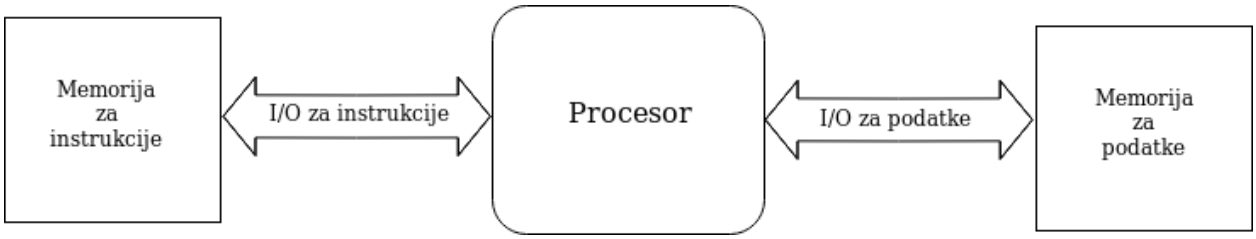
# Vežba 2
# Hardverska implementacija Single cycle RISC-V arhitekture

## 1. Uvod

Na prethodnim vežbama objašnjen je RISC-V set instrukcija i kako se na osnovu njega može napisati jednostavan program, dok će u nastavku biti objašnjeni osnovni principi i tehnike implementacije procesora koji podržava taj set instrukcija. No, da bi se oni razumeli neophodno je prvo razumeti funkcionalnost procesora počevši od njegove pojednostavljene predstave na visokom nivou abstrakcije.

Kao što je već pomenuto, procesor izvršava program predstavljen pomoću niza instrukcija, pri čemu se prilikom izvršavanja generišu određeni rezultati ili međurezultati. Da bi procesor mogao da izvršava program on mora da ima pristup memorijskim lokacijama na kojima je program smešten i mora da ima pristup memorijskim lokacijama u koje može da smešta, odnosno iz kojih može da preuzima određene podatke. Na slici 1 je prikazana konfiguracija koja ilustruje prethodno opisano:




Slika 1. Pojednostavljena predstava procesora na visokom nivou apstrakcije

Konfiguracija sa slike 1 ilustruje da procesor komunicira sa dve memorije. U prvu se smeštaju instrukcije, a u drugu podaci, pri čemu se komunikacija obavlja preko dva interfejsa:

- Interfejsa za komunikaciju sa memorijom za instrukcije, koji mora da obezbedi samo mogućnost čitanja
- Interfejsa za komunikaciju sa memorijom za podatke, koji mora da obezbedi i čitanje i pisanje.

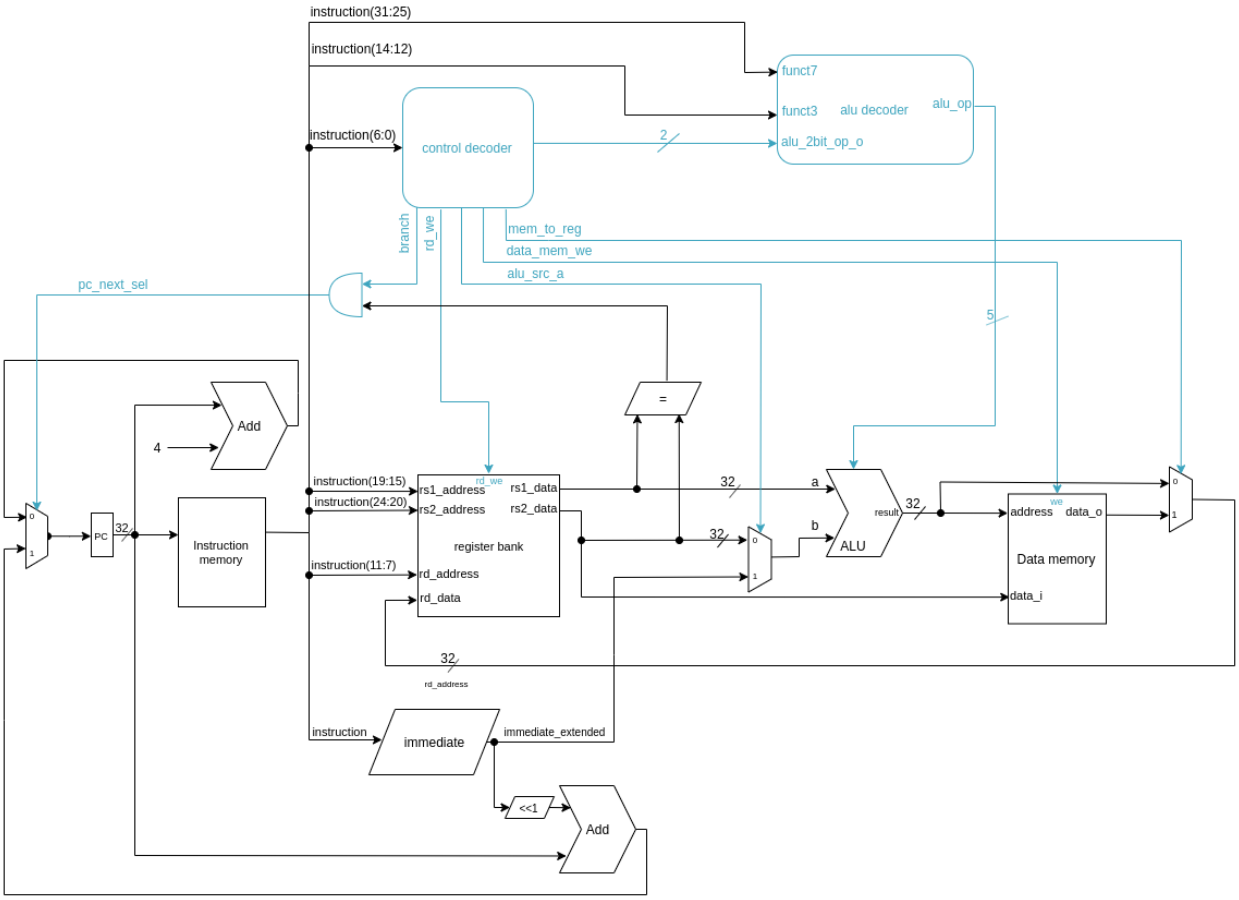
## 2. Single-cycle procesor

Prva i najjednostavnija implementacija procesora koja će biti objašnjena jeste Single-Cycle implementacija, specifična po tome da svaku instrukciju izvršava tačno jedan takt. Njena jednostavnost ima cenu, a to su lošije performanse u odnosu na druge implementacije (npr. Procesor sa protočnom obradom) i retko se koristi u savremenim procesorima, ali je korisna za edukativne svrhe jer predstavlja odličnu uvertiru u svet procesora.

Kako bi procesor izvršio određeni program, on mora da bude u stanju da prepozna svaku od instrukcija koje čine program i da izvrši određenu operaciju u zavisnosti od primljene instrukcije. No, kako je RISC-V set instrukcija veliki, na ovim i narednim vežbama biće implementiran samo osnovni podskup instrukcija, dok se čitaocu ostavlja da taj podskup instrukcija samostalno proširi. Instrukcije koje će biti implementirane su sledeće:

- Instrukcije koje omogućavaju pristup memoriji sa podacima: učitaj reč (*eng. Load word*) i skladišti reč (*eng. Store word*), sa oznakama *lw i sw* respektivno.
- Instrukcije koje izvršavaju aritmetičko logičke operacije: sabiranje, oduzimanje, logičko i i logičko ili, sa oznakama *add, sub, and, or* respektivno.
- Instrukcija uslovnog skoka *(eng. Branch if equal)* koja omogućava skok ukoliko je ispunjen uslov jednakosti, sa oznakom *BEQ*.
- Instrukcija sabiranja sa konstantom *(eng. add immediate),* sa oznakom *addi.*

Iako mali, ovaj podskup instrukcija će ilustrovati najvažnije principe korišćene u projektovanju procesora, dok će implementacija ostalih instrukcija zahtevati minimalne modifikacije. Na sledećoj slici je prikazana arhitekura procesora koja implementira prethodni set instrukcija:



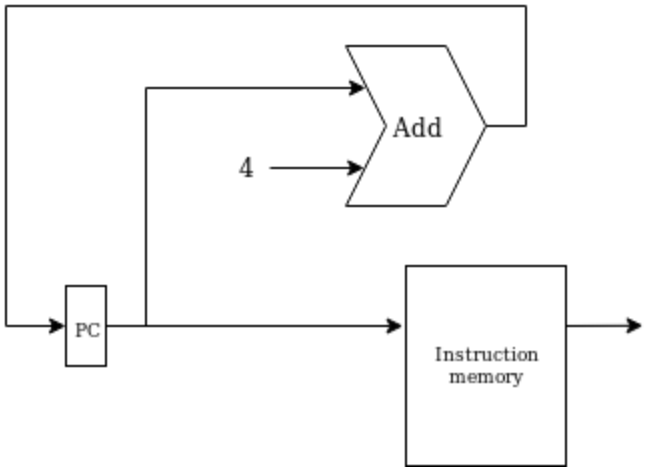

Slika 2. RISC-V procesor

Hardverska implementacija sa slike 2 je podeljena na dve velike celine: *Controlpath*(obojeno plavo na slici) i *Datapath*. *Datapath* se sastoji od osnovnih komponenti neophodnih da bi se izvršile prethodno navedene instrukcije i razlog korišćenja svake od njih biće objašnjen u nastavku. Controlpath predstavlja logiku za kontrolu komponenti u *Datapath-u.* U zavisnosti od instrukcije koju procesor trenutno izvršava, *ControlPath* generiše signale koji upravljaju komponentama u DataPath-u na takav način da se izvrši adekvatna operacija (npr. ALU će izvršiti sabiranje ukoliko je procesor prihvatio ADD instrukcija). *Controlpath* će detaljno biti objašnjen u nastavku. Na prethodnoj slici se može dobiti dojam da se **memorija za instrukcije i memorija za podatke** nalaze u datapath-u, odnosno da se nalaze unutar procesora, no, one su tu samo radi lakšeg vizuelnog prikaza i **ne pripadaju** arhitekturi procesora.

