SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
Zavod za elektroniku, mikroelektroniku, računalne i inteligentne sustave

Laboratorijske vježbe

ARHITEKTURE RAČUNALA 2

Mikroprogramiranje

Zagreb, listopad 2012.

1 Uvod

Predmet ove laboratorijske vježbe je mikroprogramiranje. Mikroprogramiranje je jedna od dvaju osnovnih tehnika izvedbe upravljačke jedinice procesora.

Da bi nam bilo jasnije o čemu se radi, trebamo najprije nešto znati o unutarnjoj građi procesora. Procesor, kao središnja komponenta računala, interno se sastoji od nekoliko komponenata: skupa registara, aritmetičko-logičke jedinice, upravljačke jedinice, te veza među njima. Razmotrimo ukratko uloge pojedinih komponenata. Uloga registara je privremeno pohranjivanje podataka, a aritmetičko-logička jedinica (ALU) omogućuje obavljanje različitih jednostavnih aritmetičkih i logičkih operacija nad podatcima iz registara. Registri i ALU međusobno su povezani jednom ili većim brojem spojnih puteva, tzv. *internih sabirnica*. Procesor će morati omogućiti i vezu prema glavnoj memoriji i možebitnim drugim komponentama računala, a ona će biti ostvarena putem posebnih registara, koji su spojeni na vanjske, adresne i podatkovne, izvode procesora. Aritmetičko-logička jedinica, registri i interne sabirnice koje ih povezuju, često se nazivaju zajedničkim imenom *put podataka*.

Ovom osnovnom strukturom procesora određene su neke elementarne operacije koje je procesor u stanju izravno, na najnižoj sklopovskoj razini podržati. Npr:

- postavljanje podatka iz jednog registra na internu sabirnicu;
- aktiviranje određene aritmetičke ili logičke operacije u ALU;
- upis podatka s interne sabirnice u neki od registara;
- aktiviranje čitanja podatka iz memorije, ili upisa podatka u memoriju.

Ovakve elementarne, izravno sklopovski podržane operacije, nazivaju se *mikrooperacijama*.

Rad procesora svodi se na dohvaćanje i izvršavanje strojnih instrukcija, koje su predstavljene numeričkim kodom i pohranjene u glavnoj memoriji. Iako su sa stajališta programera strojne instrukcije vrlo jednostavne, štoviše, najelementarnije moguće instrukcije, sa stajališta procesora one su relativno složene i često ih procesor neće moći izvršiti u jednom koraku nego će ih morati razložiti na niz elementarnih koraka koji su izravno sklopovski podržani, odnosno na niz mikrooperacija, u odgovarajućem vremenskom redoslijedu. Npr., strojna instrukcija ADD A,B,C (zbroji sadržaje registara B i C i rezultat spremi u registar A) u tipičnom se procesoru ne može izvesti u jednom koraku, nego se mora razložiti na sljedeće mikrooperacije:

- postavi sadržaj registra A na internu sabirnicu koja je spojena na jedan ulaz ALU;
- postavi sadržaj registra B na internu sabirnicu koja je spojena na drugi ulaz ALU;
- aktiviraj operaciju zbrajanja u ALU (ovisno o rezultatu operacije postavi i odgovarajuće zastavice procesora, obično su to Z (nula), V (preljev), C (prijenos), N (negativan broj));
- podatak s interne sabirnice koja je spojena na izlaz iz ALU upiši u registar C.

Svakoj mikrooperaciji u procesoru je pridružen odgovarajući *upravljački signal* koji njome upravlja (aktiviranje odnosno deaktiviranje upravljačkog signala aktivira odnosno deaktivira odgovarajuću

mikrooperaciju). Procesor dakle mora, na temelju saznanja o tome koja strojna instrukcija se treba izvršiti, generirati odgovarajući vremenski slijed upravljačkih signala. Zadaća generiranja slijeda upravljačkih signala na temelju operacijskog koda tekuće strojne instrukcije spada na *upravljačku jedinicu* procesora.

Kao što je već napomenuto, dva su osnovna načina realizacije upravljačke jedinice: sklopovski i mikroprogramski.

1. Ako je upravljačka jedinica sklopovski realizirana, ona se promatra kao sekvencijski digitalni sklop – crna kutija koja na ulazu prima operacijski kod tekuće instrukcije i sistemski takt, a na izlazu generira upravljačke signale u odgovarajućem vremenskom slijedu. Ovako izvedena upravljačka jedinica projektira se standardnim metodama koje se u digitalnoj logici koriste za projektiranje sekvencijskih digitalnih sklopova. Prednosti ovakve (sklopovske) izvedbe upravljačke jedinice su manje zauzeće površine na čipu, manja potrošnja i veća brzina rada, no nedostatak joj je u tome što je za bilo kakvu promjenu u jednom realiziranoj sklopovskoj upravljačkoj jedinici potreban potpuni redizajn.

2. Alternativni pristup izvedbi upravljačke jedinice predstavlja mikroprogramiranje. Kod mikroprogramski izvedene upravljačke jedinice, svaka strojna instrukcija predstavljena je nizom tzv. *mikroinstrukcija*. Ove mikroinstrukcije smještene su u posebnoj memoriji, tzv. *mikroprogramskoj* ili *upravljačkoj memoriji*, koja se nalazi unutar same upravljačke jedinice procesora (u stvarnim mikroprogramiranim procesorima to je tipično memorija tipa ROM).Svaka mikroinstrukcija specificira upravljačke signale koje je u nekom trenutku potrebno aktivirati. Niz mikroinstrukcija tvori tzv. *mikroprogram*. Svakoj strojnoj instrukciji odgovara, dakle, jedan mikroprogram smješten u mikroprogramskoj memoriji. Na temelju operacijskog koda trenutne strojne instrukcije, upravljačka jedinica "zna" gdje se u mikroprogramskoj memoriji nalazi mikroprogram koji odgovara baš toj strojnoj instrukciji, te ga izvršava. Strojne instrukcije nazivaju se u ovom kontekstu *makroinstrukcijama*, a programi sačinjeni od strojnih instrukcija nazivaju se *makroprogramima*. Korištenjem mikroprogramiranja, projektant procesora može realizirati vlastiti skup strojnih instrukcija. Koncept mikroprogramiranja detaljnije je opisan u predavanjima i u knjizi S.Ribarić, Građa računala, Algebra, Zagreb, 2011 (poglavlja 7.2 (str. 163-165) i 7.4, (str. 187-200)).

