



ELEKTROTEHNIČKI FAKULTET
BEOGRAD

**SIMULATOR PROCESORA RAČUNARSKOG
SISTEMA ZA RAD U LABORATORIJI**

Diplomski rad

profesor: **dr JOVAN ĐORĐEVIĆ**

student: **Zoran Živanović 2005/0404**

Beograd

oktobar, 2009.

Sadržaj

1	Uvod.....	- 3 -
2	Arhitektura i organizacija procesora.....	- 5 -
2.1	Arhitektura procesora.....	- 5 -
2.2	Organizacija procesora.....	- 6 -
2.2.1	Operaciona jedinica	- 6 -
2.2.2	Upravljačka jedinica	- 18 -
3	Korisnički interfejs.....	- 21 -
3.1	Pokretanje simulatora.....	- 21 -
3.2	Kontrola rada simulatora i navigacija	- 23 -
3.3	Pregledanje stanja registara	- 27 -
4	Softverska realizacija.....	- 28 -
4.1	Pomoćni programi.....	- 28 -
4.2	Softverska realizacija procesora	- 34 -
5	Testovi za laboratorijski rad	- 41 -
5.1	Skup testova za samostalni rad.....	- 41 -
5.2	Detaljan pregled odabranog testa	- 49 -
6	Zaključak.....	- 61 -
7	Literatura.....	- 63 -

1 UVOD

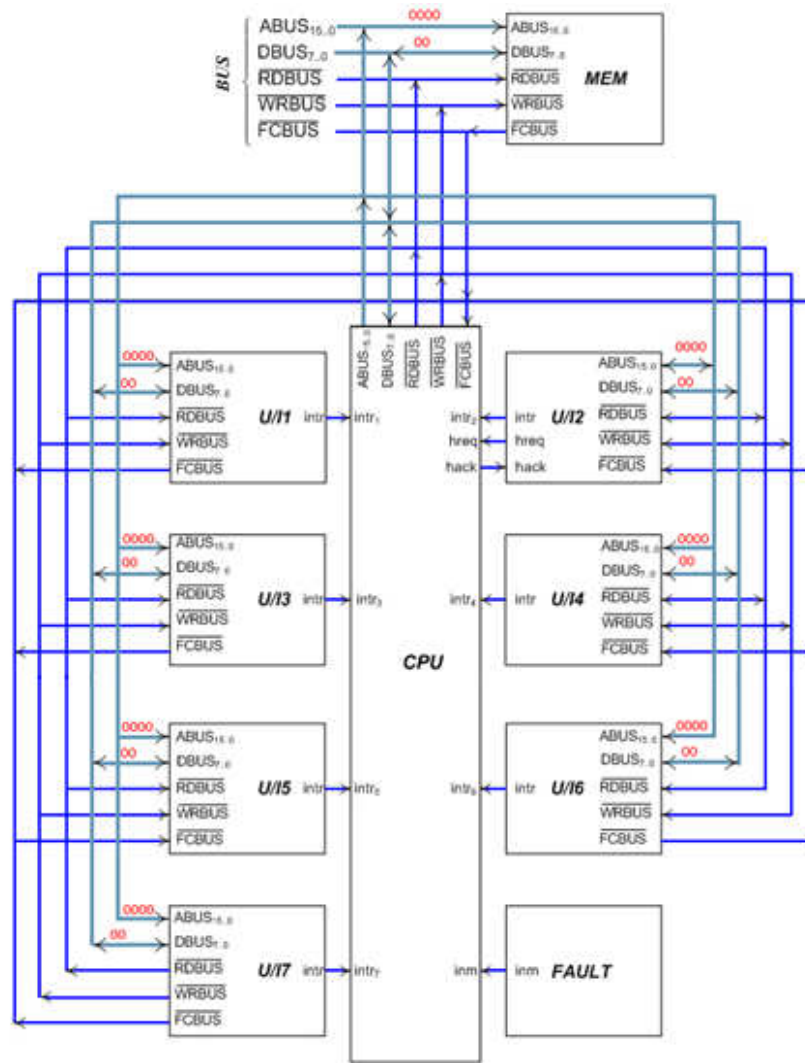
Računari su danas zastupljeni u svim sverama ljudskog društva. Dalji naučni razvoj na svim poljima je u direktnoj zavisnosti od razvoja računara, jer se zahvaljujući računarima pre same realizacije nekog projekta može testirati ispravnost rada kroz simulaciju realnih situacija. Savremeni vidovi komunikacije se ne mogu ostvariti bez upotrebe računara. Pa čak i moderna poljoprivreda zahteva upotrebu računara za regulisanje navodnjavanja obradivih površina, doziranje toplote, hrane i vode u prostorijama za uzgoj životinja, itd.

Dalji razvoj računarskih tehnologija je u direktnoj vezi sa obrazovanjem mladih inženjera. Na visokoškolskim ustanovama se izučava funkcionisanje računara i dizajn računarskih sistema. Prema preporukama IEEE se pored predavanja i vežbi na table studentima treba omogućiti i laboratorijski rad. Laboratorijski rad može da se obavlja na hardveru ili, kako je danas najčešće, korišćenjem softverskih vizuelnih simulatora. Kroz vizuelni simulator student proverava svoje znanje i na konkretnom primeru vidi sva dešavanja u sistemu.

Zadatak ovog projekta je da se za zadati dizajn računarskog sistema napravi vizuelni simulator, koji će se koristiti za učenje osnovnih principa arhitekture i organizacije računara. Računarski sistem se sastoj od procesora, memorije, 7 kontrolera za ulaz/izlaz (slika 1.). Jedan kontroler je sa direktnim pristupom memoriji. Dva kontrolera su bez direktnog pristupa memoriji. Ostala četiri kontrolera služe za generisanje takta(tajmeri). Memorija je kapaciteta 60KB i pristupa joj se preko asinhrona magistrale. Detalji o ulazno/izlaznom podsistemu i memoriji su dati u literaturi [2]. U daljem tekstu će se akcenat staviti na procesor. Takođe su kreirani test primeri koji se koriste za rad u laboratoriji.

U drugoj glavi je dat kratak pregled arhitekture i organizacije procesa. Opisani su delovi operacione i upravljačke jedinice procesora. Nabrojeni su elementi arhitekture programski dostupni registri, tipovi podataka, format instrukcija, načini adresiranja, skup instrukcija i mehanizam prekida. Organizacija procesora je podeljena na operacionu i upravljačku jedinicu. Operaciona jedinica je realizovana sa direktnim vezama, a upravljačka jedinica je realizovana kao ožičena. Deo teksta i slike su uzete iz literature [1] uz saglasnost autora dela.

U trećoj glavi je objašnjen korisnički interfejs simulatora. Interfejs je napravljan tako da bude intuitivno jasan svim grupama korisnika. Objašnjen je postupak kreiranja simulacije korišćenjem programskog paketa koji je napravljen kao dodatak simulatoru i pokretanja simulacije učitavanjem programa u memoriju simulatora. Dat je opis svih načina praćenja toka simulacije i navigaciju kroz sistem. Prikazano je kako se pregledaju i menjaju stanja registara procesora.



Slika 1. Struktura sistema

Četvrta glava sadrži prikaz softverske realizacije. Dati su dijagrami klasa i dijagrami toka karakterističnih događaja u simulatoru. Objašnjeni su pomoćni programi koji su kreirani da bi pojednostavili kreiranje simulatora ili kasniji rad sa simulatorom. Tokom izgradnje simulatora ispoštovan je interfejs koji je kreiran i važi za ceo sistem, tako da se dve celine, procesor na jednoj i ulaz/izlaz na drugoj strani, mogu spojiti u jednu funkcionalnu celinu.

U petoj glavi je dat skup test primara koji su prikladno izabrani da se najbolje prikažu svi delovi i sve situacije u kojim sistem može da se nalazi. Odabran je jedan test primer koji je detaljno prokomentarisam i za svaki karakterističan trenutak predstavljene su slike sa detaljima hardvera. Kroz niz slika je predstavljen postupak simulacije i prikazani ključni momenti tokom jednog izvršavanja instrukcije.

2 ARHITEKTURA I ORGANIZACIJA PROCESORA

U ovoj glavi je dat sažet opis arhitekture i organizacije procesora. Detaljni opisi svake od ovih stavki je dat u literatura [1].

2.1 ARHITEKTURA PROCESORA

U daljem tekstu će biti ukratko razmotren svaki od elemenata arhitekture procesora.

Programski dostupni registri procesora su: programski brojač PC, akumulator AB, akumulator AW, 32 registra opšte namene GPR, ukazivač na vrh steka SP, programska statusna reč PSW, registar maske IMR i ukazivač na tabelu adresa prekidnih rutina IVTP.

Tipovi podataka koji se koriste u ovom procesoru su celobrojne 8-mo bitne veličine sa znakom i bez znaka, 8-mo bitne binarne reči i 16-to bitne binarne reči. Celobrojne 8-mo bitne veličine sa znakom i bez znaka se koriste kod operacije prenosa, aritmetičkih operacija i operacija pomeranja i rotiranja nad 8-mo bitnim veličinama, 8-mo bitne binarne reči se koriste kod logičkih operacija i operacija pomeranja i rotiranja, a 16-to bitne binarne reči se koriste kod operacija prenosa.

U procesoru se koriste sledeći formati instrukcija: format bezadresnih instrukcija, format instrukcije prekida, format instrukcija relativnog skoka – B format, format instrukcija apsolutnog skoka – J format, format jednoadresnih registarskih instrukcija – R format, format jednoadresnih neposrednih instrukcija – IB format, format jednoadresnih neposrednih instrukcija – IW format i format jednoadresnih memorijskih instrukcija – AP format.

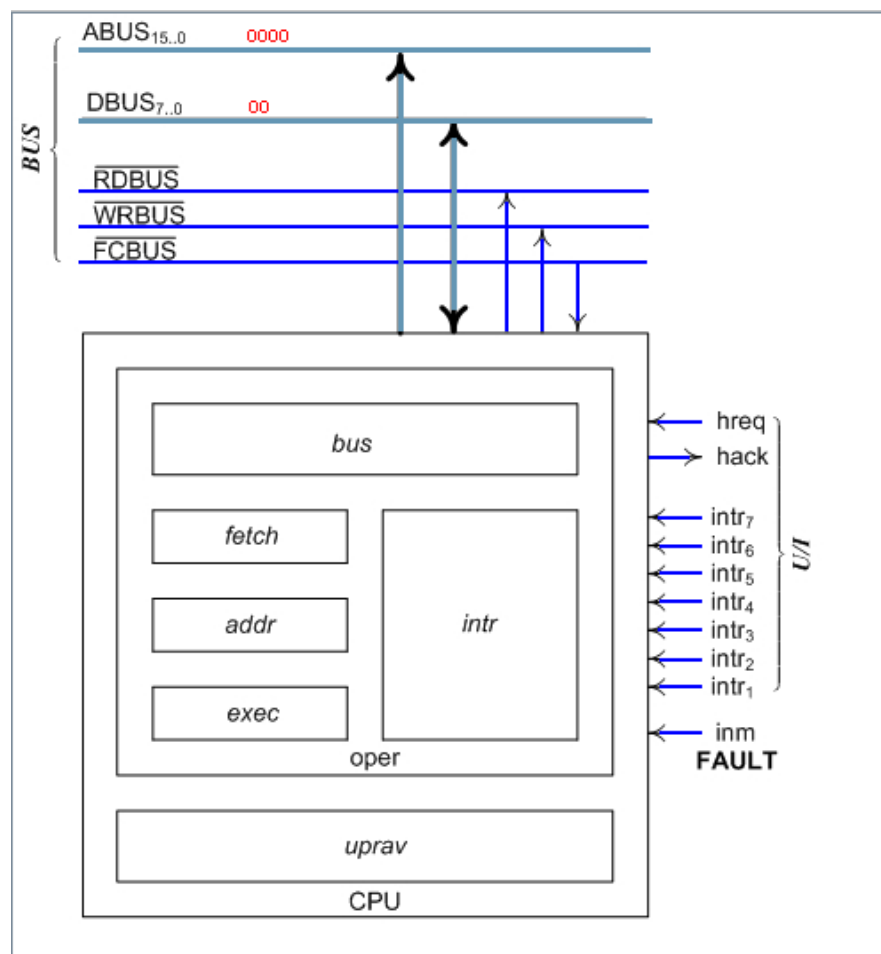
Procesor ima 8 načina adresiranja: registarsko direktno (AM:000), registarsko indirektno (AM:001), memorijsko direktno (AM: 010), memorijsko indirektno (AM: 011), registarsko indirektno sa pomerajem (AM: 100), bazno indeksno sa pomerajem (AM: 101), PC relativno sa pomerajem (AM: 110), neposredno (111).

Instrukcije procesora se mogu svrstati u sledećih osam grupa: instrukcija bez dejstva, instrukcija zaustavljanja, instrukcije skoka, instrukcije prenosa, aritmetičke instrukcije, logičke instrukcije, instrukcije pomeranja i rotiranja, instrukcije postavljanja indikatora u PSW.

Zahteve za prekid mogu da generišu: sedam kontrolera periferija po linijama $intr_7$ do $intr_1$ da bi signalizirali spremnost za prenos podataka (maskirajući prekidi), jedan uređaj računara koji kontroliše ispravnost napona napajanja (nemaskirajući prekidi), procesor, kao rezultat otkrivene nekorektnosti u izvršavanju tekuće instrukcije (nelegalan kod operacije i nelegalno adresiranje), procesor, ako je zadat takav režim rada procesora, kroz postavljanje indikatora T u programskoj statusnoj reči PSW, da se posle svake instrukcije skače na određenu prekidnu rutinu i procesor kao rezultat izvršavanja instrukcije prekida INT. Maskirajući i nemaskirajući prekidi su spoljašnji prekidi, a ostali unutrašnji. Procesor podržava gnežđenje prekida. Instrukcija RTI ne reaguje na prekid posle svake instrukcije.

2.2 ORGANIZACIJA PROCESORA

U ovoj glavi se daje organizacija procesora **CPU** koji se sastoji iz operacione jedinice **oper** i upravljačke jedinice **uprav** (slika 2.).



Slika 2. Organizacija procesora

Struktura i opis operacione i upravljačke jedinice se daju u daljem tekstu.

2.2.1 Operaciona jedinica

Operaciona jedinica **oper** je kompozicija kombinacionih i sekvencijalnih prekidačkih mreža koje služe za pamćenje binarnih reči, izvršavanje mikrooperacija i generisanje signala logičkih uslova upravljačke jedinice **uprav**.

Operaciona jedinica **oper** (slika 2.) se sastoji od sledećih blokova: blok **bus**, blok **fetch**, blok **addr**, blok **exec** i blok **intr**.

Ovi blokovi su međusobno povezani direktnim vezama.

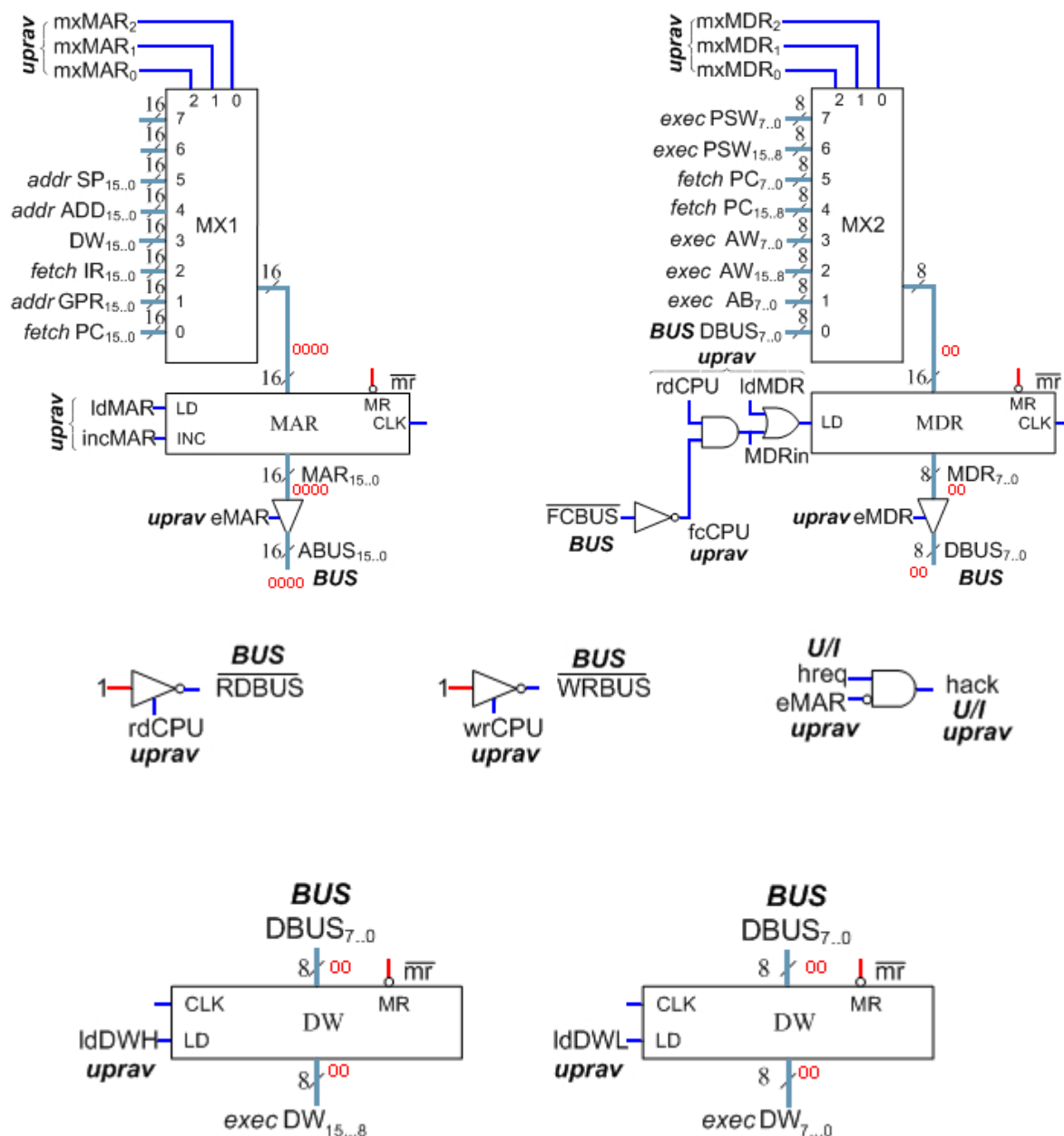
Blok **bus** (slika 3.) služi za arbitraciju procesora i ulazno/izlaznog uređaja **U/I2** sa kontrolerom za direktan pristup memoriji pri korišćenju magistrale **BUS** i realizaciju ciklusa na magistrali **BUS**. Blok **fetch** (slike 4, 5, 6) služi za čitanje instrukcije i smeštanje u prihvatni registar instrukcije. Blok **addr** (slika 7) služi za formiranje adrese operanda i čitanje operanda. Blok **exec** (slike 7, 8, 9 i 10) služi za izvršavanje operacija. Blok **intr** (slike 11, 12 i

13) služi za prihvatanje prekida i generisanje broja ulaza u tabelu sa adresama prekidnih rutina.

Struktura i opis blokova operacione jedinice *oper* se daju u daljem tekstu.

2.2.1.1 Blok bus

Blok *bus* (slika 2) sadrži registre $MAR_{15..0}$ i $MDR_{7..0}$ sa multiplekserima MX1 i MX2, respektivno, prihvatni registar $DW_{15..0}$ i kombinacione mreže za realizaciju ciklusa na magistrali **BUS** kada je procesor gazda i za arbitraciju sa kontrolerom sa direktnim pristupom memoriji pri korišćenju magistrale **BUS**.



Slika 3. Blok *bus*

Registar $MAR_{15..0}$ je 16-to razredni adresni registar, čiji se sadržaj normalno koristi pri realizaciji ciklusa čitanja ili upisa na magistrali **BUS**. Adresa se iz nekog od blokova

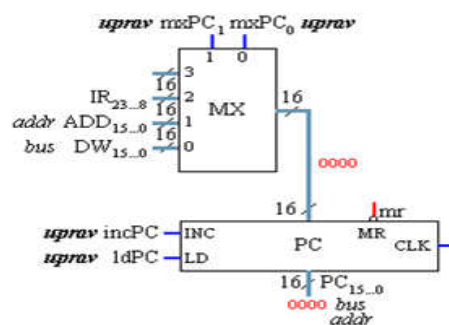
operacione jedinice **oper** preko multipleksera MX1 vodi na ulaze registra $MAR_{15...0}$ i u njega upisuje generisanjem aktivne vrednosti signala **ldMAR**. Sadržaj registra $MAR_{15...0}$ se inkrementira generisanjem aktivne vrednosti signala **incMAR**. Ovo se koristi u situacijama kada treba pročitati ili upisati 16-to bitnu veličinu koja se nalazi u dvema susednim 8-mo bitnim lokacijama. Tada se najpre u registar $MAR_{15...0}$ upisuje adresa niže lokacije, a posle se inkrementiranjem dobija adresa više lokacije. Pri realizaciji ciklusa čitanja ili upisa formira se aktivna vrednost signala **eMAR**, čime se sadržaj registra $MAR_{15...0}$ propušta kroz bafere sa tri stanja na adresne linije $ABUS_{15...0}$ magistrale **BUS**.

Registar $MDR_{7...0}$ je 8-mo razredni registar podatka u koji se generisanjem aktivne vrednosti jednog od signala **MDRin** i **ldMDR** upisuje sadržaj sa izlaza multipleksera MX2. Pri aktivnoj vrednosti signala **MDRin** i binarnoj vrednosti 000 signala **mxMDR₂**, **mxMDR₁** i **mxMDR₀** sadržaj sa linija podataka $DBUS_{7...0}$ magistrale **BUS** se propušta kroz multiplekser MX2 i upisuje u registar $MDR_{7...0}$. Pri aktivnoj vrednosti signala **ldMDR** i binarnim vrednostima 001 do 111 signala **mxMDR₂**, **mxMDR₁** i **mxMDR₀** jedan od sadržaja $AB_{7...0}$, $AW_{15...8}$, $AW_{7...0}$, $PC_{15...8}$, $PC_{7...0}$, $PSW_{15...8}$ i $PSW_{7...0}$ se propušta kroz multiplekser MX2 i upisuje u registar $MDR_{7...0}$. Signal **MDRin** je aktivan kada se generišu aktivne vrednosti signala **rdCPU** i **fcCPU**. Signal **rdCPU** je aktivan kada procesor kao gazda realizuje ciklus čitanja na magistrali, dok je signal **fcCPU** aktivan kada se na upravljačkoj liniji magistrale **FCBUS** pojavi aktivna vrednost kao indikacija od memorije ili nekog od kontrolera kao sluge da je pročitani podatak raspoloživ na linijama podataka $DBUS_{7...0}$ magistrale. Sadržaj registra $MDR_{7...0}$ se pri aktivnoj vrednosti signala **eMDR** propušta kroz bafere sa tri stanja na linije podataka $DBUS_{7...0}$ magistrale.

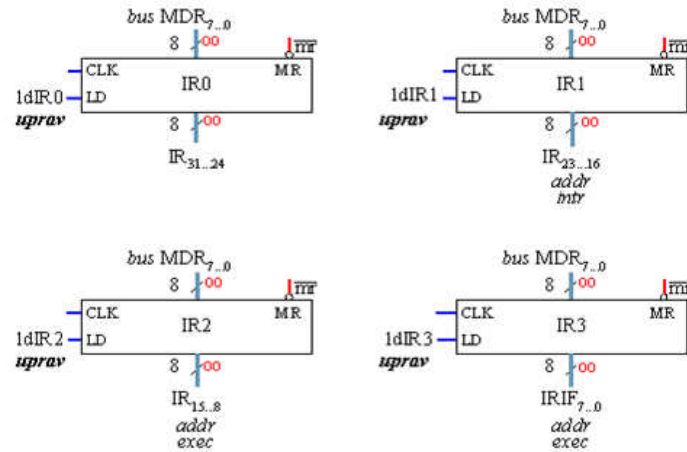
Registar $DW_{15...0}$ je 16-to razredni pomoćni registar koji se koristi za prihvatanje 16-to bitne veličine koja se dobija iz dve susedne 8-mo bitne memorijske lokacije u dva posebna ciklusa na magistrali. Takođe se koristi da se prihvati 16-to bitna adrese operanda u slučaju indirektnog memorijskog adresiranja i da se posle upiše u registar $MAR_{15...0}$. Pored toga registar $DW_{15...0}$ se koristi da se prihvati 16-to bitni operand u slučaju instrukcija koje se izvršavaju nad 16-to bitnim veličinama i da se posle upiše u registar $BW_{15...0}$. Registar $DW_{15...0}$ se koristi i da se prihvati 16-to bitna vrednost sa steka ili 16-to bitna vrednost adrese prekidne rutine iz tabele sa adresama prekidnih rutina i da se posle upiše u programski brojač $PC_{15...0}$.

2.2.1.2 Blok fetch

Blok *fetch* sadrži registar $PC_{15...0}$ sa multiplekserom MX, registre IR0, IR1, IR2 i IR3 (slika 4.), dekodere DC1 do DC11 signala logičkih uslova operacija i načina adresiranja (slika 5.) i kombinacione mreže signala logičkih uslova dužina instrukcija (slika 6.).

