Vežba 1
Uvod u RISC-V arhitekturu

1 Uvod

Na vežbama za predmet "Napredni mikroprocesorski sistemi" će biti opisane napredne tehnike dizajna procesora na primeru danas aktuelne RISC-V arhitekture skupa instrukcija (*eng. ISA - Instruction Set Architecture*). RISC-V je open-source (besplatna) specifikacija za arhitekturu procesora ali ne i za njegovu implementaciju. Ovo znači da niko nema vlasnička prava nad specifikacijom pa samim tim svako može napraviti svoju verziju procesora bez da traći često velike sume novca na dobijanje prava za korišćenje određenog seta instrukcija. RISC-V je inicijalno dizajniran za edukacione i naučno-istraživačke svrhe na *Univerzitetu Kalifornije* (*Berkely)* u SAD-u, ali nakon što je pokazao odlične performanse u prvobitnim implementacijama, izgleda kao obećavajući standard za budućnost industrije.

Na prvim vežbama ćemo krenuti od detalja RISC-V specifikacije i imaćemo priliku da izučimo instrukcije koje procesor podržava kroz interaktivni simulator. Nakon što se bolje upoznamo sa asemblerskim instrukcijama, napravićemo jednostavnu implementaciju osnovnog 32-bitnog seta instrukcija RV32I pomoću VHDL jezika. Krenućemo od jednostavne Single-Cycle implementacije, što znači da će se svaka instrukcija izvršavati tačno jednu periodu taktnog signala. Kako bismo dobili bolje performanse, modifikovaćemo prethodno pomenutu implementaciju u procesor sa protočnom obradom, a zatim prokomentarisati i razrešiti sve hazarde koji se mogu pojaviti.

Ovaj kurs prati knjigu:

"Computer Organization and Design RISC-V Edition: The Hardware Software Interface"

- David A. Petterson, UC Berkley

- John L. Hennesy, Stanford University

Za bolje razumevanje materijala se preporučuje da se pročita knjiga.

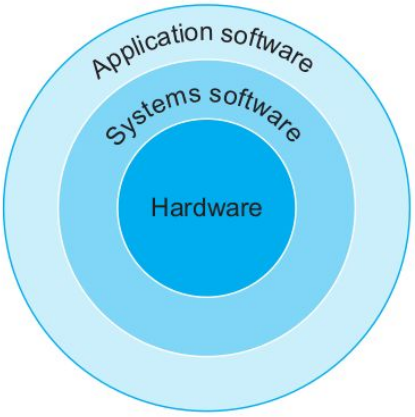
1.1 Slojevi programske podrške

Tipična današnja aplikacija u programskom jeziku visokog nivoa kao što su C, C++, Java, itd se sastoji od miliona linija koda i oslanja se na sofisticirane biblioteke koje implementiraju složene funkcije kao podršku aplikaciji. Kao što ćemo videti kasnije, hardver u računaru može da izvršava samo jako jednostavne instrukcije na niskom nivou. Kako bi se od kompleksnih aplikacija došlo do koda sastavljenog od jednostavnih instrukcija postoji nekoliko softverskih slojeva apstrakcije.

Kao što se može videti na slici 1.1 ovi slojevi su organizovani hijerarhijski sa aplikacijama u najširem prstenu, hardverom u najužem prestenu i nizom sistemskog softvera između njih. Sistemski softver je softver koji pruža servise koji su često neophodni za rad računara, a najbitniji su: operativni sistem, kompajleri, asembleri i loaderi. Operativni sistem je interfejs između korisničkih programa i hardvera, te obezbeđuje mnoge servise i nadzorne funkcije kao što su:

- rukovanje osnovnim ulazno/izlaznim operacijama
- alokacija memorije za aplikacije i prostora za skladištenje informacija
- planiranje procesorskog vremena

Primeri operativnih sistema koji se danas koriste su Linux, iOS i Windows.



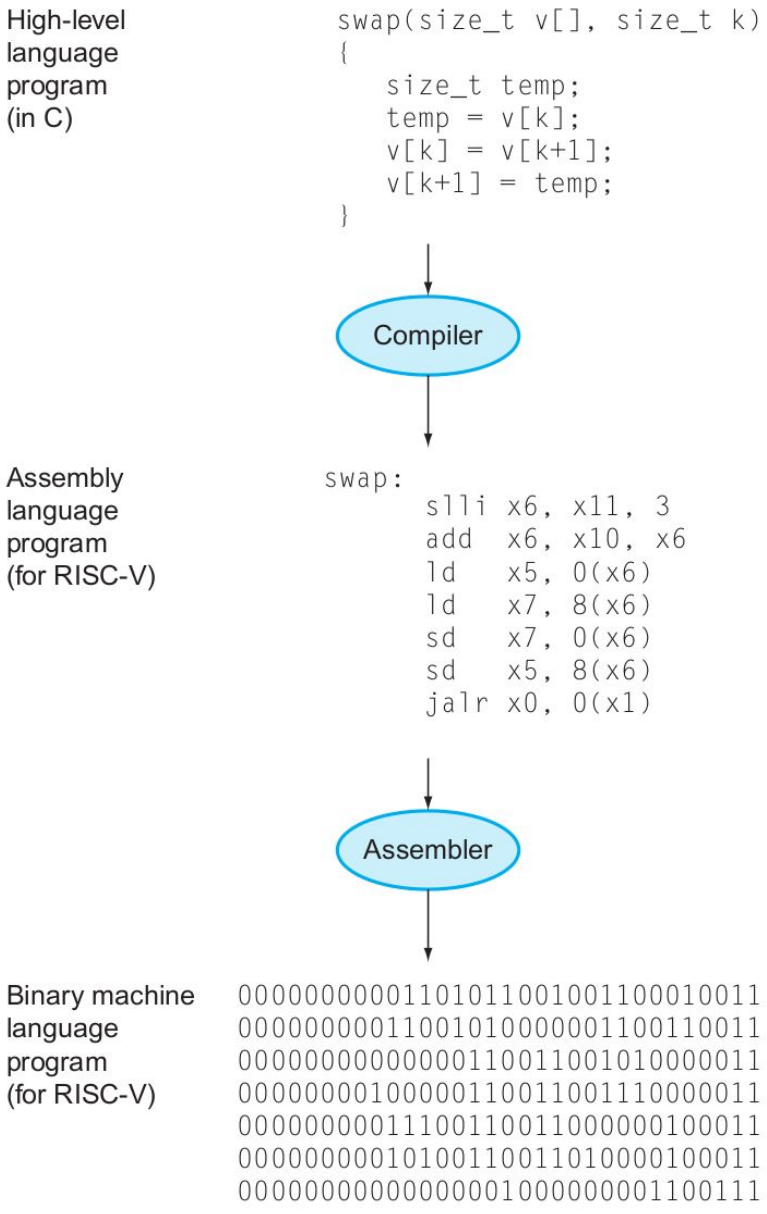
Slika 1.1: Softverski slojevi apstrakcije

Kompajleri izvršavaju podjednako bitnu funkciju: prebacivanje programa napisanog u prograskom jeziku opšte namene (C,C++, Java, Visual Basic...) u instrukcije koje hardver može da izvrši. Sagledajući kompleksnost današnjih jezika visokog nivoa i jednostavnost instrukcija koje procesor podržava, ovaj korak može biti izuzetno kompleksan.