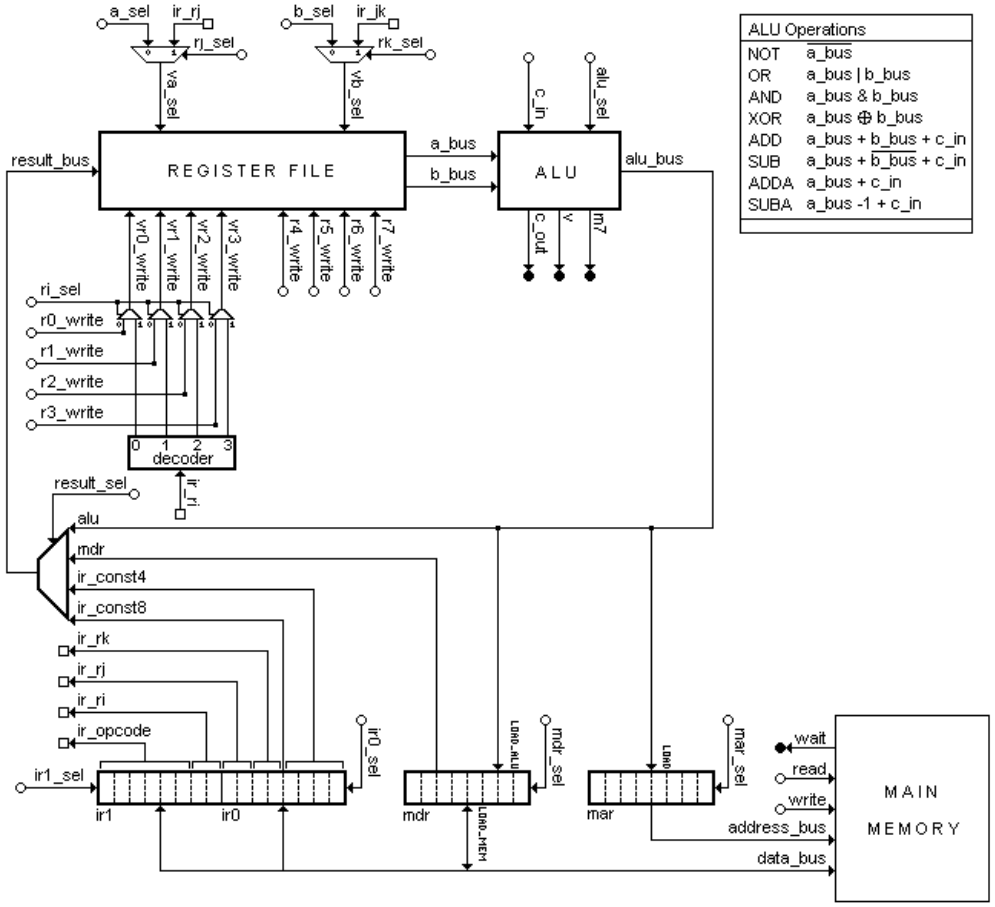
U ovoj vježbi studenti će realizirati vlastiti skup strojnih instrukcija za procesor zadane interne organizacije. Na taj način steći će osjećaj za rad procesora na najnižoj razini. Pošto nam stvarni mikroprogramirljivi procesor nije na raspolaganju, vježba će se obaviti korištenjem simulatora **MythSim**, razvijenog na University of Illinois, Chicago, USA. Simulator je pisan u programskom jeziku Java i može se pokretati na svim raširenijim operacijskim sustavima. Do simulatora se može doći na internim stranicama kolegija Arhitektura računala 2 (FER web).

2 Model mikroprogramiranog procesora

Kao što smo vidjeli u uvodu, projektant mikroprogramirane upravljačke jedinice mora za svaku strojnu instrukciju (makroinstrukciju) napisati odgovarajući mikroprogram. Mikroprogram je niz mikroinstrukcija koje će se izvesti jedna za drugom, a svaka mikroinstrukcija specificira skup upravljačkih signala koji će biti aktivirani u trenutku njezina izvođenja u svrhu pobuđivanja odgovarajućih mikrooperacija. Iz ovoga je jasno da projektant – mikroprogramer mora najprije znati

kakve mikrooperacije njegov procesor podržava, odnosno mora detaljno poznavati internu organizaciju procesora, da bi onda mogao svaku makroinstrukciju razložiti na odgovarajući slijed mikrooperacija i na temelju njih napisati mikroprogram.

Kao što smo također vidjeli u uvodu, općenito govoreći put podataka procesora sastoji se od skupa registara, aritmetičko-logičke jedinice i odgovarajućeg broja internih sabirnica koje ih povezuju. No koliko točno ima registara, koliko su oni široki (koliko bitova imaju), koliko ima internih sabirnica, koje operacije podržava ALU, ovisi o konkretnom procesoru koji se koristi.



Slika 1: **Model mikroprogramiranog procesora**

Model mikroprogramiranog procesora koji se koristi u simulatoru MythSim, odnosno njegovog puta podataka, prikazan je na slici 1. Njegove značake su sljedeće:

1. **Skup registara**. Procesor koristi skup od 8 8-bitnih registara opće namjene. Ti registri nazivaju se imenima r0 – r7. Na slici 1 ovi registri prikazani su kućicom označenom kao "register file" u gornjem lijevom dijelu slike. Osim ovih osam registara, procesor još posjeduje 8-bitni memorijski adresni registar (MAR) i 8-bitni memorijski registar podataka (MDR) koji služe za komunikaciju s memorijom, te 16-bitni instrukcijski registar (IR) koji sadrži trenutnu strojnu instrukciju. Ova posljednja tri registra prikazana su pri dnu slike 1.