Bitno je naglastiti da će arhitektura procesora biti 32-bitna, odnosno sve instrukcije i podaci će biti predstavljeni sa 32 bita.

## 2.1 Datapath

Razuman početak implementacije datapath celine jeste da se razjasne glavne komponente neophodne da se izvrši svaka klasa RISC-V instrukcija. Te komponente su zadužene za prihvat instrukcije iz memorije i prikazane su na sledećoj slici:

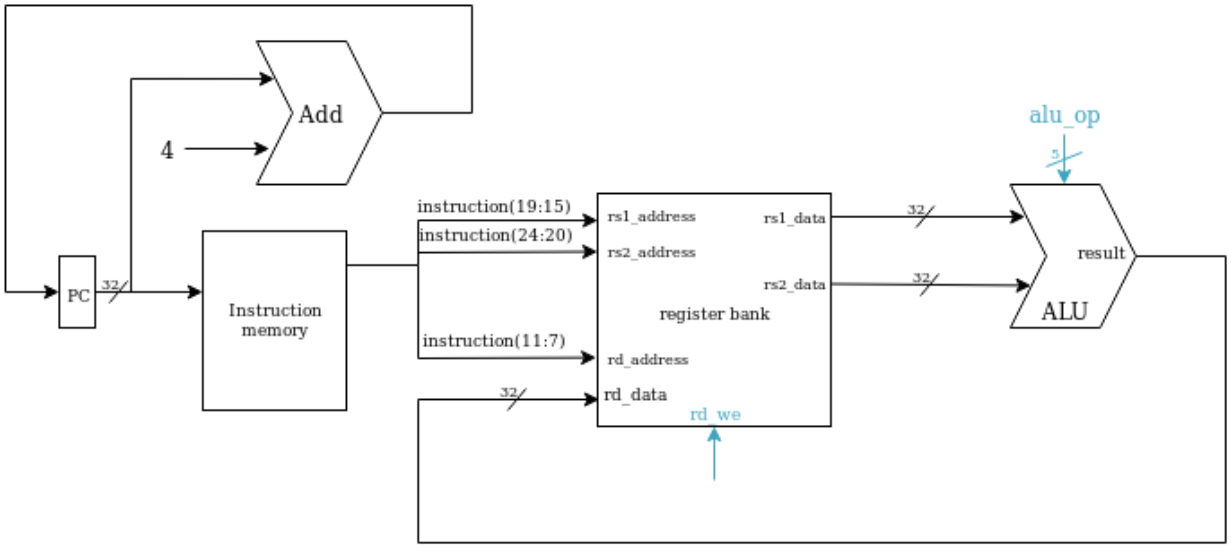



Slika 3. Inkrement programskog brojača i ekstrakcija instrukcije

Memorija za instrukcije je memorija sa asinhronim čitanjem i u njoj je smešten program koji treba da se izvrši. Koja instrukcije će biti ekstrahovana iz memorije zavisi od programskog brojača (PC na slici 3) koji generiše adresu na kojoj se nalazi naredna instrukcija. Sa slike 3 se može videti da se programski brojač uvek uvećava za 4 pomoću *Add* komponente koja predstavlja sabirač. Razlog za to je memorija za instrukcije koje je bajt adresibilna, odnosno četiri lokacije u memoriji predstavljaju jednu instrukciju i zato je naredna instrukcija pomerena u odnosu na prethodnu za 4 (instrukcija je 32-bitna, a to je 4 bajta). Takođe bitno je naglasiti da je programski brojač registar, što znači da će se naredna vrednost na njegovom izlazu pojaviti tek pri pojavi rastuće ivice takta, a kako je memorija za instrukcije memorija sa asinhronim čitanjem, pri svakoj promeni stanja programskog brojača promeniće se i izlaz memorije.

### 2.1.1 Implementacija ADD, SUB, OR, AND instrukcija

Nakon što je implementirana logika za prihvat naredne instrukcije počeće se sa proširivanjem arhitekture sa slike 3 dodatnim komponentama kako bi se podržale *add, sub, or i and* instrukcije.




Slika 4. Proširenje procesora kako bi podržao add, sub, or i and instrukcije

Slika 4 ilustruje da je neophodno dodati dve nove komponente: Registarsku banku i aritmetičko logičku jedinicu.

Registarska banka sadrži 32 registra širine 32 bita (arhitektura je 32-bitna) i uloga svakog od registara objašjena je na prethodnim vežbama. Komunikacija sa registarskom bankom odvija se preko dva interfejsa za čitanje i jednog interfejsa za upis podataka i razlog za to je format samih instrukcija. Primera radi, instrukcije koje pripadaju R tipu instrukcija moraju da pristupe na 3 lokacije istovremeno u registarskoj banci, sa dve treba da pročitaju i u treću treba da upišu rezultat operacije. Opis interfejsa je sledeći:

- Interfejsi za **čitanje** podaka sastoji se iz dva pristupa: rsN_address i rsN_data (slovo N predstavlja broj 1 ili 2). rsN_address je ulazni port koji predstavlja adresu sa koje treba pročitati podatak, dok je rsN_data izlazni port na kome će se pojaviti vrednost registra sa željene adrese. **Čitanje podataka iz memorije je asinhrono**.
- Interfejs za **upis** podataka se sastoji iz rd_address, rd_data i rd_we pristupa. Rd_address ulaz predstavlja adresu registra u koji se želi upisati podatak, rd_data predstavlja port preko koga se željena vrednost upisuje u registar i rd_we port predstavlja signal dozvole upisa u registarsku banku (Kontrolni signali

iz *ControlPath* celine kontrolišu njegovu vrednost). **Upis podataka u memoriju je sinhron.**

Sa slike 4 se može videti da je na svaki adresni ulaz registarske banke dovedeno 5 bita instrukcije, odnosno bitovi 19 do 15 su dovedeni na rs1_address port, bitovi 24 do 20 su dovedeni na rs2_address port i bitovi 11 do 7 su dovedeni na rd_address port. Da bi se razumelo zašto se baš ovi bitovi dovode na pomenute adresne ulaze, napraviće se osvrt na format R tipa instrukcije. On je prikazan na sledećoj slici:

| funct7 | rs2 | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|---|
| 7 bita(31:25) | 5 bita(24:20) | 5 bita (19:15) | 3 bita (14:12) | 5 bita(11:7) | 7 bita(6:0) |

Slika 5. Format R tipa instrukcije

U ovom trenutku, pri kreiranju *datapath* celine, bitna su nam 3 polja prikazana na slici 5 i ta polja su rs2, rs1 i rd. Unutar njih je sadržana informacija o tome iz kojih registara registarske banke se čita (rs1 i rs2) i gde se smešta rezultat izvršene operacije (rd). Odnosno ta polja predstavljaju adrese registara u registarskoj banci. Na osnovu ove informacije jasno je zašto se prethodno pomenuti bitovi instrukcije dovode na tačno određene adresne ulaze registarske banke.

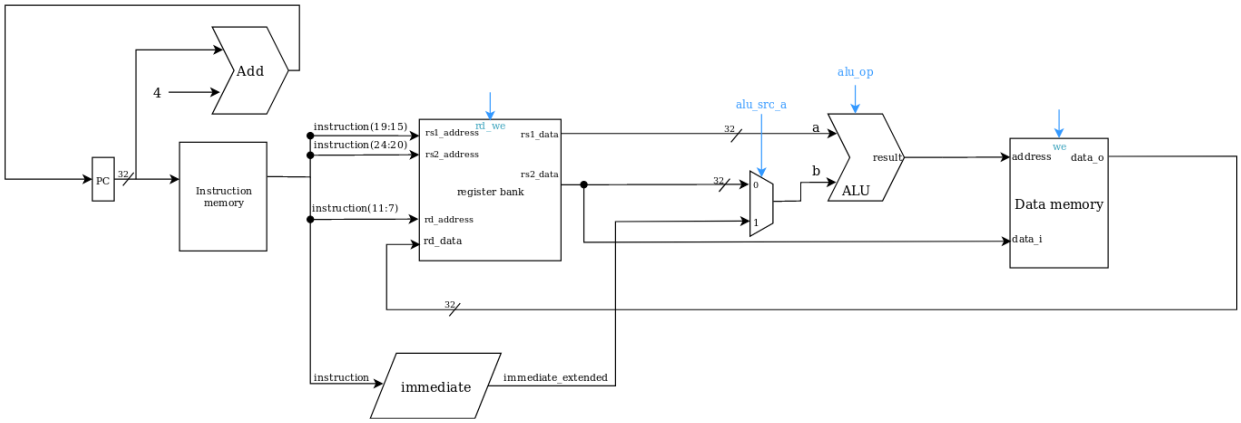
Druga komponenta koja je dodata jeste aritmetičko logička jedinica (na slici 4 označena sa ALU). Njena uloga može da se protumači iz imena, a to je obavljanje aritmetičko logičih operacija (sabiranje, oduzimanje, logičko ili, logičko i). Interfejs joj se sastoji iz sledećih linija:

- Dva ulazna porta za operande nad kojima treba da se izvrši aritmetičko logička operacija
- Ulaznog porta (*alu_op* na slici 4), za izbor operacije koju treba izvršiti.
- Izlaznog porta (*result* na slici 4) na kome se pojavljuje rezultat operacije.