Budući da računari mogu da rade samo sa binarnim vrednostima, instrukcije koje oni izvršavaju su kodovane kao niz bita u memoriji - mašinski jezik (*eng. machine language*). Prvi programeri su pisali programe u binarnom formatu, ali je ubrzo nastao simbolički način predstavljanja ovih instrukcija koji je više primeran ljudskom načinu razmišljenja. U početku su ove simboličke notacije konvertovane u binarni zapis ručno, ali je ubrzo napravljen prvi program pomoću kojega je postupak

automatizovan - asembler (*eng. assembler*). Po njemu je simbolički način predstavljanja instrukcija dobio naziv asemblerski jezik (*eng. assembly language*).

Iako je ovo bio veliki korak u apstrahovanju programa, asembler je zahtevao od programera da napiše jednu liniju koda za svaku računarsku instrukciju i samim tim da razmišlja kao računar. Shvatanje da program može biti zapisan u moćnijem jeziku nezavisnom od arhitekture računara je predstavljao jedan od najvećih uspeha u ranim danima računarstva. Upravo ovo je dovelo do nastanka programskih jezika opšte namene i kompajlera koji programe napisane u jednom takvom jeziku prevode u asemblerski jezik za konkretnu arhitekturu računara.



Slika 1.2 Prevođenje programa u mašinski kod

Programski jezici visokog nivoa imaju nekoliko prednosti u odnosu na asemblerski jezik. Prva je da dopuštaju programeru da razmišlja na prirodniji način, koristeći reči Engleskog jezika i algebarske notacije, što čini programe mnogo

čitljivijim. Takođe omogućavaju da se jezici dizajniraju u zavisnosti od njihove primene, npr: Fortran je dizajniran za naučne svrhe, Cobol za poslovne svrhe, Lisp za manipulaciju simbolima, itd... Druga prednost programskih jezika je poboljšana produktivnost programera. Ukoliko se ideja može predstaviti sa manje linija koda - vreme za razvoj programa će biti kraće. Treća i poslednja prednost je da programski jezici omogućavaju programima da budu nezavisni od arhitekture računara na kojima će se izvršavati, budući da će tu translaciju obavljati kompajler. Ove tri prednosti su toliko bitne da se danas jako malo programa piše u asembleru. 1975 godine su mnogi operativni sistemi bili napisani u asemblerskom jeziku jer su memorije bile male a kompajleri neefikasni. Današnji optimizacioni kompajleri mogu da generišu podjednako dobar program napisan u asemblerskom jeziku kao i eksperti u oblasti, a u slučajevima većih programa čak i bolji.

1.2 Arhitektura skupa instrukcija

Arhitektura skupa instrukcija (eng. *ISA, Instruction Set Architecture*) opisuje na koji način određeni procesor funkcioniše i koje su njegove mogućnosti. Ona opisuje registre koje će procesor imati kao i sve mašinske instrukcije koje će podržavati. ISA specificira do detalja šta koja instrukcija radi i na koji način će ona biti kodovana u memoriji. ISA predstavlja spregu između hardvera i softvera. Hardverski inžinjeri dizajniraju i modeluju kola koja će izvršavati dati skup instrukcija. Softverski inžinjeri pišu kod (operativne sisteme, kompajlere) bazirane na ovome skupu instrukcija.

Do danas je postojalo mnoštvo arhitektura u masovnoj primeni, pri čemu se dele u dve grupa:

RISC - Reduced Instruction Set Architecture - koristi se veći broj jednostavnih instrukcije kako bi se izvršio program, pri čemu se instrukcije izvršavaju u jednom mašinskom ciklusu. Dekodovanje instrukcija je brzo (fiksna širina kodovanja instrukcija) a implementacija protočne obrade laka. Implementacija procesora zahteva manji broj tranzistora. Veći akcenat se stavlja na softver (kompajler). Koristi registarsko - registarski model, pri čemu se aritmetičko-logičke operacije izvršavaju strogo nad registrima. Instrukcije LOAD i STORE se koriste za prebacivanje sadržaja iz memorije u registre i suprotno. Kodovi su često veliki i zazimaju mnogo prostora u RAM.

- ARM (ARM Holdings)
- SPARC (Sun - Oracle)
- PowerPC (IBM)
- MIPS
- Alpha (DEC)
- AVR (Atmel)

CISC - Complex Instruction Set Architecture - koristi se manji broj kompleksnih instrukcija kako bi se izvršio program, pri čemu se instrukcije izvršavaju više taktova. Dekodovanje i implementacija instrukcija je komplikovana (promenljiva širina kodovanja instrukcija), te zahteva veći broj tranzistora. Operacije nisu strogo registarsko-registarske, te je moguće da su operandi u memoriji. Veličina koda je manja.

- x86 (AMD, Intel)
- VAX (DEC)
- System 360 (IBM)

Svaki od prethodno pomenutih setova instrukcija zahteva plaćanje prava za korišćenje. Mnogi bitni detalji su često nepristupačni generalnoj populaciji i njihova široka primena dugi niz godina nosi mnoge nepogodnosti zbog potrebe da bude kompatabilan sa ranijim verzijama programa. Otkada su navedeni ISA napravljeni, dizajn računara je umnogome napredovao i nove tehnologije izrade čipova su promenile kakve se odluke donose pri dizajnu procesora.

RISC-V projekat je nastao kako bi rešio prethodno pomenute probleme. Cilj je bio da se napravi moderan skup instrukcija koji sadrži sve do danas najbolje ideje u dizajnu procesora. RISC-V je morao biti dosta jednostavniji i manji od onih koji su danas u upotrebi, a istovremeno praktičan za moderne primene i namenjen za dizajn što brže hardverske implementacije. Još jedan cilj je bio da se napravi "čista" (pure) RISC arhitektura gde će se svaka instrukcija izvršiti za tačno jedan takt, pa instrukcije moraju biti jednostavne i ograničene. Jedna od većih prednosti RISC-V je to da je ovaj set instrukcija otvorenog koda "open-source". Ovo znači da velike korporacije više ne bi mogli kontrolisati i naplaćivati korišćenje njihovog seta instrukcija. Ovakav pristup znači da bi mnoge firme mogle da se takmiče u implementaciji jednog seta instrukcija, što bi rezultovalo u mnogim pogodnostima koje se mogu primetiti kod drugih open-source projekata.

RISC-V dizajn nije jedinstven, potpuno specificiran ISA. Dizajneri su shvatili da postoje različite primene za procesore sa različitim ograničenjima. Na primer za određeni embeded procesor je možda neophodno da bude jeftin, pouzdan, jednostavan, i da ne zahteva podršku za operativni sistem, 64-bitne operacije ili više jezgara. RISC-V pristupa ovome problemu tako što omogućava više opcija koje se mogu uključiti u ISA koji će biti implementiran na procesoru. Time RISC-V postaje modularan ISA koji omogućava dizajneru da iz kompletne specifikacije izabere ekstenzije koje su potrebne za neku specifičnu primenu. Dizajner će odlučiti koje ekstenzije da implementira, a koje da odbaci a zatim će u dokumentaciji procesora naglasiti koji delovi ISA su implementirani i na koji način. Bitno je naglasiti da kada se završi sa radom na nekoj od ekstenzija, ona se “zamrzne” što znači da se nikada u budućnosti neće menjati. Ovo daje stabilnost RISC-V setu instrukcija koji garantuje da će jednom napisani kod biti komaptibilan sa svim budućim verzijama procesora.

