obične naredbe

- SPREMANJE na stog

```
addi    sp, sp, -4      ; smanjivanje SP
sw      x7, 0(sp)       ; spremanje podatka na stog
```

- ČITANJE sa stoga

```
lw      x7, 0(sp)       ; spremanje podatka na stog
addi    sp, sp, 4       ; uvećavanje SP
```

Primjer

- Potprogram za računanje izraza 128*A+B

Napisati potprogram koji prima dva ulazna parametra preko stoga (nazovimo ih A i B). Parametar A je onaj kojeg pozivatelj prvog stavlja na stog, a parametar B je onaj kojeg stavlja drugog.

Parametri su 32-bitni brojevi u zapisu NBC. Potprogram treba izračunati vrijednost izraza 128*A+B i vratiti rezultat preko registra x5.

Pretpostavite da neće doći do prekoračenja opsega od 32 bita. Potprogram treba čuvati stanja registara, a parametre treba uklanjati pozivatelj.

Glavni program treba pozvati potprogram pri čemu kao parametar A treba poslati vrijednost iz memorijske lokacije 0x1000, a kao parametar B podatak iz memorijske lokacije 0xE000. Rezultat treba spremiti na lokaciju 0x1F00.

```
glavni  addi    sp, x0, 0x100           ; inicijalizacija
        slli    sp, sp, 8               ; stoga na 0x10000

        auipc   x5, %pcrel_hi(POD_A)    ; upisivanje adrese pod A
        addi    x5, x5, %pcrel_lo(POD_A); u x5
        lw      x5, 0(x5)               ; i učitavanje A

        auipc   x6, %pcrel_hi(POD_B)    ; upisivanje adrese pod B
        addi    x6, x6, %pcrel_lo(POD_B); u x6
        lw      x6, 0(x6)               ; i učitavanje B

        addi    sp, sp, -4              ; stavljanje učitanih
        sw      x5, 0(sp)               ; parametara na vrh
        addi    sp, sp, -4              ; stoga
        sw      x6, 0(sp)               ;

        jal     ra, potp                ; poziv potprograma

        addi    sp, sp, 8               ; uklanjanje parametara sa stoga
        auipc   x6, %pcrel_hi(REZ)      ; upisivanje adrese rezultata
        addi    x6, x6, %pcrel_lo(REZ)  ; u x6
        sw      x5, 0(x6)               ; i spremanje rezultata
        halt
```


```
potp    addi    sp, sp, -4          ; spremanje konteksta
        sw      x6, 0(sp)           ;

        lw      x6, 0x4(sp)         ; učitavanje B
        lw      x5, 0x8(sp)         ; učitavanje A

        slli    x5, x5, 7           ; množenje A sa 128
        add     x5, x5, x6          ; zbroj 128*A + B

        lw      x6, 0(sp)           ; obnova konteksta
        addi    sp, sp, 4           ;

        jalr    x0, 0(ra)           ; povratak iz potprograma

        org     0x1000
POD_A   dw   0x002B47EF

        org     0x1F00
REZ     ds   4

        org     0xE000
POD_B   dw   0x0000F12C
```