Na slici 4 je prikazano da su na ulaze ALU jedinice dovedeni izlazi registarske banke (rs1_data i rs2_data), vrednost na *alu_op* ulazu kontroliše *ControlPath* (on određuje tip operacije koji treba se izvrši) i izlaz (result) je doveden rd_data ulaz registarske banke.

### 2.1.2 Implementacija instrukcije za upis u memoriju za podatke (sw)

Na slici 6 se može primetiti memorija za podatke koja prilikom implementiranja dosadašnjih instrukcija nije bila pominjana. Razlog je što instrukcije koje su do sada implementirane ne koriste memoriju za podatke, no kako je pristup ovoj memoriji neophodan, u nastavku će se implementirati instrukcije za čitanje i upis u nju. Prva koja će biti implementirana jeste instrukcija za upis (*sw*) u memoriju i na sledećoj slici biće prikazano proširenje slike 4 sa četiri dodatne komponente kako bi se pomenuta instrukcija podržala:




Slika 6. Proširenje procesora kako bi podržao *SW* instrukciju

Pre opisivanja pojedinačnih komponenti, napraviće se osvrt na S format instrukcija kome pripada *sw:*

| imm(11:5) | rs2 | rs1 | funct3 | imm(4:0) | opcode |
|---|---|---|---|---|---|
| 7 bita(31:25) | 5 bita(24:20) | 5 bita (19:15) | 3 bita (14:12) | 5 bita(11:7) | 7 bita(6:0) |

Slika 7. S format instrukcije

Polja koja su od interesa za datapath celinu su imm(11:5), rs2, rs1 i imm(4:0). Rs2 predstavlja adresu registra u registarskoj banci čiju vrednost treba upisati u memoriju za podatke. Imm(11:5) i imm(4:0) predstavljaju jednu celinu koja se zove immediate(konstanta), odnosno posmatraju se kao imm(11:0) i ta celina sabira se sa vrednošću na adresi rs1 u registarskoj banci kako bi se dobila adresa na koju je potrebno upisati podatak. Mašinski kod jedne takve instrukcije je sledeći:

Primer: sw  a0, 4(a1).

Ovde a0 predstavlja registar rs2 čiju vrednost treba smestiti u memoriju, 4 predstavlja immediate polje ($000000000100_2$ = $4_{10}$) i a1 predstavlja rs1. Odnosno

ukoliko je a0=5 i a1=4 to znači da će se na adresu 8 (a1 + immediate) memorije za podatke upisati vrednost registra na adresi 5 u registarskoj banci.

Sada se može preći na opis dodatnih komponenti na slici 6 neophodnih da bi se izvršila *sw* instrukcija. Memorija za podatke ima jako jednostavan interfejs koji se sastoji iz sledećih portova:

- *Address* port određuje sa koje adrese se čita, odnosno na koju adresu se piše.
- *Data_i* port predstavlja ulazni port za upis podataka. **Upis je sinhron**
- *Data_o* port predstavlja izlazni port čitanja podataka. **Čitanje je asinhrono**
- *Data_we* port predstavlja ulazni port dozvole upisa i kontrolisan je od strane signala koji potiče iz *controlpath* celine. Ukoliko je on na logičkoj jedinici pisanje je dozvoljeno, u suprotnom nije.

Sa slike 6 se može videti da je izlaz ALU jedinice doveden na adresni ulaz memorije sa podacima, razlog tome je sabiranje vrednosti registra na adresi rs1 i konstante (immediate) pri čemu je neophodno koristiti ALU jedinicu. Na *data_i* ulaz memorije sa podacima doveden je rs2_data izlaz registarske banke, jer je to informacija koju je potrebno upisati.

Sledeća komponenta je *immediate*. Njena uloga je da proširi konstantu (immediate) sa 12 bita na 32, pri čemu se gornji biti proširuju nulama ili jedinicama u zavisnosti od najvišeg bita immediate polja kako bi se održao predznak konstante. Nezavisno od formata instrukcije, najviši bit immediate polja će se uvek poklapati sa najvišim bitom instrukcije (instruction(31)). Ukoliko je on logička jedinica immediate se do 32 bita proširuje sa logičkim jedinicama, u suprotnom se proširuje sa logičkim nulama.

| *11111111111111111111* | *immediate(11:0)* |
|---|---|
| 20 bita | 12 bita |

Slika 8. Proširivanje jedinicama ukoliko je instruction(31) = 1

| *00000000000000000000* | *immediate(11:0)* |
|---|---|
| 20 bita | 12 bita |

Slika 9. Proširivanje nulama ukoliko je instruction(31) = 0

Proširenje je neophodno uraditi kako bi ALU jedinica mogla da sabere immediate polje sa rs1_data izlazom registarske banke jer ona očekuje da oba ulazna operanda budu iste širine od 32 bita. Interfejs immediate komponente se sastoji od dva porta:

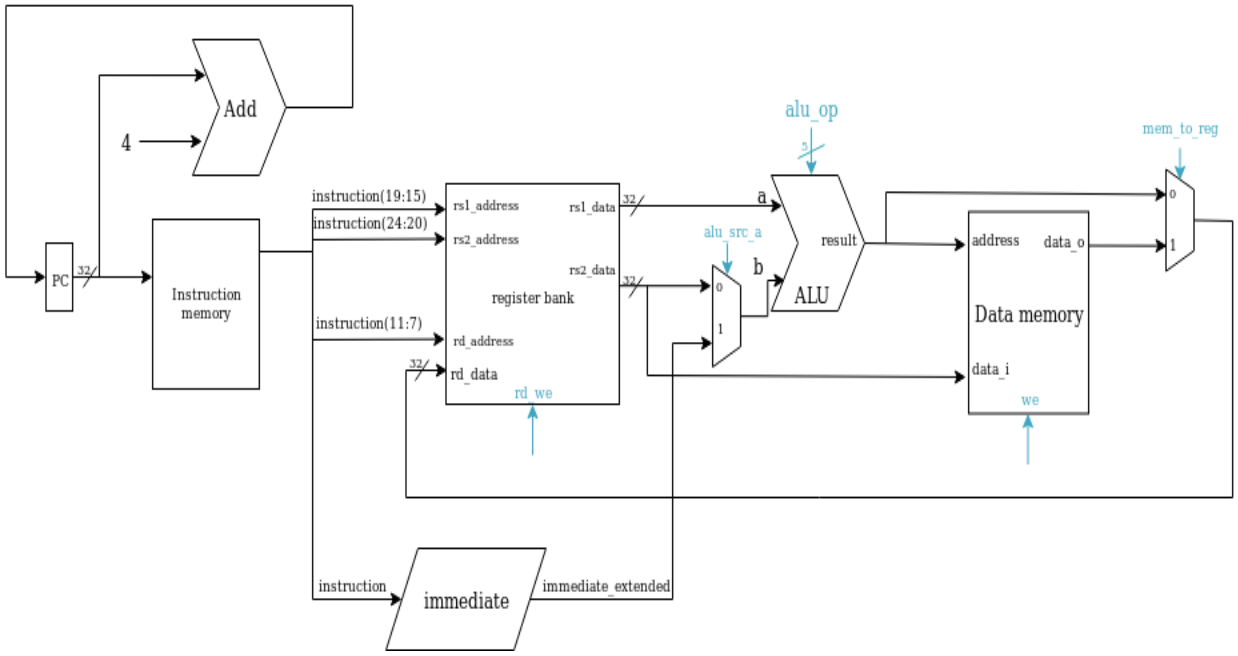
- *Instruction* predstavlja ulazni port preko koga se prima instrukcija.
- *Immediate_extended* izlazni port predstavlja prošireno immediate polje.

Immediate komponenta preko instruction porta dobija instrukciju iz koje izvlači immediate polje koje treba proširiti, ali takođe dobija i *opcode* polje. *Opcode* polje je neophodno jer *immediate* komponenta na osnovu njega zaključuje koji format instrukcije je u pitanju i shodno tome vrši proširivanje. Različiti formati zahtevaju različit način proširivanja. U slučaju sw instrukcije, koja ima S format, neophodno je izvršiti konkatanaciju polja imm(11:5) i polja imm (4:0), i onda ih proširiti do 32 bita.

Poslednja komponenta je multiplekser 2 na 1 i on je tu kako bi multipleksirao šta se dovodi na b ulaz ALU jedinice(*immediate_extend* ili *rs2_data*). Njega kontroliše *controlpath* preko *alu_src_a* ulaza.

### 2.1.3 Implementacija instrukcije za čitanje iz memorije za podatke (lw)

Proširenje slike 6 za *lw* instrukciju je sledeće:




Slika 10. Proširenje procesora kako bi podržao *LW* instrukciju

Uloga *lw* instrukcije jeste da smesti podatak iz memorije za podatke u određeni registar u registarskoj banci i da bi se to izvršilo neophodno je ubaciti dodatne komponente. Da bi se razumela potreba za dodatnim multiplekserom napraviće se osvrt na I format instrukcija kome pripada lw:

| imm(11:0) | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|
| 12 bita(31:20) | 5 bita (19:15) | 3 bita (14:12) | 5 bita(11:7) | 7 bita(6:0) |

Slika 11. I format instrukcija

Polja koja su u ovom trenutku od interesa su imm(11:0), rs1 i rd. Rd predstavlja registar unutar registarske banke u koji će se smestiti podatak iz memorije, dok suma između rs1 i immediate polja predstavlja adresu u memoriji za podatke sa koje treba da se učita informacija.

U slučaju ove instrukcije immediate komponenta samo treba da proširi imm(11:0) polje instrukcije do 32 bita. Da li će se proširiti logičkim nulama ili jedinicama zavisi od imm(11) bita i to na isti način kao kod prethodne instrukcije.