   Primijetimo da procesor ne raspolaže posebnim registrom za programsko brojilo. Pošto je programsko brojilo neizostavno potrebno za realizaciju procesora von Neumannove arhitekture, bit će potrebno jedan od registara opće namjene r0 – r7 (tipično r7, ali projektant može odlučiti i drukčije) koristiti kao programsko brojilo. Isto razmatranje vrijedi i za statusni

registar i kazalo stoga – u osnovnom putu podataka nema registara koji bi odgovarali ovim ulogama, te, ako su nam takvi registri potrebni, moramo u tu svrhu koristiti neki od registara iz registarskog skupa r0 – r7.

2. **Aritmetičko-logička jedinica**. Procesor raspolaže 8-bitnom ALU (na slici 1 desno od skupa registara) koja podržava 8 aritmetičkih i logičkih operacija. Podržane operacije navedene su u tablici u desnom gornjem kutu slike. Neke operacije (konkretno, logička operacija NOT te aritmetičke operacije ADDA i SUBA) djeluju samo nad jednim operandom, dok ostale djeluju na dva operanda. Odabir konkretne aritmetičke ili logičke operacije koju ALU izvodi u određenom trenutku vrši se trobitnim upravljačkim signalom alu_sel. ALU također ima i dodatni upravljački signal c_in (ulazni bit prijenosa) koji se koristi u aritmetičkim operacijama i koji se može postaviti u 0 ili 1. Izlazi iz ALU su: 8-bitni rezultat aritmetičke ili logičke operacije (izlaz alu_bus) te bitovi c_out (engl. carry out - prijenos s najvišeg bita), v (engl. overflow – aritmetički preljev) te m7 (najznačajniji bit rezultata, koji predstavlja informaciju o predznaku rezultata).

3. **Interne sabirnice**. Procesor koristi tri interne sabirnice (trosabirnička arhitektura): dvije sabirnice operanda – na slici označene kao a_bus i b_bus (između skupa registara i ALU) i jednu sabirnicu rezultata - na slici označena kao result_bus (ulaz u skup registara). Tijekom svakog procesorskog takta, na sabirnice operanada postavljaju se sadržaji određenih registara iz skupa r0 - r7. Odabir konkretnih registara čiji će sadržaji biti postavljeni na sabirnice operanada vrši se upravljačkim signalima a_sel, b_sel, rj_sel i rk_sel. Točna semantika ovih signala i način njihove uporabe bit će objašnjeni kasnije, u poglavlju o strukturi upravljačke jedinice.

   Tijekom svakog procesorskog takta moguće je upisati sadržaj sa sabirnice rezultata u jedan ili više registara iz skupa r0 – r7. Odabir takvih odredišnih registara obavlja se upravljačkim signalima ri_sel, te r0_write – r7_write. Podatak koji se nalazi na sabirnici rezultata može biti rezultat aritmetičke ili logičke operacije (izlaz iz ALU) ili podatak iz memorijskog registra podataka, ili pak konstanta (8-bitna ili predznačno pršrena 4-bitna) sadržana u instrukcijskom registru kao sastavni dio instrukcije (tzv. usputna konstanta). Odabir jedne od ovih četiriju mogućnosti vrši se pomoću multipleksora kojim upravlja dvobitni upravljački signal result_sel.

4. **Veza s memorijom**. Veza s memorijom ostvarena je preko već spomenutih registara MAR (memorijski adresni registar, spojen na adresnu sabirnicu) i MDR (memorijski podatkovni registar, spojen na podatkovnu sabirnicu), te upravljačkih signala read, write i wait.

   (a) Operacija čitanja. Prilikom operacije čitanja, procesor mora postaviti adresu ciljne memorijske lokacije u MAR, te aktivirati signal read. Memoriji je potrebno određeno vrijeme prije nego što odgovori i postavi podatak na podatkovnu sabirnicu procesora (vrijeme pristupa), koje procesoru nije unaprijed poznato. Stoga memorija drži aktivnim signal wait sve dok pristup podatku nije završen, nakon čega postavlja podatak na podatkovnu sabirnicu i deaktivira signal wait. Deaktiviranje signala wait znak je procesoru da je traženi podatak postavljen na podatkovnu sabirnicu i da ga procesor može preuzeti. Primijetimo da podatak koji je memorija postavila na podatkovnu sabirnicu prilikom operacije čitanja, procesor može upisati u memorijski registar podataka MDR (aktiviranjem signala mdr_sel) ili u viši ili niži bajt instrukcijskog registra (aktiviranjem upravljačkih signala ir1_sel, odnosno ir0_sel).

   (b) Operacija pisanja. Prilikom operacije pisanja, procesor postavlja i adresu ciljne memorijske lokacije u MAR, i podatak u MDR, te aktivira signal write. Memorija drži aktivnim signal wait sve dok podatak nije upisan, signalizirajući time procesoru koliko

dugo traje operacija pisanja. Za razliku od čitanja, upis podatka u memoriju moguće je jedino preko registra MDR.

Ako je trenutna mikrooperacija aktivirala signal read (ili write), a odabir sljedeće mikrooperacije ovisi o signalu wait, pretpostavljamo da će se signal wait aktivirati tijekom istog perioda takta.

Organizacija procesora osigurava da se isti registar može koristiti i kao izvorni i kao odredišni operand, bez pojave nedeterminizma. Za potrebe zadovoljenja znatiželje bistrih studenata ovdje ćemo navesti dva načina da se to provede u praksi. Prvi način se temelji na generatoru slijedova uz $k = 2$. Čitanje sabirnice operanada i upisivanje u interni registar ALU provelo bi se u φ_1, dok bi se upisivanje preko sabirnice rezultata provelo u φ_2. U drugom načinu elemente registarskog skupa izveli bismo bridom okidanim bistabilima. Upisivanje vrijednosti proveli bismo na brid signala takta koji odgovara kraju instrukcijskog ciklusa.