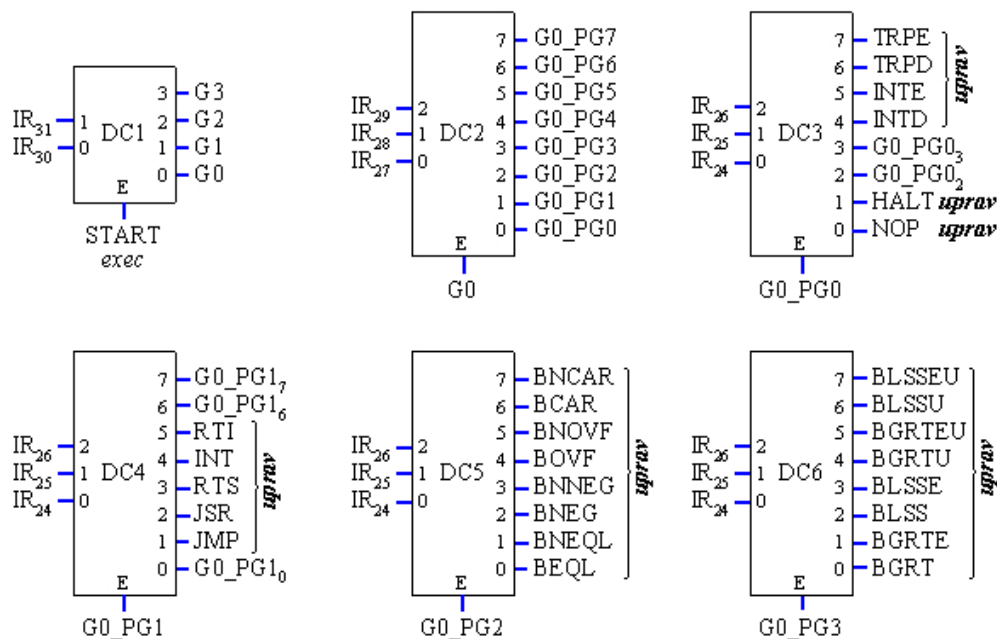


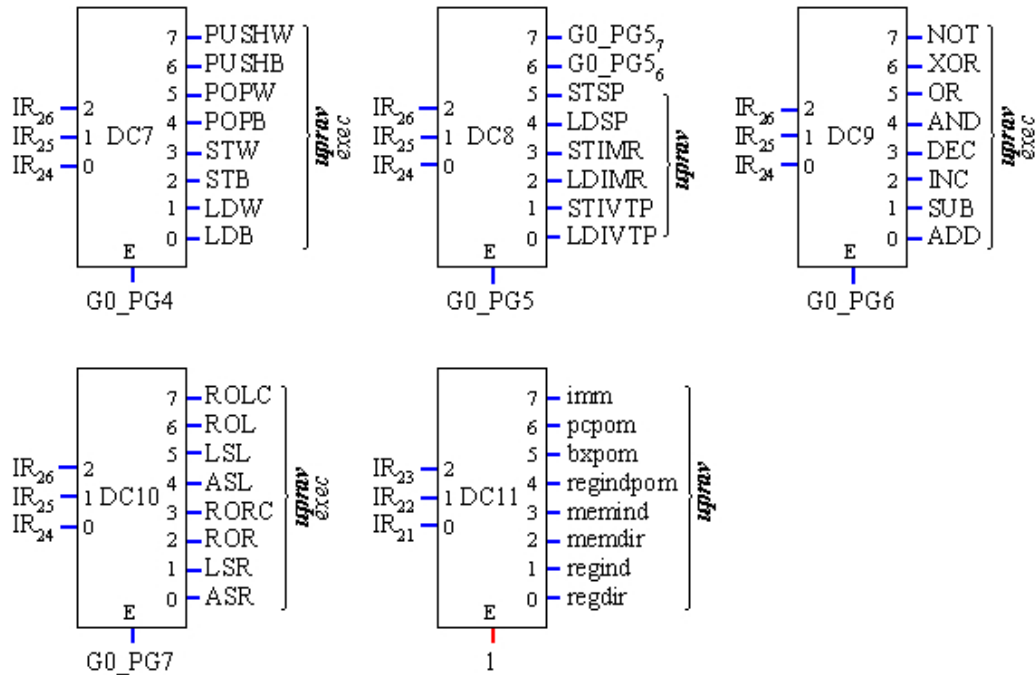
Slika 4. Blok *fetch* (prvi deo)

Slika 4(nastavak) . Blok **fetch** (prvi deo)

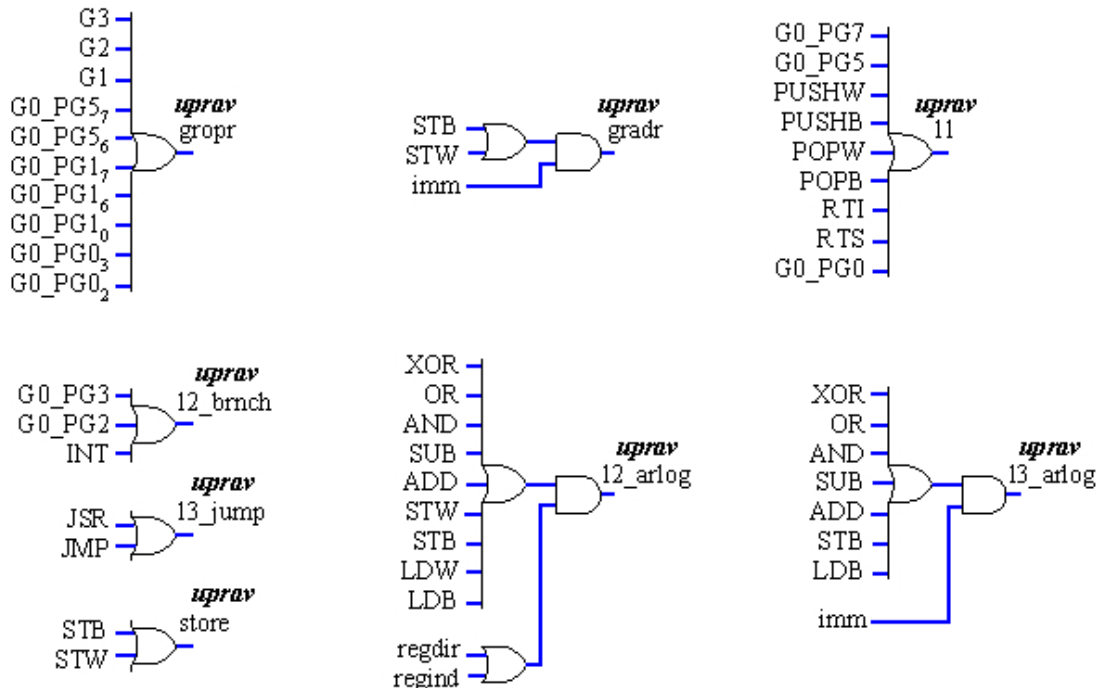
Registar $PC_{15...0}$ je 16-to razredni programski brojač čiji sadržaj predstavlja adresu memorijske lokacije počev od koje treba pročitati jedan do četiri bajta instrukcije. Adresa skoka u programu se iz nekog od blokova operacione jedinice **oper** preko multipleksera MX vodi na ulaze registra $PC_{15...0}$ i u njega upisuje generisanjem aktivne vrednosti signala **ldPC**. Sadržaj registra $PC_{15...0}$ se inkrementira generisanjem aktivne vrednosti signala **incPC**. Ovo se koristi prilikom čitanja svakog bajta instrukcije koji se nalaze u susednim 8-mo bitnim lokacijama. Sadržaj registra $PC_{15...0}$ se koristi u bloku **addr** i za formiranje adrese memorijske lokacije kada se za adresiranje operanda koristi PC relativno adresiranje

Registri IR0, IR1, IR2 i IR3 su 8-mo razredni registri koji formiraju razrede 31...24, 23...16, 15...8 i 7...0, respektivno, prihvatnog registra instrukcije $IR_{31...0}$. Instrukcije mogu, u zavisnosti od formata instrukcije, da budu dužine 1, 2, 3 ili 4 bajta. Saglasno formatu instrukcije prvi, drugi, treći i četvrti bajt instrukcije se smeštaju redom u registre IR0, IR1, IR2 i IR3.

Slika 5 . Blok **fetch** (drugi deo)

Slika 5(nastavak) . Blok **fetch** (drugi deo)

Dekoderi DC1 do DC12 se koriste za dekodovanje instrukcija i formiranje signala logičkih uslova operacija NOP, ... , ROLC i načina adresiranja regdir, ..., imm (slika 5) saglasno načinu kodiranja instrukcija i načina adresiranja.

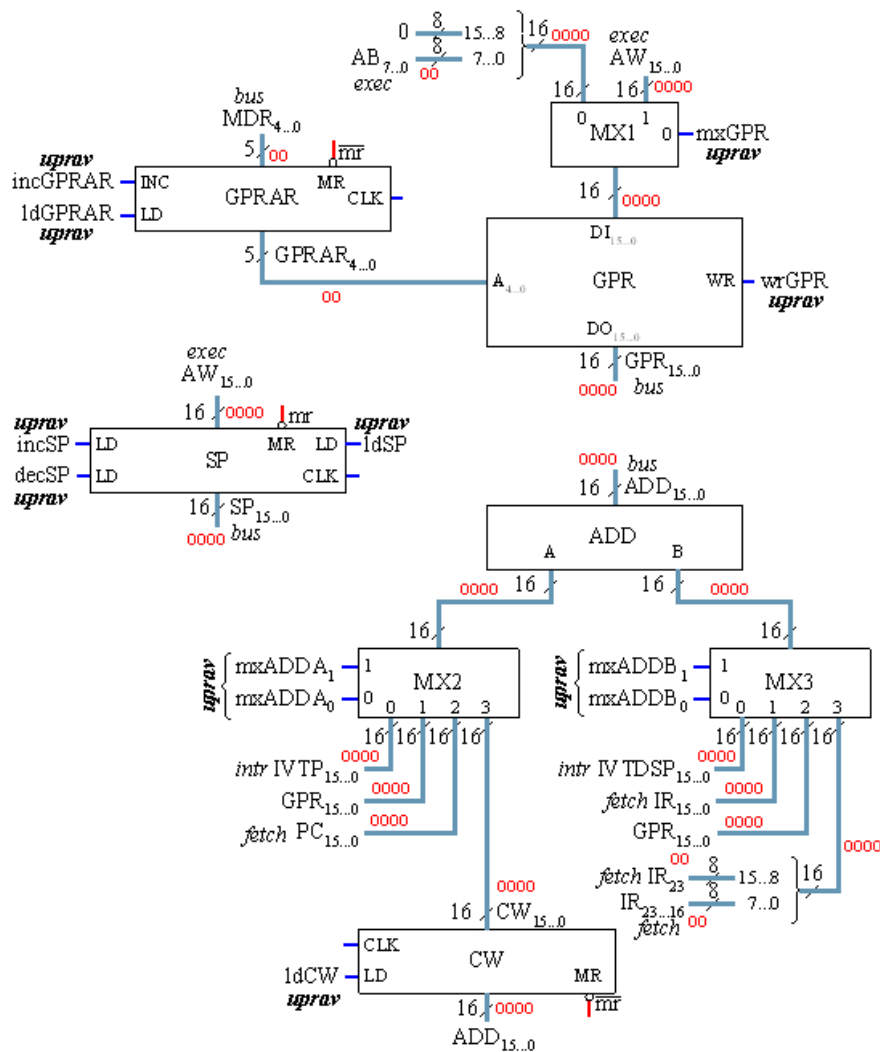
Slika 6 . Blok **fetch** (treći deo)

Kombinacione mreže signala logičkih uslova greška operacije, greška adresiranja i dužine instrukcija formiraju signale koji ukazuje da li postoji greška u pročitanoj instrukciji i ukoliko ne postoji kolika je dužina instrukcije u bajtovima (slika 6).

2.2.1.3 Blok addr

Blok addr sadrži registre opšte namene GPR sa adresnim registrom registara opšte namene GPRAR4...0 i multiplekserom MX1, sabirač ADD sa multiplekserima MX2 i MX3, pomoćni registar CW15...0 i ukazivač na vrh steka SP15...0 (slika 7).

Adresni registar opšte namene GPRAR4...0 je 5-to razredni registar čiji se sadržaj koristi kao adresa registra registarskog fajla prilikom upisa u registarki fajl i čitanja iz registarskog fajla. U registar GPRAR4...0 se pri aktivnoj vrednosti signala **ldGPRAR** upisuju razredi 4...0 prihvatnog registra podatka MDR7...0 bloka *bus*. Sadržaj registra GPRAR4...0 se inkrementira pri aktivnoj vrednosti signala **incGPRAR**, što se koristi samo ukoliko se radi o bazno indeksnom adresiranju sa pomerajem. Tada se registar opšte namene čija je adresa zadata bitovima 5...0 drugog bajta instrukcije koristi kao bazni registar, a registar opšte namene sa prve sledeće adrese kao indeksni registar. Sadržaj registra GPRAR4...0 se vodi na adresne linije A4...0 registarskog fajla GPR.



Slika 7. Blok addr

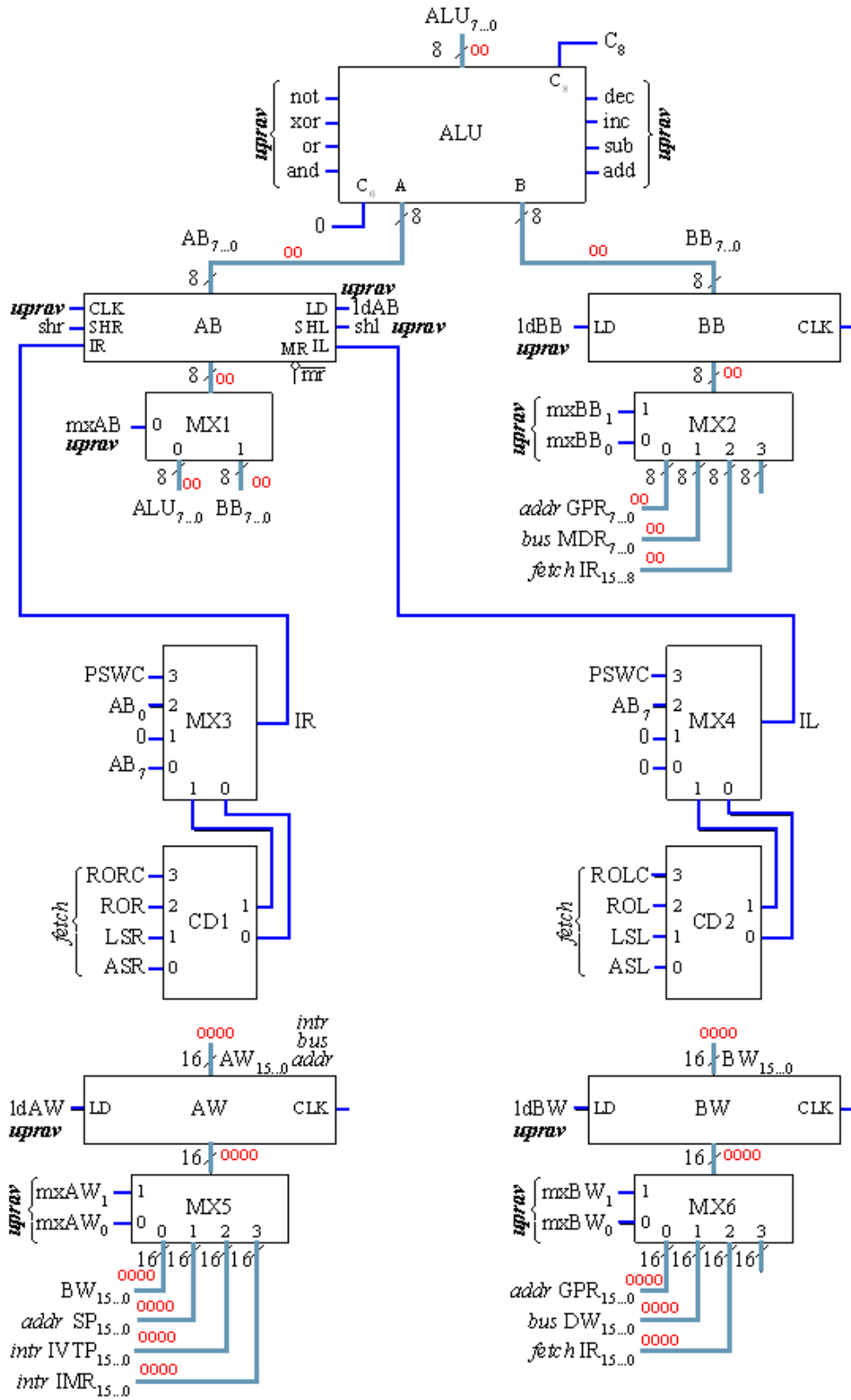
2.2.1.4 Blok *exec*

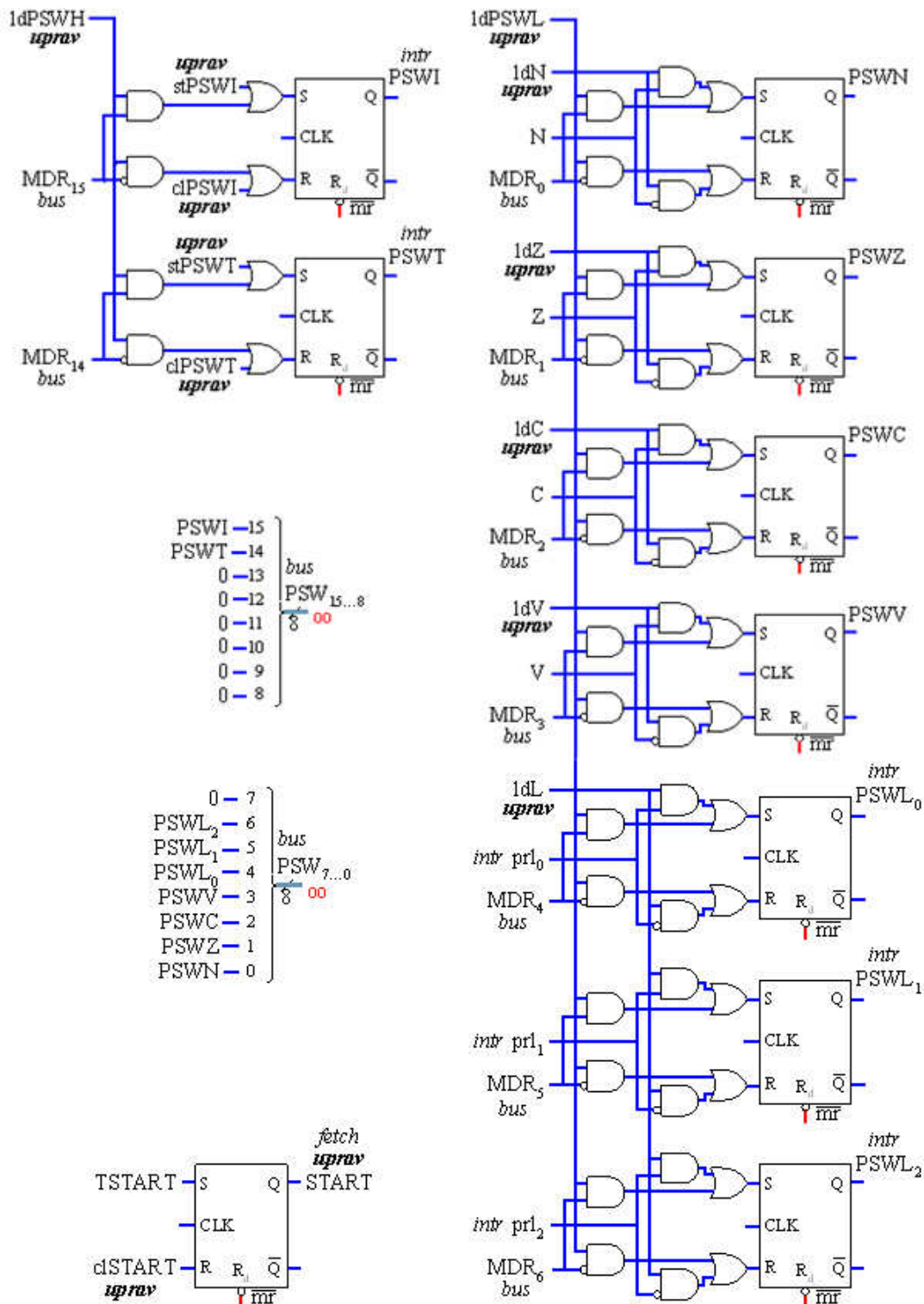
Blok *exec* sadrži aritmetičko logičku jedinicu ALU, registre $AB_{7...0}$ i $BB_{7...0}$, multipleksere MX1, MX2, MX3 i MX4, kodere DC1 i DC2, registre $AW_{15...0}$ i $BW_{15...0}$, multipleksere MX5 i MX6 (slika 8), registar $PSW_{15...0}$, flip-flop START (slika 9), kombinacione mreže za formiranje signala postavljanja indikatora N, Z, C i V (slika 10) i kombinacione mreže za formiranje signala rezultata operacija **eql**, ..., **nneg**, **brpom** (slika 11).

Aritmetičko logička jedinica ALU realizacije četiri aritmetičke i četiri logičke mikrooperacije nad sadržajima registara $AB_{7...0}$ i $BB_{7...0}$ (slike 6). Mikrooperacija koju treba realizovati se specificira aktivnom vrednošću ili jednog od upravljačkih signala **add**, **sub**, **inc** i **dec** za jednu od aritmetičkih mikrooperacija sabiranja, oduzimanja, inkrementiranja i dekrementiranja, respektivno, ili jednog od upravljačkih signala **and**, **or**, **xor** i **not** za jednu od logičkih mikrooperacija I, ILI, ekskluzivno ILI i komplementiranja, respektivno. Rezultat realizovane mikrooperacije se dobija na linijama $ALU_{7...0}$. Na ulaz C_0 je dovedena vrednost 0, pri čemu je vrednost signala C_0 , bitna samo u slučaju aritmetičkih mikrooperacija, dok ta vrednost nije bitna u slučaju logičkih mikrooperacija. U slučaju aritmetičkih mikrooperacija sabiranja i inkrementiranja na izlazu C_8 se dobija prenos, a u slučaju aritmetičkih mikrooperacija oduzimanja i dekrementiranja na izlazu C_8 se dobija pozajmica. U slučaju logičkih mikrooperacija signal na izlazu C_8 nema smisla.

Registar $AB_{7...0}$ je 8-mo razredni akumulator koji se koristi kao implicitno izvorište i odredište u aritmetičkim, logičkim i pomeračkim instrukcijama. Sadržaj sa izlaza multipleksera MX1 se vodi na ulaze registra $AB_{7...0}$ i u njega upisuje generisanjem aktivne vrednosti signala **ldAB**. Sadržaj registra $AB_{7...0}$ se pomera udesno za jedno mesto generisanjem aktivne vrednosti signala **shR**. Tada se u najstariji razred registra AB_7 upisuje signal **IR** koji dolazi sa izlaza multipleksera MX3. Sadržaj registra $AB_{7...0}$ se pomera ulevo za jedno mesto generisanjem aktivne vrednosti signala **shL**. Tada se u najmlađi razred registra AB_0 upisuje signal **IL** koji dolazi sa izlaza multipleksera MX4. Sadržaj registra $AB_{7...0}$ se vodi na ulaze ALU gde se koristi kao prvi izvorišni operand u slučaju aritmetičkih i logičkih operacija.

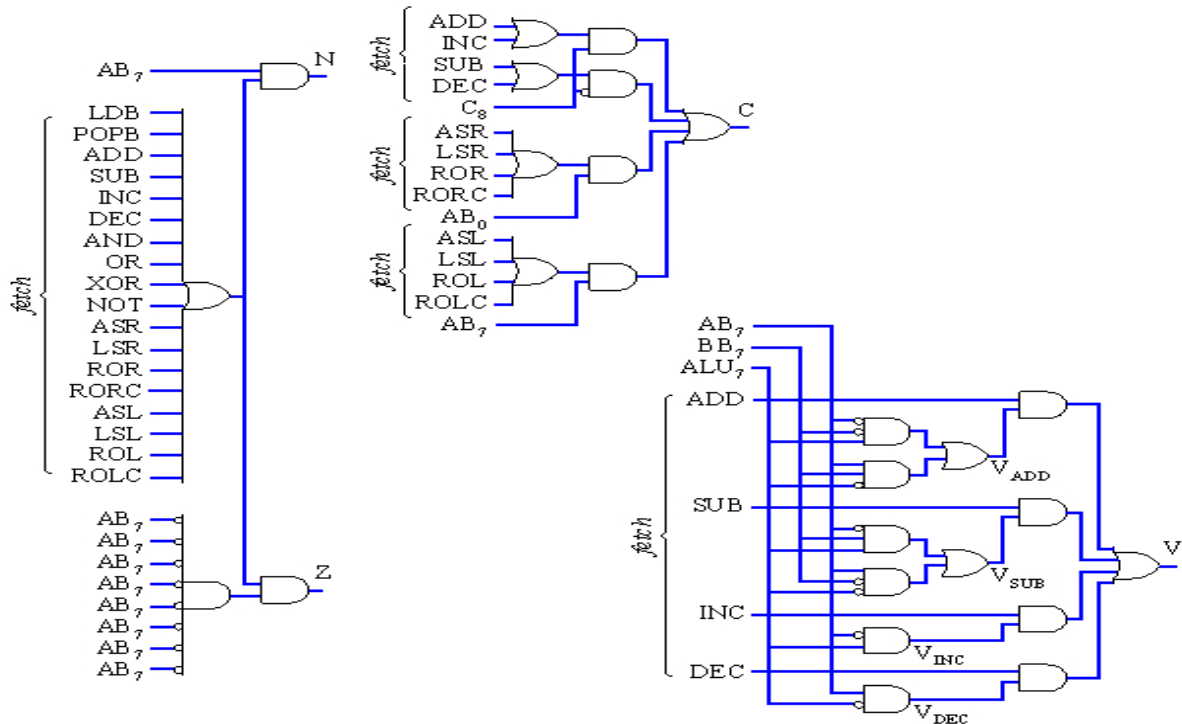
Registar $PSW_{15...0}$ je 16-to razredni registar koji sadrži indikatore programske statusne reči procesora (slika 9). Registar $PSW_{15...0}$ se sastoji od flip-flopora PSWI, PSWT, $PSWL_2$, $PSWL_1$, $PSWL_0$, PSWV, PSWC, PSWZ i PSWN, koji predstavljaju razrede $PSW_{15.14}$ i $PSW_{6...0}$, respektivno, dok razredi $PSW_{13...7}$ ne postoje. Flip-flop START služi da zadrži signal startovanja procesora koji se zadaje preko tastera.