Dodatna komponenta koja je neophodna jeste multiplekser 2 na 1 i on multipleksira šta će biti upisano u registarsku banku(rezultat alu jedinice ili podatak iz memorije). Njegov selekcioni ulaz je takođe kontrolisan od strane signala iz *controlpath* celine.

### 2.1.4 Implementacija ADDI instrukcije

Implementacija ADDI instrukcije ne zahteva dodatnu logiku u datapath celini. ADDI instrukcija pripada I formatu instrukcija (slika 11) zbog čega dodatne komponente nisu neophodne i sledeći primer će to ilustrovati:

Primer:

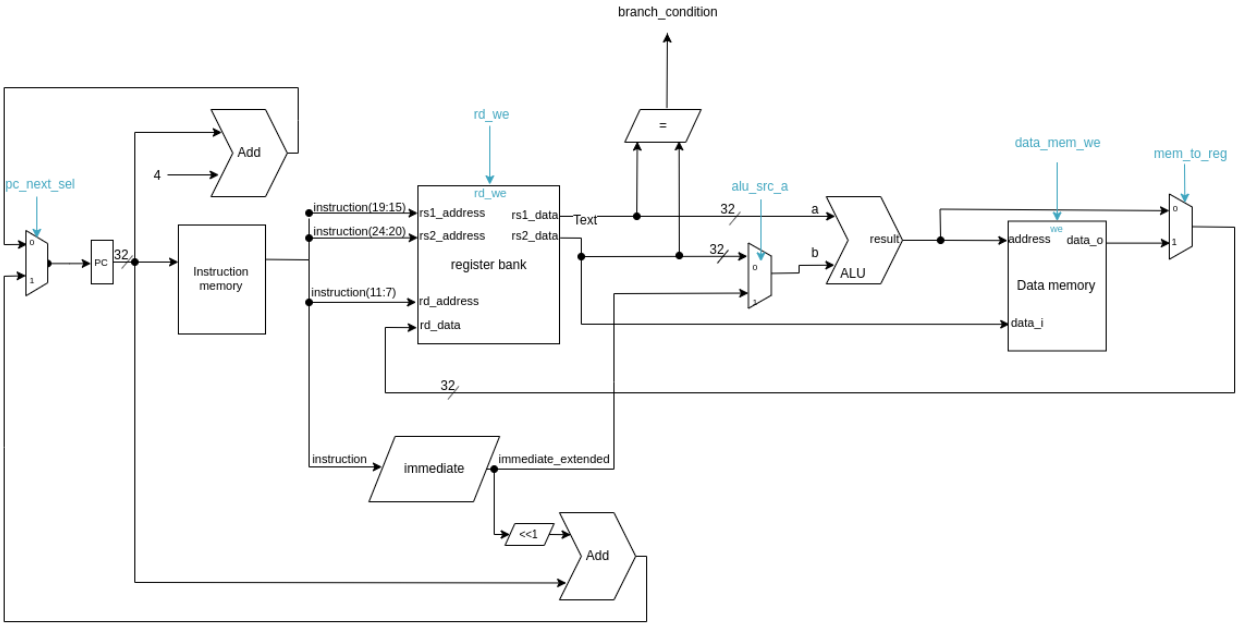
```
ADDI x10, x11, 10
```

Prethodna instrukcija sumira vrednost u registru X11 sa konstantom (immediate poljem) i rezultat smešta u registar x10. Na osnovu ovoga se zaključuje da je za sumiranje potrebna ALU jedinica, potrebno je multipleksirati “b” ulaz ALU jedinice[1] i potrebno je proslediti rezultat sumiranja u registarsku banku. Sve nabrojane komponente su već prisutne .

[1] Multipleksiranje je neophodno kako bi se na “b” ulaz ALU jedinice prosledila proširena vrednost immediate polja umesto rs2_data izlaza registarske banke.

## 2.1.5 Implementacija instrukcije uslovnog skoka (BEQ)

Slika 12 ilustruje proširenje slike 8 neophodnim komponentama kako bi procesor podržao instrukciju uslovnog skoka:



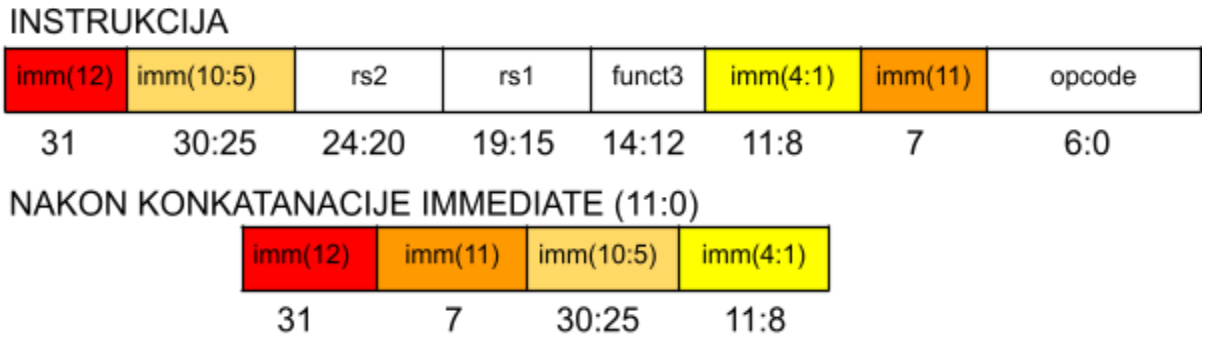

Slika 12. Proširenje procesora kako bi podržao *BEQ* instrukciju

Prilikom implementiranja svih instrukcija do sada, programski brojač (PC) se uvek uvećavao za konstantnu vrednost, odnosno instrukcije su uvek izvršavane sekvencijalno (jedna za drugom). Sada, dodavanjem instrukcijoe uslovnog skoka (BEQ), omogućava se skok na proizvoljnu instrukciju u memoriji ukoliko je uslov jednakosti ispunjen. Kako bi se razumele dodatne komponente napraviće se osvrt na SB format instrukcija kome BEQ pripada:

| imm(12,10:5) | rs2 | rs1 | funct3 | imm(4:1,11) | opcode |
|---|---|---|---|---|---|
| 7 bita(31:25) | 5 bita(24:20) | 5 bita (19:15) | 3 bita (14:12) | 5 bita(11:7) | 7 bita(6:0) |

Slika 13. SB format instrukcije

Polja koja nisu od interesa u ovom trenutku su opcode i funct3. Rs1 i rs2 polja sadrže u sebi adrese registara koji se porede i od njihove jednakosti zavisi da li će se desiti skok ili ne. U *immediate* poljima (imm(12, 10:5) i imm(4:1, 11)) krije se informacija koliko bajtova treba skočiti do željene instrukcije. Da bi se ta informacija dobila *immediate* komponenta mora da izvrši konkatanaciju *immediate* polja instrukcije i nakon toga proširivanje do 32 bita. Konkatanacija se vrši spajanjem  31:25 i 11:7 bitova instrukcije, vodeći računa o poziciji bitova. Odnosno treba uraditi sledeće:




Slika 14. Ilustracija konkatanacije immediate bita

Proširivanje do 32 bita se radi kao i u prethodnim slučajevima, umeću se sa leve strane sve jedinice ili nule u zavisnosti od najvišeg bita instrukcije (instruction(31)):

| *11111111111111111111* | *immediate(11:0)* |
|---|---|
| 20 bita | 12 bita |

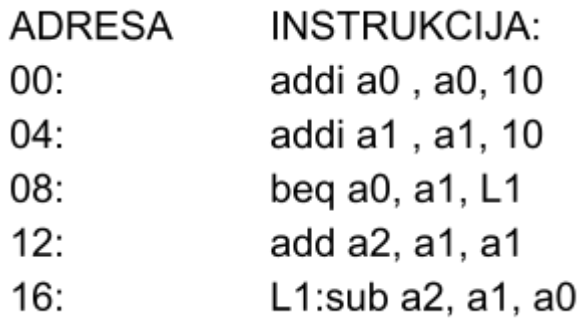
Slika 15. Proširivanje jedinicama ukoliko je instruction(31) = 1

| *00000000000000000000* | *immediate(11:0)* |
|---|---|
| 20 bita | 12 bita |

Slika 16. Proširivanje nulama ukoliko je instruction(31) = 0

Još jedna operacija koju je potrebno dodati jeste pomeranje proširene vrednost za jednu poziciju u levo i ako se pogleda slika 12, može se videti da je nakon *immediate* komponente dodat pomerač u levo[2]. Rezultat nakon pomeranja predstavljaće broj bajtova od trenutne vrednosti programskog brojača do naredne instrukcije koju treba izvršiti. Smisao pomeranja u levo za jednu poziciju jeste proširenje za duplo opsega instrukcija do kojih može da se skoči. Sledeći mašinski kod ilustruje zašto je pomeranje u levo neophodno:

[2] Na slici 12 nalazi se pomerač u levo, ali to je samo radi lakše ilustracije, u realnom slučaju on se nalazi unutar immediate komponente.