3 Struktura strojne instrukcije (makroinstrukcije)

Prije analiziranja mikroprogramirane izvedbe upravljačke jedinice, pogledajmo strukturu strojnih instrukcija (makroinstrukcija). Kao što je spomenuto u prethodnom poglavlju, instrukcijski registar ovog modela procesora je 16-bitni, iz čega proizlazi da će i strojne instrukcije biti 16-bitne. Budući da je glavna memorija bajtno organizirana (jedna adresa odgovara jednom bajtu, što se vidi iz činjenice da je podatkovni registar 8-bitni), svaka strojna instrukcija će u memoriji biti pohranjena na dvije susjedne adrese. Dohvat svake instrukcije tijekom faze "pribavi" također će se morati obaviti u dva koraka (najprije jedan bajt, zatim drugi), jer se komunikacija s glavnom memorijom odvija preko 8-bitne podatkovne sabirnice.

Instrukcija se dakle pohranjuje u 16-bitni instrukcijski registar IR. Na slici 1 možemo uočiti da taj instrukcijski registar ima specifičnu strukturu:

- Najviših 6 bitova instrukcijskog registra (bitovi 15 – 10) uvijek predstavljaju operacijski kod instrukcije. To znači da razmatrani procesor podržava moguće najviše $2^6 = 64$ različitih instrukcija.
- Preostalih 10 bitova može poslužiti za specifikaciju registara i konstanti koji sudjeluju u operaciji, u skladu s jednim od dva moguća formata instrukcije:

  1. Troadresna strojna instrukcija (specificira tri registra koji sudjeluju u operaciji – dva registra operanada i jedan za odredište; moguće je dodatno specificirati i 4-bitnu konstantu koja se predznačno proširuje na 8 bita prije sudjelovanja u operaciji).

15 10	9 8	7 6	5 4	3 0
ir_opcode	**ir_ri**	**ir_rj**	**ir_rk**	**ir_const4**

  Dva bita neposredno iza operacijskog koda (bitovi 9 – 8, odnosno polje ir_ri) specificiraju odredišni registar. Ako instrukcija nema potrebu za odredišnim registrom, ovi bitovi se ne koriste. Primijetimo da je za specifikaciju odredišnog registra unutar makroinstrukcije na raspolaganju samo 2 bita, što znači da je na ovaj način u makroinstrukciji moguće kao odredište koristiti samo 4 registra: registre r0 – r3 (u ostale registre je također moguće upisivati rezultat, ali ih nije moguće specificiratiu u makroinstrukciji kao odredište, nego ih je moguće koristiti samo na razini mikroinstrukcija, kao što ćemo objasniti kasnije).

Dvije grupe od po dva bita u nižem bajtu instrukcijskog registra (bitovi 7 – 6 i 5 – 4, odnosno polja ir_rj i ir_rk) mogu služiti za specifikaciju dvaju registara operanada. Ako instrukcija ima potrebu samo za jednim operandom, onda koristi samo jedno od tih dvaju polja, dok drugo ostaje neiskorišteno. Opet, s obzirom da su ove grupe dvobitne, moguće je kao operande u makroinstrukciji specificirati samo 4 registra: r0 – r3.

Četiri najniža bita instrukcije (bitovi 3 - 0, odnosno polje ir_const4) mogu se koristiti za specificiranje konstante.

2. Jednoadresna strojna instrukcija (specificira samo jedan registar i dodatno 8-bitnu konstantu).

15 10	9 8	7 0
ir_opcode	**ir_ri**	**ir_const8**

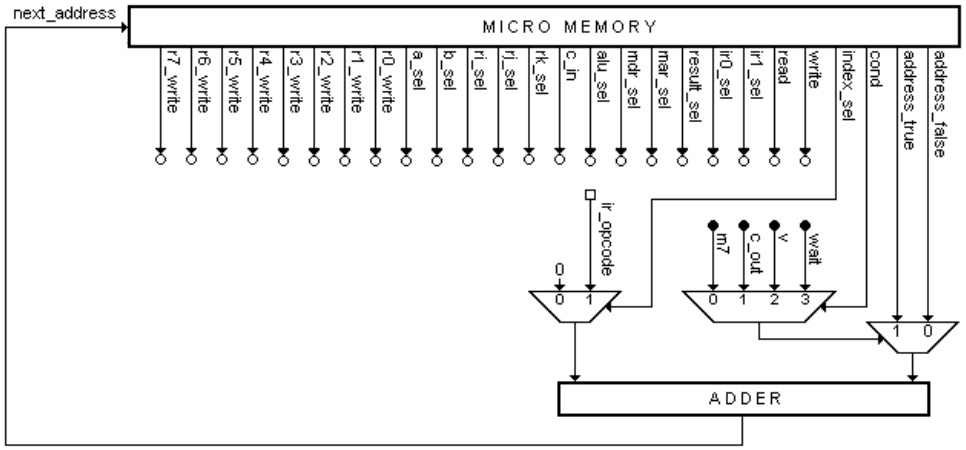
Neke instrukcije nemaju registarskih operanada (npr. instrukcija JMP adresa), pa je kod takvih instrukcija moguće svih 8 nižih bitova (bitovi 7 – 0, odnosno polje ir_const8) koristiti kao 8-bitnu konstantu. Ako instrukcija nema potrebe ni za konstantom ni za registarskim operandima (npr. instrukcija NOP), ovi bitovi mogu ostati neiskorišteni.

4 Mikroprogramirana upravljačka jedinica

Razmotrimo, konačno, upravljačku jedinicu procesora. Središnja komponenta mikroprogramirane upravljačke jedinice je mikroprogramska (ili upravljačka) memorija, koja sadrži mikroinstrukcije (ta memorija nalazi se u samom procesoru, u njegovoj upravljačkoj jedinici, i nema nikakve veze s glavnom memorijom računala). Kao što je već rečeno, svakoj strojnoj instrukciji odgovara jedan niz mikroinstrukcija u mikroprogramskoj memoriji – mikroprogram. Adresa početka mikroprograma jednoznačno je određena na temelju operacijskog koda makroinstrukcije.