Slika 8. Blok *exec* (prvi deo)

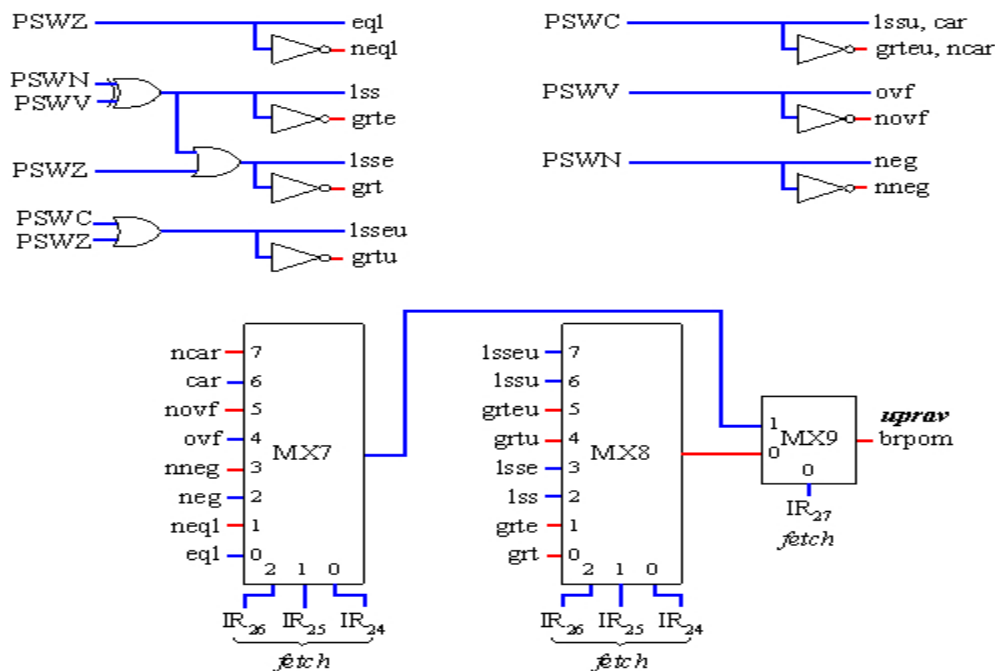


Slika 9. Blok exec (drugi deo)

Kombinacione mreže signala postavljanja indikatora N, Z, C i V se sastoje od logičkih elemenata (slika 10). Na izlazima kombinacionih mreža se formiraju signali koji se upisuju u flip-flobove PSWN, PSWZ, PSWC i PSWV programske statusne reči u okviru izvršavanja određenih instrukcija.

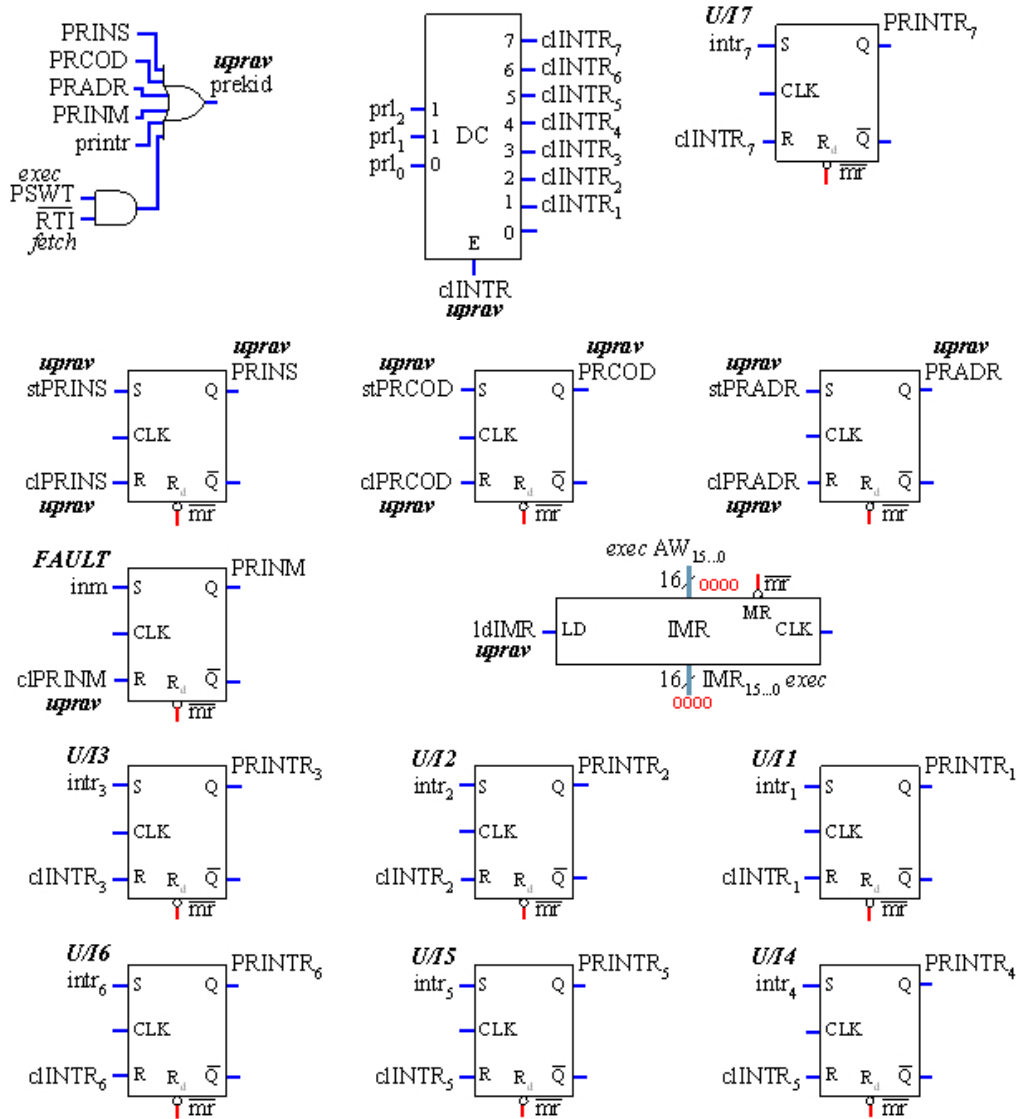
Slika 10. Blok *exec* (treći deo)

Kombinacione mreže za formiranje signala rezultata operacija **eql**, ..., **nneg** i **brpom** se sastoje od logičkih elemenata i multipleksera MX7, MX8 i MX9 (slika 11). Logičkim elementima se na osnovu sadržaja flip-floпова PSWN, PSWZ, PSWC i PSWV programske statusne reči formiraju signali **eql**, ..., **nneg**. Multiplekserima MX7, MX8 i MX9 se na osnovu razreda $IR_{27...24}$ prihvatnog registra instrukcije kojima se specificira kod operacije instrukcije uslovnog skoka BEQL, ..., BNNEQ selektuje jedan od signala **eql**, ..., **nneg** i pojavljuje kao signal **brpom** koji se koristi u upravljačkoj jedinici *uprav* da bi se prilikom izvršavanja instrukcija uslovnog skoka utvrdilo da li je uslov za skok ispunjen ili nije.

Slika 11. Blok *exec* (četvrti deo)

2.2.1.5 Blok intr

Blok *intr* sadrži logički ILI element za formiranje signala prekida **prekid**, kombinacione i sekvencijalne mreže za prihvatanje unutrašnjih prekida i spoljašnjih nemaskirajućih i maskirajućih prekida, registar $IMR_{15...0}$ (slika 12), kombinacione mreže za formiranje signala spoljašnjih maskirajućih prekida **printr** (slika 13), kombinacione i sekvencijalne mreže za formiranje pomeraja za tabelu sa adresama prekidnih rutina $IVTDSP_{15...0}$ i registar $IVTP_{15...0}$ (slika 14).

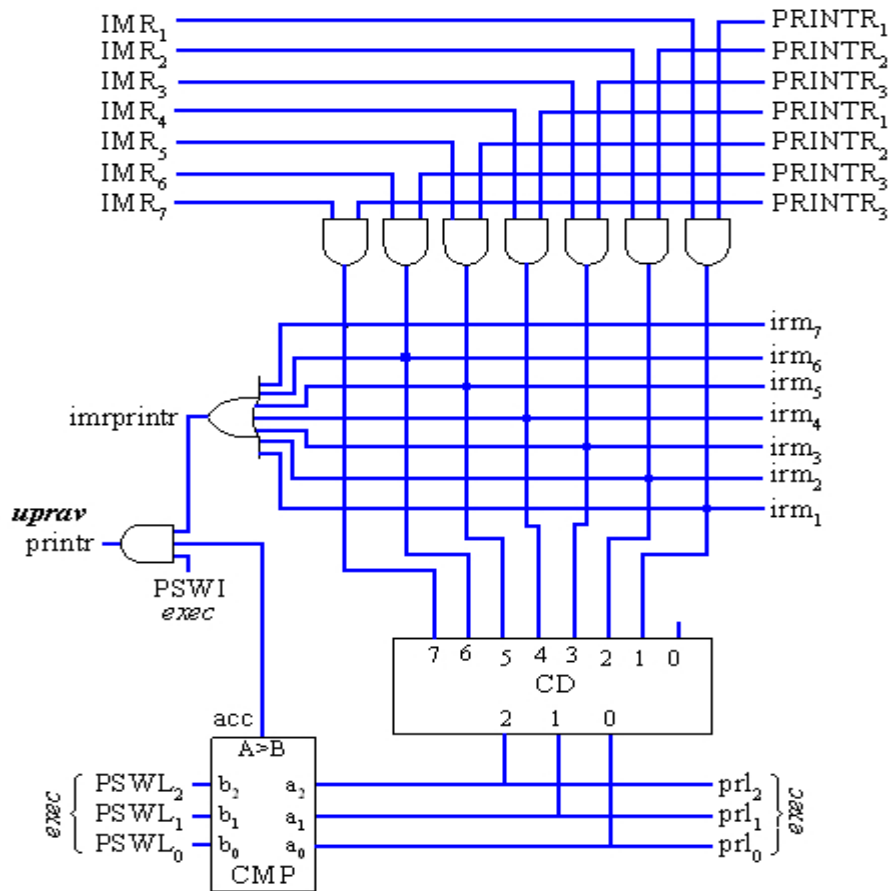


Slika 12. Blok *intr* (prvi deo)

Kombinacione prekidačke mreže za formiranje signala spoljašnjih maskirajućih prekida **printr** se sastoje od logičkih ILI i I elemenata, koda CD i komparatora CMP (slika 13).

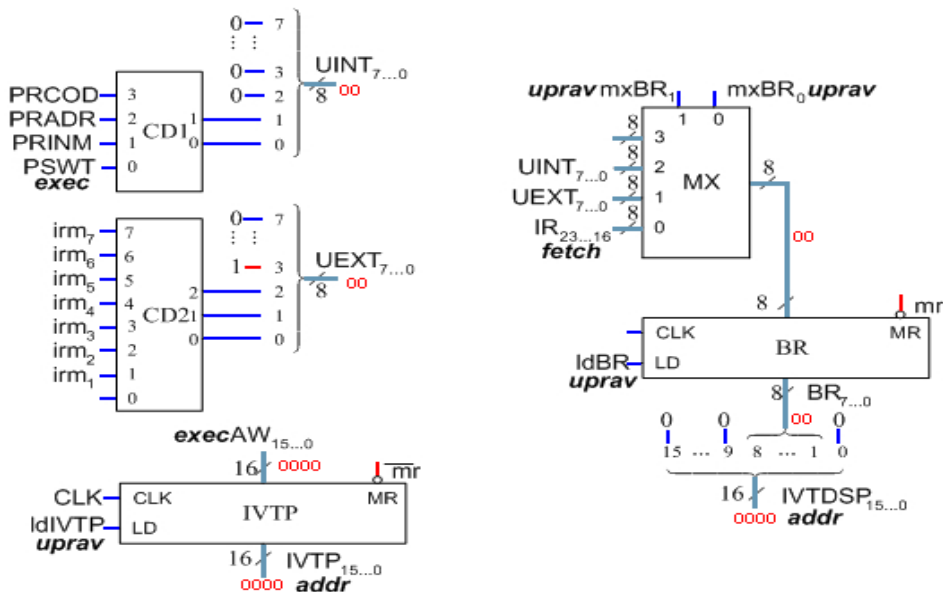
Signal maskirajućeg prekida **printr** ima aktivnu vrednost ukoliko signali **imrprintr**, **acc** i **PSWI** imaju aktivne vrednosti. Signal **imrprintr** ima aktivnu vrednost ukoliko je aktivna vrednost barem jednog od signala **irm₁** do **irm₇**. Ovo će se desiti ukoliko se barem u jednom od flip-flova $PRINTR_1$ do $PRINTR_7$ (slika 12) nalazi aktivna vrednost i ukoliko je u odgovarajućem razredu IMR_1 do IMR_7 registra maske $IMR_{15...0}$ takođe aktivna vrednost. Signal **acc** na izlazu komparatora CMP ima aktivnu vrednost ukoliko je prioritet pristiglog

spoljašnjeg maskirajućeg zahteva za prekid najvišeg prioriteta koji nije maskiran viši od prioriteta tekućeg programa.



Slika 13. Blok *intr* (drugi deo)

Kombinacione i sekvencijalne mreže za formiranje pomeraja IVTDSP_{15...0} za tabelu sa adresama prekidnih rutina se sastoje od kodera CD1 i CD2, i registra BR_{7...0} sa multiplexerom MX (slika 14).

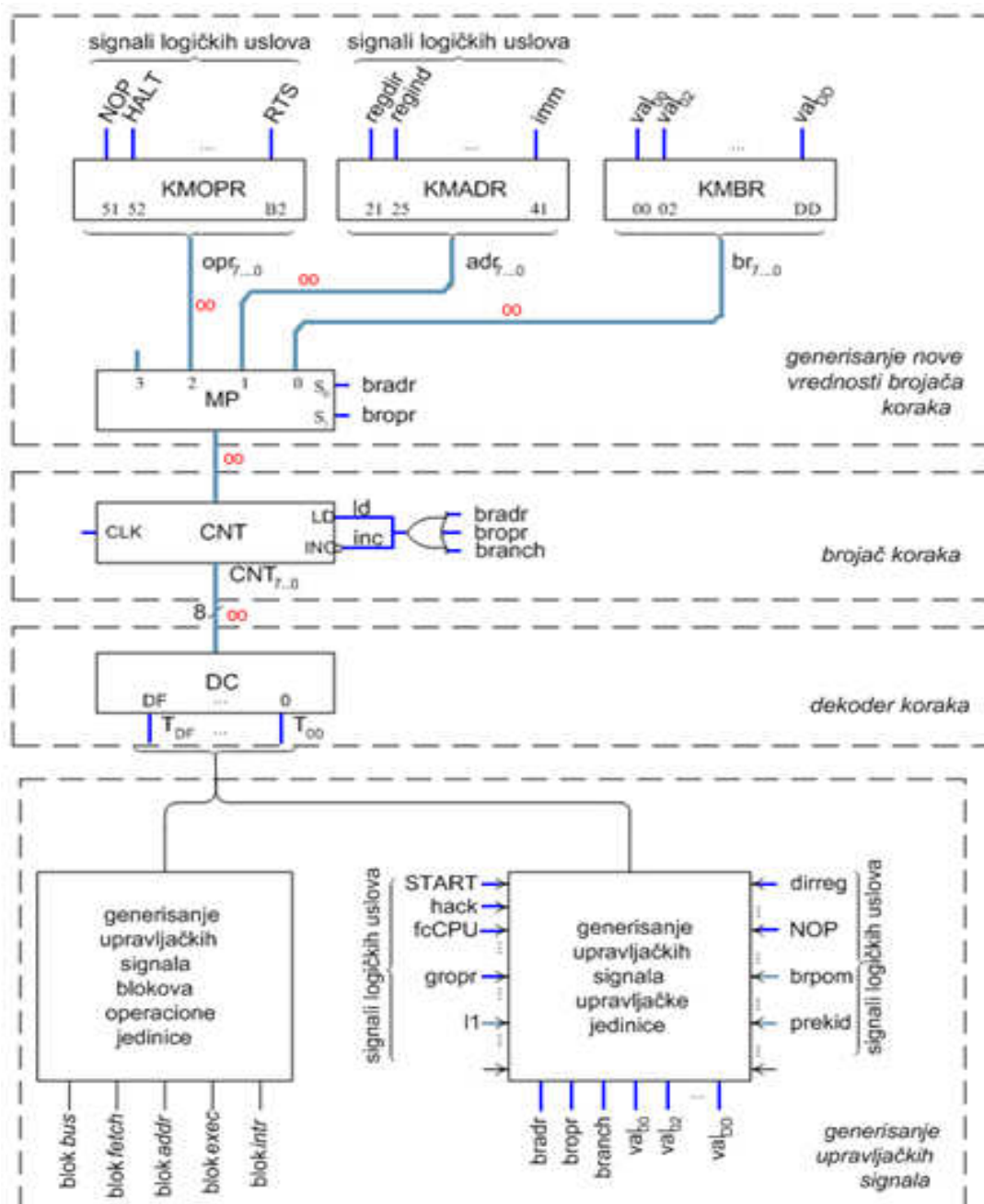


Slika 14. Blok *intr* (drugi deo)

2.2.2 Upravljačka jedinica

Upravljačka jedinica **uprav** je kompozicija kombinacionih i sekvencijalnih prekidačkih mreža koje služe za generisanje upravljačkih signala operacione jedinice **oper** na osnovu algoritama operacija i signala logičkih uslova.

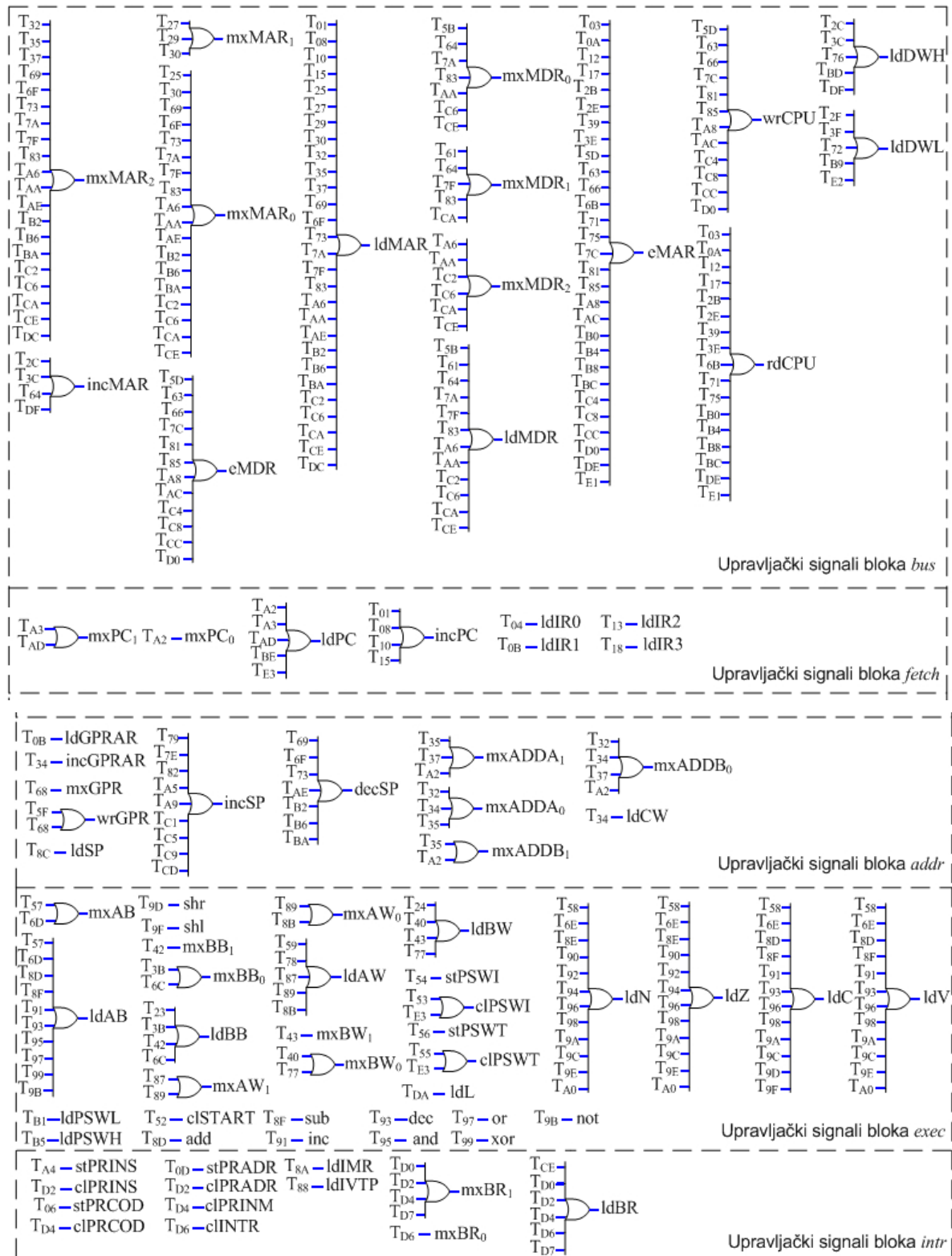
Dijagram toka i mikro kod upravljačke jedinici je dat detaljno u literaturi [1]. Ovde će biti predstavljene samo slike strukture upravljačke jedinice (slika 15), načini generisanja upravljačkih signala operacione jedinice (slika 16) i upravljačkih signala upravljačke jedinice (slika 17).



Slika 15. Struktura upravljačke jedinice

Kola KMOPR, KMADR i KMBR služe za generisanje vrednosti u slučaju višestrukog uslovnog skoka pri određivanju operacije koja se izvršava, načina adresiranja i u slučaju uslovnih i безусловnih skokova, respektivno.

Slede slike koje koristeći kombinaciona prekidačka kola generišu upravljačke signale operacione i upravljačke signale upravljačke jedinice.



Slika 16. Generisanje upravljačkih signala operacione jedinice

Na slici 17 su dekoderi uvedeni da bi slika izgledala manja i optimalnija jer su prethodne pretpostavke bile da neće postojati slika sa više od 100 logičkih kola. Ukoliko bi

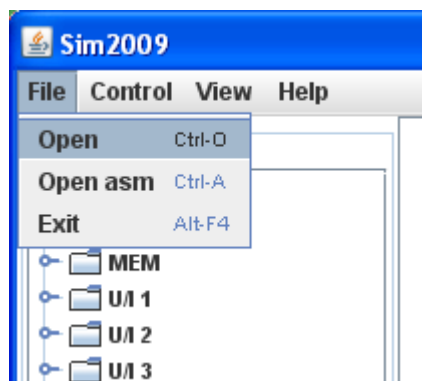
3 KORISNIČKI INTERFEJS

U ovoj glavi je predstavljen korisnički interfejs. Objašnjen je postupak pokretanja simulacije, kontrola rada simulatora sa navigacijom i pregledanje stanja registara.

3.1 POKRETANJE SIMULATORA

Simulator je predviđen za brže učenje i bolje razumevanje rada procesora jednog računarskog sistema, koji u sebi sadrži pored procesora i kontroler bez i sa direktnim pristupom memoriji kao i generatore takta. Takođe se očekuje da korisnik bude upoznat sa arhitekturom i organizacijom sistema pre nego što počne da koristi simulator.

Za pokretanje simulatora je potrebno da korisnik ima instalirano *Java Runtime* okruženje u kojem se pokreće simulator. Pokretanje simulatora se vrši dvostrukim klikom na ikonicu *sim2009.jar*. Kada se sistem pokrene, iz menija, tj. iz stavke *File* bira se ili opcija *Open* (slika 18) ili *Open asm* (slika 19).



Slika 18. Biranje stavke *Open*



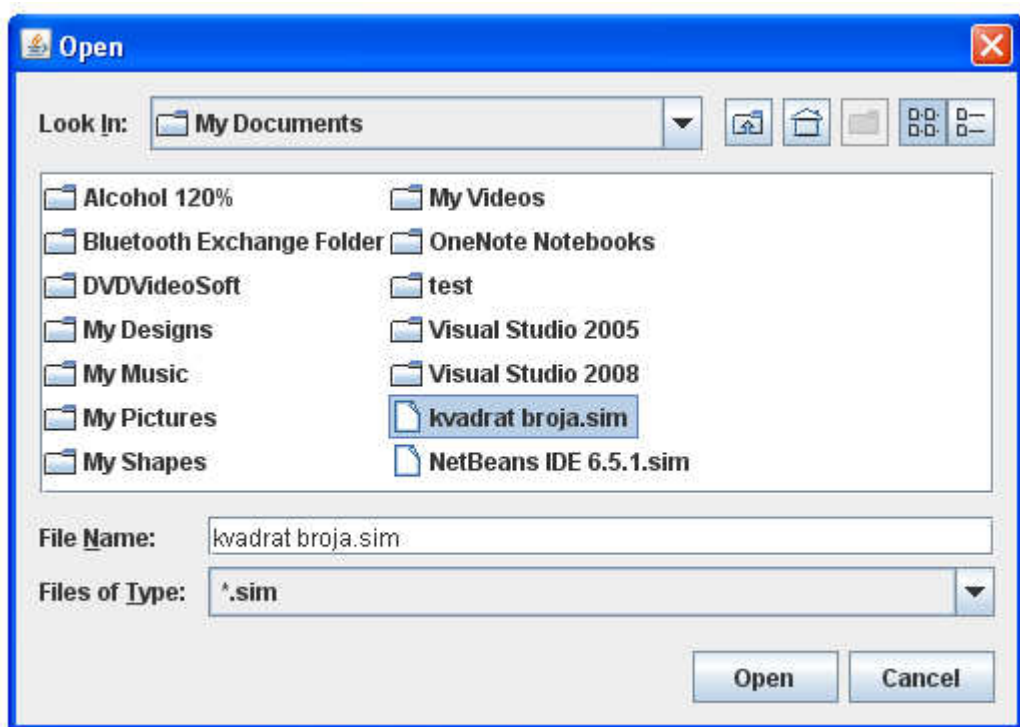
Slika 19. Biranje stavke *Open asm*

Ako korisnik ima asemblirani binarni fajl *.sim koristiće opciju *Open* iz menija. Pri odabiru opcije *Open* korisniku se prikazuje prozor za izbor fajla sa željenom simulacijom (slika 20). Na prikazanoj slici je selektovana simulacija *kvadriranja broja*.