Slika 17. Primer mašinskog koda

U ovom slučaju BEQ instrukcija treba da skoči na 16 bajt, odnosno treba da preskoči jednu instrukciju. Kada se BEQ instrukcija pretvori u binarni kod dobija se sledeće :

| 0000000 | 01011 | 01010 | 000 | 01000 | 1100011 |
|---|---|---|---|---|---|
| 7 bita(31:25) | 5 bita(24:20) | 5 bita (19:15) | 3 bita (14:12) | 5 bita(11:7) | 7 bita(6:0) |

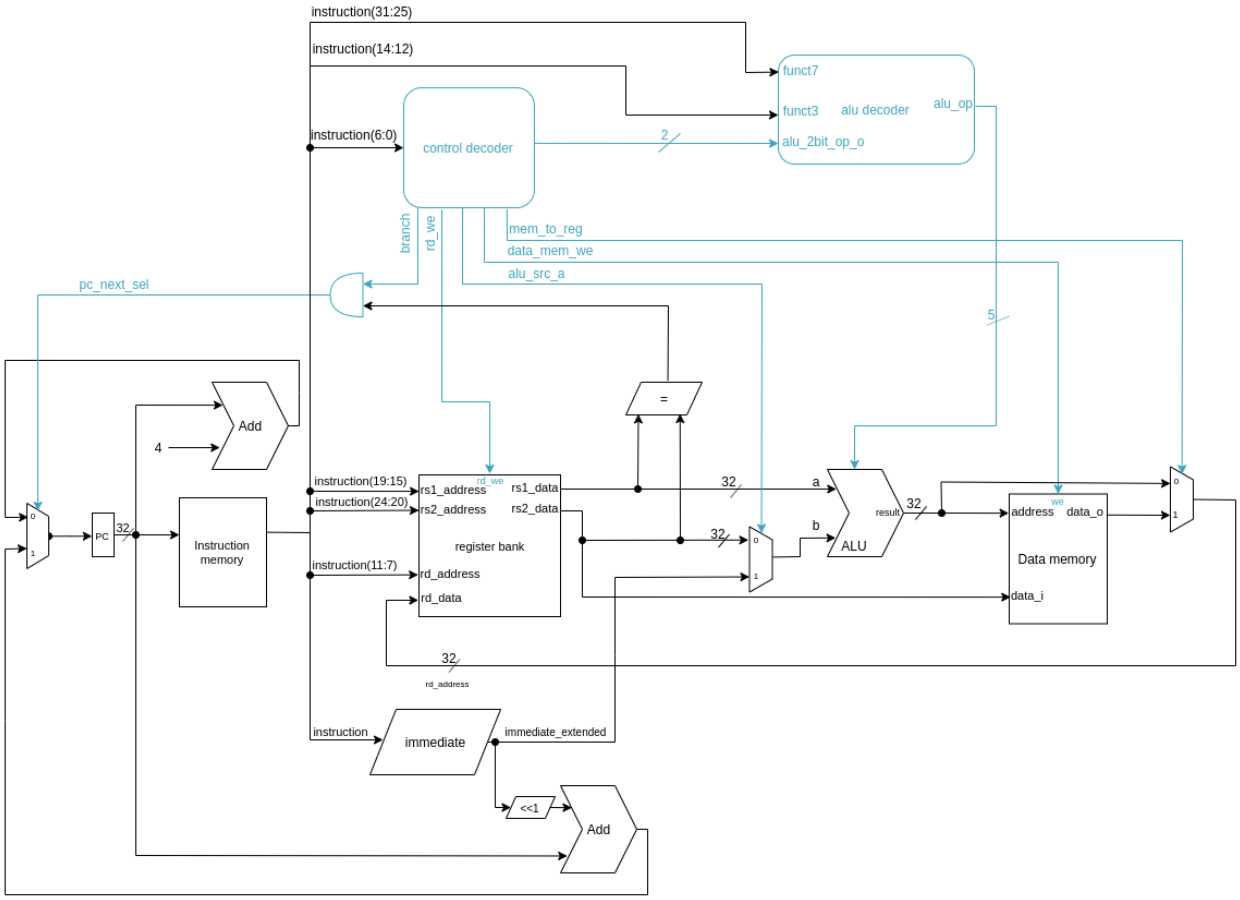
Slika 18. Primer BEQ instrukcije

Nakon što se izvrši ekstrahovanje *immediate* polja instrukcije i izvrši se prethodno opisani postupak konkatanacije i proširivanja, dobija se vrednost $00000004_{16} = 4_{10}$. Kako je potrebno preskočiti 8 bajtova a ne 4, neophodno je izvrišiti pomeranje u levo za jednu poziciju(ekvivalentno množenju sa 2).

Nakon pomeranja u levo za jednu poziciju, ta vrednost se sabira sa tretnutnom vrednošću programskog brojača i trebalo bi da se prosledi do ulaza PC registra. Da li će se proslediti ili ne zavisi od multipleksera koji se nalazi ispred PC registra čiji je selekcioni ulaz kontrolisan od strane *pc_next_sel* kontrolnog signala. Njega generiše *controlpath* i da bi to ispravno uradio neophodna mu je informacija o tome da li su operandi koje se porede (primer: a0 i a1 na slici 17) jednaki. Tu informaciju dobija u vidu statusnog signala *branch_condition* (slika 12) koji je izlaz iz komparatora koji poredi prethodno pomenute operande.

## 2.2 Controlpath

Nakon kreiranja *datapath* celine neophodno je kreirati *controlpath* koji će kontrolisati prethodno pomenute kontrolne signale. Ubacivanjem komponenti koje pripadaju controlpath celini dobija se sledeća slika:




Slika 18. Proširenje procesora sa *controlpath* komponentama

Sa slike 18 se može videti da *controlpath* čine dve komponente: *Control decoder* i *alu decoder*[3].

***Alu decoder*** komponenta je zadužena za kontrolisanje ALU jedinice, odnosno ona određuju koju instrukciju ALU treba da izvrši. Njen interfejs se sastoji iz sledećih portova:

- alu_2bit_op: 2-bitni ulazni port kontrolisan od strane *control decoder* komponente..

[3] Razdvajanje na dve komponente nije neophodno, odnosno *control decoder* i *alu decoder* su mogli biti jedna komponenta.

- funct3: 3-bitni ulazni port preko koga se primaju 3 bita instrukcije koji predstavljaju funct3 polje (instruction(14:12)).
- funct7: 7-bitni ulazni port preko koga se prima 7 bita instrukcije koji predstavljaju funct7 polje (instruction(31:25)).
- alu_op: 5-bitni izlazni port koji generiše kontrolne signale za upravljanje ALU jedinicom.

Sledeća slika ilustruje kako se u zavisnosti od ulaznih portova generiše *alu_op* kontrolni signal:

| Alu_2bit_op ulaz | | Funct7 ulaz | | | | | | | Funct3 ulaz | | | Alu_op izlaz |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 2 | 1 | 0 | |
| 0 | 0 | X | X | X | X | X | X | X | X | X | X | 00010 |
| X | 1 | X | X | X | X | X | X | X | X | X | X | 00110 |
| 1 | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00010 |
| 1 | X | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00110 |
| 1 | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 00000 |
| 1 | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 00001 |

Slika 19. Tablica istinosti za alu_op izlaz

Slika 20 daje informaciju koju operaciju ALU treba da izvrši u zavisnosti od kombinacije koju na svom izlazu generiše *alu_op*.

| Alu_op | operacija |
|---|---|
| 00000 | Logičko “I” |
| 00001 | Logičko “ILI” |
| 00010 | Sabiranje |
| 00110 | Oduzimanje |

Slika 20. Veza *alu_op* kontrolnih signala i operacije koje ALU izvršava

***Control_decoder*** komponenta je zadužena za generisanje kontrolnih signala kojima se kontrolišu sve ostale komponente u *datapath* celini i njen interfejs se sastoji iz sledećih portova:

- *opcode*: ulazni port preko koga se prima donjih 7 bita instrukcije (*instruction(6:0)*).

- *branch:* Izlazni port koji se postavlja na visok logički nivo ukoliko je iz instrukcione memorije zahvaćena *BEQ* instrukcija.
- *rd_we*: izlazni port koji se postavlja na visok logički nivo ukoliko instrukcija prihvaćena iz instrukcione memorije upisuje podatak u registarsku banku.
- *alu_src_a:* izlazni port koji kontroliše multiplekser ispred aritmetičko logičke jedinice i koji se postavlja na visok logički nivo ukoliko na “b” ulaz ALU jedinice treba da se prosledi *immediate_extended* signal.
- *data_mem_we*: izlazni port koji se postavlja na visok logički nivo ukoliko instrukcija prihvaćena iz instrukcione memorije upisuje podatak u memoriju za podatke.
- mem_to_reg: izlazni port koji kontroliše multiplekser ispred memorije za podatke i koji se postavlja na visok logički nivo ukoliko je u registarsku banku potrebno upisati podatak iz memorije za podatke, a na nizak ukoliko je potrebno upisati izlaz ALU jedinice
- alu_2bit_op: 2-bitni izlazni port koji se prosleđuje *alu_decoder* komponenti, i na osnovu koga ona zaključuje koju operaciju ALU jedinica treba da izvrši.