Svaka mikroinstrukcija pak nije ništa drugo nego jedan dugački niz bitova, grupiranih u određena polja, pri čemu svako polje upravlja jednim upravljačkim signalom ili grupom upravljačkih signala i tako pobuđuje mikrooperacije u procesoru.

Na slici 2 prikazana je struktura mikroprogramirane upravljačke jedinice procesora koji se koristi u ovoj laboratorijskoj vježbi.



Slika 2: **Mikroprogramirana upravljačka jedinica**

Pri vrhu slike vidimo mikroprogramsku memoriju ("micro memory"). Kao i kod svake druge me-

morije, da bi se pročitao sadržaj određene memorijske lokacije, potrebno je specificirati adresu s koje želimo čitati (oznaka "next_address" na slici 2, gore lijevo). Adresiranjem mikroprogramske memorije, na njenim podatkovnim linijama pojavljuje se mikroinstrukcija pohranjena na toj adresi (još se naziva i mikroriječ). Ona se sastoji od većeg broja polja bitova, što na slici 2 odgovara strelicama s natpisima koje izlaze iz mikroprogramske memorije. Opišimo ukratko ulogu pojedinih grupa bitova u mikroinstrukciji, no radi lakšeg praćenja, opisivat ćemo ih malo drukčijim redoslijedom nego što se nalaze na slici 2.

Krenimo najprije od četiri krajnje desne grupe signala (četiri krajnje desne linije koje izlaze iz mikroprogramske memorije na slici 2). Ove četiri grupe bitova odnose se na određivanje adrese sljedeće mikroinstrukcije. Naime, za razliku od makroinstrukcija, koje se u pravilu izvode slijedno jedna za drugom (o čemu se brine programsko brojilo), kod mikroinstrukcija nije unaprijed osigurano slijedno izvođenje, nego svaka mikroinstrukcija eksplicitno mora specificirati adresu sljedeće mikroinstrukcije. Da bi se, međutim, omogućilo i uvjetno grananje unutar mikroprograma, u razmatranom modelu procesora mikroinstrukcija sadrži dva polja za adresu sljedeće mikroinstrukcije: address_true (adresa sljedeće mikroinstrukcije ako je uvjet ispunjen) i address_false (adresa sljedeće mikroinstrukcije ako uvjet nije ispunjen), te dvobitno polje cond koje služi za specifikaciju uvjeta o kojemu ovisi hoće li adresa sljedeće mikroinstrukcije biti određena jednim ili drugim adresnim poljem. Bezuvjetno grananje, koje nam je potrebno ćešće od uvjetnog, ostvaruje se tako da se oba polja address_true i address_false postave na istu vrijednost, uz proizvoljnu vrijednost polja cond. Sam uvjet može biti jedan od četiri moguća, kao što je vidljivo sa slike 2 (gdje se vidi da polje cond upravlja multipleksorom 4/1): može ovisiti o rezultatu aritmetičke ili logičke operacije (m7 – najviši bit rezultata je 1, odnosno rezultat je negativan; c_out – pri rezultatu se pojavio prijenos; v - pojavio se aritmetički preljev), te o stanju memorije (wait – memorijska operacija još nije završila). Konačno, posljednje polje za specifikaciju adrese sljedeće mikroinstrukcije je jednobitno polje index_sel, koje, kada je postavljeno, specificira da se adresa sljedeće mikroinstrukcije ne određuje na temelju polja address_true i address false, nego na temelju operacijskog koda makroinstrukcije u instrukcijskom registru.

Uloga preostalih signala i signalnih grupa mikroinstrukcije jasna je već iz slike 1 – oni naime izravno odgovaraju pojedinim upravljačkim signalima koji upravljaju odgovarajućim, već ranije opisanim elementima puta podataka:

- Signali rj_sel i rk_sel te dva trobitna polja (grupe signala) a_sel i b_sel specificiraju koji od registara će biti postavljeni na sabirnice operanada (a_bus i b_bus – ulazi u ALU, vidi sliku 1). Ove registre moguće je, naime, specificirati na dva načina:

  (i) Izravno iz mikroinstrukcije upravljačkim signalima a_sel i b_sel, te

  (ii) Iz makroinstrukcije (strojne instrukcije), na temelju polja ir_rj (bitovi) i ir_rk u instrukcijskom registru (ovim načinom moguće je specificirati samo registre r0 – r3, jer u makroinstrukciji imamo na raspolaganju samo po dva bita za specifikaciju registara).

  Koji od ovih dvaju načina će se koristiti pri izvođenju dane mikroinstrukcije, određeno je dodatnim upravljačkim bitovima rj_sel (za prvi operand) i rk_sel (za drugi operand). Ako su ovi bitovi u 0, koristi se način (i); ako su pak postavljeni u 1, koristi se način (ii). O ovoj funkcionalnosti brinu se dva multipleksora koja vidimo na samom vrhu slike 1, iznad skupa registara.

- Trobitna grupa signala alu_sel odabire jednu od osam mogućih funkcija ALU.

- Upravljački signal c_in postavlja u 0 ili 1 ulaz c_in u ALU.