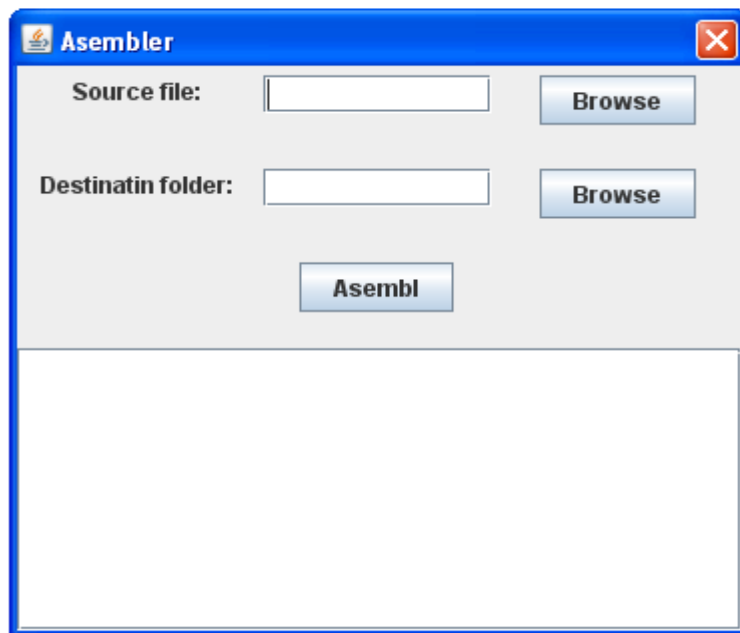
Ukoliko je korisnik napisao asemblerski kod koji želi da testira i posmatra kako se taj program izvršava prvo mora da izvrši asembliranje datog fajla. Izborom opcije *Open asm* iz menija se otvara prozor za asembliranje fajlova (slika 21). Dugmadima *Browse* se bira fajl koji se asemblira i destinacioni folder u koji će biti smešten fajl *.sim koji je rezultat rada asemblera. Kada se izvrši izbor asm fajla i destinacija, pritiskom na taster *Asembl* se vrši

asembliranje. Po završetku asembliranja se dobija poruka o tome kako je proteklo asembliranje. Ukoliko je došlo do greške saopštava se korisniku u kom je redu asemblerskog fajla greška (slika 22). Po završenom asembliranju korisnik može da otvori *.sim fajl kao što je predhodno već opisano.

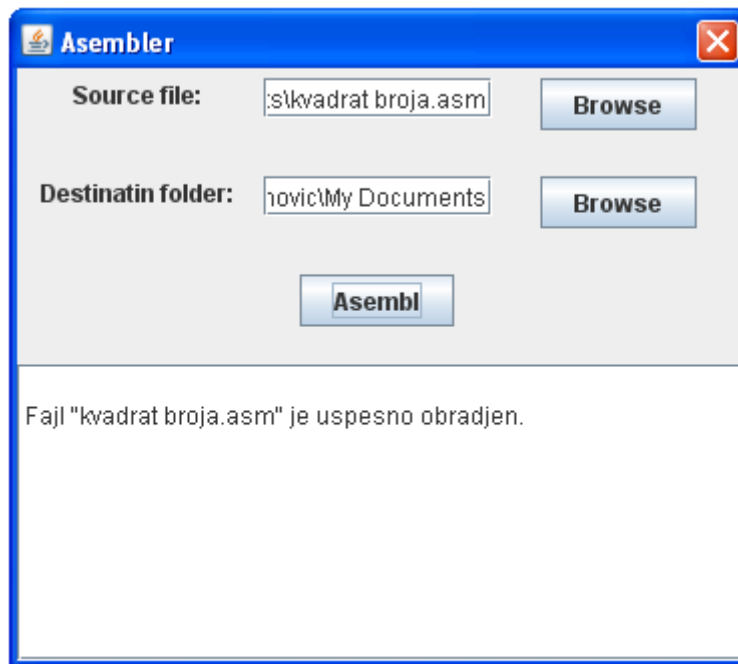
Po otvaranju *.sim fajla u memoriju sistema se upisuje mašinski kod i sistem se priprema za izvršavanje. Signal stSTART se postavlja na 1 i traje tokom jednog perioda signala takta, a potom se vraća na 0. Posle je aktivan signal START i procesor kreće da radi.



Slika 20. Prozor za izbor simulacije



Slika 21. Prozor za asembliranje programa koji se želi pokrenuti na simulatoru



Slika 22. Prozor za asembliranje programa koji se želi pokrenuti na simulatoru – rezultat asembliranja

3.2 KONTROLA RADA SIMULATORA I NAVIGACIJA

U ovom delu su objašnjeni načini navigacije kroz sistem i način kontrole rada sistema.

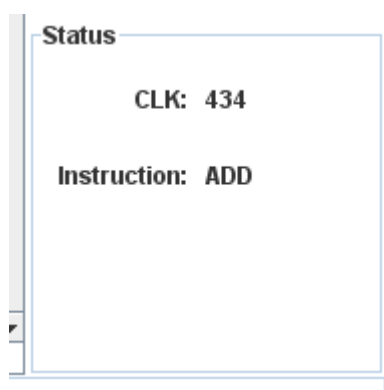
Za kontrolu rada sistema se koriste dugmad iz panela *Kontrola* (slika 23). Dugmad koja se nalaze sa leve i desne strane oznake *CLK* služe da se sistem pomera za jedan takt

unapred (desno) ili unazad (levo). Dugmad koja se nalaze sa leve i desne strane oznake *INS* služe da se sistem pomera za jednu instrukciju unapred (desno) ili unazad (levo). Dugme *PROGRAM* služi za izvršavanje celog programa odjednom. Dugme *GOTO* služi za izvršavanje programa do zadatog takta koji može biti manji ili veći od trenutnog takta. U polje pored dugmeta *GOTO* se unosi željeni takt.



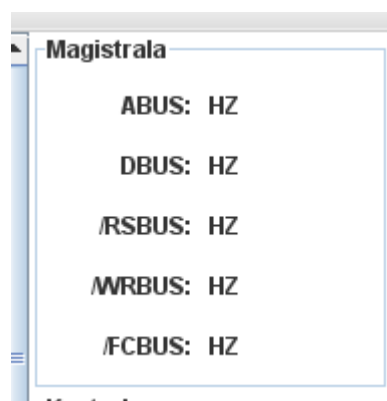
Slika 23. Kontrola rada sistema

Informacija o tome koja se instrukcija izvršava i koji je trenutni takt je data u panelu *Status* (slika 24).



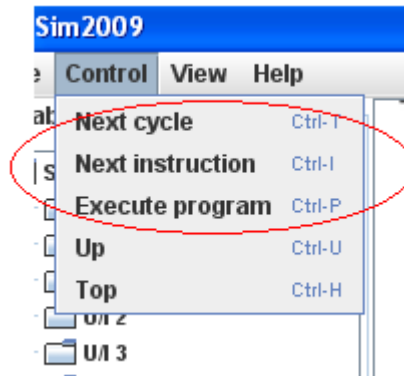
Slika 24. Izgled panela Status

U svakom trenutku se može videti stanje na magistrali sistema koje je prikazano u panelu *Magistrala* (slika 25).



Slika 25. Izgled panela Magistrala

Kontrola se može vršiti i kroz izbor odgovarajuće opcije stavke menija *Control* (slika 26).



Slika 26. Izbor kontrole u meniju

Ukoliko bismo želeli da simuliramo nekakav kvar u sistemu to ćemo uraditi pritiskom na dugme *set fault* (slika 27). To nam omogućava da mi zadamo asihrono zahtev za nemaskirajući prekid. Taj zahtev se može poništiti sa dugmetom *reset fault*.



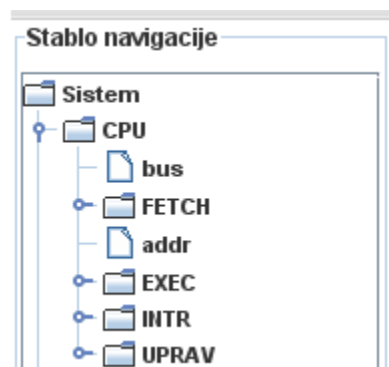
Slika 27. Deo za generisanje nemaskirajućeg prekida

Tokom rada simulatora prikazuje se koji je korak upravljačke jedinice procesora trenutno aktivan. To je predstavljeno u paleti *Mikro kod* (slika 28).



Slika 28. Mikro kod upravljačke jedinice

Kroz sistem se može vršiti navigacija na više načina. Prvi način je koristeći stablo hijerarhije koje se nalazi u paleti sa leve strane (slika 29). Klikom na odgovarajuću stavku iz stabla u glavnom prozoru se prikazuje izabrani deo sistema.

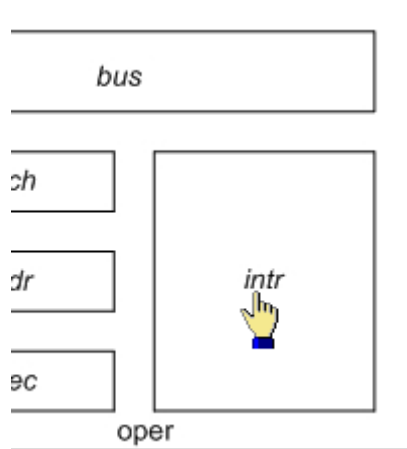


Slika 29. Stablo navigacije

Drugi način za prolazak kroz sistem jeste klikom na željeni blok sistema unutar kojeg se želi pregledati stanje, ili ukoliko odgovarajući blok ima više delova klikom na željeni broj dela u koji želi da se ide. Primeri su dati na sledecim slikama.

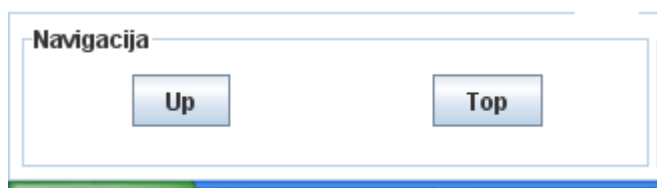


Slika 30. Odlazak na prvi deo nekog bloka a trenutno se nalazi na drugom delu

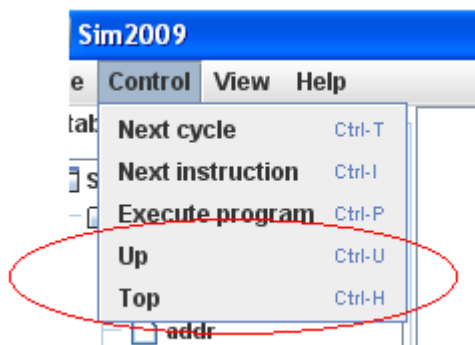


Slika 31. Ulazak u intr blok operacione jedinice procesora

Takođe, ako se želi kretati naviše u hijerarhiji pritiskom na dugme *Up* iz palete *Navigacija* (slika 32). Ukoliko se želi izaći na vrh hijerarhije pritisne se dugme *Top*. Takođe se to može postići i iz menija izborom istoimenih opcija (slika 33).



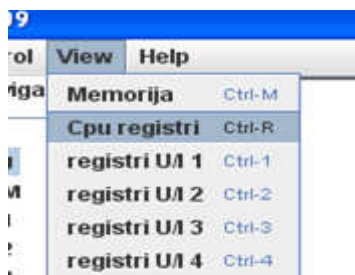
Slika 32. Paleta za navigaciju



Slika 33. Opcije u meniju za navigaciju

3.3 PREGLEDANJE STANJA REGISTARA

U ovom delu dato je objašnjenje kako se može pristupiti i kako se mogu menjati registri procesora u toku rada sistema. Pristupanje paleti sa registrima se vrši na sledeći način. Iz menija izborom stavke *View* i opcije *Cpu registry* (slika 34) otvara se paleta.




Slika 34. Opcije u meniju za pregled registara

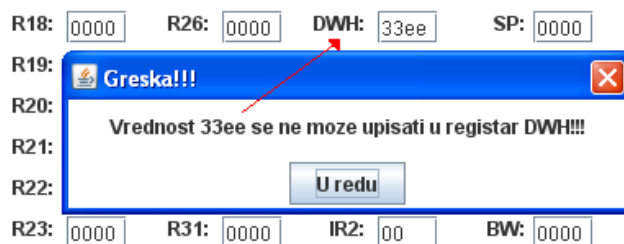
Kada se otvori paleta prikazuju se svi relevantni registri u sistemu i trenutni sadržaj svakog od njih (slika 35).



Slika 35. Pregled registara procesora

Najveći deo zauzimaju registri opšte namene, ovde su obeleženi sa R0 do R31. Kada se želi promeniti vrednost nekog registra potrebno je u polje pored imena registra upisati novu vrednost. Kada se u sve registre, čije vrednosti korisnik želi da izmeni, upiše željena vrednost, pritiskom na dugme *Izmeni* vrši se promena vrednosti u registrima. Ukoliko se pre pritiska dugmeta *Izmeni* pritisne dugme *Izadji* ili se pritisne standardno dugme za izlazak iz menija , promene koje su unete neće biti evidentirane i neće se ništa menjati u sistemu.

Ukoliko se unese vrednost koja nije heksadecimalan broj ili je uneti broj prevelik da se upiše u dati registar korisnik će biti opomenut za tu grešku (slika 36).



Slika 36. Ispis upozorenja o grešci

Kada se greška dogodi vrednost u svim registrima ostaje nepromenjena dok se data greška ne ispravi. Ostale opcije iz menija *View* su objašnjene u projektu u kome se obradljivao memorijski i ulazno izlazni sistem literatura [2].

4 SOFTVERSKA REALIZACIJA

U ovoj glavi se razmatra softverska realizacija simulatora. Predstavljani su pomoćni program koji su kreirani radi ubrzavanja izgradnje simulatora, kao i sam simulator. Dati su dijagrami klasa i sekvencijalni dijagram karakterističnih procesa u simulatoru.

4.1 POMOĆNI PROGRAMI

Projekat je realizovan u programskom jeziku Java. Razvojno okruženje je program *Eclipse SDK*. Sistemska biblioteka koja je korišćena je Java Runtime System Library, verzija 6. Za generisanje dijagrama korišćeni su primeri iz literature [4].

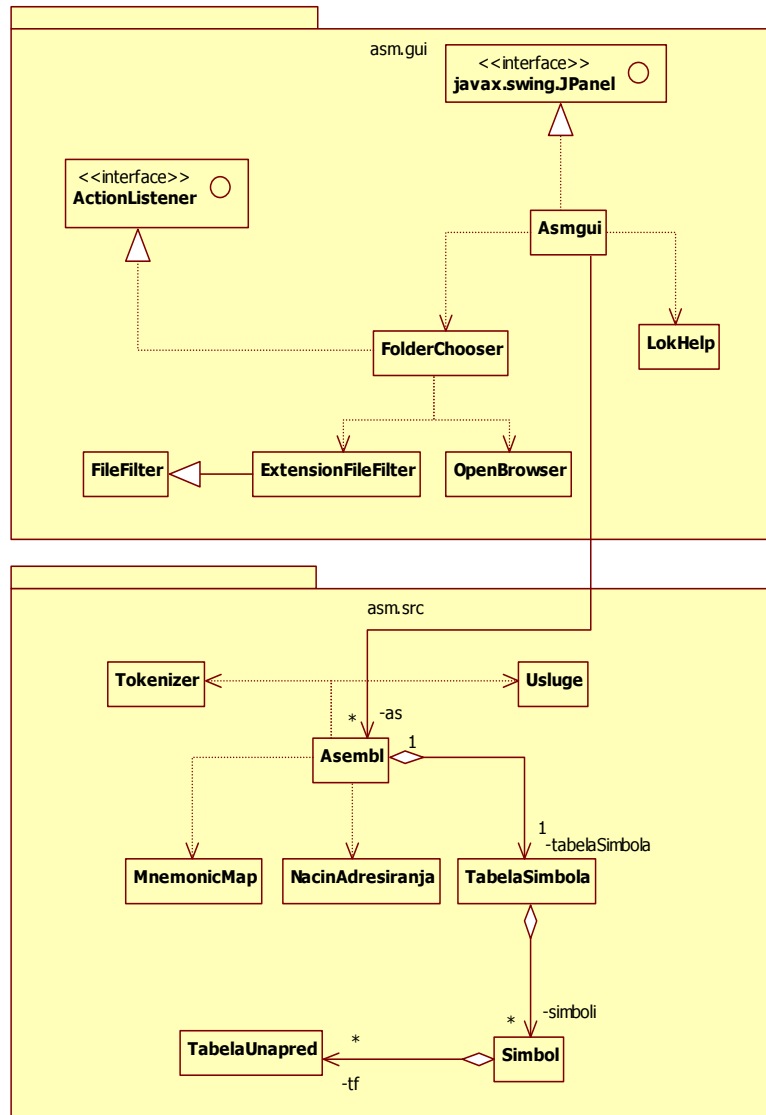
Da bi se mogao praviti sim fajl potrebno je bilo napraviti assembler koji će da prevodi pravilno napisan asemblerski kod. Projektovan je jednoprolazni assembler. Skup instrukcija koje assembler sadrži je ekvivalentan onom koji omogućava procesor. U sistem su dodate direktive za rezervisanje reči (2 bajta) DW i 1 bajta DB sa sledećom sintaksom:

```
labela: DW val;
```

```
labela: DB val;
```

Labela služi da se imenuje rezervisan prostor a val je vrednost koja će se upisati.

Dijagram klasa dat je na slici 37.



Slika 37. Dijagram klasa za assembler

Klasa `Asembl` je srce assemblera sa stanovišta logike, a klasa `Asmgui` sa stanovišta korisničkog interfejsa.

Klasa `MnemonicMap` ima u sebi heš tabelu mnemonika instrukcija i direktiva koje assembler datog procesora podržava. Posедуje metode koje za zadati `String` vraćaju mašinski kod date instrukcije, ili ako je u pitanju direktiva vraćaju kod direktive na osnovu koje se odlučuje dalja akcija.

Klasa `NacinAdresiranja` ima metodu za proveru načina adresiranja. Kao rezultat provere vraća se niz `String` vrednosti koji predstavljaju kod načina adresiranja, neposrednu vrednost ako je neposredno adresiranje u pitanju, ako je PC relativno adresiranje u pitanju vraća se pomeraj, ako je registarsko indirektno sa pomerajem ili bazno indeksno sa pomerajem ili registarsko direktno onda se vraća kod registra.

Klasa `Tokenizer` ima zadatak da zadati string izdeli na celine koje se očekuju. Jer se očekuje da je korisnik korektno uneo assemblerski kod. Celine na koje se deli linija su: labela, mnemonik i operand.

Klasa `Usluga` služi da olakša manipulaciju sa bitima prilikom formiranja mašinske instrukcije i za manipulacije sa stringovima i brojevima i za konverzije iz stringa u broj i obratno.

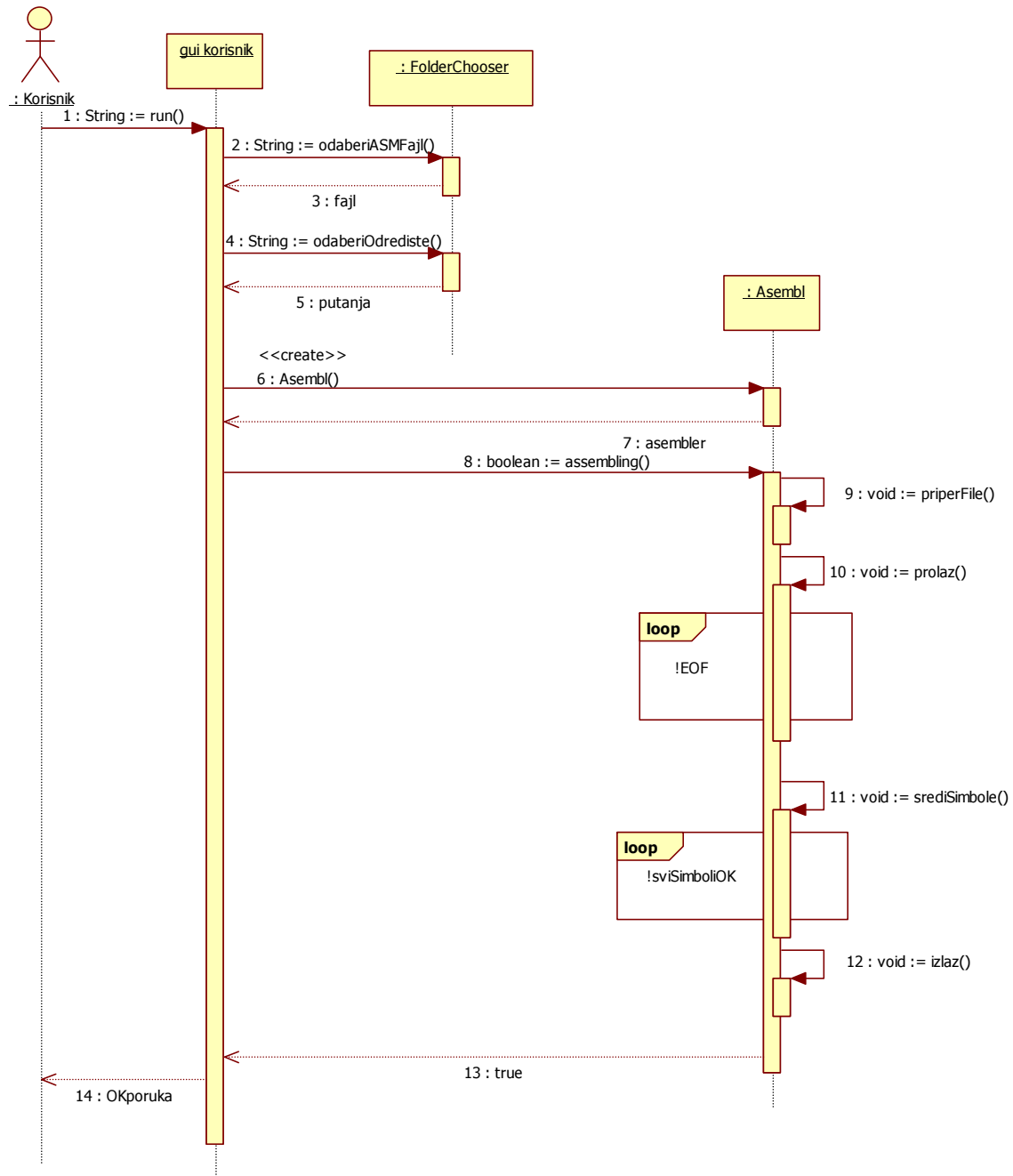
Klase `Simbol`, `TabelaSimbola` i `TabelaUnapred` služe da se vodi evidencija o pojavama simbola njihovim oznakama i vrednostima, kao i adresi memorijskog prostora koji se označavaju datim simbolom. `TabelaSimbola` poseduje sve simbole u sistemu i prilikom pronalaženja novog simbola u sistemu u ovu tabelu se dodaje simbol. Ukoliko je pronadjen kao definicija onda se proverava da li se pre definicije simbola taj simbol negde upotrebljavao i ako jeste onda mu se postave parametri koji se nisu popunili pri prvom pronalasku. `TabelaUnapred` pamti sve pojave simbola pre nego sto se naidje na definiciju.

Klasa `Asembl` u sebi realizuje algoritam za assembler u jednom prolazu. Algoritam se može podeliti na sledece celine:

1. Čitanje jedne linije asemblerskog koda,
2. Propustanje kroz tokenizer,
3. Određivanje tipa instrukcije,
4. Određivanje načina adresiranja,
 - 4.1. Provera ako je simbol, da li je već definisan ili da li je to definicija,
5. Pripremanje mašinskog koda i
6. Vraćanje na stavku 1. algoritma.

Po završetku asembliranja i kada se svi simboli ažuriraju formira se bajt kod. Tako formirani bajt kod se upisuje u fajl koji ima isto ime kao i izvorišni fajl samo sa ekstenzijom *sim*. Tako dobijeni fajl se koristi u simulatoru za popunjavanje radne memorije.

Što se tiče korisničkog interfejsa on je predhodno objašnjen i nece biti dužeg zadržavanja na opisu klasa koje su za to zaslužne jer to je skoro standardno odradjeno. Klase u paketu `asm.gui` samo su obuhvatile klasu `Asembl` i pružile korisniku lak način da se fajl prosledi, da se odredi određište *.sim fajla i da se izvrši asembliranje. Takođe kao pomoć korisniku data je i informacija u kojoj se liniji nalazi greška i opis verovatnog uzroka greške.

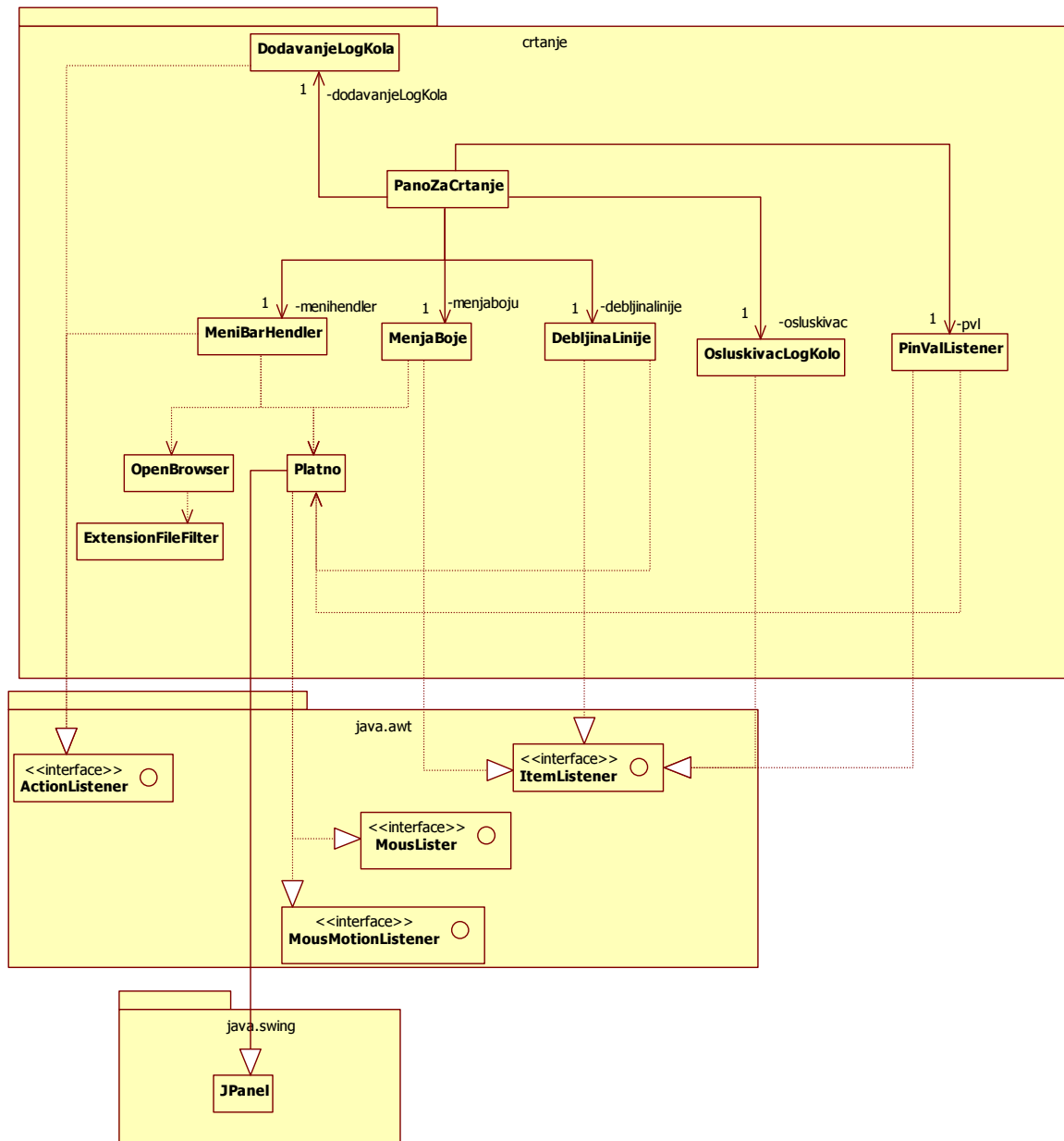


Slika 38. Dijagram sekvence assemblera

Na slici 38 dat je dijagram sekvence koji predstavlja jedan poziv assembleru za obradom. Ovaj se zahtev može ponoviti neograničen broj puta.