Dakle, dokumentacija za RISC-V procesor mora da odgovori sledeća pitanja:

- Koje su širine registri procesora?
    - Odgovor: 32 bita, 64 bita ili 128 bita
- Koliko registara procesor poseduje?
    - Odgovor: 16 ili 32 registra
- Sa koliko bita su kodovane instrukcije?
    - Odgovor: 32 bita obavezno, 16-bitni format se opciono može uključiti
- Da li je uključen hardver za množenje i deljenje celobrojnih brojeva?
    - Odgovor: Da ili Ne
- Da li je uključena podrška za operacije sa pokretnim zarezom (floating point)?
    - Odgovor: Nisu podržane, jednostruka tačnost (single-precision) ili dvostruka tačnost (double-precision)

Konvencija za imenovanje RISC-V seta instrukcija:

Kao što je prethodno pomenuto, dizajner koji pravi procesor sa RISC-V arhitekturom bira koje će ekstenzije implementirati a koje ne. Kako bi se lakše opisalo šta je prisutno u datoj hardverskoj implementaciji svaka od ovih ekstenzija ima sebi pridruženo veliko slovo abecede.

Svaki RISC-V skup instrukcija kreće slovima RV a zatim je praćen širinom registara u datoj implementaaciji:

- RV32 32-bitni procesor
- RV64 64-bitni procesor
- RV128 128-bitni procesor

Nakon širine registara se određuju skupovi instrukcija koji će biti implementirani:

- I Osnovni set instrukcija za rad sa celim brojevima (*integer*)
- M Množenje i deljenje celobrojnih podataka
- A Atomičke operacije za interprocesnu sinhronizaciju
- F Operacije brojevima sa pokretnim zarezom, jednostruka preciznost (*float*)
- D Operacije brojevima sa pokretnim zarezom, dvostruka preciznost (*double*)

Skup instrukcija "I" je osnovni i neophodan je dok su ostali samo njegove ekstenzije.Na primer RV32IMAFD je procesor koji podržava sve prethodno opisane operacije. Ova verzija procesora se često implementira te ima skraćenicu G (RV32G).

Sledeća proširenja procesora su takođe podržane:

- S Nadzorni (*supervisor*) mod (podrška za rad sa operativnim sistemima)
- Q Četvorostruka (128-bitna) preciznost za operacije sa pokretnim zarezom
- C Kompresovane (16-bitne) instrukcije
- E Embeded mikroprocesor, broj registara smanjen na 16

U dokumentaciji za RISC-V ISA su takođe naglašene sledeće modifikacije koje sada nisu podržane ali će biti specificirane u budućnosti:

- L Decimalna aritmetika
- V Vektorske instrukcije
- P Pakovane SIMD (*eng. Single Instruction Multiple Data*) instrukcije
- B Operacije za manipulisanje bitima
- T Podrška za transakcioni memorijski model
- J Podrška za dinamičko prevođenje instrukcija
- N Podrška za prekide na korisničkom nivou

RISC-V dokumentacija (https://content.riscv.org) pruža mnoge uvide u odluke prilikom specificiranja dizajna. Savetujemo studenta da pročita dokumentaciju ukoliko ga zanimaju pojedinosti specifikacije, ili ukoliko se isti planira ozbiljnije baviti dizajnom RISC-V procesora.

Tri univerzalna principa hardverskog dizajna kojih su se držali autori RISC-V arhitekture su:

1. Jednostavnost nastaje iz regularnosti
2. Manje je brže
3. Dobar dizajn zahteva dobar kompromis

Na ovome kursu će biti implementiran 32-bitni procesor sa osnovnim setom instrukcija - RV32I.

1.3 Operandi RISC-V procesora

Za razliku od promenljivih u nekom programskom jeziku visokog nivoa, operandi instrukcija iz ISA su ograničeni na 32 registra koja se nalaze u procesoru. Po RISC-V specifikaciji se ova 32 registra opšte namene označavaju imenima x0 do x31. Samo je registar x0 specifičan jer uvek sadrži vrednost 0 koju je nemoguće promeniti. Ova dizajnerska odluka je donešena zbog potrebe da u se u svakom trenutku može iskoristiti operand sa vrednošću 0. Dakle, sve instrukcije procesora su na RTL (register transfer level) nivou, pa stoga u svakom trenutku mogu biti na raspolaganju samo 32 operanda. Aritmetičko-logičke instrukcije vrše aritmetičku ili logičku operaciju nad vrednostima bilo koja dva registra a zatim rezultat smeštaju u proizvoljni registar.

Na primer: instrukcija *add x5, x20, x6* sabira vrednosti registara x20 i x6 a zatim rezultat smešta u registar x5. Zašto onda samo 32 registra? Odgovor se krije u drugom od tri glavna principa u dizajnu hardvera - "Manje je brže"! Naime, veliki broj registara u procesoru može da uspori maksimalnu frekvenciju rada, iz razloga što se povećava propagaciono kašnjenje kroz registarsku banku u kojoj se oni nalaze. Drugi razlog je povećanje broja bita kojima se registar može adresirati u instrukciji. RISC-V instrukcije su fiksne širine od 32 bita što umnogome olakšava njihovo dekodovanje.

Procesor može da čuva jako malo informacija u registrima, i najčešće su to samo podaci koji se trenutno obrađuju. Iz ovog razloga postoji memorija koja može da čuva na milijarde programskih promenljivih. Budući da se aritmetičke instrukcije izvršavaju samo među registrima, RISC-V mora posedovati i instrukcije za premeštanje podataka između registara i memorije. Ove instrukcije se zovu instrukcije za prenos podataka (*eng. data transfer instructions*). Podaci koji se prebacuju mogu biti različitih širina pa se stoga definišu nazivi za određene širine podataka:

- bajt (*byte*) 8 bita
- polu reč (*halfword*) 16 bita
- reč (word) 32 bita
- dupla reč (*doubleword*) 64 bita (samo kod RV64 arhitektura)

Ove veličine su uzete zato što su standardi za predstavljanje određenih tipova podataka u programskim jezicima visokog nivoa (u C-u: char - 8 bita, short - 16 bita, int - 32 bita, long - 64 bita ...itd). Memorija je napravljena da bude bajt-adresibilna

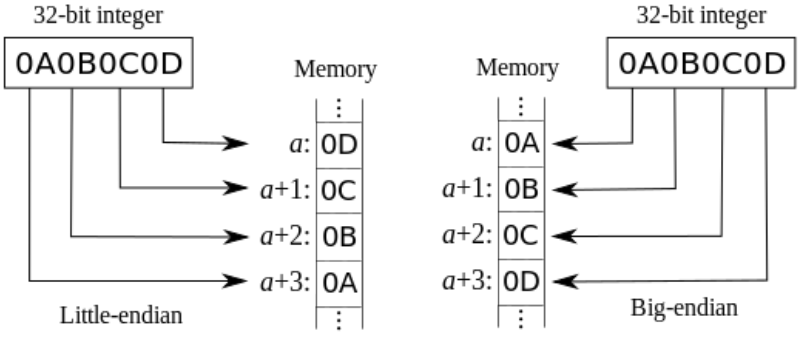
kako bi bio omogućen upis i čitanje podataka prethodno pomenutih širina. U ovom slučaju se reči nalaze na adresama koje su umnošci četiri (4 bajta = 32 bita), a polu-reči na adresama koje su umnošci dva (2 bajta = 16 bita).