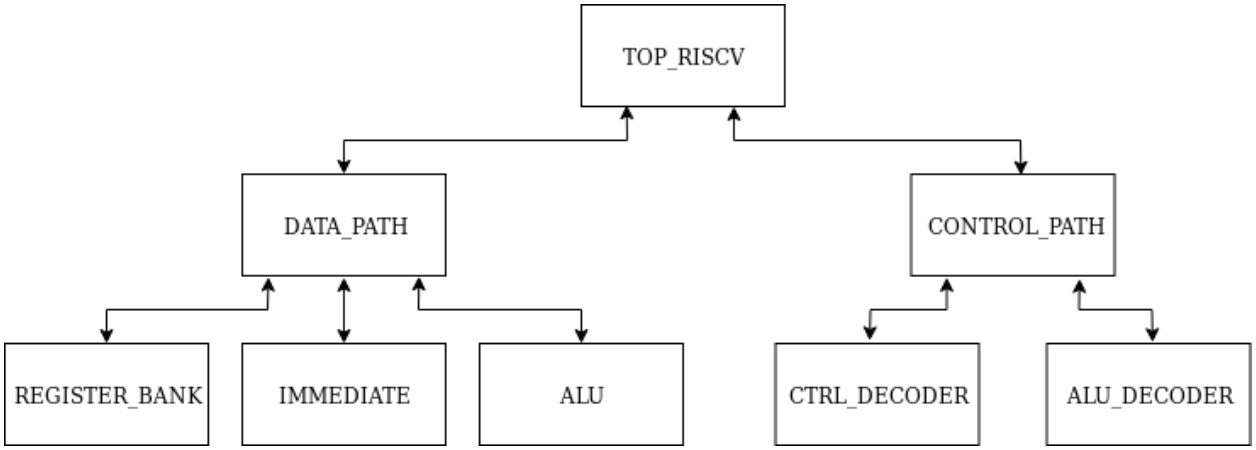
Sledeća slika ilustruje koji kontrolni signali se generišu u zavisnosti od tipa instrukcije (vrednosti *opcode* polja):

| | Ime signala | R-format | lw | sw | beq | addi |
|---|---|---|---|---|---|---|
| Ulazi | opcode(6) | 0 | 0 | 0 | 1 | 0 |
| | opcode(5) | 1 | 0 | 1 | 1 | 0 |
| | opcode(4) | 1 | 0 | 0 | 0 | 1 |
| | opcode(3) | 0 | 0 | 0 | 0 | 0 |
| | opcode(2) | 0 | 0 | 0 | 0 | 0 |
| | opcode(1) | 1 | 1 | 1 | 1 | 1 |
| | opcode(0) | 1 | 1 | 1 | 1 | 1 |
| Izlazi | Alu_src | 0 | 1 | 1 | x | 1 |
| | Mem_to_reg | 0 | 1 | x | 0 | 0 |
| | rd_we | 1 | 1 | 0 | 0 | 1 |
| | data_mem_we | 0 | 0 | 1 | 0 | 0 |
| | branch | 0 | 0 | 0 | 1 | 0 |
| | Alu_2bit_op(1) | 1 | 0 | 0 | 0 | 1 |
| | Alu_2bit_op(0) | 0 | 0 | 0 | 1 | 1 |

Slika 21. Veza Kontrolnih signala i opcode polja instrukcije

# 3 Hijerarhijska predstava procesora

Na sledećoj slici je prikazana hijerarhijska predstava procesora, odnosno kako su prethodno opisane celine međusobno povezane. U narednim sekcijama biće napisani interfejsi svih blokova sa slike 22. **Prilikom realizovanja svake od komponenti procesora ne modifikovati interfejs ukoliko to nije neophodno**!




Slika 22. RISCV hijerarhija

## 3.1 REGISTER_BANK

Interfejs registarske banke je već opisan (pogledati 2.1.1) i sledeći kodni listing entity deklaraciju napisanu u VHDL jeziku.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
--***************OPIS MODULA*********************
--Registarska banka sa dva interfejsa za citanje
--podataka i jednim interfejsom za upis podataka.
--Broj registara u banci je 32.
--WIDTH je parametar koji odredjuje sirinu poda-
--data u registrima
--***********************************************
entity register_bank is
   generic (WIDTH : positive := 32);
   port (clk           : in  std_logic;
         reset         : in  std_logic;
         -- Interfejs 1 za citanje podataka
         rs1_address_i : in  std_logic_vector(4 downto 0);
```

```
        rs1_data_o    : out std_logic_vector(WIDTH - 1 downto 0);
        -- Interfejs 2 za citanje podataka
        rs2_address_i : in  std_logic_vector(4 downto 0);
        rs2_data_o    : out std_logic_vector(WIDTH - 1 downto 0);
        -- Interfejs za upis podataka
        rd_we_i       : in  std_logic;  -- port za dozvolu upisa
        rd_address_i  : in  std_logic_vector(4 downto 0);
        rd_data_i     : in  std_logic_vector(WIDTH - 1 downto 0));

end entity;
```

## 3.2 IMMEDIATE

Kao i kod registarske banke, interfejs immediate komponente i njena funkcionalnost su već opisani (pogledati 2.1.2) i sledeći kodni listing je samo vhdl deklaracija entity dela. Može se primetiti da je deo koda zakomentarisan. Ukoliko se čitalac odluči da proširi set instrukcija koji trenutno podržava ova implementacija RISC-V procesora te linije može otkomentarisati.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity immediate is
   port (instruction_i        : in  std_logic_vector (31 downto 0);
         immediate_extended_o : out std_logic_vector (31 downto 0));
end entity;
```

## 3.3 ALU

Sledeći kodni listing predstavlja entity deklaraciju ALU jedinice.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
use ieee.math_real.all;
use work.alu_ops_pkg.all;


ENTITY ALU IS
   GENERIC(
      WIDTH : NATURAL := 32);
   PORT(
      a_i    : in STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0); --prvi operand
```

```
      b_i    : in STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0); --drugi operand
      op_i   : in STD_LOGIC_VECTOR(4 DOWNTO 0); --port za izbor operacije
      res_o  : out STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0); --rezultat
      zero_o : out STD_LOGIC; -- signal da je rezultat nula
      of_o   : out STD_LOGIC); -- signal da je doslo do prekoracenja opsega
END ALU;
```

## 3.4 DATAPATH

Sledeći kodni listing predstavlja vhdl deklaraciju entity dela *datapath* celine. Unutar DATAPATH modula (slika 22) treba da budu instancirane opisane komponente (ALU, REGISTER_BANK i IMMEDIATE), implementirana dodatna logika (sabirači, multiplekseri, komparatori, idt.) i sve to povezano na način koji je objašnjen u sekcijama 2.1 do 2.1.5.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity data_path is
   generic (DATA_WIDTH : positive := 32);
   port(
      -- ********* Globalna sinhronizacija ******************
      clk                 : in  std_logic;
      reset               : in  std_logic;
      -- ********* Interfejs ka Memoriji za instrukcije *****
      instr_mem_address_o : out std_logic_vector(31 downto 0);
      instr_mem_read_i    : in  std_logic_vector(31 downto 0);
      instruction_o       : out std_logic_vector(31 downto 0);
      -- ********* Interfejs ka Memoriji za podatke *****
      data_mem_address_o  : out std_logic_vector(31 downto 0);
      data_mem_write_o    : out std_logic_vector(31 downto 0);
      data_mem_read_i     : in  std_logic_vector(31 downto 0);
      -- ********* Kontrolni signali ************************
      mem_to_reg_i        : in  std_logic;
      alu_op_i            : in  std_logic_vector(4 downto 0);
      pc_next_sel_i       : in  std_logic;
      alu_src_i           : in  std_logic;
      rd_we_i             : in  std_logic;
      -- ********* Statusni signali *************************
      branch_condition_o  : out std_logic
    -- ***************************************************
      );

end entity;
```

## 3.5 CTRL_DECODER

Sledeći kodni listing predstavlja vhdl deklaraciju entity dela *ctrl_decoder* modula.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ctrl_decoder is
   port (
      ------------ Opcode polje instrukcije------------
      opcode_i      : in  std_logic_vector (6 downto 0);
      ------------ Kontrolni signali------------------
      branch_o      : out std_logic;
      mem_to_reg_o  : out std_logic;
      data_mem_we_o : out std_logic;
      alu_src_o     : out std_logic;
      rd_we_o       : out std_logic;
      alu_2bit_op_o : out std_logic_vector(1 downto 0)
      );
end entity;
```

## 3.5 ALU_DECODER

Sledeći kodni listing predstavlja vhdl deklaraciju entity dela *alu_decoder* modula. Njegova funkcionalnost opisana je u poglavlju 2.2 ovih vežbi.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.alu_ops_pkg.all;

entity alu_decoder is
   port (
      -------- Controlpath ulazi *********
      alu_2bit_op_i : in  std_logic_vector(1 downto 0);
      -------- Polja instrukcije *******
      funct3_i      : in  std_logic_vector (2 downto 0);
      funct7_i      : in  std_logic_vector (6 downto 0);
      -------- Datapath izlazi ********
      alu_op_o      : out std_logic_vector(4 downto 0));
end entity;
```