- Dvobitni signal result_sel upravlja multipleksorom koji određuje što će se pojavitina sabirnici rezultata (result_bus) radi upisa u odredišni registar. Na slici 1 uočavamo da se o tome brine multipleksor 4/1 koji, ovisno o vrijednosti polja resutl_sel predviđa četiri mogućnosti što će se pojaviti na sabirnici rezultata: (a) rezultat aritmetičke ili logičke operacije iz ALU (koji se nalazi na sabirnici alu_bus), (b) sadržaj memorijskog podatkovnog registra (MDR), odnosno podatak pročitan iz memorije, (c) 4-bitna usputna konstanta iz instrukcijskog registra, predznačno proširena na 8 bita, te (d) 8-bitna usputna konstanta iz instrukcijskog registra.
- Signali ri_sel te r0_write – r7_write omogućuju upis podatka sa sabirnice rezultata u registre iz skupa r0 – r7. Ako je aktivan signal ri_sel, odredišni registar je određen na temelju dvobitnog polja ri u instrukcijskom registru, odnosno makroinstrukciji (na ovaj način ponovno je moguće pristupiti samo registrima r0 – r3), a ako je signal ri_sel neaktivan, odredišne registre određuju signali r0_write – r7_write. Pošto je riječ o osam nezavisnih signala koje je moguće aktivirati istovremeno i neovisno jedne o drugima, moguće je isti podatak sa sabirnice rezultata istovremeno upisati i u više registara (iako je to rijetko kada korisno).
- Signal mar_sel aktivira upis podataka sa sabirnice alu_bus (izlaz iz ALU) u memorijski adresni registar (MAR).
- Signal mdr_sel je dvobitni signal koji aktivira upis u memorijski podatkovni registar (MDR). Podatak koji se upisuje u MDR može biti ili podatak sa sabirnice alu_bus ili podatak s vanjske podatkovne sabirnice procesora (podatak iz memorije).
- Signali ir1_sel i ir0_sel aktiviraju upis podatka s vanjske podatkovne sabirnice (iz memorije) u viši, odnosno niži bajt instrukcijskog registra.
- Signali read i write upućuju se memoriji i aktiviraju čitanje, odnosno pisanje.

5 Pisanje mikroprograma

Mikroprogrami su nizovi mikroinstrukcija, pri čemu su pojedine mikroinstrukcije nizovi bitova u skladu s formatom opisanim u prethodnom poglavlju.

No da bi se mikroprogrameru olakšao posao, mikroprogramirani sustavi u pravilu ne zahtijevaju da se mikroprogrami pišu izravno na razini pojedinačnih bitova. Umjesto toga, većina sustava podržava unos mikroprograma u nekom simboličkom formatu, a sam sustav se brine o prevođenju takvog simboličkog formata u pojedinačne bitove mikroinstrukcije. Takav je slučaj i u simulatoru MythSim.

Mikroprogram za simulator MythSim piše se u obliku obične tekstualne datoteke (korištenjem bilo kojeg uređivača teksta – npr. notepad, gedit, vi, ...) u skladu sa sljedećim pravilima:

- Naziv datoteke ima ekstenziju .ucode.
- Komentari počinju oznakom // i protežu se do kraja retka.
- Svaka mikroinstrukcija može biti označena labelom, da bi se omogućilo grananje na tu naredbu. Labela se nalazi na početku retka i odvojena je od ostatka mikroinstrukcije dvotočkom. Npr:

```
labela: // ovdje slijedi nastavak mikroinstrukcije ...
```

- Mikroinstrukcije se međusobno odvajaju oznakom ; (točka-zarez).
- Pojedina mikroinstrukcija se sastoji od niza oznaka koje predstavljaju upravljačke signale. Oznake su međusobno odvojene zarezima. Ako je riječ o jednobitnim signalima, samo navođenje imena signala specificira njegovo postavljanje u 1, a izostavljanje tog imena u mikroinstrukciji označava da signal poprima vrijednost 0. Npr., ako se u liniji koja odgovara mikroinstrukciji nalazi oznaka c_in, znači da će signal c_in biti postavljen u 1. Ako nema oznake c_in u mikroinstrukciji, taj signal će biti u 0. Ako je riječ o višebitnim signalima, koristi se sintaksa `ime=vrijednost`, pri čemu se vrijednost može pisati u simboličkom obliku. Npr. `alu_sel=ADD` znači da će selekcijski ulazi aritmetičko-logičke jedinice biti postavljeni na trobitnu vrijednost koja odgovara operaciji zbrajanja (korisnik ne mora znati koja je to točno vrijednost, nego koristi simbolički naziv ADD). Primjer jedne mikroinstrukcije:

  ```
  labela: ri_sel, rj_sel, rk_sel, alu_sel=ADD;
  ```

  To znači: signali ri_sel, rj_sel i rk_sel će biti postavljeni u 1, a signal alu_sel u vrijednost koja odgovara operaciji zbrajanja. Svi ostali (jednobitni i višebitni) signali koji nisu navedeni u mikroinstrukciji poprimit će vrijednost 0. Npr. ako je signal result_sel izostavljen, pretpostavlja se `result_sel=ALU`. **Popis svih upravljačkih polja u mikroinstrukciji, njihovih značenja i vrijednosti koje mogu poprimiti dan je u tabličnom obliku u dodatku na kraju ovih uputa.**
- Za specificiranje adrese sljedeće mikroinstrukcije koristi se simbolička oznaka "goto *labela*". Ako ovakve oznake u mikroinstrukciji nema, simulator podrazumijeva da je sljedeća mikroinstrukcija ona koja se nalazi neposredno iza tekuće u tekstualnoj datoteci.
- Za ostvarivanje uvjetnog grananja koristi se sintaksa

  ```
  if signal_uvjeta then goto labela endif; ili
  if signal_uvjeta then goto labela1 else goto labela2 endif;
  ```

- Dopuštene su prazne linije između mikroinstrukcija.

Tipični mikroprogram sastojat će se od tri dijela, kao što ćemo pokazati na primjeru malo niže:

1. Dio koji se odnosi na fazu "pribavi" nalazi se uvijek na početku .ucode datoteke.
2. Slijedi dio koji se odnosi na operacijske kodove pojedinih instrukcija ("opcode" dio), pri čemu svakoj instrukciji (svakom operacijskom kodu) odgovara točno jedna linija (tj. jedna mikroinstrukcija) u ovom dijelu datoteke.
3. Ako mikroprogram za neku strojnu instrukciju zahtijeva više od jedne mikroinstrukcije, ostale mikroinstrukcije (osim prve) moraju se nalaziti ispod "opcode" dijela koja sadrži po jednu liniju za svaki operacijski kod. Ovaj treći dio naziva se "opcode extension" dio. Prva mikroinstrukcija (koja se nalazi u dijelu (2), tj. "opcode" dijelu) mora u tom slučaju sadržavati "goto" na odgovarajuću labelu gdje se nalazi nastavak mikroprograma.