Instrukcija za prenos podataka od memorije ka registru se zove *load*. U RISC-V setu instrukcija postoje različite verzije load instrukcije u zavisnosti od širine podatka koji se učitava (*lw- load word, lh - load halfword i lb - load byte*). Ova instrukcija zahteva ime registra u koji će podatak biti upisan, praćen registrom i konstantom za adresiranje memorije. Adresa sa koje će biti učitan podatak je vrednost registra sabrana sa vrednošću konstante (*offset*).

Na primer: instrukcija *lw x9, 8(x22)* preuzima 32 bita sa memorijske lokacije *x22+8* i tu vrednost smešta u registar *x9.* Ukoliko nam registar x22 sadrži adresu statički alociranog niza (pokazivač u C-u), možemo prolaziti kroz elemente niza samo uvećavajući konstantu u koracima od četiri. U ovome trenutku se može i primetiti korisnost registra x0 koji je uvek jednak nuli. Ukoliko u prethodnoj instrukciji registar x22 zamenimo sa x0, možemo adresirati memoriju apsolutno, samo pomoću konstante!

Komplementarna instrukcija, koja prenosi podatak iz registra u memoriju se zove *store.* Ekvivalentno, za različite širine podataka postoje varijacije: *sd- store doubleword, sw store word, sh - store halfword* i *sb - store byte*. Adresiranje memorije se vrši na isti način kao kod load instrukcije.

Bitno je pomenuti da se računari dele u one koji pojedinačne bajtove u memoriji adresiraju od bajta najnižeg značaja (LSB) ka bajtu najvišeg značaja - *little endian*, i one koji adresiraju od bajta najvišeg značaja ka bajtu najnižeg značaja - *big endian.* Ukoliko bi pokušali upisati reč 0x0A0B0C0D na neku memorijsku lokaciju "*a"* u memoriji, razlika u rezultatima se može primetiti na sledećoj slici. RISC-V je *little-endian* arhitektura.



Slika 1.3 Little endian i big endian arhitekture

U ovom trenutku je potrebno pomenuti da sem 32 registra opšte namene koji su prethodno opisani, svaki RISCV procesor ima i programski brojač (*PC, Program Counter*). Programski brojač čuva adresu instrukcije koja se treba izvršiti. U njemu se čuva adresa koja pokazuje na lokaciju u memoriji na kojoj se čuva kodirana instrukcija. Pri sekvencijalnom izvršavanju instrukcija se uvećava za 4 (4 bajta = 32

bita) kako bi pokazao na sledeću instrukciju. Njegovu vrednost mogu promeniti samo instrukcije uslovnih i bezuslovnih skokova.

1.4 Instrukcije i njihovo kodovanje

Instrukcije za RISCV procesore su 32 bitne, bilo da se radi o RV32, RV64 ili RV128 arhitekturi. Postoji 16-bitni kompresovani format za instrukcije, ali on neće biti predmet razmatranja ovog kursa. Instrukcije se izvršavaju isključivo nad registrima. RISCV je takozvana arhitektura sa tri adrese. Ovo znači da se za izvršavanje neke aritmetičke instrukcije moraju naznačiti tri registra: dva izvorišna (operandi) i jedan ciljni (destinacioni). Ciljni registar će sadržati rezultat izvršene operacije i u asemblerskom jeziku se često navodi prvi, odmah nakon oznake za instrukciju.

Na primer: kod operacije sabiranje add x10, x11, x12, vrši se sabiranje dva izvorišna registra x11 i x12, a zatim se rezultat smešta ciljni registar x10.

Za kodovanje instrukcija se koristi nekoliko različitih formata koji se mogu videti na slici 1.4. Instrukcije su podeljene na polja u zavisnosti od funkcije koju oni imaju u instrukciji:

- opcode operation code, kod instrukcije, vrši osnovnu podelu instrukcija
- funct3 dodatna 3 bita polju opcode, za detaljnije definisanje instrukcije
- funct7 dodatnih 7 bita polju opcode, za detaljnije definisanje instrukcije
- rs1 prvi izvorišni registar (operand)
- rs2 drugi izvorišni registar (operand)
- rd ciljni registar, u koji se smešta rezultat instrukcije
- imm immediate, polje koje sadrži konstantu

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Slika 1.4 Formati instrukcija RV32I arhitekture

Polja za adresiranje registara su 5-bitna kako bi se mogla adresirati sva 32 registra. Opcode uvek zauzima najnižih 7 bita u instrukciji. Ovi bitovi se prvi dekoduju u procesoru čime se saznaje o kojem se tipu instrukcije radi a samim tim i koji se od prethodno navedenih formata koristi. Konstante (immediate) imaju različite širine polja u zavisnosti od tipa instrukcije koji se koristi. U formatima instrukcija se najbolje vidi prvi princip dizajna hardvera “Jednostavnost nastaje iz regularnosti”. Instrukcije su fiksne širine i može se primetiti da se polja često nalaze na istom mestu unutar instrukcije. Ove osobine će umnogome olakšati dekodovanje instrukcija jer će se podaci u većini slučajeva nalaziti na istom mestu. To će zatim umanjiti bespotrebno rutiranje podataka kroz multipleksere, što kao rezultat daje malo propagaciono kašnjenje. Ova osobina je toliko uticajna da su biti za predstavljanje konstanti (immediate) često podeljeni na više polja ili postavljeni takvim redosljedom da se regularnost ispoštuje u što većoj meri.

1.4.1 Aritmetičko-logičke instrukcije

Aritmetičke i logičke operacije koriste R-format i prikazane su na slici 1.5. Sve instrukcije imaju isto opcode polje, dva izvorišna i jedan ciljni registar. Instrukcije se razaznaju na osnovu funct3 i funct7 polja.

- **ADD** (add) Sabira vrednosti registara rs1 i rs2 i rezultat smešta u registar rd. Predpostavlja se da su operandi dati u formatu komplementa dvojke (označeni brojevi) pa ne postoji razlika između sabiranja brojeva sa i bez predznaka.
- **SUB** (substract) Oduzima vrednost registra rs2 od registra rs1 i smešta rezultat u registar rd.
- **SLT** (set less than) Postavlja vrednost registra rd na 1 ukoliko je vrednost registra rs1 manja od rs2, u suprotnom će rd biti nula. Vrednosti operanada se interpretiraju u komplementu dvojke.
- **SLTU** (set less than unsigned) Postavlja vrednost registra rd na 1 ukoliko je vrednost registra rs1 manja od rs2, u suprotnom će rd biti nula. U ovom slučaju se vrednosti operanada interpretiraju u neoznačenom formatu (kao pozitivni brojevi).
- **XOR** (exclusive or) Vrši logičku operaciju “ekskluzivno ili” bit po bit (bitwise) nad vrednostima registara rs1 i rs2, rezultat smešta u registar rd.
- **OR** (or) Vrši logičku operaciju “ili” bit po bit (bitwise) nad vrednostima registara rs1 i rs2, rezultat smešta u registar rd.
- **AND** (and) Vrši logičku operaciju “i” bit po bit (bitwise) nad vrednostima registara rs1 i rs2, rezultat smešta u registar rd.