## 3.5 CONTROL_PATH

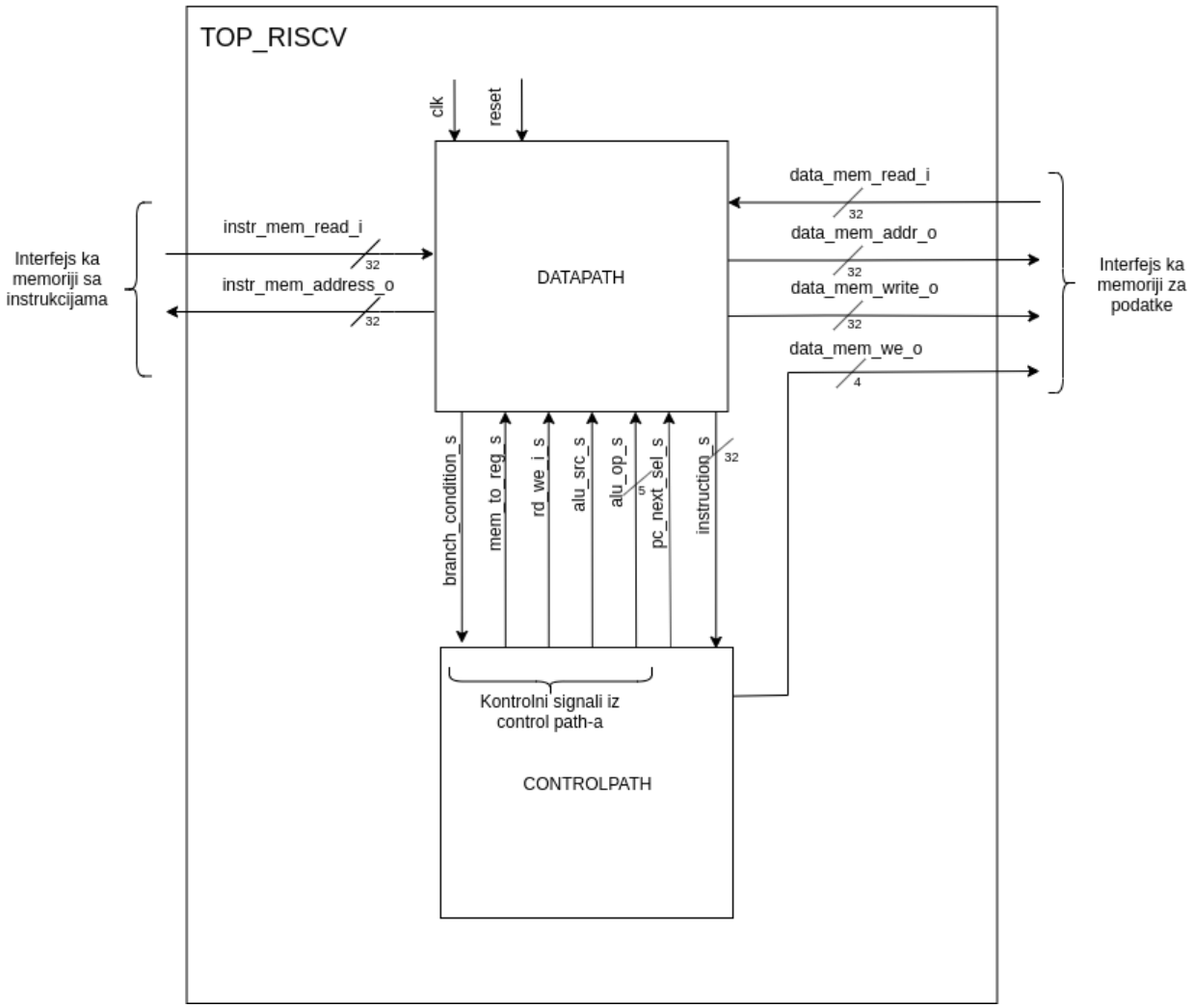
Unutar control_path celine treba da budu instancirani CTRL_DECODER i ALU_DECODER i izvršeno je njihovo povezivanje na način objašnjen u 2.2. Sledeći kodni listing predstavlja vhdl deklaraciju entity dela control_path celine:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity control_path is
   port (clk                : in  std_logic;
         reset              : in  std_logic;
         -- ********* Interfejs za prihvat instrukcije iz datapath-a*********
         instruction_i      : in  std_logic_vector (31 downto 0);
         -- ********* Kontrolni intefejs ***********************************
         mem_to_reg_o       : out std_logic;
         alu_op_o           : out std_logic_vector(4 downto 0);
         pc_next_sel_o      : out  std_logic;
         alu_src_o          : out std_logic;
         rd_we_o            : out std_logic;
         --********** Ulazni Statusni interfejs
************************************
         branch_condition_i : in  std_logic;
         --********** Izlazni Statusni interfejs
************************************
         data_mem_we_o      : out std_logic_vector(3 downto 0)
         );
end entity;
```

## 3.6 TOP_RISCV

Na samom kraju neophodno je instancirati i povezati DATA_PATH i CONTROL_PATH unutar TOP_RISCV modula. Njegov interfejs je dat u nastavku:




Slika 23. TOP_RISCV

Kao što je prethodno pomenuto interfejs procesora je sačinjen od portova za komunikaciju sa memorijom za podatke i portova za komunikaciju sa memorijom za instrukcije.

Sa memorijom za instrukcije se komunicira preko dva porta:

- *instr_mem_read_i* porta preko koga se primaju instrukcije
- instr_mem_addr_o porta preko koga se šalje adresa naredne instrukcije

Komunikacija sa memorijom za podatke se obavlja preko 4 porta:

- *data_mem_read_i* ulazni port preko koga se prima podataka iz memorije
- data_mem_addr_o izlazni port preko koga se šalje adresa sa koje podatak čita ili na koju se podatak upisuje.
- data_mem_write_o izlazni port preko koga se šalje vrednost koju treba upisati u memoriju.
- data_mem_we izlazni port preko koga se šalje signal dozvolje upisa.

Ako se pogleda slika 23 može se primetiti da je port dozvole upisa data_mem_we_o četvorobitan i razlog za to je memorija sa kojom se komunicira. Ona je napravljena tako da se u nju odjednom može upisati 1, 2, 3 ili 4 bajta u zavisnosti od vrednosti tog porta (informacija koja se upisuje nalazi se na data_mem_write_o portu). Sledeći primer ilustruje kako vrednost tog porta određuje koliko bajtova se upisuje:

data_mem_we_o = $1111_2$ => upisuju se 4 bajta
data_mem_we_o = $1110_2$ => upisuju se gornja 3 bajta
data_mem_we_o = $0110_2$ => upisuju se središnja 2 bajta bajta

U slučaju ovog procesora data_mem_we_o može da ima dve vrednosti: $0000_2$ ili $1111_2$. To je zato što za sada samo jedna instrukcija vrši upis u procesor (*sw*) i ona vrši upis informacije od 32 bita. Ukoliko se čitalac odluči da proširi set instrukcija koji procesor podržava i reši da implementira *sh (store halfword)* instrukciju, onda će prilikom izvršavanja ove instrukcije morati da na data_mem_we_o port postavi vrednost $0011_2$.

Sledeći kodni listing predstavlja vhdl deklaraciju entity dela TOP_RISCV celine:

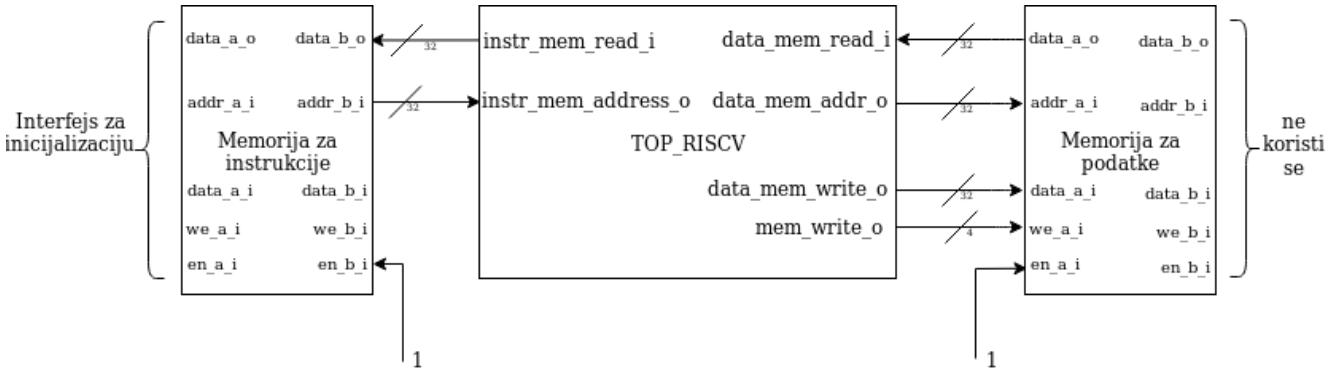
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity TOP_RISCV is
   generic (DATA_WIDTH : positive := 32);
   port(
      -- ********* Globalna sinhronizacija ******************
      clk                : in  std_logic;
```

```
      reset               : in  std_logic;
      -- ********* Interfejs ka Memoriji za instrukcije *****
      instr_mem_address_o : out std_logic_vector (31 downto 0);
      instr_mem_read_i    : in  std_logic_vector(31 downto 0);
      -- ********* Interfejs ka Memoriji za podatke *********
      data_mem_we_o       : out std_logic_vector(3 downto 0);
      data_mem_address_o  : out std_logic_vector(31 downto 0);
      data_mem_write_o    : out std_logic_vector(31 downto 0);
      data_mem_read_i     : in  std_logic_vector (31 downto 0));
end entity;
```

## 3.8 Verifikaciono okruženje.

Da bi se testirao procesor napisan je jednostavan *testbench.* Unutar njega je instanciran TOP_RISCV modul i dve memorije koje predstavljaju memoriju sa instrukcijama i memoriju sa podacima. Slika 24 to ilustruje:




Slika 24. Verifikaciono okruženje

Testiranja ispravnog rada procesore izvršeno je tako što se u memoriju za instrukcije upiše određeni program i nakon upisivanja pusti se da procesor izvrši taj program. Sledeći kodni listing implementira verifikacioni okruženje:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use work.txt_util.all;

entity TOP_RISCV_tb is
-- port ();
end entity;


architecture Behavioral of TOP_RISCV_tb is
   -- Operand za pristup asemblerskom kodu programa
   file RISCV_instructions             : text open read_mode is
"../../../../../RISCV_tb/assembly_code.txt";
   -- Globalni signali
   signal clk                          : std_logic := '0';
   signal reset                        : std_logic;
   -- Signali memorije za instrukcije
   signal ena_instr_s, enb_instr_s     : std_logic;
   signal wea_instr_s, web_instr_s     : std_logic_vector(3 downto 0);
   signal addra_instr_s, addrb_instr_s : std_logic_vector(9 downto 0);
   signal dina_instr_s, dinb_instr_s   : std_logic_vector(31 downto 0);
   signal douta_instr_s, doutb_instr_s : std_logic_vector(31 downto 0);
   signal addrb_instr_32_s             : std_logic_vector(31 downto 0);
   -- Signali memorije za podatke
   signal ena_data_s, enb_data_s       : std_logic;
   signal wea_data_s, web_data_s       : std_logic_vector(3 downto 0);
   signal addra_data_s, addrb_data_s   : std_logic_vector(9 downto 0);
   signal dina_data_s, dinb_data_s     : std_logic_vector(31 downto 0);
   signal douta_data_s, doutb_data_s   : std_logic_vector(31 downto 0);
   signal addra_data_32_s              : std_logic_vector(31 downto 0);

begin

   -- Memorija za instrukcije
   -- Pristup A : Koristi se za inicijalizaciju memorije za instrukcije
   -- Pristup B : Koristi se za citanje instrukcija od strane procesora
   -- Konstante:
   ena_instr_s   <= '1';
   enb_instr_s   <= '1';
   addrb_instr_s <= addrb_instr_32_s(9 downto 0);
   web_instr_s   <= (others => '0');
   dinb_instr_s  <= (others => '0');
   -- Instanca:
   instruction_mem : entity work.BRAM(behavioral)
```