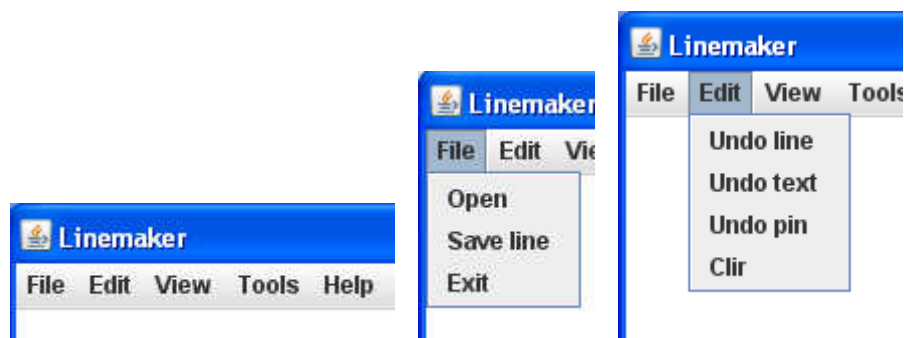
Drugi program koji je morao biti napravljen je nazvan *LineMaker*. Cilj pravljenja ovog programa je da olakša posao crtanja linija i generisanja logičkih komponenti koje te linije povezuju.

Dijagram klasa sa relacijama je dat na slici 39.



Slika 39. Dijagram klasa "LineMaker"

MeniBarHendler je glavni osluškivač menija(slika 40). Prihvatao je događaje za sledeće opcije iz menija: Open, Save line, Exit, Undo line, Undo text, Undo pin, Clir.



Slika 40. Meni Bar

MenjaBoje je osluškivač koji reaguje na zahtev za promenu boje kojom želi korisnik da iscrtava datu liniju. Na slici 41. je prikazan deo koji podešava promenu boje linije (crvenom elipsom okruženo).

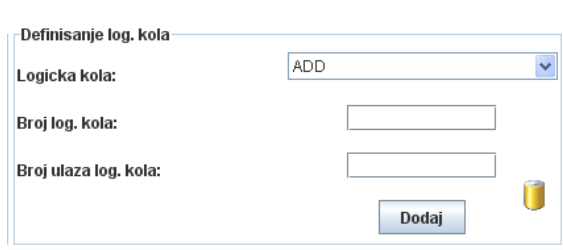


Slika 41. Panel za podešavanje linije

DebljinaLinije je osluškivač koji reaguje na promenu izborne padajuće liste za debljinu linije izraženo u tačkama (point). Na slici 41. je prikazan deo koji podešava debljine linije (plavom elipsom okruženo).

PinValListener je osluškivač koji reaguje na promenu izborne padajuće liste za izbor tipa pina. Tip može biti jednobitni, dvobitni,...,tridestdvobitni. Na slici 41. je prikazan deo koji podešava tip pina koji se iscrtava (zelenom elipsom okruženo).

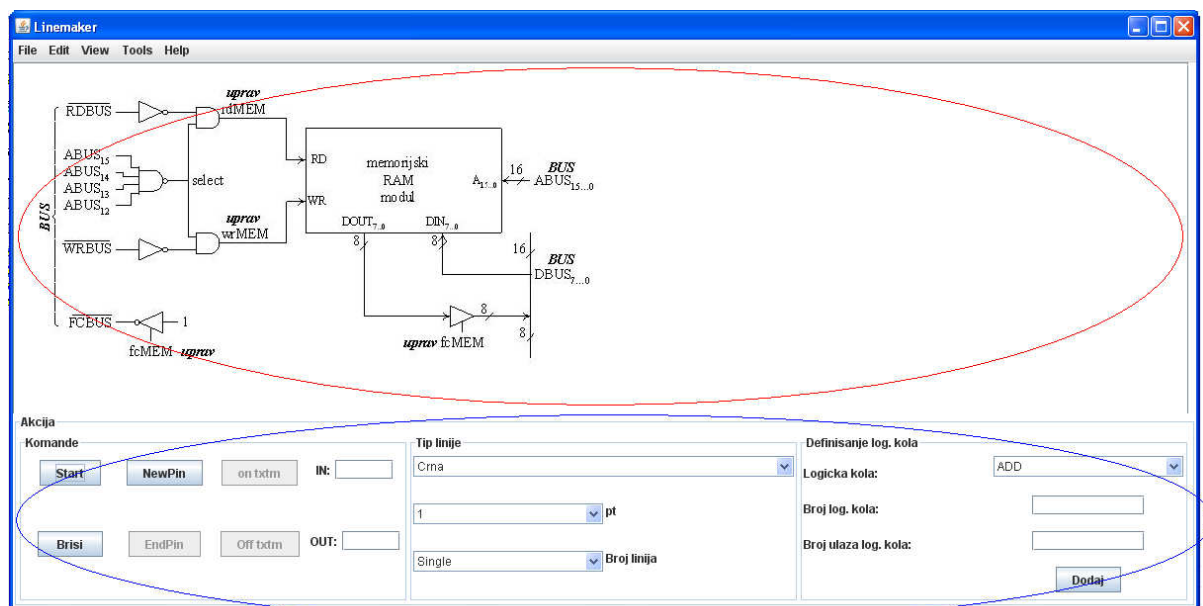
OsluskivacLogKola i DodavanjeLogKola su osluškivači koji reaguju kada se zadaju kola koja postoje u sistemu koji se iscrtava. Na slici 42 je dat pregled panoa koji služi da se generišu kola koja su u sistemu zastupljena. Za prve tri stavke je zadužen OsluskivacLogKola, a za dugme *Dodaj* je zadužen osluskivač DodavanjeLogKola.



Slika 42. Panel za definisanje logičkih kola

Klasa Platno je zadužena da predstavi sliku sa logičkim kolima i da omogući da se po njoj vrši crtanje linija i teksta na mestima gde je predviđeno da se linije i tekst nalaze. Ona vodi evidenciju o svemu šta se dešava na radnoj površini slike koja se iscrtava.

Klasa PanoZaCrtanje je omotač koji pruža lakši interfejs korisniku koji radi sa ovima programom. Izgled korisničkog interfejsa je dat na slici 43. Crvenom elipsom je naznačen radni prostor a plavom paleta sa alatima za crtanje.

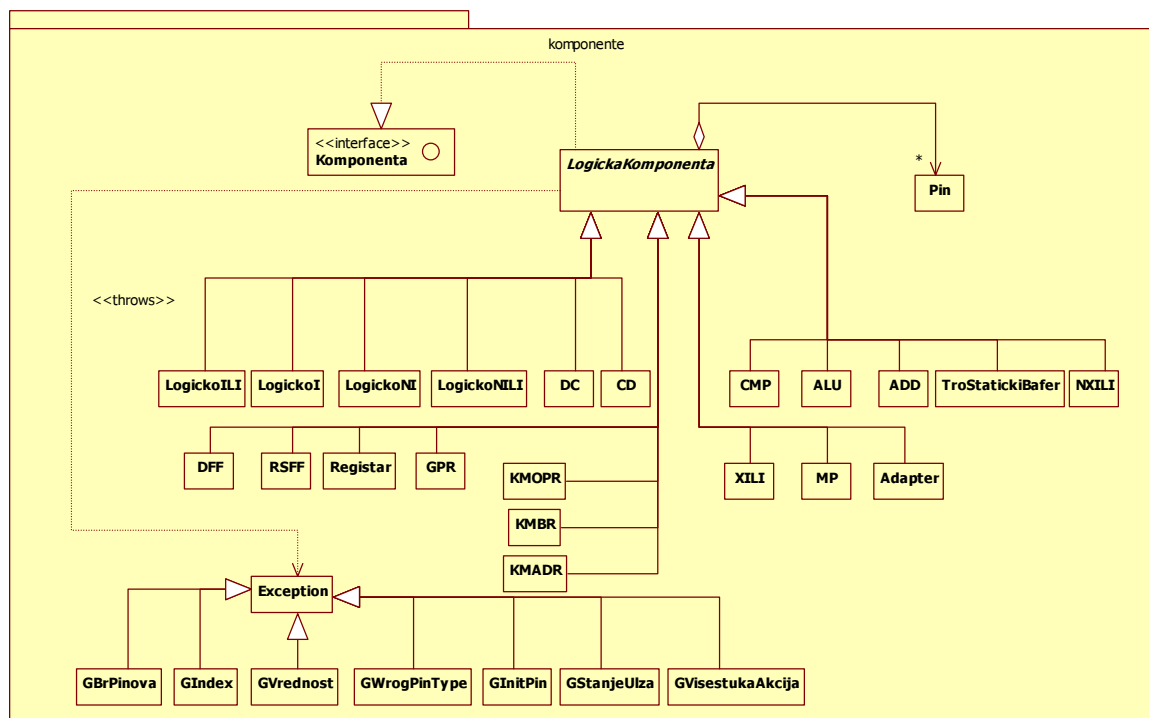


Slika 43. Korisnički interfejs Linemaker

4.2 SOFTVERSKA REALIZACIJA PROCESORA

Dizajn se može podeliti na dva nivoa a to su logički nivo i nivo prezentovanja. Na logičkom nivou se sve komponente koje postoje u procesoru preslikavaju u objekte odgovarajućih klasa koje simuliraju rad tih komponenti. Na nivou prestavljanja se grafički korisniku predstavi raspored signala i logičkih komponenti tako da bude intuitivno jasno i razumljivo kako je odgovarajuća vrednost dobijena.

Na slici 44 dat je dijagram klasa logičkih komponenti i veza izmedju njih.



Slika 44. Dijagram klasa logički komponenti

Sva logička kola se izvode iz jedne klase bilo da su sekvencijalna ili kombinaciona kola u pitanju. Svaka logička komponenta je u obavezi da zadovolji interfejs koji je propisan i predstavljen je na slici 45.

LogickaKomponenta
#in: Pin[*] {JavaDimensions = 1} #out: Pin[*] {JavaDimensions = 1}
<<create>>+LogickaKomponenta(in: int, out: int) +povezan(): boolean {JavaThrows = Exception} +fun(): int {JavaThrows = Exception} +setInputPins(index: int, pin: Pin) {JavaThrows = Exception} +setOutputPins(index: int, pin: Pin) {JavaThrows = Exception} +getMyID(): String +getOut(): Pin +getIn(): Pin

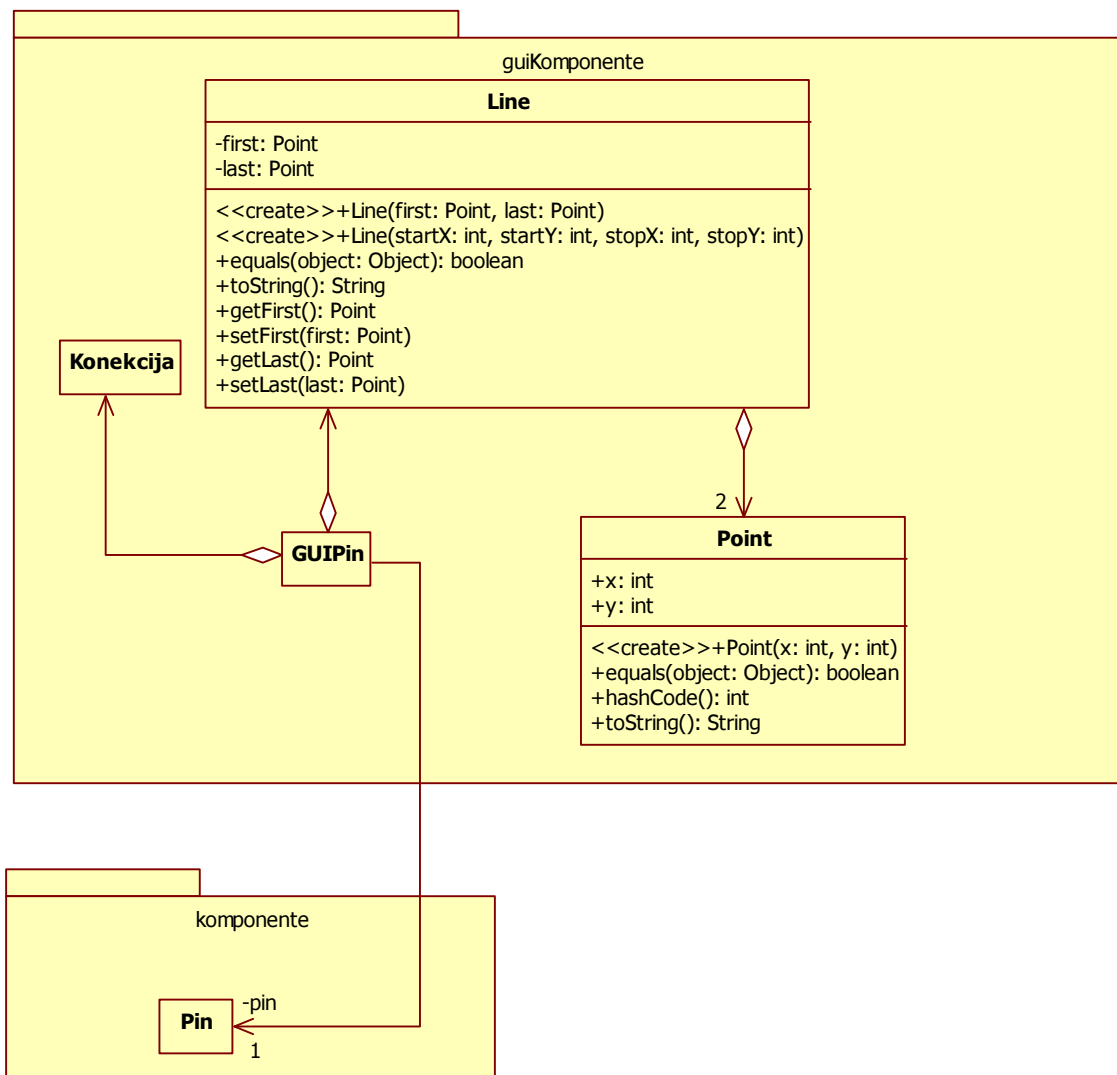
Slika 45. Interfejs logičke komponente

Metoda `povezan()` i metoda `fun()` nisu implementirane u ovoj klasi već svaka klasa koja proširuje ovu klasu ima zadatak da na njoj svojstven način implementira ove metode. Metoda `povezan()` se koristi samo u fazi testiranja i provere da li je komponenta povezana u sistem, dok je metoda `fun()` zadužena da simulira logiku rada konkretnog logičkog kola. Na primer, ukoliko se radi o dvoulaznom I logičkom kolu na izlaznom pinu će biti vrednost koja zadovoljava sledeću formulu $out \leq a \& b$.

Klase označene sa KMOPR, KMADR i KMBR su kombinacione mreže koje daju na izlazu vrednost koja je selektovana aktivnim ulaznim pinom. U jednom trenutku samo je jedan ulazni pin aktivan u jednom logičkom kolu. Oni su specifični i samo se koriste u upravljačkoj jedinici procesora.

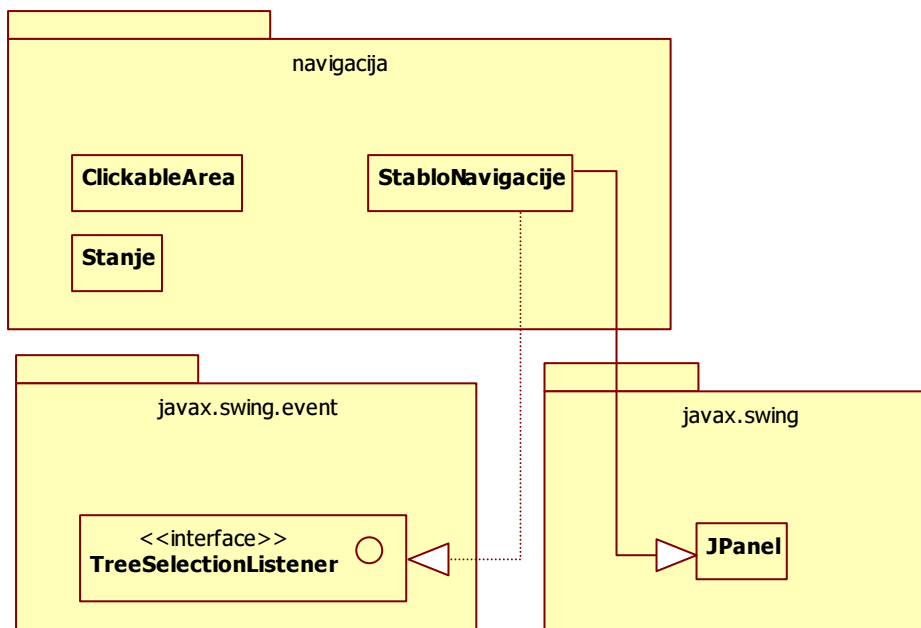
Pin predstavlja preslikan hardverski izvod iz kućišta komponente u logički svet. Svaka komponenta mora da za definisan ulaz i izlaz u sebi pridoda pin koji prenosi vrednost do te komponente i od te komponente do svake koja koristi njen rezultat. Postoje dve vrste pinova, to su pinovi koji podržavaju visoku impedansu i pinovi koji ne podržavaju visoku impedansu nego samo dva stanja. Oni koji podržavaju visoku impedansu su magistrale podataka, adresna magistrala i kontrolna magistrala.

Sledeći paket koji ćemo opisati je paket za grafičko predstavljanje pinova na slikama blokova procesora.



Slika 46. Grafičko prikaz pinova

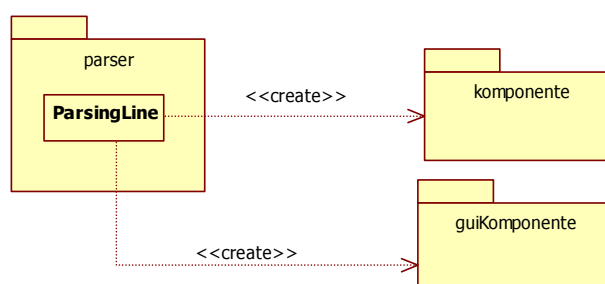
Klasa `GUIPin` čuva informaciju o tome gde koji pin treba da se ucrti i sa kojim logičkim kolima treba da se poveže. Klasa `Line` i `Point` služe da zapamte koordinate linije koja može biti i izlomljena. Klasa `Pin` služi za povezivanje logičkih komponenti kako je prethodno već objašnjeno. Ona se od strane `GUIPin`-a koristi da se odredi kojom će se bojom iscrtavati koja linija. Ukoliko se radi o pinu koji ima 1 bit i ukoliko je vrednost pina *true* linija će se bojiti u crveno, a ako je vrednost *false* bojiće se u plavo. Ukoliko se radi o pinu koji ima više od jednog bita, vrednost koju trenutno ima pin biće napisana na odgovarajućem mestu pored linije, koja je u ovom slučaju obojena svetlo plavom bojom i ima debljinu od 3pt.



Slika 47. Klase za navigaciju i pamćenje stanja

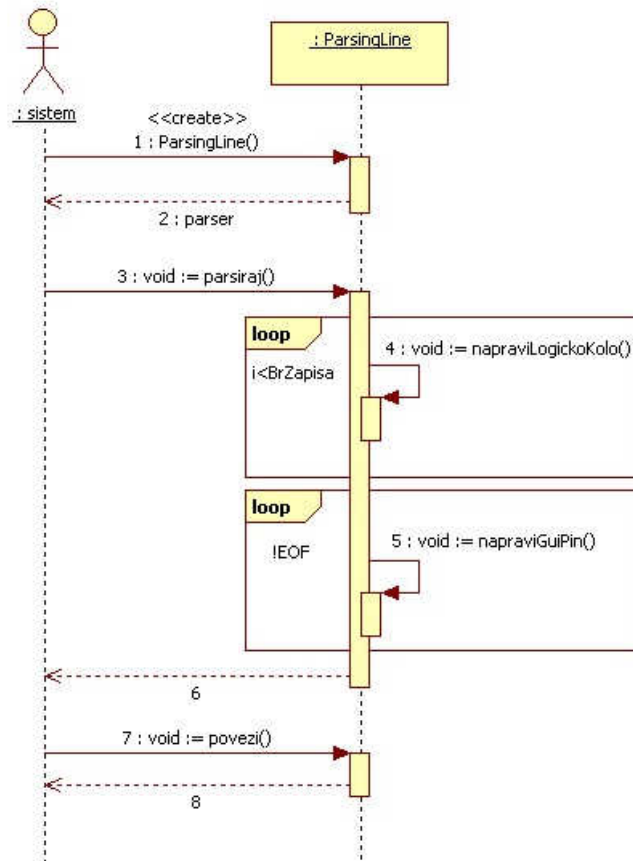
Klasa *StabloNavigacije* omogućava da se u obliku stabla predstavi hijerarhija procesora. Na slici 29. je prikazano stablo hijerarhije. Klasa *ClickabilArea* je zadužena da se prostim klikom (slika 31) na željeni blok prikaže prva strana tog bloka, ili da klikom na željeni deo tog bloka pređemo sa trenutnog na odabrani deo (slika 30).

Klasa *Stanje* se ne koristi za navigaciju kroz hijerarhiju sistema već za navigaciju kroz vreme. Kada se želi preći na sledeći takt, trenutno stanje se zapamti. Pamte se samo sekvencijalne komponente, a vrednosti kombinacionih komponenti se izračunavaju na osnovu njih. Ukoliko se želi ići unazad u vremenu, podaci koje pamti ova klasa se restauriraju za željeno stanje (koristi se princip keša za čuvanje stanja).



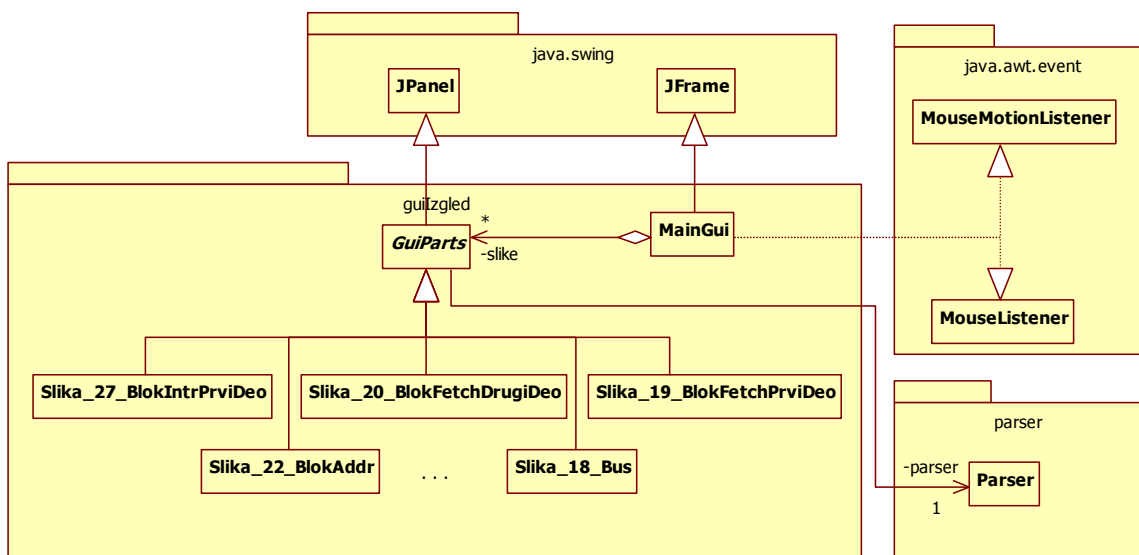
Slika 48. Paket parser

Kada se koristeći predhodno opisan *LineMaker* program iscrtaju slike koje vizuelno reprezentuju hardver sistema, po određenom formatu se sve informacije o linijama, logičkim kolima i vezama između logičkih kola upišu u fajlove. Kasnije interpretaciju sadržaja tih fajlova radi objekat klase *ParsingLine* i to za svaku sliku u sistemu postoji drugi objekat. Na slici 49 je dat dijagram sekvence za primer jednog poziva.