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Slika 1.5 Aritmetičko-logičke instrukcije

Za sve aritmetičko-logičke operacije sem oduzimanja postoji i *immediate* verzija instrukcije, gde se kao drugi operand, umesto vrednosti registra rs2 uzima konstanta koja se koduje u “*imm*” polju. Oduzimanje nije implementirano zato što se konstanta može prebaciti u negativan broj čime se operacija sabiranja (ADDI) pretvara u oduzimanje. Ove instrukcije koriste I-format i mogu se videti na slici 1.6. Kako bi se izvršila odgovarajuća operacija, 16-bitna konstanta se mora proširiti na 32-bita. Ukoliko se konstanta interpretira kao neoznačena vrednost (instrukcije: SLTU, XOR, OR, AND), ostatak bita se popunjava nulama. U suprotnom slučaju, ukoliko se konstanta iterpretira u formatu komplementa dvojke (instrukcije: ADD, SLT), najznačajniji (15-ti bit) se replicira kako bi se održao znak operacije.

31 20	19 15	14 12	11 7	6 0	
imm[11:0]	rs1	funct3	rd	opcode	I-type
imm[11:0]	rs1	000	rd	0010011	ADDI
imm[11:0]	rs1	010	rd	0010011	SLTI
imm[11:0]	rs1	011	rd	0010011	SLTIU
imm[11:0]	rs1	100	rd	0010011	XORI
imm[11:0]	rs1	110	rd	0010011	ORI
imm[11:0]	rs1	111	rd	0010011	ANDI

Slika 1.6 Aritmetičko-logičke instrukcije sa konstantom

1.4.2 Instrukcije pomeranja (shift)

Instrukcije pomeranja imaju R-format i mogu se videti na slici 1.7.

- SLL (shift left logic) Logičko pomeranje vrednosti registra rs1 za vrednost registra rs2 mesta u levo. Rezultat se smešta u registar rd.
- SRL (shift right logic) Logičko pomeranje vrednosti registra rs1 za vrednost registra rs2 u desno. Rezultat se smešta u registar rd.
- SRA (shift right arithmetic) Aritmetičko pomeranje vrednosti registra rs1 za vrednost registra rs2 u desno. Rezultat se smešta u registar rd.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA

Slika 1.7 Instrukcije pomeranja (shift)

Kao i u sličaju aritmetičko-logičkih instukcija, ove instrukcije takođe imaju *immediate* verzije, gde se umesto drugog operanda koristi konstanta. Razlika je što u ovom sličaju nema potrebe da se koristi I-format već se ponovono koristi R-format. Ovo je slučaj zato što je maskimalni broj pomeranja 32-bitne vrednosti 32, pa se ta vrednost može direktno proslediti pomoću 5 bita u polju rs2 - sada zvano *shamt* (shift amount). Ove instrukcije se od prethodnih razlikuju po polju opcode.

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

Slika 1.8 Instrukcije pomeranja (shift) sa konstantom

1.4.3 Instrukcije za preuzimanje podataka iz memorije (load)

Instrukcije za prebacivanje podataka iz memorije u registre (*load*) koriste I-format i mogu se videti na slici 1.9.

- LW (load word) Sa adrese *rs1+imm* u memoriji se preuzima 32-bitni podatak i smešta u registar rd
- LH (load halfword) Sa adrese *rs1+imm* u memoriji se preuzima 16-bitni podatak, proširuje na 32 bita i smešta u registar rd. Proširivanje se se vrši replikacijom najznačajnijeg (15. bita) kako bi se održao znak.
- LHU (load halfword unsigned) Sa adrese *rs1+imm* u memoriji se preuzima 16-bitni podatak, proširuje nulama na 32 bita i smešta u registar rd.
- LB (load byte) Sa adrese *rs1+imm* u memoriji se preuzima 8-bitni podatak, proširuje na 32 bita i smešta u registar rd. Proširivanjese se vrši replikacijom najznačajnijeg (7. bita) kako bi se održao znak.
- LBU (load byte unsigned) Sa adrese *rs1+imm* u memoriji se preuzima 8-bitni podatak, proširuje nulama na 32 bita i smešta u registar rd.

31 20	19 15	14 12	11 7	6 0	
imm[11:0]	rs1	funct3	rd	opcode	I-type
imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

Slika 1.9 *Load* instrukcije

1.4.4 Instrukcije za smeštanje podataka u memoriju (store)

Instrukcije za prebacivanje podataka iz registra u memoriju (*store*) koriste S-format i mogu se videti na slici 1.10. Primetite da je imm polje u S-formatu instrukcije rastavljeno na dva dela: imm[11:5] određuje gornjih 7 bita dok imm[4:0] određuje donjih 5 bita konstante.

- SW (store word) Vrednost registra rs2 se upisuje na adresu [rs1 + imm] u memoriji.
- SH (store halfword) Nižih 16 bita registra rs2 se upisuje na adresu [rs1 + imm] u memoriji.
- SB (store byte) Najnižih 8 bita registra rs2 se upisuje na adresu [rs1 + imm] u memoriji.

31 25	24 20	19 15	14 12	11 7	6 0	
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

Slika 1.10 *Store* instrukcije

1.4.5 Instrukcije za donošenje odluka, uslovni skokovi (branch)

Ono što razlikuje kompijuter od kalkulatora je mogućnost kontrolisanja toka izvršavanja programa. U programskim jezicima visokog nivoa je najčešće zastupljena “if-else” naredba, koja se nakon kompajliranja u asemblerskom jeziku zamenjuje jednostavnijim “if-goto” naredbama. U RISC-V asemblerskom jeziku se skup takvih instrukcija zove “branch” (grananje). Instrukcija grananja poredi dva operanda (rs1 i rs2) i u zavisnosti od zadovoljenosti uslova koji je određen tipom branch instrukcije, adekvatno menja vrednost PC registra. Ukoliko je uslov zadovoljen PC dobija vrednost *PC+imm* dok u suprotnom slučaju dobija *PC+4*. Dakle, kada je uslov zadovoljen, tok programa izvršava skok unapred ili unazad za vrednost *imm* polja, u suprotnom prelazi na sledeću instrukciju memoriji. Immediate polje se koduje u komlementu dvojke kako bi bilo moguće izvršiti skok unazad (neophodno za petlje).

- **BEQ** (branch if equal) PC dobija vrednost PC+imm ako je vrednost registra rs1 jednaka vrednosti registra rs2, u suprotnom slučaju PC dobija PC+4 (sledeća instrukcija).
- **BNE** (branch if not equal) PC dobija vrednost PC+imm ako je vrednost registra rs1 različita od vrednosti registra rs2, u suprotnom slučaju PC dobija PC+4.
- **BLT** (branch if less than) PC dobija vrednost PC+imm ako je vrednost registra rs1 manja od vrednosti registra rs2, u suprotnom slučaju PC dobija PC+4. Vrednosti operanada se interpretiraju u komplementu dvojke.
- **BGE** (branch if greater or equal) PC dobija vrednost PC+imm ako je vrednost registra rs1 veća ili jednaka vrednosti registra rs2, u suprotnom slučaju PC dobija PC+4. Vrednosti operanada se interpretiraju u komplementu dvojke.
- **BLTU** (branch if less than unsigned) PC dobija vrednost PC+imm ako je vrednost registra rs1 manja od vrednosti registra rs2, u suprotnom slučaju PC dobija PC+4. Vrednosti operanada se interpretiraju u neoznačenom formatu (kao pozitivni brojevi).
- **BGEU** (branch if greater or equal unsigned) PC dobija vrednost PC+imm ako je vrednost registra rs1 veća ili jednaka vrednosti registra rs2, u suprotnom slučaju PC dobija PC+4. Vrednosti operanada se interpretiraju u neoznačenom formatu (kao pozitivni brojevi).