To da nastavci mikroprograma moraju biti ispod svih prvih mikroinstrukcija, proizlazi iz arhitekture mikroprorgamirane upravljačke jedinice, odnosno načina određivanja adrese početka mikroprograma na temelju operacijskog koda makroinstrukcije. To vidimo iz slike 2: ako je aktivan signal index_sel, operacijski kod smješten u instrukcijskom registru se pribraja adresi sljedeće mikroinstrukcije. To znači da se prva mikroinstrukcija za strojnu naredbu s operacijskim kodom 0 mora

nalaziti neposredno iza zadnje mikroinstrukcije za fazu pribavi, prva mikroinstrukcija za strojnu naredbu s operacijskim kodom 1 točno jednu adresu iza mikroinstrukcije za strojnu naredbu s operacijskim kodom 0 itd..

Primjer mikroprograma:

```
// ====================== PRIBAVI ======================

fetch0: a_sel=7, b_sel=7, alu_sel=OR, mar_sel=LOAD;       // MAR <- PC
fetch1: ir1_sel=LOAD, read, if wait then goto fetch1 endif; // IR_high <- MEM(MAR)
fetch2: a_sel=7, c_in, alu_sel=ADDA, r7_write;            // PC <- PC+1
fetch3: a_sel=7, b_sel=7, alu_sel=OR, mar_sel=LOAD;       // MAR <- PC
fetch4: ir0_sel=LOAD, read, if wait then goto fetch4 endif; // IR_low <- MEM(MAR)
fetch5: a_sel=7, c_in, alu_sel=ADDA, r7_write, goto opcode[IR_OPCODE]; // PC <- PC+1

// ============= DIO OPERACIJSKIH KODOVA =============

// 0) NOP
opcode[0]: goto fetch0;

// 1) LOAD_IMMEDIATE (ri <- ir_const8)
opcode[1]: result_sel=IR_CONST8, ri_sel, goto fetch0;

// 2) ADD (ri <- rj + rk)
opcode[2]: ri_sel, rj_sel, rk_sel, alu_sel=ADD,
       if m_7 then goto opcode2.1 else goto opcode 2.2 endif;

// 3) HALT
opcode[3]: goto opcode[3];

// ================ DIO EKSTENZIJE ================
// postavi zastavicu N
opcode2.1: a_sel=4, b_sel=4, alu_sel=XOR, r4_write; // pomocni registar r4 <- 0
       a_sel=4, c_in, alu_sel=ADDA, r6_write, goto fetch0; // r4=0 + c_in=1 -> r6 (SR)

// obrisi zastavicu N
opcode2.2: a_sel=4, b_sel=4; alu_sel=XOR, r4_write; // pomocni registar r4 <- 0
       a_sel=4, alu_sel=ADDA, r6_write, goto fetch0; // r4=0 -> r6 (SR)
```

Prvih šest mikroinstrukcija (označenih labelama fetch0 – fetch5) odgovara fazi "pribavi". Prve tri mikroinstrukcije čitaju prvi (viši) bajt makroinstrukcije u instrukcijski registar, dok druge tri čitaju niži bajt (ovdje pretpostavljamo da procesor koristi big endian način zapisivanja podataka; primjeri koji dolaze uz simulator pretpostavljaju little endian način).

Promotrimo pobliže mikrokod za fazu "pribavi".

Prva mikroinstrukcija (fetch0) prebacuje sadržaj programskog brojila u memorijski adresni registar. Kako procesor nema posebnog registra za programsko brojilo, mi kao programsko brojilo koristimo registar r7. No s obzirom na strukturu puta podataka, sadržaj bilo kojeg registra mora najprije proći kroz ALU, kako bi završio na sabirnici alu_bus, prije nego što ga je moguće upisati u MAR. Da bi sadržaj registra r7 pri tom prolasku kroz ALU ostao nepromijenjen, mi koristimo jednostavan "trik": specificiramo operaciju OR (logičko ILI) i dovodimo na oba ulaza ALU sadržaj registra r7. Logičko ili nekog binarnog broja samog sa sobom daje isti taj broj, pa će se na izlazu iz ALU pojaviti nepromijenjen sadržaj registra r7 (alternativno, mogli smo odabrati i operaciju AND, ili čak ADDA uz c_in=0, odnosno SUBA uz c_in=1; rezulat bi bio isti).

Druga mikroinstrukcija (fetch1) aktivira čitanje iz memorije (signal read) i upisuje pročitani podatak u viši bajt instrukcijskog registra (ir1_sel=LOAD). Sama operacija čitanja može zahtijevati više vremena, jer je memorija tipično sporija od procesora. Zbog toga druga mikroinstrukcija ostvaruje petlju: ona provjerava stanje signala wait kojim memorija signalizira je li operacija pristupa memoriji završena ili još traje. Ako memorija još nije završila operaciju (signal wait je aktivan), mikroinstrukcija skače sama na sebe i tako se vrti u petlji sve dok pristup memoriji ne završi. Kao što je napomenuto na kraju odjeljka 2, signal wait se aktivira tijekom istog perioda takta, pa procesor neće prijeći na treću mikroinstrukciju prije nego što memorija stvarno dojavi da je postavila vrijednost na sabirnicu.

Treća mikroinstrukcija (fetch2) uvećava programsko brojilo (tj. registar r7) za 1, kako bi bio spreman za adresiranje sljedećeg bajta strojne instrukcije. U tu svrhu na jedan ulaz ALU (a_bus) postavlja se sadržaj registra r7 i aktivira se operacija ADDA uz c_in=1. Svejedno je što se nalazi na drugom ulazu ALU (b_bus), jer operacija ADDA djeluje samo nad jednim operandom. Tako uvećani sadržaj sprema se nazad u programsko brojilo (r7_write).