Slika 49. Sekvencijalni dijagram parsiranja jednog fajla

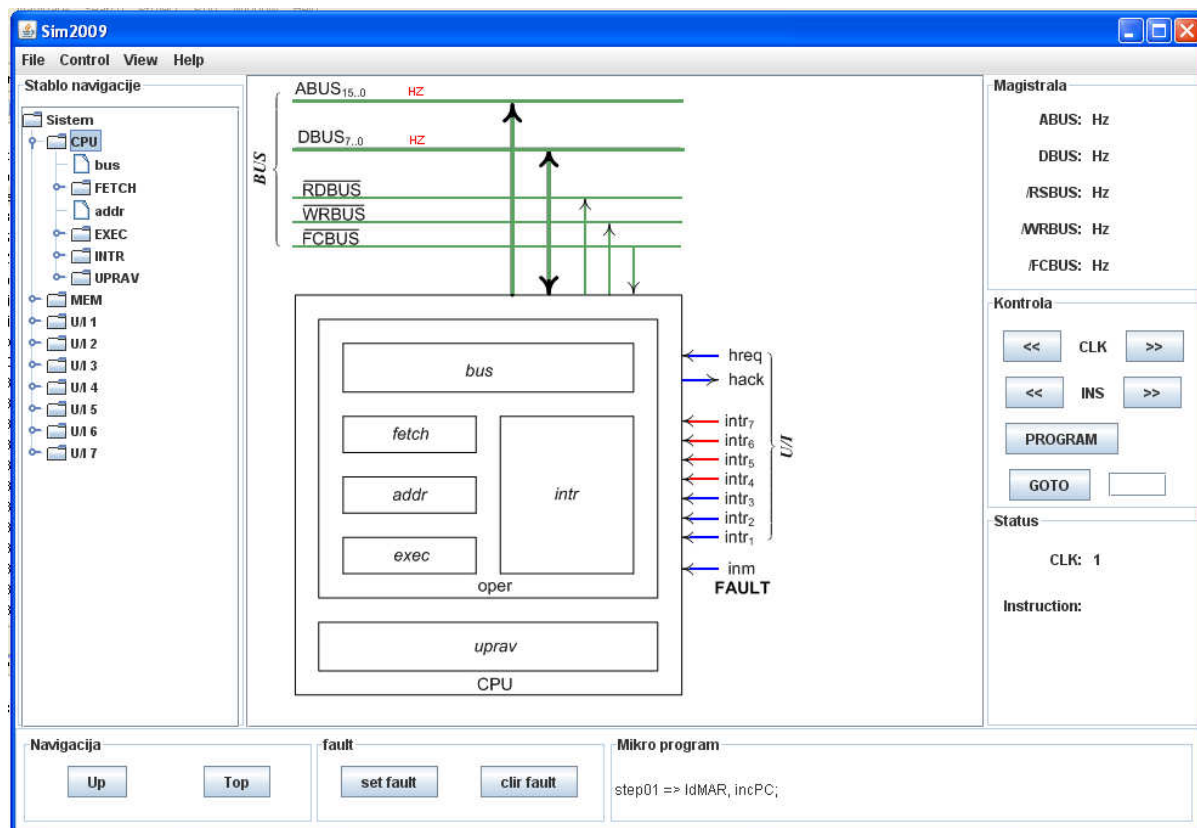
Glavni paket u sistemu je `guiIzgled`. On sadrži sve elemente koji se predstavljaju korisniku. Na Slici 50 je predstavljen paket sa klasama koje su relevantne.

Slika 50. Paket `guiIzgled`

Apstraktna klasa `guiParts` predstavlja osnovnu klasu za grafičko prikazivanje delova sistema. Ona prevazilazi metodu `paint(Graphics g)`, koja je standardno implementirana u `JPanel` klasi jer postoje elementi koji se posebno iscrtavaju kao što su

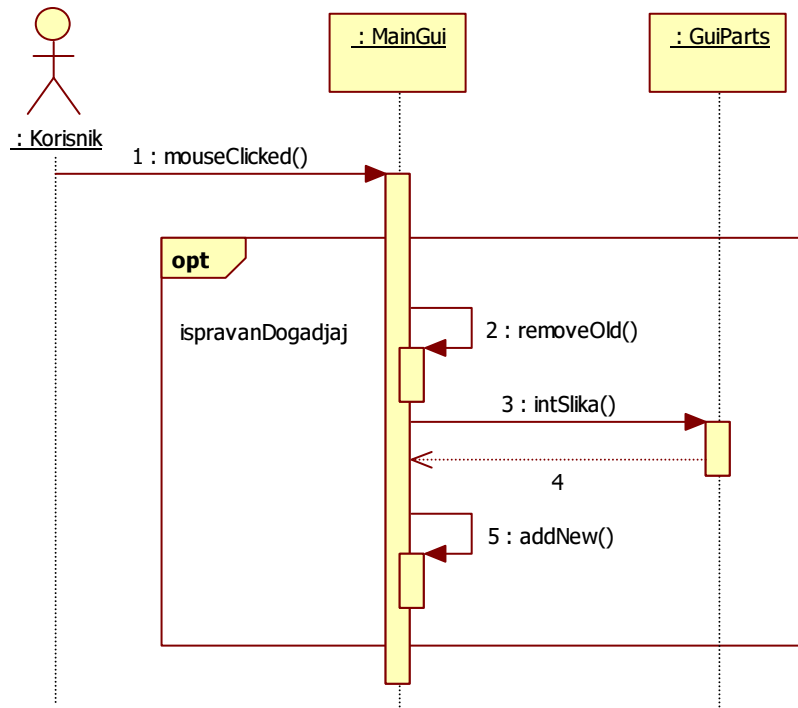
pozadinske slike, linije u odgovarajućoj boji i vrednosti pored odgovarajućih linija. Svaka klasa koja se izvede mora da implementira apstrakne metode koje je propisala ova klasa, a to su metode za inicijalizaciju i prilagođavanje veza u logičkom delu sistema koje nisu urađene automatski od strane programa LineMaker (ranije objašnjen), kao i metode za uvoz i izvoz pinova. Pored iscrtavanja slika, istovremeno se radi i uvoženje slika. Slike se uvoze u sistem tek u trenutku kada se poželi prikaz date slike i time se štedi memorija, jer se ne baferišu slike koje se ne koriste.

MainGui je omotač oko svih delova sistema i realizuje izgled i funkcionalnosti prema korisniku. Izgled koji on pruža korisniku je predstavljen je na slici 51.



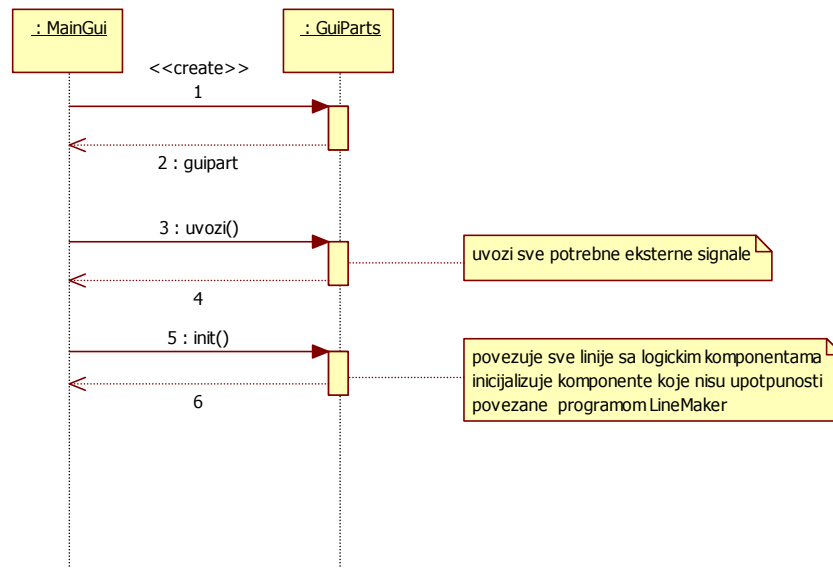
Slika 51. Izgled simulatora

Primer rada učitavanja jedne slike u sistem je dat slici 52. Dijagram sekvence uopšteno predstavlja učitavanje jedne selektovane slike u centralni, radni, deo simulatora. Ako korisnik strelicom misa pređe preko aktivnog regiona kursor menja svoj znak u šaku i time korisnik zna da je data oblast preko koje prelazi aktivna. U suprotnom ako selektuje neku drugu regiju neće doći ni do kakve akcije.



slika 52. Promena glavnog ekrana

Na slici 53 je prikazano kako se kreira objekat tipa koji je nastao iz tipa GuiParts. Ovde nije prikazano, ali se neposredno posle kreiranja objekta pokupe svi signali koje dati objekat izvozi, pa se potom kada se svi objekti kreiraju radi uvoz i inicijalizacija tih signala.



Slika 53. Primer kreiranja dela procesora

5 TESTOVI ZA LABORATORIJSKI RAD

U ovoj glavi dat je skup testova za rad u laboratoriji. Svaki test primer je propraćen objašnjenjem o karakteristikama koje se prikazuju datim testom. Na jednom, pogodno izabranom, test primeru prikazani su svi detalji rada sistema.

5.1 SKUP TESTOVA ZA SAMOSTALNI RAD

U ovoj glavi predstavljeni su primeri koje studenti mogu koristiti za učenje i samotestiranje. Dati su primeri za aritmetičke, logičke, pomeračke i instrukcije uslovnog skoka i skoka na potprogram, kao i maskirajući prekidi, maskiranje svih maskirajućih prekida, selektivno maskiranje svih maskirajućih prekida, prioriteti maskirajućih prekida, prekid posle svake instrukcija, instrukcija INT, gnzždenje maskirajućih prekida, prekid usled greške u načinu adresiranja

Prvi primer prikazuje aritmetičku instrukciju sabiranja i indeksno adresiranje sa pomerajem kao i registarsko i memorijsko direktno adresiranje. U programu se pojavljuju direktive *prg* i *db* koje označavaju početak programa i rezervaciju jednog bajta, respektivno. Instrukcije koje se koriste u programu su *ldb* za punjenje akumulatora, *stb* za smeštanje vrednosti akumulatora na zadatu lokaciju i aritmetička instrukcija sabiranja *add*. Prvo se koristeći registarsko indirektno adresiranje sa pomerajem dovuče u akumulator prvi operand, pa se potom on smesti u registar R1. potom se dovlači drugi operand i sabira sa prvim operandom koji se nalazi u registru R1. Rezultat se iz akumulatora potom prenosi na memorijsku adresu na kojoj se nalazio drugi operand.

```

;      aritmeticka instrukcija
;
; _____
;      GLAVNI_PROGRAM
;
prg      ; -KOMENTAR-----KOD-----ADRESE---
ldb [R4+R5]0x10 ; acc<=mem[r4+r5+5h] | 20 A4 00 10 | 100 |
stb R1      ; r1<=acc | 22 01 | 104 |
ldb podatak ; acc<<mem[podatak] | 20 40 01 11 | 106 |
add R1      ; acc<=acc add r1 | 30 01 | 10A |
stb podatak ; mem[podatak]<=acc | 22 40 01 11 | 10C |
halt       ; zaustavlja se CPU | 01 | 110 |
;
; _____
;      ALOKACIJA_PROSTORA
;
podatak: db 0x5 ; | 05 | 111 |
;
; _____

```

Sledeći kod služi za prikazivanje logičkih instrukcija. Program je identičan kao i prethodni samo što se umesto *add* koristi instrukcija *or*, logičko *ili*. . Prvo se koristeći registarsko indirektno adresiranje sa pomerajem dovuče u akumulator prvi operand, pa se potom on smesti u registar R1. potom se dovlači drugi operand i izvrši se logička operacija *ili* sa prvim operandom koji se nalazi u registru R1. Rezultat se iz akumulatora potom prenosi na memorijsku adresu na kojoj se nalazio drugi operand.

; logicka instrukcija			
;			
; _____ GLAVNI_PROGRAM _____			
prg	<i>; -KOMENTAR-----KOD-----ADRESE---</i>		
ldb [R4+R5]0x10	<i>; acc<=mem[r4+r5+5h]</i>	20 A4 00 10	100
stb R1	<i>; r1<=acc</i>	22 01	104
ldb podatak	<i>; acc<<mem[podatak]</i>	20 40 01 11	106
or R1	<i>; acc<=acc add r1</i>	35 01	10A
stb podatak	<i>; mem[podatak]<=acc</i>	22 40 01 11	10C
halt	<i>; zaustavlja se CPU</i>	01	110
;			
; _____ ALOKACIJA_PROSTORA _____			
podatak: db 0x7 ;		07	111
;			

U nastavku slede instrukcije rotiranja i pomeranja. Instrukcija *asr* radi aritmetičko pomeranje u desno, a instrukcija *rorc* vrši rotiranje u desno kroz C fleg PSW registra. Za očitavanje podatka korišćeno je memorijsko indirektno adresiranje, a za smeštanje rezultata memorijsko direktno. Koristeći memorijsko indirektno adresiranje se u akumulator smešta operand. Potom se izvrši aritmetičkim rotiranjem u desno pa se rotira u desno ali ovoga puta kroz C bit PSW registra. Dobijeni rezultat se smešta na adresu na kojoj se nalazio operand koristeći memorijsko direktno adresiranje.

; pomeracke instrukcije			
;			
; _____ GLAVNI_PROGRAM _____			
prg	<i>; -----KOMENTAR-----KOD-----ADRESE---</i>		
ldb [adresa]	<i>; acc<=mem[mem[adresa]]</i>	20 60 01 0C	100
asr	<i>; acc>>=acc</i>	38	104
rorc	<i>; acc>>=acc kroz C PSW-a</i>	3B	105
stb podatak	<i>; mem[podatak]<=acc</i>	22 40 01 0b	106
halt	<i>; zaustavlja se CPU</i>	01	10A
;			
; _____ ALOKACIJA_PROSTORA _____			
podatak: db 0x5 ;		05	10B
adresa: dw podatak;		01 0B	10C
;			

Slede instrukcije uslovnog skoka i skoka na potprogram. Instrukcija *jsr* je instrukcija skoka na potprogram, a ime potprograma je dato kao operand. Instrukcija *bneql* je instrukcija uslovnog skoka. Za učitavanje vrednosti u akumulator se koristi PC relativno adresiranje. Potom se skače u podprogram gde se smanjuje vrednost akumulatora za jedan. Sve dok je uslov za skok ispunjen program će pozivati prekidnu rutinu i smanjivati vrednost akumulatora. Kada uslov za skok nebude bio *true* završiće se program.

; instrukcije skoka			
;			
; _____ GLAVNI_PROGRAM _____			
prg	<i>; --KOMENTAR-----KOD-----ADRESE---</i>		
ldb &0x8	<i>; acc<=2h</i>	20 C0 00 08	100

loop:	jsr podprog	;skok na podprogram	0A 01 0A	104
bneql	loop	;skoks ako je acc<>0	11 FB	107
halt		;zaustavlja se procesor	01	109
; _____				
; _____PODPROGRAM_____				
podprog:				
dec		;acc--	33	10A
rts		;povratak iz podprgrama	0B	10B
; _____				
; _____ALOKACIJA_PROSTORA_____				
podatak:	db 0x2		02	10C
adresa:	dw podatak;		01 0C	10D
; _____				

Sledeći primer obuhvata sve prethodne primere i načine adresiranja. Takođe su pridodati adresiranja registarsko indirektno sa pomerajem i neposredno koji se nisu javljali u predhodnim testovima.

;instrukcije: uslovnog skoka, skoka u podprogram, aritmetičke,
;logičke i pomeracke

; _____				
; _____GLAVNI_PROGRAM_____				
prg		; -----KOMENTAR-----	-----KOD-----	---ADRESE---
ldw [R4]0x150		;acc<=mem[R4+150]	21 84 01 50	100
stw R1		;R1<=acc	23 01	104
ldw #0x2		;acc<=2	21 E0 00 02	106
stw R2		;R2<=acc	23 02	10A
or R1		;acc<= acc or R1	35	10C
asr		;acc>>	38	10D
rolc		;acc<<PSWC	3F	10E
stw R3		;R3<=acc	23 03	10F
jsr loop2		;podprogram loop2	0A 01 23	111
loop: ldw R1		;acc<=R1	21 01	114
add R2		;acc<=acc+R2	30 02	116
stw R1		;R1<=acc	23 01	118
blss loop		;if r1<0 goto loop	1A F8	11A
ldw R1		;acc<= R1	21 01	11C
stw 0x152		;mem[152]=acc	23 40 01 52	11E
halt			01	122
; _____				
; _____PODPROGRAM_____				
loop2: rts		;povratak iz podprog	0B	123
; _____				
; _____ALOKACIJA_PROSTORA_____				
pod: dw 0xffffd		; mem[150]<= -3	FE FD	124
; _____				

U sledećim primerima su predstavljene sve situacije sa maskirajućim prekidima, greškama u načinu adresiranja, prekidom posle svake instrukcije i instrukcije *INT*. Za generisanje prekida se koriste periferije tajmeri koji daju prekide po liijama 4 do 7.

Sledeći primer obuhvata opsluživanje zahteva za prekid i maskiranje svih maskirajućih prekida. Maskiranje svih maskirajućih prekida se radi naredbom *INTE*. Prvo se inicijalizuje registar *IVTP* i *IV* tabela. Potom se u tajmer upiše broj koliko taktova od trenutaka startovanja periferije treba da protekne do trenutka kada se zada prekid. Startuje se periferija i setuje se bit maskiranja svih maskirajućih prekida *PSWE*. U prvoj fazi izvršavanja instrukcije sabiranja dolazi do prekida koji se evidentira a po izvršavanju instrukcije se i prihvata prekid. Odlazi u prekidnu rutinu izvršava se rutina i vraća se u glavni program gde u nekoliko iteracija kroz petlju i smeštanjem rezultata završava program. Tokom rada sa ovim program treba posebnu pažnju usmeriti na smeštanja i skidanje konteksta glavnog programa sa steka.

;opsluzivanje prekida i povratak iz prekidne rutine i maskiranje			
;svih maskirajucih prekida			
;			
; _____ GLAVNI_PROGRAM _____			
prg	;-----KOMENTAR-----KOD-----ADRESE-		
xor R1	<i>;acc<=0;</i>	36 01	100
stivtp	<i>;ivtp<=0;</i>	29	102
ldw adresaPRR	<i>;podesavanje IV tabeli</i>	21 40 01 28	103
stw 0x0d	<i>;mem[0]<=[adresaPRR]</i>	23 40 00 00	107
ldb #0x1	<i>;acc<=1h</i>	20 E0 00 01	10B
stb 0xF143	<i>;WCRL(5)<=acc</i>	22 40 F1 43	10F
stb 0xF140	<i>;startuje tajmer u/i 5</i>	22 40 F1 40	113
INTE	<i>;omogucen prekid</i>	05	117
;			
loop: add R2	<i>;acc<=acc+R2</i>	30 02	118
blss loop	<i>;skok ako je manje</i>	1A FC	11A
stw 0x150	<i>;mem[150]<=acc</i>	23 40 01 50	11C
halt	<i>;zaustavlja procesor</i>	01	110
;			
; _____ PREKIDNA_RUTINA _____			
prekid:	;		
ldw #0x20	<i>;acc<=20</i>	21 E0 00 20	121
stw R2	<i>;R2<=acc</i>	23 02	125
rti	<i>;povracaj iz prekidne rutine</i>	0D	127
;			
; _____ ALOKACIJA_PROSTORA _____			
;alokacija prostora			
adresaPRR: dw prekid	<i>;</i>	01 21	128
;			

Sledeći primer prikazuje selektivno maskiranje maskirajućih prekida manipulacijom vrednostima registra **IMR**. Procesoru dolazi zahtev za prekid od više periferija ali su neki od njih maskirani. Procesor prihvata prekid najvišeg prioriteta od zahteva za prekid koji nisu maskirani. Prvo se inicijalizuje ulaz u tabeli prekidnih rutina. Potom se setuju određeni biti **IMR** na nulu tako da se prođe prekid samo od jedne a da se prekid od druge periferije blokira. Setuje se bit maskiranja svih maskirajućih prekida *PSWE*. I startuju periferije koje broje vreme do prekida i generišu isti. Tokom izvršavanja instrukcije *add* dolaze oba zahteva za prekid jedan je maskiran i ne prihvata se a drugi je odmaskiran i prihvata se. Opslužuje se zahtev za prekid. Vraća se iz prekidne rutine i završava program. Više se neće desiti prekid jer iako je aktivan i postoji prekid neće biti prihvaćen jer je maskiran.

;selektivno maskiranje		
;		
; _____ GLAVNI_PROGRAM _____		
prg	<i>-----KOMENTAR-----</i>	<i>---KOD--- ---ADRESE---</i>
xor R1	<i>; acc<=0;</i>	36 01 100
stivtp	<i>; ivtp<=0;</i>	29 102
ldw adresaPRR	<i>; ulaza u IV tabeli</i>	21 40 01 3D 103
stw 0x0D	<i>; mem[0d]<=[adresaPRR]</i>	23 40 00 0D 107
stw 0x0E	<i>; mem[0e]<=[adresaPRR]</i>	23 40 00 0E 10B
ldb #0x9	<i>; acc<=9h</i>	20 E0 00 09 10F
stb 0xF143	<i>; WCRL(5)<=acc</i>	22 40 F1 43 113
ldb #0x5	<i>; acc<=5h</i>	20 E0 00 05 117
stb 0xF183	<i>; WCRL(6)<=acc</i>	22 40 F1 83 11B
ldw #0x0020	<i>; acc<=0020h</i>	21 E0 00 20 11F
stIMR	<i>; IMR<=acc</i>	2B 123
INTE	<i>; omogucen prekid</i>	05 124
stb 0xF140	<i>; startuje tajmer u/i 5</i>	22 40 F1 40 125
stb 0xF148	<i>; startuje tajmer u/i 6</i>	22 40 F1 48 129
;		
loop: add R2	<i>; acc<=acc+R2</i>	30 02 12D
blss loop	<i>; skok ako je manje</i>	1A FC 12F
stw 0x150	<i>; mem[150]<=acc</i>	23 40 01 50 131
halt	<i>; zaustavlja procesor</i>	01 135
;		
; _____ PREKIDNA_RUTINA _____		
prekid:	<i>;</i>	
ldw #0x20	<i>; acc<=20</i>	21 E0 00 20 136
stw R2	<i>; R2<=acc</i>	23 02 13A
rti	<i>; povracaj iz prekidne rutine</i>	0D 13C
;		
; _____ ALOKACIJA_PROSTORA _____		
; alokacija prostora		
adresaPRR: dw prekid	<i>;</i>	01 36 13D
;		

U sledećem primeru će biti prikazani prioriteti maskirajućih prekida. Od dve periferije dolazi zahtev za prekid ali će biti prihvaćen onaj koji ima veći prioritet. Tajmeri periferija se postavljaju tako da se zahtevi za prekid generišu istovremeno od obe periferije. Kada se bude odlučivalo koji će se prekid od ova dva prihvatiti pošto su obadva masikrajuća prihvata se onaj koji ima viši prioriteti. Po završetku obrade tog prekida prihvata se zahtev i od drugog tajmera i kada se i on obradi kontekst se vraća glavnom programu.

;prioritiranje maskirajucih prekida		
;		
; _____ GLAVNI_PROGRAM _____		
prg	<i>-----KOMENTAR-----</i>	<i>---KOD--- ---ADRESE---</i>
xor R1	<i>; acc<=0;</i>	36 01 100
stivtp	<i>; ivtp<=0;</i>	29 102
ldw adresaPRR	<i>; ulaza u IV tabeli</i>	21 40 01 3A 103
stw 0x0D	<i>; mem[0d]<=[adresaPRR]</i>	23 40 00 0D 107
stw 0x0F	<i>; mem[0e]<=[adresaPRR]</i>	23 40 00 0F 10B

ldb #0x9 ; <i>acc<=9h</i>	20 E0 00 09	10F
stb 0xF143 ; <i>WCRL(5)<=acc</i>	22 40 F1 43	113
ldb #0x5 ; <i>acc<=5h</i>	20 E0 00 05	117
stb 0xF183 ; <i>WCRL(6)<=acc</i>	22 40 F1 83	11B
INTE ; <i>omogucen prekid</i>	05	11F
stb 0xF140 ; <i>startuje tajmer u/i 5</i>	22 40 F1 40	120
stb 0xF180 ; <i>startuje tajmer u/i 6</i>	22 40 F1 80	124
;		
loop: add R2 ; <i>acc<=acc+R2</i>	30 02	128
blss loop ; <i>skok ako je manje</i>	1A FC	12A
stw 0x150 ; <i>mem[150]<=acc</i>	23 40 01 50	12C
halt ; <i>zaustavlja procesor</i>	01	130
;		
; <u>PREKIDNA_RUTINA</u>		
prekid: ;		
ldw #0x20 ; <i>acc<=20</i>	21 E0 00 20	131
stw R2 ; <i>R2<=acc</i>	23 02	135
stw R3 ; <i>R3<=acc</i>	23 03	137
rti ;	0D	139
;		
; <u>ALOKACIJA_PROSTORA</u>		
; <i>alokacija prostora</i>		
adresaPRR: dw prekid ;	01 31	13A
;		

Prekid posle svake instrukcije je prikazan u sledećem primeru. Ovaj prekid se može koristiti za brojanje instrukcija, tipova instrukcija ili neku validaciju. U ovom primeru se inicijalizuje IV tabela i postavi se PSWT bit na *true*. Potom se posle svake instrukcije odlazi u prekidnu rutinu i inkrementira se vrednost registra R1. Na kraju se u tom registru nalazi vrednost koja označava koliko je puta pozvana ta prekidna rutina, odnosno koliko je instrukcija izvršeno dok PSWT nije postavljen na *false*.