31 25	24 20	19 15	14 12	11 7	6 0	
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

Slika 1.11 *Branch* instrukcije

U B-formatu instrukcija je kodovanje *immediate* polja još kompleksnije, jer sem što je polje podeljeno na dva dela, čak su i biti izmešani. Pažljivi čitlac će primetiti da u *immediate* polju postoje biti [12:1] dok će nulti bit uvek biti jednak nuli. Naime, instrukcije su 32-bitne pa se u memoriji nalaze na lokacijama deljivim sa četiri (0,4,8,12,16...itd). Iz ovog razloga kada se desi skok, želimo da uvećamo ili umanjimo adresu memorije (PC) za broj koji je deljiv sa četiri. Ako nam je *imm* polje uvek umnožak četiri, nema potrebe za kodovanjem bita [1:0] zato što će oni uvek biti jednaki nulama. Zbog čega onda postoji bit sa indeksom 1? Ako se vratimo na poglavlje 1.2 možemo primetiti da specifikacija RISCV mora da podržava proširenje “C” - za kompresovane 16-bitne instrukcije! U ovom slučaju se instrukcije nalaze u memoriji na adresama deljivim sa dva (0,2,4,6...itd) pa nam je u *imm* polju samo nulti bit uvek jednak nuli. Uklanjanje nultog bita nam daje prostor za dodavanje trinaestog, pa je time opseg skoka branch instrukcija duplo proširen.

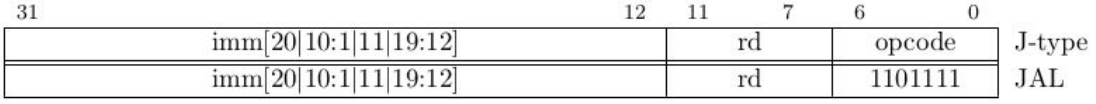
1.4.6 Instrukcije za rad sa procedurama, bezuslovni skokovi (jump)

Procedura ili funkcija je alat koji programeri koriste kako bi strukturisali programe. Korisćenje procedura pravi kod koji je lakši za razumeti, a i povećava se mogućnost ponovnog korišćenja već napisanog koda. U izvršavanju funkcije program mora pratiti sledećih 6 koraka:

1. Postavi parametre na mesto gde im procedura može pristupiti
2. Predaj kontrolu izvršavanja proceduri
3. Rezerviši memorijske resurse za proceduru
4. Izvrši željeni zadatak
5. Postavi rešenje izvršavanja procedure na mesto gde im originalni program može pristupiti
6. Vrati kontrolu izvršavanja na originalno mesto poziva procedure

Registri su najbrži elementi za skladištenje podataka, pa ih želimo koristiti što je više moguće. RISCV asembler koristi sledeću konvenciju za korišćenje registara. Registri x10-x17 se koriste za smeštanje parametara pri pozivu procedure kao i povratnih vrednosti pri povratku u originalni program. Registar x1 se koristi za smeštanje povratne adrese, tj. adrese na koju se program treba vratiti nakon što se procedura izvrši. To je adresa instrukcije koja sledi nakon instrukcije za poziv procedure.

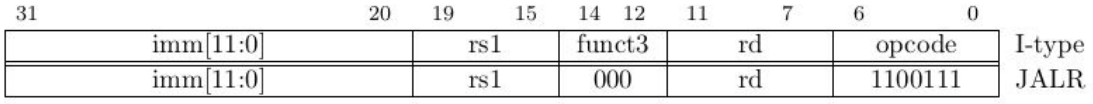
- **JAL** (jump and link) Koristi se za “poziv procedure”. Ova instrukcija izvršava dve stvari. Vrednost PC+4 (adresa sledeće instrukcije) se upiše u registar rd, nakon čega PC dobija vrednost kodovanu u *immediate* polju (početna adresa procedure). Ova instrukcija koristi specijalan J-format instrukcije sa 20-bitnim *immediate* poljem. Odredišni registar rd je registar u koji će se smestiti povratna adresa.



31 — 12	11 — 7	6 — 0	
imm[20|10:1|11|19:12]	rd	opcode	J-type
imm[20|10:1|11|19:12]	rd	1101111	JAL

Slika 1.12 *Jump and link* instrukcija

- **JALR** (jump and link register) Koristi se često za “povratak iz procedure”. Ova instrukcija izvršava dve stvari. U registar rd se upisuje vrednost PC+4, a zatim PC dobija vrednost registra *rs1+imm*. Ukoliko se koristi za povratak iz procedure, *imm* polje se postavi na nulu, a kao vrednost *rs1* se postavi registar u koji je sačuvana povratna adresa pri pozivu procedure pomoću JAL instrukcije.



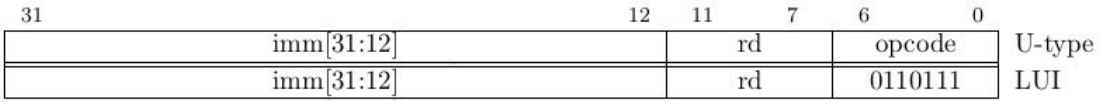
31 — 20	19 — 15	14 — 12	11 — 7	6 — 0	
imm[11:0]	rs1	funct3	rd	opcode	I-type
imm[11:0]	rs1	000	rd	1100111	JALR

Slika 1.13 *Jump and link register* instrukcija

1.4.7 Instrukcije za podržavanje velikih konstanti

Recimo da želimo da inicijalizujemo registar x17 na vrednost 42. Najlakši način za uraditi ovo je instrukcija *ADDI* gde bismo polje rs1 postavili na nulu (registar x0), na polje rd postavili 17 (registar x17) a na immediate 42 (addi x17,x0,42). U ovom slučaju se koristi I-format instrukcije gde je immediate polje 12-bitno, dok su registri 32-bitni. Šta raditi u slučaju da je vrednost na koju želimo postaviti registar mnogo veća od opsega 12-bitne *immediate* konstante? Zbog ovakvih slučajeva postoji instrukcija LUI. Ovde se najbolje vidi treći univerzalni princip dizajna hardvera: “Dobar dizajn zahteva dobar kompromis”. Neke arhitekture bi napravile promenljivu širinu instrukcija da bi razrešili ovakve probleme. Kako bi održao fiksnu 32-bitnu širinu instrukcije, u RISCV specifikaciji je napravljen kompromis da se u ovim situacijama moraju izvršiti dve instrukcije umesto jedne. LUI koristi U-format instrukcije.

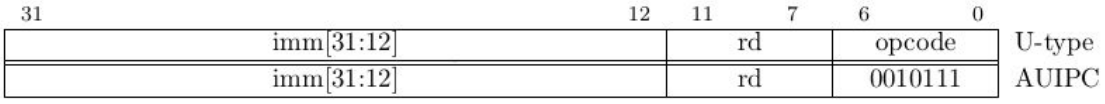
- **LUI** (load upper immediate) Na gornjih 20 bita destinacionog registra rd smešta vrednost immediate polja, dok donjih 12 bita postavlja na nule. Naime, napravi 32-bitnu konstantu kao konkatanaciju 20-bitnog *imm* polja i 12 nula a zatim je smesti u registar rd. Često se koristi odmah pre ADDI instrukcije koja na vrednost registra dodaje donjih 12 bita neke konstante.



Slika 1.14 Load upper immediate

Isti problem postoji kada se treba adresirati procedura čija je adresa van domašaja JAL instrukcije. Za razrešavanje ovog problema je definisana još jedna instrukcija koja koristi U-format.