```
   generic map(WADDR => 10)
   port map (clk      => clk,
             -- pristup A
             en_a_i   => ena_instr_s,
             we_a_i   => wea_instr_s,
             addr_a_i => addra_instr_s,
             data_a_i => dina_instr_s,
             data_a_o => douta_instr_s,
             -- pristup B
             en_b_i   => enb_instr_s,
             we_b_i   => web_instr_s,
             addr_b_i => addrb_instr_s,
             data_b_i => dinb_instr_s,
             data_b_o => doutb_instr_s);


-- Memorija za podatke
-- Pristup A : Koristi procesor kako bi upisivao i citao podatke
-- Pristup B : Ne koristi se
-- Konstante:
addra_data_s <= addra_data_32_s(9 downto 0);
addrb_data_s <= (others => '0');
dinb_data_s  <= (others => '0');
ena_data_s   <= '1';
enb_data_s   <= '1';
-- Instanca:
data_mem : entity work.BRAM(behavioral)
   generic map(WADDR => 10)
   port map (clk      => clk,
             -- pristup A
             en_a_i   => ena_data_s,
             we_a_i   => wea_data_s,
             addr_a_i => addra_data_s,
             data_a_i => dina_data_s,
             data_a_o => douta_data_s,
             -- pristup B
             en_b_i   => enb_data_s,
             we_b_i   => web_data_s,
             addr_b_i => addrb_data_s,
             data_b_i => dinb_data_s,
             data_b_o => doutb_data_s);


-- Top Modul - RISCV procesor jezgro
TOP_RISCV_1 : entity work.TOP_RISCV
   port map (
      clk   => clk,
```

```
        reset => reset,

        instr_mem_read_i    => doutb_instr_s,
        instr_mem_address_o => addrb_instr_32_s,

        data_mem_we_o      => wea_data_s,
        data_mem_address_o => addra_data_32_s,
        data_mem_read_i    => douta_data_s,
        data_mem_write_o   => dina_data_s);

   -- Inicijalizacija memorije za instrukcije
   -- Program koji ce procesor izvrsavati se ucitava u memoriju
   read_file_proc : process
      variable row : line;
      variable i   : integer := 0;
   begin
      reset       <= '0';
      wea_instr_s <= (others => '1');
      while (not endfile(RISCV_instructions))loop
         readline(RISCV_instructions, row);
         addra_instr_s <= std_logic_vector(to_unsigned(i, 10));
         dina_instr_s  <= to_std_logic_vector(string(row));
         i             := i + 4;
         wait until rising_edge(clk);
      end loop;
      wea_instr_s <= (others => '0');
      reset       <= '1' after 20 ns;
      wait;
   end process;

   -- klok signal generator
   clk_proc : process
   begin
      clk <= '1', '0' after 100 ns;
      wait for 200 ns;
   end process;

end architecture;
```

Ako se pogleda kod, može se videti da su u njemu instancirani i povezani procesor i dve memorije. Pored toga, napisan je jednostavan kod koji vrši čitanje fajla u kome se nalazi program, njime se inicijalizuje memorija sa instrukcijama, odnosno vrši se upis instrukcija u memoriju.

Verifikacionom okruženju je neophodna VHDL implementacija RAM memorije i sledeći kodni listing predstavlja jednu vrstu implementacije:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity BRAM is
   generic
      (
         WADDR : natural := 10
         );
   port
      (
         clk      : in  std_logic;
         en_a_i   : in  std_logic;
         en_b_i   : in  std_logic;
         data_a_i : in  std_logic_vector(31 downto 0);
         data_b_i : in  std_logic_vector(31 downto 0);
         addr_a_i : in  std_logic_vector(WADDR - 1 downto 0);
         addr_b_i : in  std_logic_vector(WADDR - 1 downto 0);
         we_a_i   : in  std_logic_vector(3 downto 0);
         we_b_i   : in  std_logic_vector(3 downto 0);
         data_a_o : out std_logic_vector(31 downto 0);
         data_b_o : out std_logic_vector(31 downto 0)
         );

end BRAM;

architecture behavioral of BRAM is type ram_type is array(0 to 2**WADDR - 1)
                                       of std_logic_vector(7 downto 0); signal ram_s
:
                                                                          ram_type
:= (others => (others => '0'));

begin

   -- sinhroni upis
   process(clk)
   begin
      if(rising_edge(clk)) then
         if(en_a_i = '1') then
            if(we_a_i(3) = '1') then
               ram_s(to_integer(unsigned(addr_a_i)+3)) <= data_a_i(31 downto 24);
            end if;
            if(we_a_i(2) = '1') then
               ram_s(to_integer(unsigned(addr_a_i)+2)) <= data_a_i(23 downto 16);
            end if;
            if(we_a_i(1) = '1') then
               ram_s(to_integer(unsigned(addr_a_i)+1)) <= data_a_i(15 downto 8);
```

```
            end if;
            if(we_a_i(0) = '1') then
               ram_s(to_integer(unsigned(addr_a_i))) <= data_a_i(7 downto 0);
            end if;
         end if;
         if(en_b_i = '1') then
            if(we_b_i(3) = '1') then
               ram_s(to_integer(unsigned(addr_b_i)+3)) <= data_b_i(31 downto 24);
            end if;
            if(we_b_i(2) = '1') then
               ram_s(to_integer(unsigned(addr_b_i)+2)) <= data_b_i(23 downto 16);
            end if;
            if(we_b_i(1) = '1') then
               ram_s(to_integer(unsigned(addr_b_i)+1)) <= data_b_i(15 downto 8);
            end if;
            if(we_b_i(0) = '1') then
               ram_s(to_integer(unsigned(addr_b_i))) <= data_b_i(7 downto 0);
            end if;
         end if;
      end if;
   end process;

   -- asinhrono citanje
   process(en_a_i, en_b_i, addr_a_i, addr_b_i)
   begin
      if(en_a_i = '1') then
         data_a_o(31 downto 24) <= ram_s(to_integer(unsigned(addr_a_i)+3));
         data_a_o(23 downto 16) <= ram_s(to_integer(unsigned(addr_a_i)+2));
         data_a_o(15 downto 8)  <= ram_s(to_integer(unsigned(addr_a_i)+1));
         data_a_o(7 downto 0)   <= ram_s(to_integer(unsigned(addr_a_i)));
      end if;
      if(en_b_i = '1') then
         data_b_o(31 downto 24) <= ram_s(to_integer(unsigned(addr_b_i)+3));
         data_b_o(23 downto 16) <= ram_s(to_integer(unsigned(addr_b_i)+2));
         data_b_o(15 downto 8)  <= ram_s(to_integer(unsigned(addr_b_i)+1));
         data_b_o(7 downto 0)   <= ram_s(to_integer(unsigned(addr_b_i)));
      end if;
   end process;

end behavioral;
```

## 3.9 Uputstvo za pokretanje simulacije

Kada se realizuju TOP_RISCV celina, odnosno kada se realizuje procesor, on se može instancirati u prethodno opisano verifikaciono okruženje i testirati. U tački 3.8 je rečeno da verifikaciono okruženje prilikom pokretanja simulacije učitava u memoriju za instrukcije program koji je potrebno izvršiti. Fajl iz koga se program čita zove se *assembly_code.txt* i u njemu su upisane instrukcije u binarnom formatu. Svaka linija u fajlu predstavlja jednu instrukciju. Ukoliko čitalac želi da promeni program koji procesor izvršava, neophodno je da promeni *assembly_code.txt* fajl i upiše u njega druge instrukcije. Jednostavan način da se to uradi jeste da se iskoristi simulator opisan u prethodnoj vežbi, koji može da generiše mašinski kod u binarnom formatu.Kada se to uradi samo je potrebno kopirati binarni kod koji je generisao simulator u *assembly_code.txt* fajl i pokrenuti simulaciju.

**Napomena1:** Prilikom korišćenja ovog fajla, neophodno je navesti putanju do njega kako bi Vivado mogao da ga uključi prilikom kompajliranja, iz tog razloga sledeću liniju u verifikacionom okruženju treba promeniti na sledeći način:

```
file RISCV_instructions             : text open read_mode is
"../../../../../RISCV_tb/assembly_code.txt";
                        |
                        | gornju liniju promeniti u donju (navesti svoju apsoultnu putanju)
                        |
file RISCV_instructions             : text open read_mode is
"/home/nikola/Documents/git_repos/RISCV_VHDL/RV32I/RISCV_tb/assembly_code.txt";
```

Da bi se zaključilo da procesor ispravno radi, potrebno je posmatrati stanje na I/O portovima procesora i njegove unutrašnje signale.

**Napomena1: Da bi vefikaciono okruženje funkcionisalo neophodno je dodati txt_util.vhd fajl u projekat. On se nalazi u priloženom materijalu.**