;prekid posle svake instrukcije		
;		
;		
; <u>GLAVNI_PROGRAM</u>		
prg ; -----KOMENTAR-----	KOD -----	ADRESE -----
ldw adresaPRR ; <i>ulaza u IV tabeli</i>	21 40 01 1D	100
stw 0x00 ; <i>mem[0d]<=[adresaPRR]</i>	23 40 00 00	104
trpe ; <i>prekid posle svake instrukcije</i>	07	108
ldb #0x45 ; <i>acc<=45h</i>	20 E0 00 45	109
stb 0x150 ; <i>mem[150]<=acc</i>	22 40 01 50	10D
trpd ; <i>zabranjen prekid T</i>	06	111
xor #0x45 ; <i>acc<=acc xor 45h</i>	36 E0 00 45	112
halt ;	01	116
;		
; <u>PREKIDNA_RUTINA</u>		
prekid: ;		
ldb R0 ; <i>acc<=R0</i>	20 00	117
inc ; <i>acc<=acc+1</i>	32	119
stw R0 ; <i>R0<=acc</i>	23 00	11A
rti ; <i>povratak u glavni program</i>	0D	11C

i _____ GLAVNI_PROGRAM			
prg	i -----KOMENTAR-----	KOD	ADRESE
ldw	adresaPRR ; ulaza u IV tabeli	21 40 01 1F	100
stw	0x04 ; mem[0d]<=[adresaPRR]	23 40 00 04	104
ldb	#0x45 ; acc<=45h	20 E0 00 45	108
stb	0x150 ; mem[150]<=acc	22 40 01 50	10C
INT	#0x2 ; br. ulaza 2	0C 02	10E
xor	#0x45 ; acc<=acc xor 45h	36 E0 00 45	112
halt	;	01	116
i _____			
i _____ PREKIDNA_RUTINA			
prekid: ;			
ldb	R0 ; acc<=R0	20 00	117
inc	; acc<=acc+1	32	119
stw	[R0]0x130 ; mem[R0+130h]<=acc	23 80 01 30	11A
rti	; povratak u glavni program	0D	11E
i _____			
i _____ ALOKACIJA_PROSTORA			
;alokacija prostora			
adresaPRR:	dw prekid ;	01 17	11F
i _____			

```
;gnezdenje maskirajucih prekida
;
```

- 47 -

stw 0x0D ; mem[0d]<=[adresaPRR1]	23 40 00 0D	107
ldw adresaPRR2 ;	21 40 01 61	10B
stw 0x0F ; mem[0f]<=[adresaPRR2]	23 40 00 0F	10F
ldw adresaPRR3 ;	21 40 01 63	113
stw 0x11 ; mem[11]<=[adresaPRR3]	23 40 00 11	117
ldb #0x20 ; acc<=20h	20 E0 00 20	11B
stb 0xF143 ; WCRL(5)<=acc	22 40 F1 43	11F
ldb #0x15 ; acc<=15h	20 E0 00 15	123
stb 0xF183 ; WCRL(6)<=acc	22 40 F1 83	127
ldb #0x2 ; acc<=2h	20 E0 00 02	12B
stb 0xF383 ; WCRL(7)<=acc	22 40 F3 83	12F
INTE ; omogucen prekid	05	133
stb 0xF140 ; startuje tajmer u/i 5	22 40 F1 40	134
stb 0xF180 ; startuje tajmer u/i 6	22 40 F1 80	138
stb 0xF380 ; startuje tajmer u/i 7	22 40 F3 80	13C
;		
loop: add R2 ; acc<=acc+R2	30 02	140
blss loop ; skok ako je manje	1A FC	142
stw 0x150 ; mem[150]<=acc	23 40 01 50	144
halt ; zaustavlja procesor	01	148
;		
; _____ PREKIDNE_RUTINA _____		
prekid1: ;		
ldw #0x10 ; acc<=10	21 E0 00 10	149
dec ; acc--	33	14D
stw R2 ; R2<=acc	23 02	14E
rti ;	0D	150
prekid2: ;		
ldw #0x20 ; acc<=20	21 E0 00 20	151
stw R2 ; R2<=acc	23 02	155
rti ;	0D	157
prekid3: ;		
ldw #0x30 ; acc<=30	21 E0 00 30	158
stw R2 ; R2<=acc	23 02	15C
rti ;	0D	15E
;		
; _____ ALOKACIJA_PROSTORA _____		
;alokacija prostora		
adresaPRR1: dw prekid1 ;	01 49	15F
adresaPRR2: dw prekid2 ;	01 51	161
adresaPRR3: dw prekid3 ;	01 58	163
;		

Prekid usled greške u načinu adresiranja je sledeći test primer. Cilj ovog primera je da prikaže kako procesor reaguje na unutrašnje prekide i to na grešku u načinu adresiranja kod instrukcije *stb* gde je odredišni operand neposredna vrednost. Namerno je podešeno da se smešta operand na neposrednu vrednost a to je nemoguće pa se onda desi greška. Obrada greške se vrši u prekidnoj rutini tako što se preskoči ta instrukcija u kojoj je greška.

greska u nacinu adresiranja

; _____ GLAVNI_PROGRAM _____			
prg	<i>; -----KOMENTAR-----</i>	<i>-----KOD-----</i>	<i>--ADRESE--</i>
ldw adresaPRR	<i>; ulaza u IV tabeli</i>	21 40 01 1B	100
stw 0x04	<i>; mem[04d]<=[adresaPRR]</i>	23 40 00 04	104
;			
stb #0x75	<i>; imm<=acc ERROR</i>	22 E0 75	108
ldb #0x45	<i>; acc<=45h</i>	20 E0 45	10B
halt	<i>;</i>	01	10E
; _____			
; _____ PREKIDNA_RUTINA _____			
prekid:	<i>;</i>		
popw	<i>; acc<=PSW</i>	25	10F
stw R0	<i>; R0<=PSW</i>	23 00	110
popb	<i>; acc<=PCL</i>	24	112
add #0x1	<i>; acc<=acc+4</i>	30 E0 01	113
pushb	<i>; mem[sp]<=acc</i>	26	116
ldw r0	<i>;</i>	21 00	117
pushw	<i>;</i>	27	119
rti	<i>; povratak u glavni program</i>	0D	11A
; _____			
; _____ ALOKACIJA_PROSTORA _____			
; alokacija prostora			
adresaPRR: dw prekid	<i>;</i>	01 0F	11B
; _____			

5.2 DETALJAN PREGLED ODABRANOG TESTA

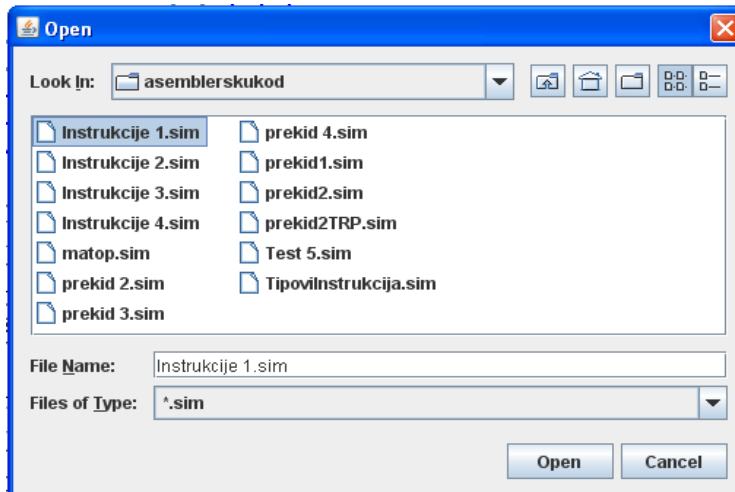
U ovoj glavi je dat detaljan opis simulacije aritmetičke operacije i niza adresiranja. Dok se asemblerski programi kojim se simuliraju logičke, pomeračke, instrukcije uslovnog skoka, skoka u podprogram, maskirajući prekidi, prekidi usled greške u adresiranju, prekidi posle svake instrukcije i softverski izazvani prekidi daju u prilogu. Obrađeni su skupovi instrukcija, adresiranja i mehanizam prekida. Svi testovi su napravljeni tako da se izvršavaju od adrese **100h**.

Dat je asemblerski kod sa komentarima:

; aritmeticka instrukcija			
; _____ GLAVNI_PROGRAM _____			
prg	<i>; -KOMENTAR-----</i>	<i>-----KOD-----</i>	<i>-----ADRESE-----</i>
ldb [R4+R5]0x10	<i>; acc<=mem[r4+r5+5h]</i>	20 A4 00 10	100
stb R1	<i>; r1<=acc</i>	22 01	104
ldb podatak	<i>; acc<<mem[podatak]</i>	20 40 01 11	106
add R1	<i>; acc<=acc add r1</i>	30 01	10A
stb podatak	<i>; mem[podatak]<=acc</i>	22 40 01 11	10C
halt	<i>; zaustavlja se CPU</i>	01	110
; _____			
; _____ ALOKACIJA_PROSTORA _____			
podatak: db 0x5	<i>;</i>	05	111
; _____			

Pošto je simulacija napravljena pokrećemo simulator i odabiramo simulaciju pod nazivom instrukcije 1.sim. Postupak pokretanja je sledeći:

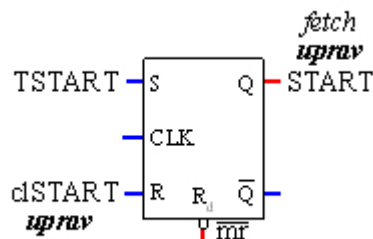
Izborom opcije *File* u meniju i stavke *Open* (slika 18), otvara se dijalog prozor za izbor simulacije. Klikom na željenu simulaciju i pritiskom na dugme *Open* pokreće se inicijalizacija simulacije i sistem je spreman za praćenje simulacije.



Slika 54. Otvaranje simulacije

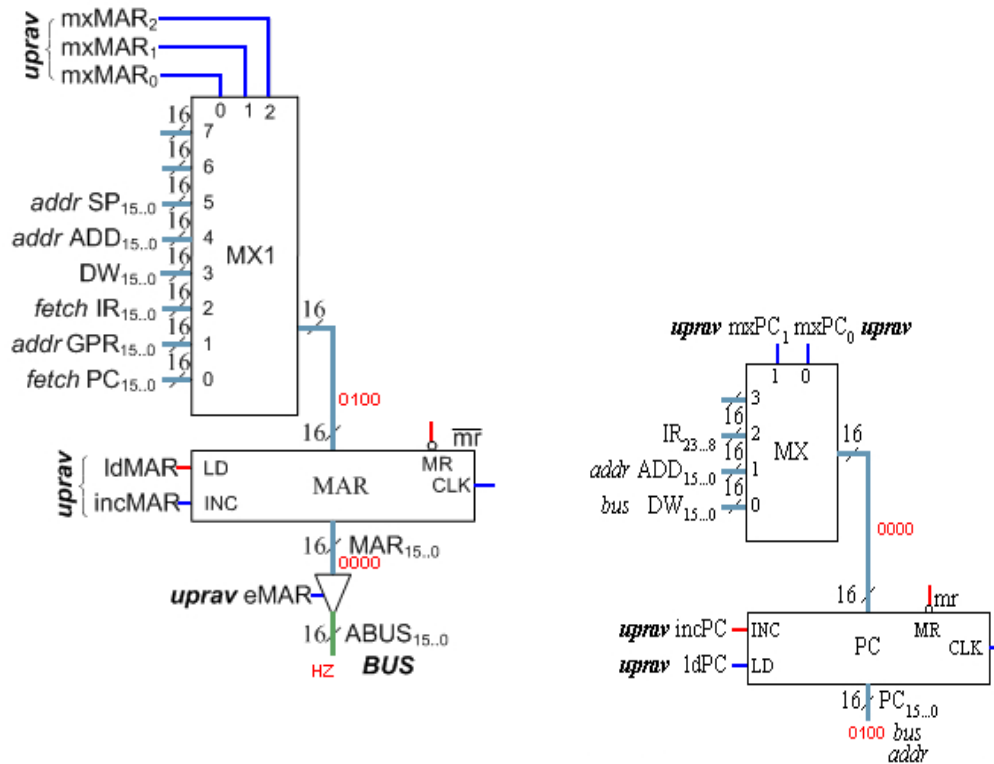
Postupak čitanja instrukcije iz memorije je prikazan kroz niz situacija koje treba da se dogode pre samog čitanja kao i trenutak kada je podatak u prihvatnom registru podataka MDR. Detaljno je prikazan prvi bajt a postupak čitanja ostalih reči se obavlja na isti način.

Prvo s proverava da li je sistem startovan slika 55.



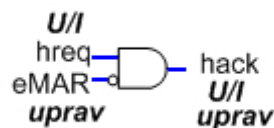
Slika 55. Flip-flop za pamćenje START komande

Pošto je sistem startovan unosi se adresa u registar MAR, propuštena kroz multiplekser MP1 iz registra PC i inkrementira se vrednost PC (slika 56).



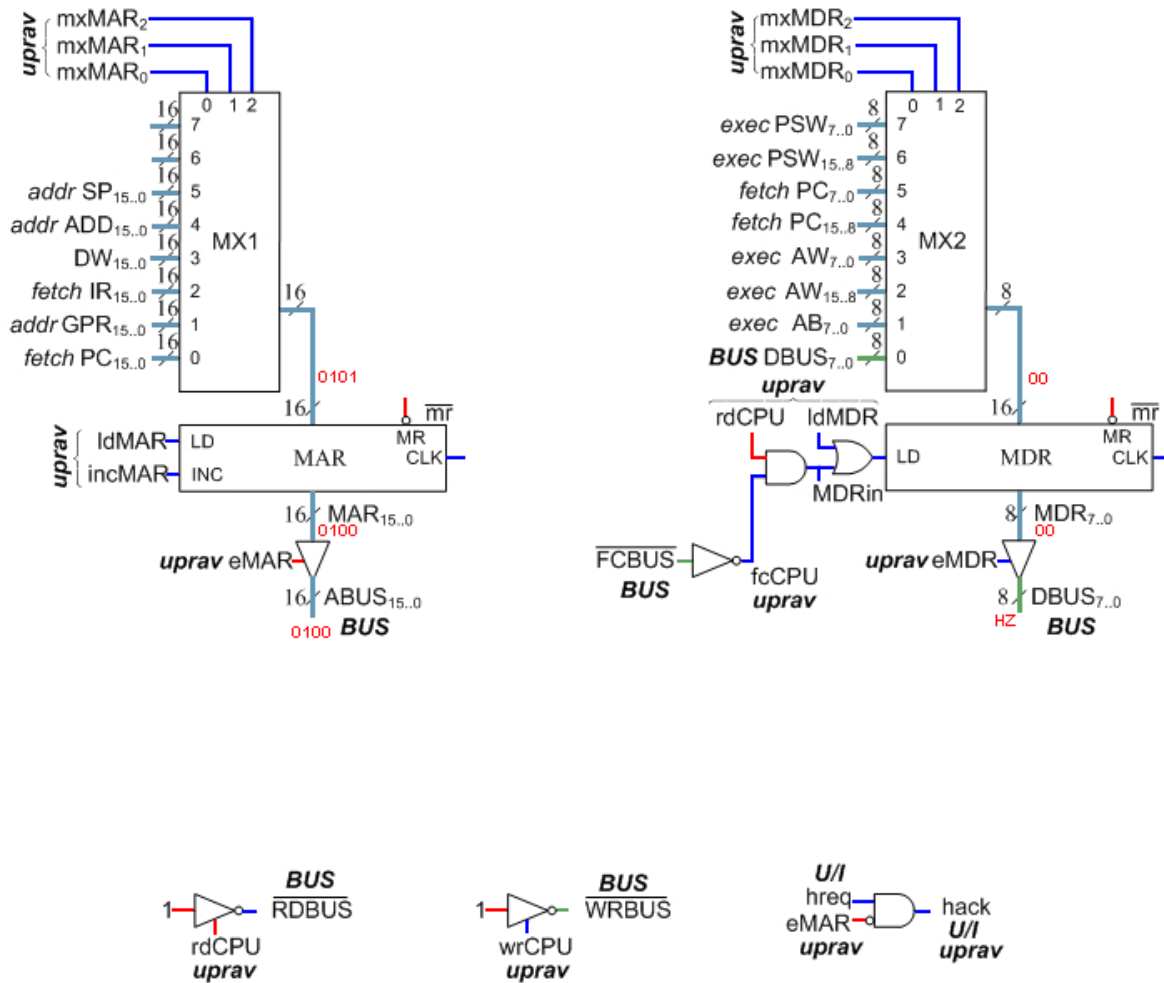
Slika 56. Registri MAR i PC

Ispituje se potom da li je magistral slobodna odnosno da li kontroler sa DMA obavlja ciklus na magistrali. U slučaju da je magistrala zauzeta čeka se dok je kontroler ne oslobodi (slika 57).



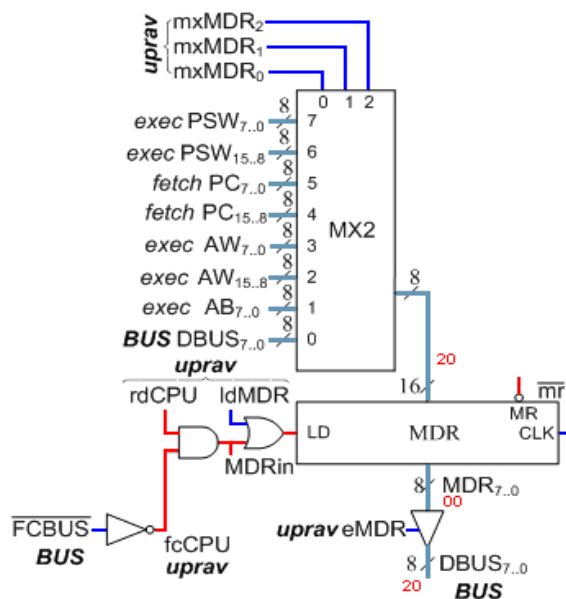
Slika 57. Mali arbitrator

U ovom slučaju je magistrala je slobodna i prelazi se na sledeći korak, a to je postavljanje adrese i zahteva za čitanje na adresne i kontrolne linije, respektivno.



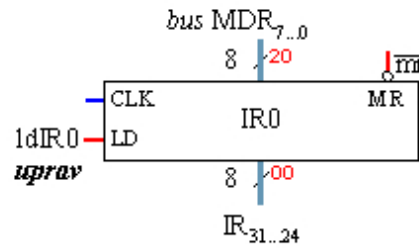
Slika 58. Blok bas u situaciji čekanja odgovora od memorije

Sada procesor čeka da memorije postavi podatak na magistralu i preko kontrolne magistrale /FCBUS javi procesoru da je podatak koji se čita ispravan (slika 59).



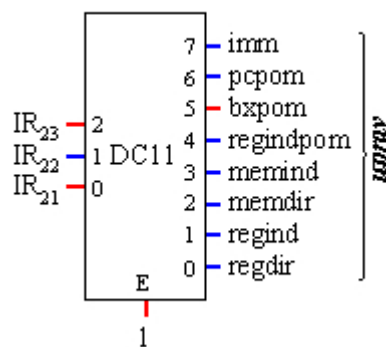
Slika 59. Registar MDR u trenutku očitavanja podatka iz memorije

Podatak se na sledeći signal takta unosi u registar MDR. Odakle će se potom uneti u registar instrukcija IR1 (slika 60).



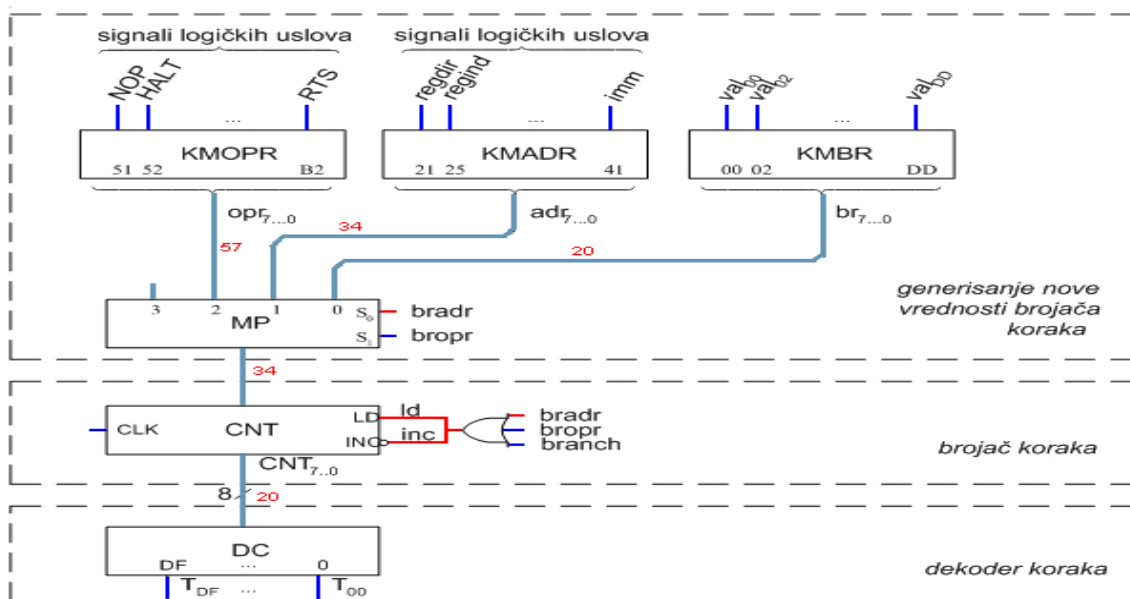
Slika 60. Registar IR0

Dalje se postupak čitanja obavlja na isti način. Kada se instrukcija u potpunosti očita sleduje deo za formiranje adrese i čitanje operanda. Prvo se dekoduje način adresiranja.



Slika 61. Dekoder načina adresiranja.

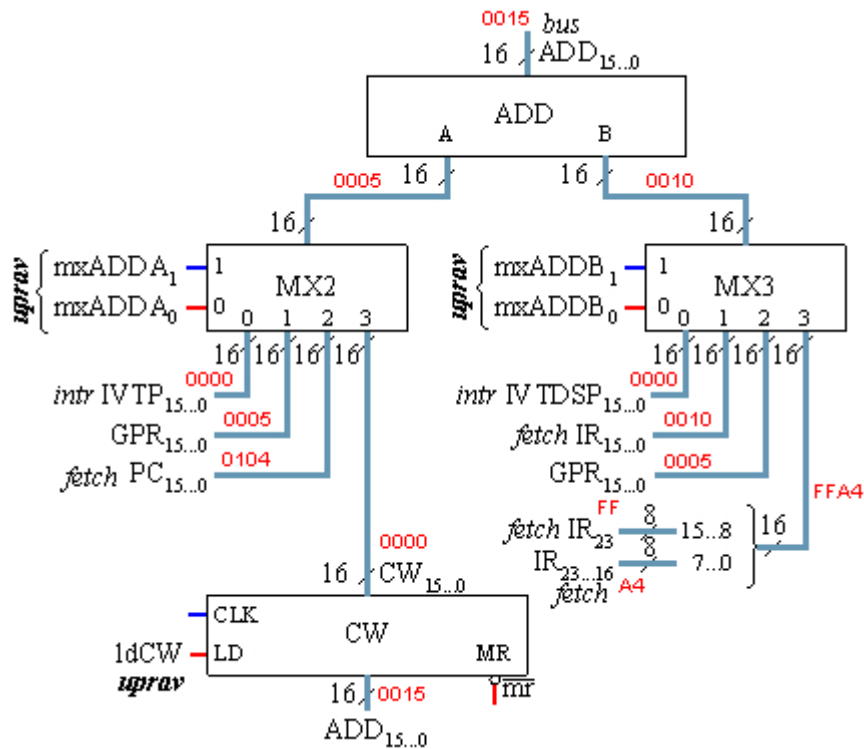
Na osnovu informacije o načinu adresiranja iz bloka za dohvaćanje instrukcije se u upravljačkoj jedinici (slika 62) generiše skok na određeni korak za obradu selektovanog načina adresiranja.



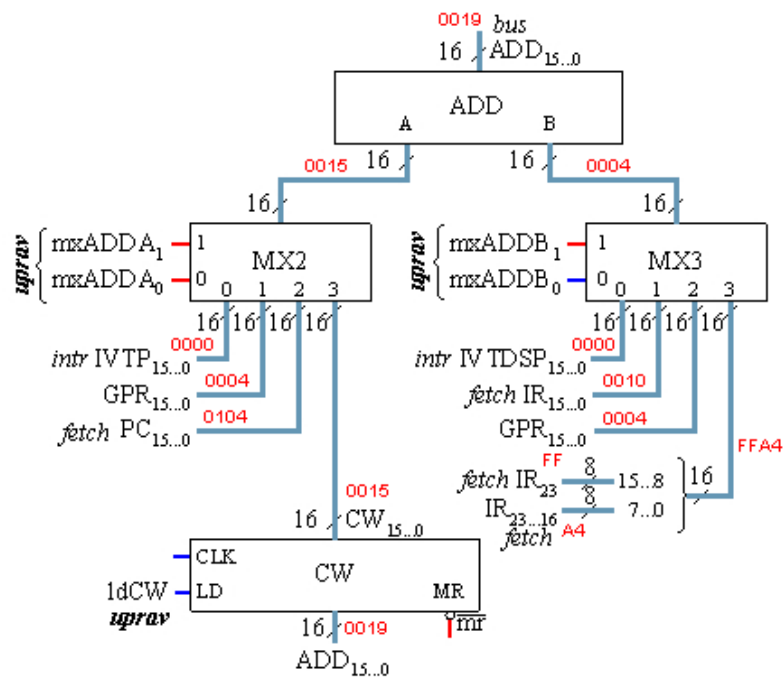
Slika 62. Deo upravljačke jedinice

U ovom se slučaju radi o bazno indeksnom adresiranju sa pomerajem. I generisanje adrese je malo složenije i dato je u sledećim slikam.

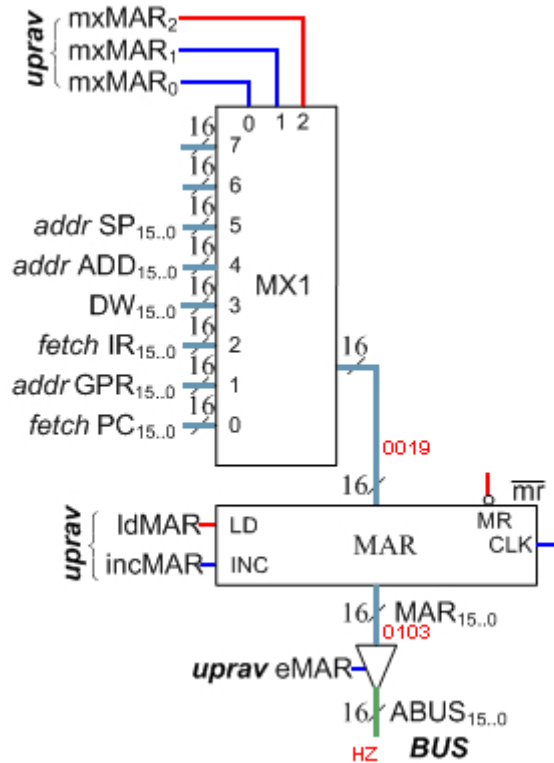
Prvo se sabiraju prvi registar i pomeraj a rezultat smešta u pomoćni registar CW (slika 63). Potom se selektuje sledeći registar i njegova vrednost sabira sa predhodno dobijenim rezultatom i upisuje novodobijna vrednost u registar MAR (Slike 64 i 65)



Slika 63. Prvo sabiranje

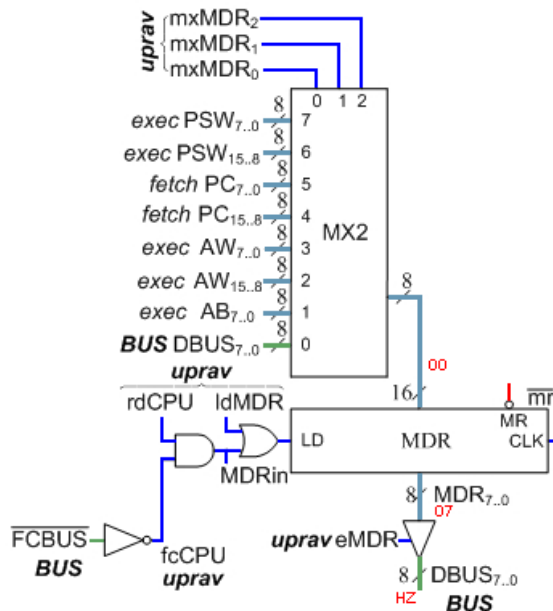


Slika 64. Drugo sabiranje



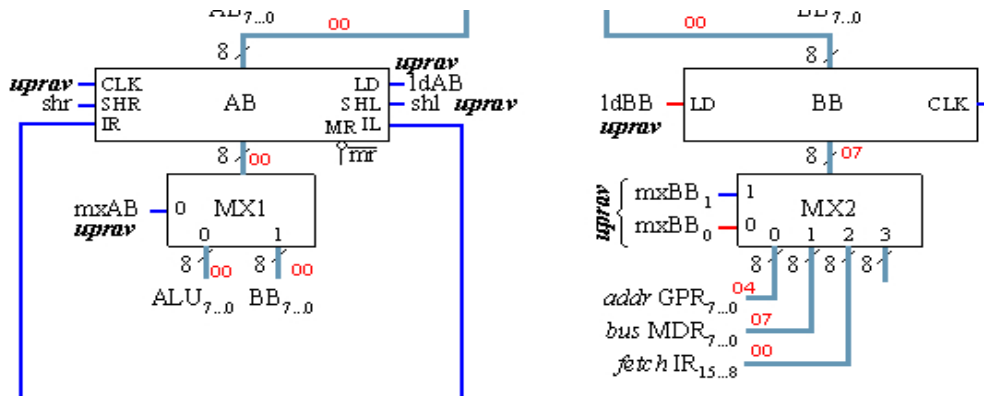
Slika 65. Upisivanje u MAR adrese podatka

Potom sledi čitanje operanda iz memorije, pošto su delovi isti kao i u slučaju kada se čita deo instrukcije prikazaćemo samo registar MDR sa očitanim podatkom.