- **AUIPC** (add upper immediata to PC) Napravi 32-bitnu konstantu kao konkatanaciju 20-bitnog *imm* polja i 12 nula, sabere je sa vrednošću PC-ja i rezultat smesti u registar rd. Ova instrukcija se koristi za PC-relativno adresiranje memorije. U kombinaciji sa addi instrukcijom moguće je adresirati bilo koji bajt u okviru 4GiB opsega PC-ja.



Slika 1.15 *AUIPC* instrukcija

1.5 Registri

RISC-V specifikacija definiše 32 registra opšete namene x0-x31. Jedino se izdvaja registar x0 po osobini da uvek ima vrednost nula. Iako se u asemblerskom jeziku registri x1-x31 mogu koristiti proizvoljno u svim instrukcijama, RISC-V kompajleri često koriste neke od registara na isti način i za specifičnu namenu, pa stoga svi registri imaju i alternativan naziv. Alternativni nazivi registara i njihova funkcija se mogu videti na slici 1.16.

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

Slika 1.16 Registri RISC-V procesora

Registar x0 se u asembleru može označavati kao “zero” budući da uvek ima vrednost nula. Kao što je pomenuto u prethodnom poglavlju, za povratnu adresu poziva procedure se često koristi registar x1, i ukoliko se u instrukcijama JAL i JALR ne specificira u kom registru se nalazi povratna adresa, asembler će predpostaviti da se radi o registru x1 (ra - return address) i pri asembliranju će u mašinskom kodu upotpuniti instrukciju shodno a time. Svaki program mora pamtiti vrednost pokazivača steka kako bi se pravilno izvršavale instrukcije push i pop. Za tu namenu je rezervisan registar x2 (sp - stack pointer) koji se često na početku programa inicijalizuje na odgovarajuću vrednost u memoriji koja predstavlja vrh steka. Global pointer (gp, x4) se koristi za pokazivanje na statičke promenljive u programu, a thread pointer (tp, x5) se koristi za pamćenje thred-a (niti) koji se trenutno izvršava. Registri x5-x7 i x28-x31 se koriste za lokalne promenljive koje neće biti sačuvane pri pozivu procedure te imaju alternativan naziv t0-t6 (temporary). Registri x8-x9 i x18-x27 se koriste za promenljive koje će biti sačuvane pri pozivu procedure te imaju alternativna imena s0-s11 (saved). Registri x10-x17 su takođe pomenuti u prethodnom poglavlju i koriste se za prosleđivanje argumenata proceduri kao i za povratnu vrednost po završetku iste.

1.6 Simulator

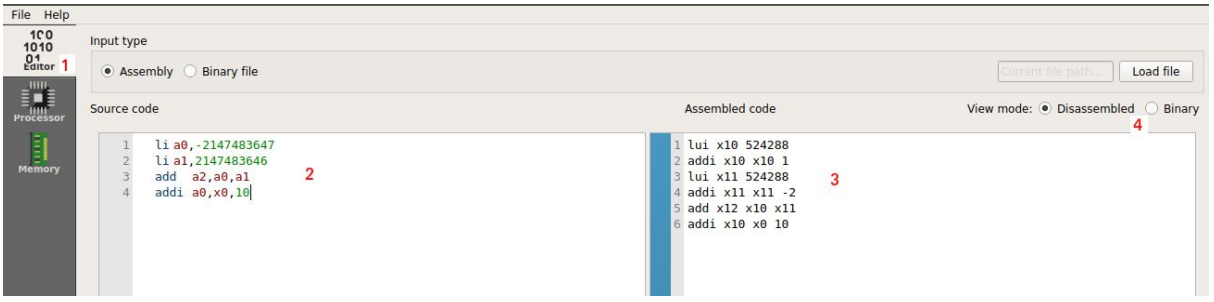
Za interaktivnu simulaciju RISC-V procesora kao i asembliranje koda se može sa sledećeg hyperlinka skinuti “Ripes” program:

https://github.com/mortbopet/Ripes/releases

Simulator se sastoji od tri glavne kartice koji se mogu selektovati na levoj strani:

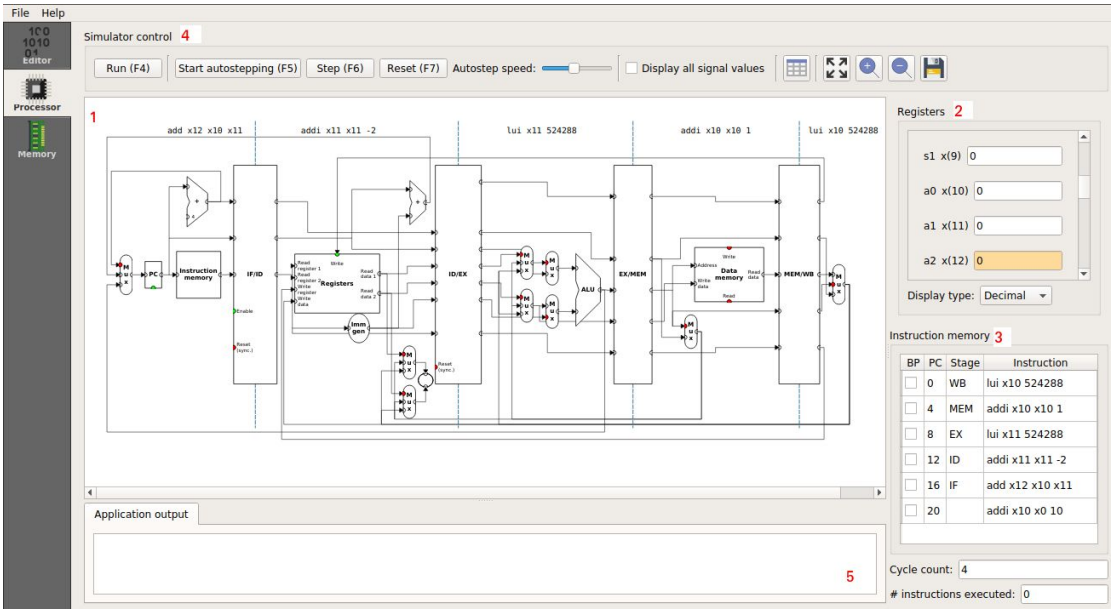
- Editor
- Processor
- Memory

“Editor” prozor se sastoji od 2 tekst editora. Na levoj strani [2] se može pisati asemblerski kod, dok će se na desnoj strani [3] u realnom vremenu isti kod asemblirati i prikazati u kao mašinski kod koji se sastoji samo od instrukcija podržanih u ISA. Mašinski kod se može prikazati u dva formata: simboličko ili binarno, što se može izabrati klikom na dugme “View mode” [4].