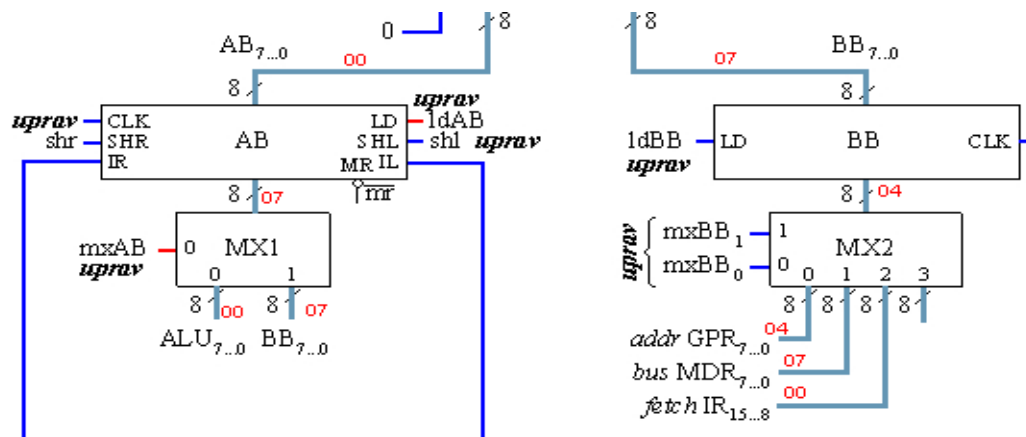


Slika 66. Učitani operand u MDR

Podatak se iz MDR upisuje u pomoćni registar BB pa se potom u sklopu izvršavanja instrukcije LDB premešta u akumulator AB (slike 67 i 68).

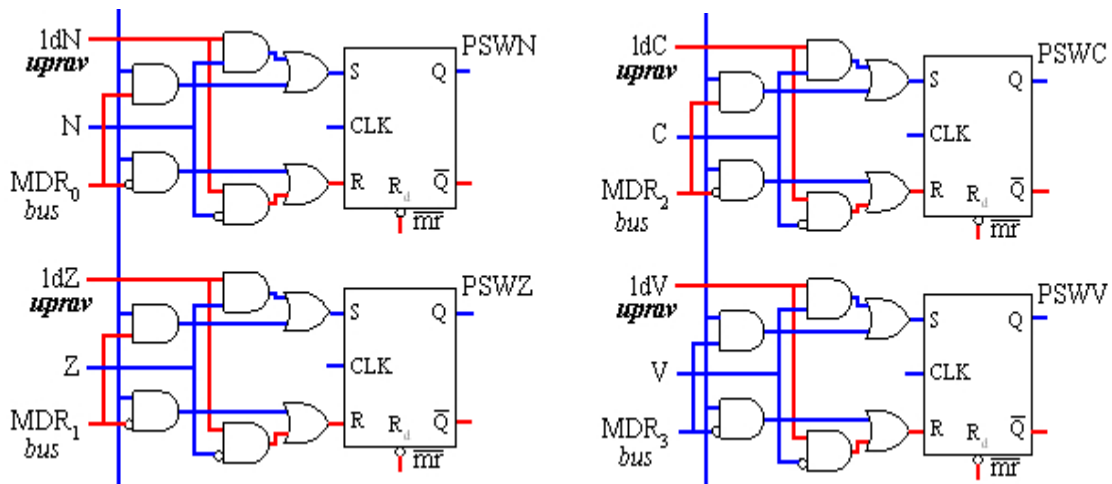


Slika 67. Upis u registar BB



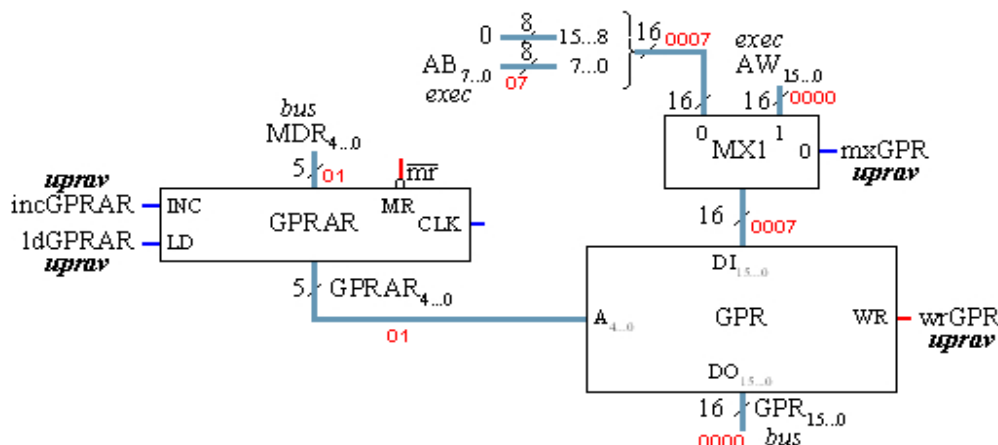
Slika 68. Izvršavanje instrukcije LDB

Potom se ažuriraju flegovi registra PSW slika 67 i provjerava da li ima prekida u ovom slučaju nema prekida.



Slika 67 Ažuriranje PSW-a

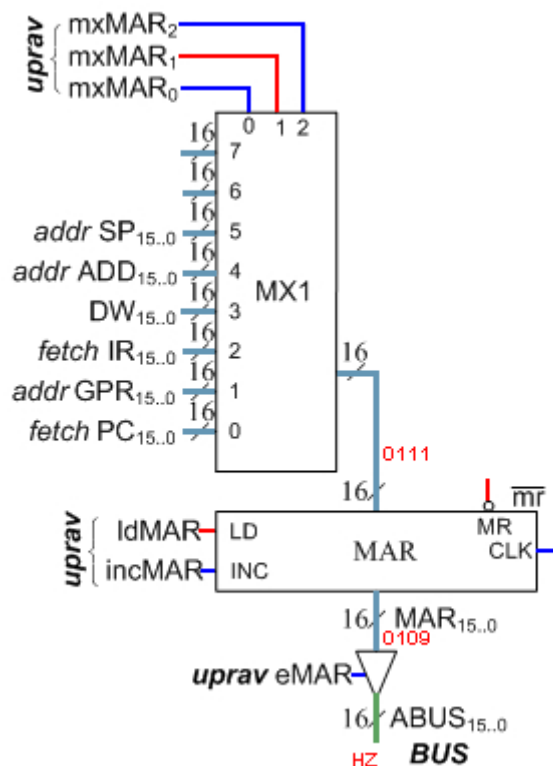
Izvršavanje instrukcije STB za slučaj registarskog adresiranja. Na slici 68 je prikazan upis u registarski fajl.



Slika 68. Upis u registarski fajl

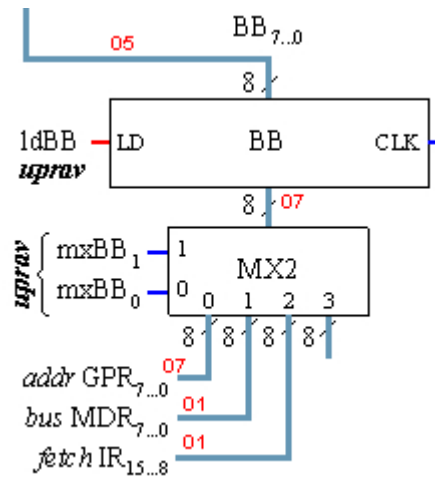
Sledeća instrukcija u programu je *LDB* sa memorijskim direktnim adresiranjem. Dohvaćanje instrukcije je preskočeno a detaljnije je opisan postupak dohvaćanja operanda i izvršavanje instrukcije.

Učitava se adresa operanda direktno iz registra instrukcija (slika 69.). Nadalje je postupak izvršavanja isti kao u predhodnom opisu instrukcije *LDB*. Na kraju se u akumulatoru *AB* nalazi vrednost očitana sa adrese *111h* u memoriji.

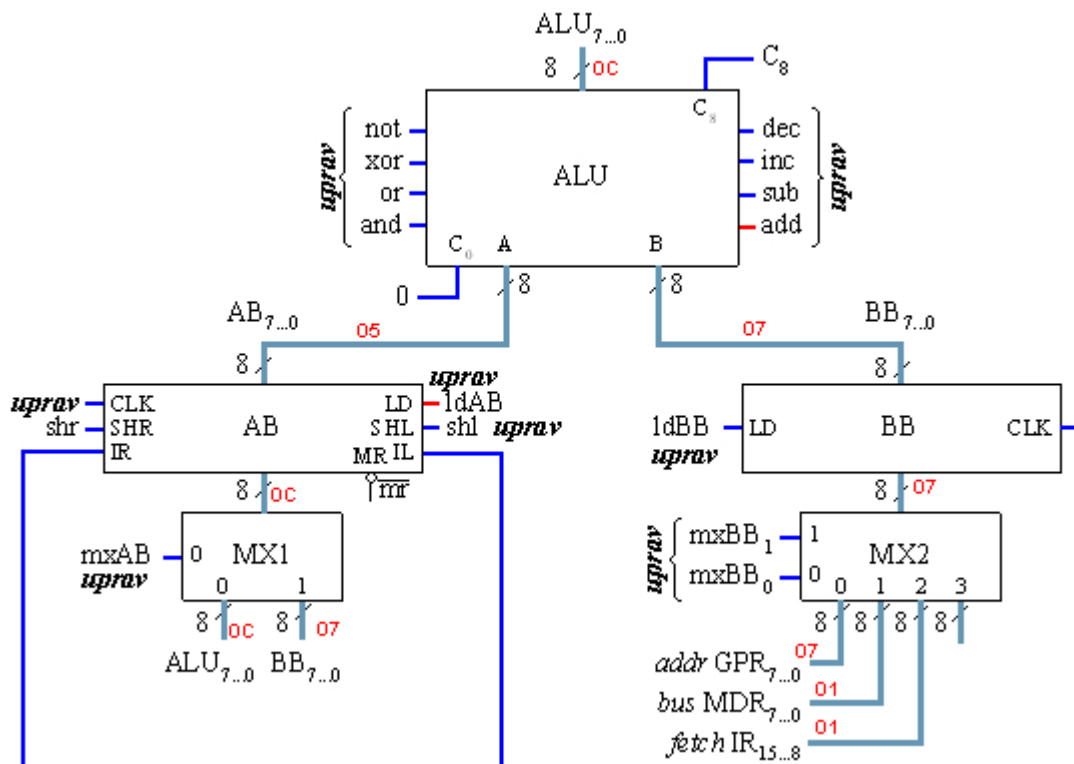


Slika 69. MAR registar

Sledi instrukcija sabiranja sa direktnim registarskim adresiranjem. Pošto se svi operandi nalaze u procesoru, vrednost selektovanog registra se propušta u pomocni registar *BB* (slika 70), i time se završava postupak dovlaćenja operanda sledi izvršavanje same instrukcij (slika 71) i ažuriranje flegova. Pošto se nijedan fleg nije generisao i sve ostaje po starom nisam prikazivao slike sa tim delovima sistema.

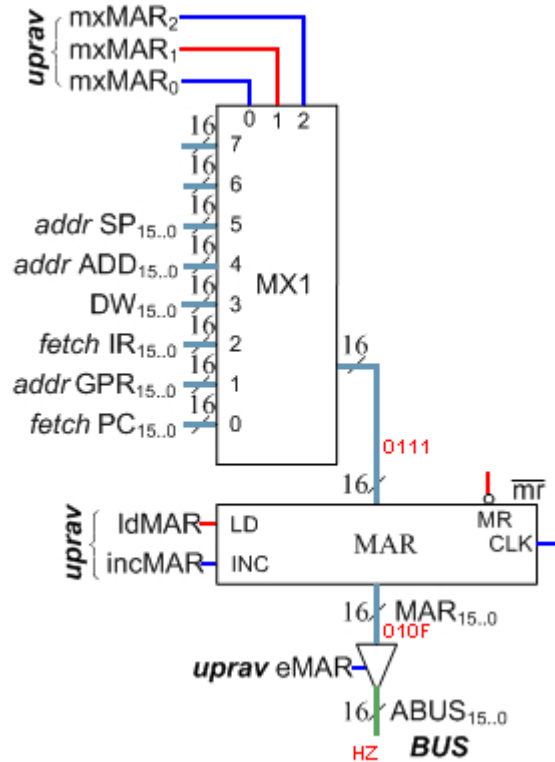


Slika 70. Dohvatanje drugog operanda



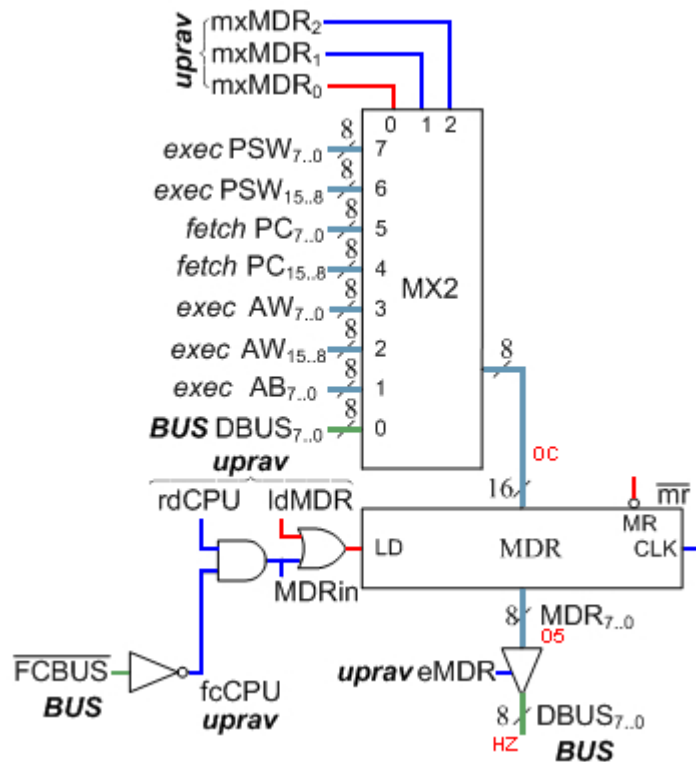
Slika 71. Izvršavanje sabiranja

Sledi instrukcija upisa bajta u memoriju *STB* sa direktnim memorijskom adresiranjem. U okviru postupka dohvaćanja operanda u MAR se upiše vrednost adrese na koju treba upisivati podatak i skoči se na deo za izvršavanje instrukcija (slika 72.).



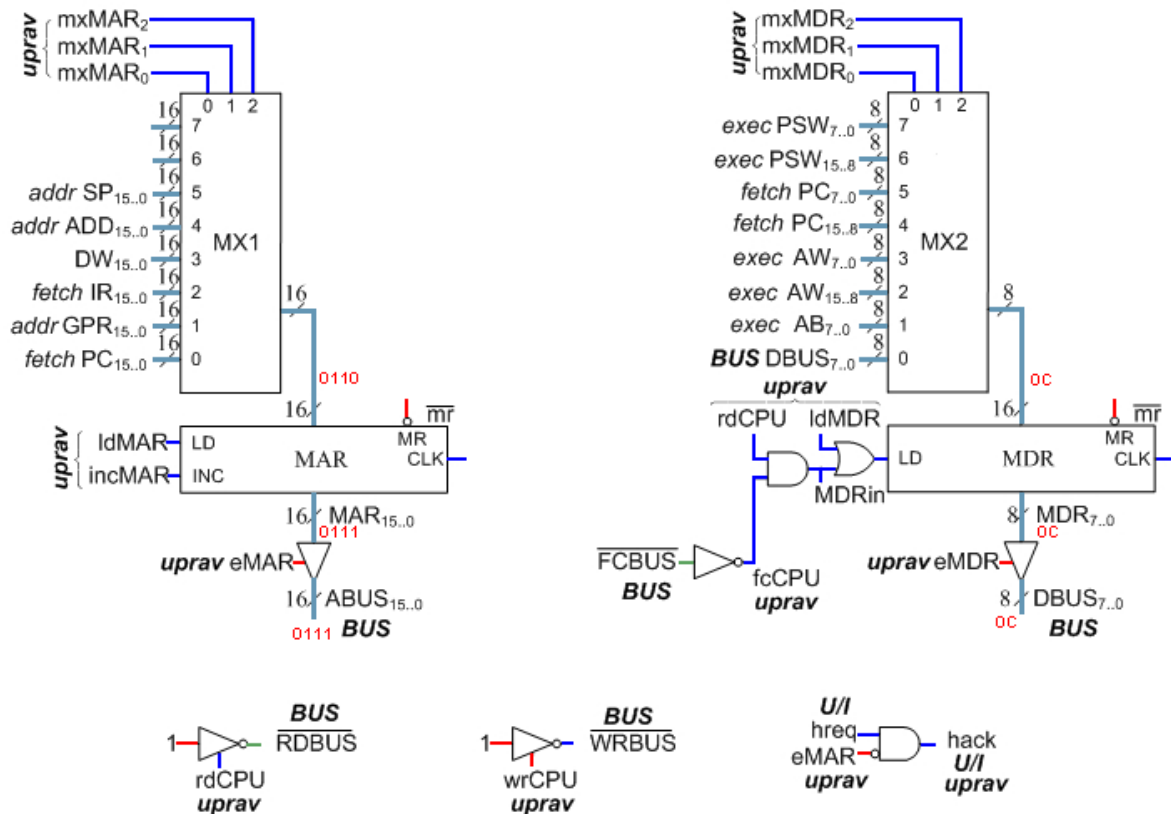
Slika 72. Unos adrese za smeštanje podatka

U izvršavanju vrednost iz registra AB propušta kroz multiplexer MX2 i upisuje u MDR i time je podatak spreman za slanje(slika 73).



Slika 73. Upis podatka u MDR

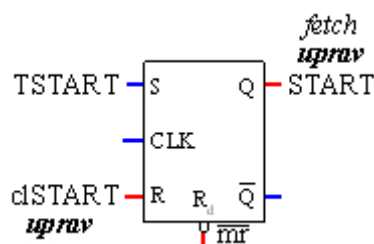
Pošto magistralu niko ne koristi procesor zauzima magistralu i postavlja adresu i podatke, kao i kontrolni signal upisa WRBUS na aktivnu vrednost (slika 74.).



Slika 74. Upis u memoriju

Memorija po završetku upisa šalje signal procesoru po magistrali FCBUS i postupak upisa je time završen.

Poslednja instrukcija u programu je HALT koja zaustavlja procesor tako što resetuje Flip-Flop START (slika 75).



Slika 75. Resetovanje signala START

Ovim je simulacija završena. Vrednost memorijske adrese 111h je promenjena i sada je 0Ch.

6 ZAKLJUČAK

Na osnovu gotovog dizajna računarskog sistema napravljen je vizuelni simulator. Arhitekturu procesora čine programski dostupni registri, tipovi podataka, formati instrukcija, načini adresiranja, skup instrukcija i mehanizam prekida. Od programskih dostupnih registara imamo 32 registra opšte namene, PC, akumulatori AB i AW, pokazivač na vrh steka SP, registar maski IMR i pokazivač na IV tabelu. Tipovi podataka su osmобitni i šesnaestobitne celobrojne veličine sa i bez znaka. Aritmetičke, logičke i pomeračke operacije rade sa osmобitnim, a šesnaestobitne vrednosti koriste se u operacijama prenosa. Postoji osam formata instrukcija i osam načina adresiranja. Instrukcije procesora se mogu svrstati u sledećih osam grupa instrukcija: bez dejstva, zaustavljanja, skoka, prenosa, aritmetičke instrukcije, logičke instrukcije, instrukcije pomeranja i rotiranja, instrukcije postavljanja indikatora u PSW. Prekidi uključuju unutrašnje i spoljašnje prekide sa maskiranjem i prioritiranjem prekida, pri čemu se kontekst procesora čuva na steku, a adresa prekidne rutine utvrđuje tehnikom vektorisanog mehanizma prekida. Organizacija procesora je podeljena u dve celine, a to su operaciona i upravljačka jedinica. Operaciona jedinica realizovana je sa direktnim vezama. Upravljačka jedinica je urađena u ožičenoj realizaciji. Simulator je dizajniran u programskom jeziku Java. Pored simulatora kreirani su i pomoćni programi radi ubrzavanja izrade simulatora. Program *Asembler* služi za prevodenje asemblerskog koda u binarni oblik. Drugi pomoćni program je *LineMaker* koji služi za generisanje grafičkog izgleda i dela logičkih komponenti simulatora. Korisnički interfejs simulator je dizajniran tako da korisniku bude intuitivno jasan i jednostavan za upotrebu. Navigacija je napravljena tako da korisnik jednostrukim klikom prelazi sa jednog na drugi deo sistema, a radi lakšeg snalaženja dato je i stablo hijerarhije. Dizajn se sastoji iz dve celine to su logički nivo gde se obavlja sama simulacija i prezentacioni nivo gde se korisniku grafički predstavlja rezultat simulacija. Za rad u laboratoriji priložen je skup test primera. Svi primeri su porkomentarisani i objašnjena je njihova namena. Izabran je jedan test i detaljno prikazan kroz skrinšotove i objašnjen svaki karakterističan korak.

Sam grafički dizajn simulatora je nastao kao posledica ličnog iskustva prilikom rada sa vizuelnim simulatorima i uočeni nedostaci na drugim simulatorima su u velikoj meri ispravljani na ovom simulatoru. Tokom testiranja ispravnosti realizovanog simulatora i funkcionalnosti dizajna zaključio sam da je omgućeno lako kretanje kroz hijerarhiju sistem, praćenje simulacije kao i pregledanje i menjanje stanja u sistemu. Skladan izbor boja u sistemu daje korisniku jasnu informaciju o dešavanjima u sistemu i ne zamara korisnika. Brzina izvršavanja

Postupak kreiranja simulatora bi mogao biti pojednostavljen unapređivanjem pomoćnih programa. Prikazani simulator procesora se može smatrati osnovom na koju se daljom nadogradnjom može dobiti računarski sistem. Taj računarski sistem se može razraditi do sitnijih detalja, simulator sa logičkog nivoa se može spustiti na elektronski nivo. Koristeći aproksimativne formule može se računati potrošnja svakog od kola u procesoru i sistemu u celini. Danas se pored smanjivanja površine čipa radi i na smanjenju potrošnje, pa bi se na ovaj način moglo doći do simulatora koji proračunava energetsку efikasnost, a i potrošnju zavisnu od tipa instrukcija koje se na njemu izvršavaju. Koristeći specijalno odabran skup test vektora može se odrediti najbolja upotreba datog procesora.

Daljim usavršavanjem programa *LineMaker* može se napraviti razvojno okruženje za projektovanje kombinacionih i sekvencijalnih logičkih šema, koje se mogu spojiti u celinu koja predstavlja neki mikrokontroler ili procesor. Takođe, ovaj program može se doraditi tako da se koristi u obrazovne svrhe. Studentu se zada da isprojektuje nekakvo logičko kolo koje zadovoljava određenu logičku formulu u nesređenom obliku, a da potom to isto uradi za sređeni oblik formule, pa da se tako dobijene šeme testiraju. Na osnovu rezultata koji se dobiju za zadate testove upoređuje se energetska efikasnost i vreme propagacije kroz sistem i na osnovu toga se potvrđuje da je minimalizovani oblik bolji od nesređene formule.

Program za prevođenje je takođe moguće unaprediti dodavanjem sintaksnog i semantičkog analizatora. Takođe, može se dodati veći skup direktiva ili neke nove instrukcije, koje će se prevoditi u niz drugih (osnovnih) instrukcija, a sve sa ciljem da se korisniku olakša rad. Moguće je uvođenje instrukcije petlje ili direktive petlje koja bi imala određeni registar kao podazumevani brojač.

7 LITERATURA

- [1] Jovan Đorđević, *Računarski sistem za rad u laboratoriji*, Elektrotehnički fakultet, Beograd, 2007.
- [2] Ivan Marković, *Simulator memorijskog i ulazno izlaznih modula računarskog sistema za rad u laboratoriji*, Elektrotehnički fakultet, Beograd, 2009.
- [3] www.java.sun.com
- [4] Laslo Kraus, Zbirka zadataka iz projektovanja softvera, Akademska misao, Beograd, 2007.
- [5] J. Djordjevic, A. Milenkovic, N. Grbanovic, "An Integrated Educational Environment for Teaching Computer Architecture and Organisation," IEEE MICRO, May 2000, pp. 66-74.
- [6] J. Đorđević, B. Nikolić, *Vizuelni simulator edukacionog računara*, Zbornik radova IT 2001, Žabljak, Jugoslavija, Mart 2001.
- [7] www.starume.com
- [8] <http://www.eclipse.org>
- [9] N. Grbanovic, J. Djordjevic, B. Nikolic, *Software package for an edicational computer system*, International Journal of Electrical Engineering Education, 40/4 October 2003, pp 270-284.
- [10] . Stallings, W., *Computer Organization and Architecture*, 6th edition, Prentice-Hall, Englewood Cliffs, New Jersey, USA, 2003.