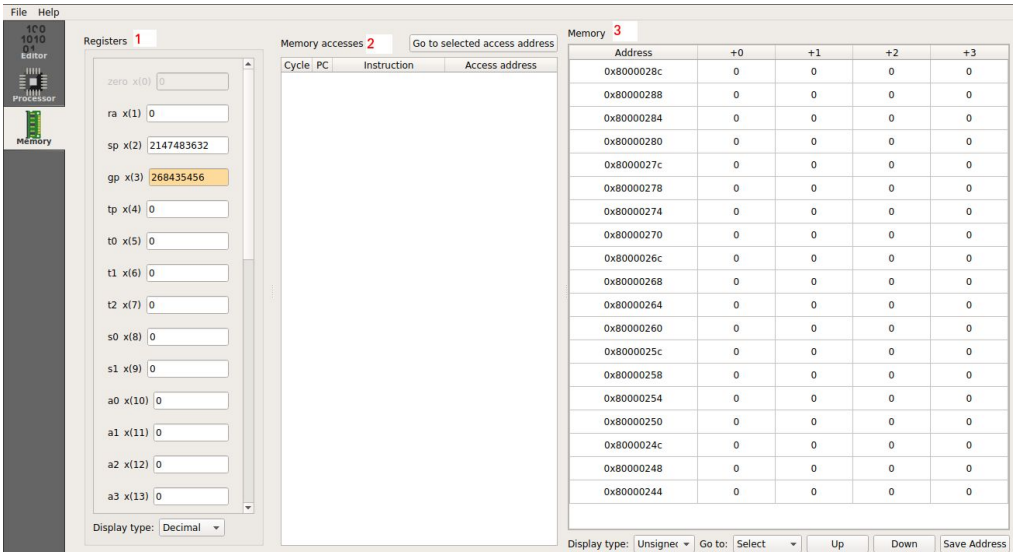
Slika 1.17 “Editor” prozor simulatora

U centru “Processor” prozora se nalazi jedna implementacija procesora sa protočnom obradom [1]. Iznad svake faze protočne obrade će biti prikazana instrukcija koja se trenutno izvršava. Crvenom tačkom su označeni ulazi mltipleksera koji se tenutno propuštaju na izlaz, a zelenom tačkom kontrolni ulazi koji su aktivni. Sa desne strane su prikazane trenutne vrednosti registara [2] i odmah ispod, memorija za instrukcije sa oznakama faza protočne obrade za trenutno simulaciono vreme. Na vrhu “processor” prozora nalaze se kontrole izvršavanja simulacije [4]. Program se može izvršiti u celosti (F4), izvršavati korak po korak (F5, F6) i resetovati (F7). U donjem desnom uglu [5] se nalazi brojač perioda takta koje su protekle od početka simulacije, kao i brojač izvršenih instrukcija. Simulator uvek izvršava program koji je napisan u “editor” prozoru. Za početak se preporučuje da studenti izvršavaju čitav program (F4) ukoliko nisu upoznati sa protočnom obradom.



Slika 1.18 “Processor” prozor simulatora

“Memory” prozor prikazuje trenutnu vrednost svih memorijskih elemenata. Na levoj strani [1] su prikazane vrednosti registara, a na desnoj strani [3] je prikazano trenutno stanje memorije. Na sredini prozora [2] će biti izlistane sve memorijske transakcije (npr. *load* i *store* operacije).



Slika 1.19 “Memory” prozor simulatora

1.7 Pisanje programa

Asemblerski jezik je nivo asptrakcije iznad mašinskog koda te se pisanje programa može znatno zakomplikovati u odnosu na jednostavne instrukcije koje su prethodno opisane. Zvanični opis asemblerskog jezika se može naći na sledećem hyperlink-u: https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md

Budući da cilj ovog kursa nije asembler već dizajn procesora, u pojedinosti asemblerskog jezika se neće zalaziti. Preporučuje se da se pre pisanja prvog programa prođe kroz uputstvo na datom linku bez zalaženja u detalja.

Za sve aritmetičko-logičke instrukcije se prvo piše oznaka instrukcije, zatim odredišni registar praćen operandima. Za ime registara se mogu koristiti svi nazivi opisani u poglavlju 1.5. Oznake mogu biti odvojene zarezom ili *space* karakterom. Konstante se mogu pisati u decimalnom, heksadecimalnom ili binarnom formatu (15, 0xf, 0b1111). Primeri instrukcija:

```
add x10, x11, x12
addi x12 x13 5
xor x5, x0, x8 = xor t0, zero, s0
```

Bitno je razumeti da asembler implementira dodatne (pseudo) instrukcije koje se na jednostavan način mogu prebaciti u jednu ili više mašinskih instrukcija. Jedan primer ovoga je instrukcija “load immediate” (npr. *LI x17, 5*) koja postavlja vrednost registra na neku konstantu. Ona će biti direktno prevedena u *ADDI* instrukciju gde je registar rs1=x0.

```
li x17, 5 = addi x17, x0, 5
```

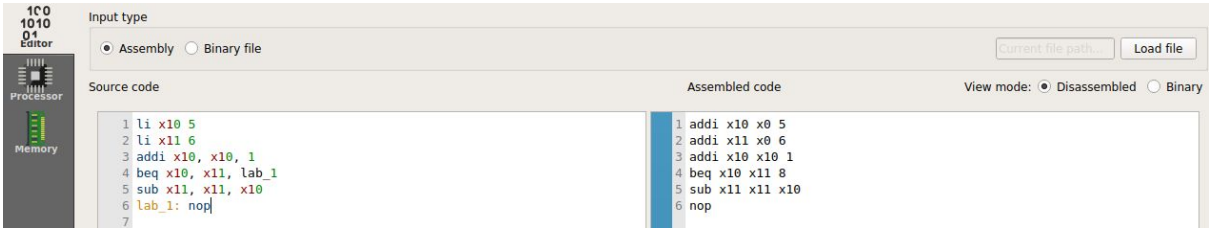
U slučajevima velikih konstanti ova instrukcija će biti asemblirana u dve instrukcije: *LUI* i *ADDI* (ovo je objašnjeno u poglavlju 1.4.7).

```
li x17, 1234567 =  lui x17 li x17, 302
                   addi x17 x17 1671
```

Još jedna prednost sa kojom smo se stretali na ranijim kursevima je da se pri skokovima prepusti asembleru da izračuna tačne vrednosti *imm* polja, a da se u programu koriste *labele*. Primer upotrebe labele:

```
1   li x10 5
2   li x11 6
3   addi x10, x10, 1
4   beq x10, x11, lab_1
5   sub x11, x11, x10
6   lab_1: nop
```

Kada se izvrše prve tri linije koda, registri x10 i x11 će imati iste vrednoati. Beq instrukcija u liniji 4 će u ovom primeru skočiti na liniju 6 označenu labelom *lab_1* te se instrukcija u liniji 5 neće izvršiti. Pri asembliranju programa će se izračunati da se labela nalazi dve instrukcije nakon skoka, te će se u mašinskom kodu *lab_1* zameniti brojem 8 (videti sliku 1.20)



Slika 1.20 Primer korišćenja labela

1.7 Zadaci

Kako bi se bolje upoznali sa mašinskim instrukcijama koje će biti implementirane na ovom kursu, za vežbu rešiti sledeće zadatke koristeći samo instrukcije iz RV32I ISA.

1) U registre t0, t1 i t2 postaviti vrednosti 2, 4 i 5, a zatim izračunati sledeće izraze i rezultate smestiti u registre s0, s1, s2.
    a) s0 = t0 + t1 - t2
    b) s1 = 1 + t0 + t1 * t2
    c) s2 = 2 ^ t2
2) U registar t0 učitati neki pozitivan broj. Izračunati njegov faktorijal a zatim rezultat smestiti u registar s0.
3) Napisati “proceduru” za računanje minimuma tri broja. Brojeve postaviti u registre namenjene za argumente, a zatim iskoristiti JAL za poziv procedure a JALR za povratak iz iste. Labelu koja će predstavljati proceduru proizvoljno imenovati.