Preostale tri mikroinstrukcije za fazu "pribavi" (fetch3 – fetch5) su identične strukture kao i prve tri, samo s tom razlikom da učitavaju drugi bajt instrukcije u niži bajt instrukcijskog registra (ir0_sel=LOAD). Jedina značajnija razlika nalazi se na samom kraju posljednje instrukcije faze pribavi, a sastoji se u tome da ova posljednja naredba ima dio "goto opcode[IR_OPCODE]". To zapravo znači da će tijekom izvođenja te mikroinstrukcije biti aktivan signal index_sel, koji će daljnje izvođenje mikroprograma usmjeriti ovisno o operacijskom kodu pročitane makroinstrukcije.

Preostale mikroinstrukcije odgovaraju fazi "izvrši" četiriju strojnih instrukcija (makroinstrukcija): NOP, LOAD_IMMEDIATE, ADD i HALT. Tri od ove četiri strojne instrukcije imaju samo jednu mikroinstrukciju; jedino instrukcija ADD ima nastavak u dijelu ekstenzije (ovaj nastavak odnosi se samo na postavljanje zastavice N – najnižeg bita u registru r6, koji koristimo kao statusni registar). Objašnjenja pojedinih mikroinstrukcija dana su u komentarima uz same mikroinstrukcije, iz čega bi trebala biti jasna njihova funkcionalnost. Zadnja mikroinstrukcija svakog mikroprograma za fazu "izvrši" sadrži oznaku goto fetch0, čime se osigurava ponovni prelazak procesora u fazu "pribavi" nakon dovršetka faze "izvrši" za trenutnu strojnu instrukciju.

6 Pisanje makroprograma

Da bi se testirala funkcionalnost mikroprograma, potrebno je napisati i odgovarajući makroprogram, odnosno program koji se sastoji od strojnih instrukcija (makroinstrukcija) čiju smo funkcionalnost prethodno ostvarili mikroprogramiranjem. U simulatoru MythSim, makroprogrami se također pišu u obliku običnih tekstualnih datoteka s kodiranjem ASCII (nepodržano kodiranje uzrokuje pogrešku kod parsiranja u MythSimu), u skladu sa sljedećim pravilima:

- Ime datoteke mora imati nastavak .mem (simulator takve datoteke zove "memory files").
- Početni dio datoteke, prije oznake %, može se koristiti za komentare
- Nakon oznake %, slijedi numerički sadržaj memorije koji odgovara makroprogramu; komentari se mogu pisati tako da započinju oznakom // i protežu se do kraja retka.
- Linije koje sadrže kod **moraju** biti označene slijednim brojevima, počevši od 0, koji se tumače kao adrese memorijskih lokacija, iza kojih slijedi dvotočka (:) i sadržaj memorijske lokacije.

- Sami sadržaji memorijskih lokacija (koji odgovaraju instrukcijama makroprograma u numeričkom obliku) mogu se pisati dekadski ili binarno. Ako se pišu binarno, moguće je grupe bitova odvajati prazninama radi lakše čitljivosti (npr. 6 bitova operacijskog koda odvojiti prazninom od dvobitnog polja za specifikaciju odredišnog registra, ili pak dva dvobitna polja za registre operanada odvojiti međusobno razmacima i od 4-bitne konstante u nižem bajtu instrukcijskog registra).
- Instrukcije su tipično 16-bitne (16-bitni instrukcijski registar) no memorija je bajtno adresirljiva; stoga će jedna instrukcija u memoriji biti predočena s dva bajta na slijednim adresama.
- Dopuštene su prazne linije.

Napomena: Ako linija sadrži samo komentar (bez adrese i sadržaja memorijske lokacije prije njega), ne smije sadržavati dvotočku. Inače će doći do pogreške jer će MythSim dio komentara prije dvotočke pokušati parsirati kao adresu.

Primjer makroprograma:

```
Pocetni dio datoteke s makroprogramom, prije znaka %
moze se koristiti za globalne komentare makroprograma.
Pribroji neposredne konstante registru r0.

%             // Pocetak koda
// LOAD_IMMEDIATE r0<- 3
0: 000001 00  // vrijednosti mogu biti binarne (s razmacima)
1: 3          // ili dekadske

// LOAD_IMMEDIATE r1<- 2
2: 000001 01
3: 2

// ADD r0 <- r0 + r1
4: 000010 00
5: 00 01 0000         // bin. vrijednosti mogu sadrzavati razmake

// HALT
6: 000011 00          // U r0 bi se sada trebalo nalaziti 5
7: 0

// Kraj
```

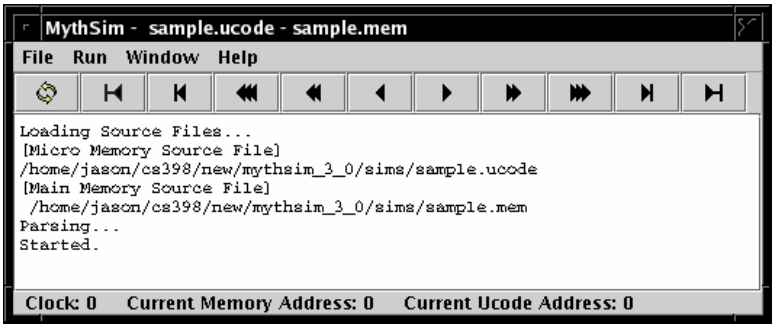
Na adresi 0 nalazi se prvi bajt prve instrukcije. On se sastoji od 6-bitnog operacijskog koda (000001 = 1) i 2-bitne adrese odredišnog registra (00, pošto se radi o registru r0), pa su radi lakše čitljivosti ova dva dijela odvojena razmakom. Drugi bajt (na adresi 1) predstavlja neposrednu konstantu (vrijednosti 3) koja je zapisana dekadski. Na sličan način moguće je, čitajući i komentare, protumačiti i sadržaj ostatka datoteke, što ostavljamo studentu za vježbu.

7 Korištenje simulatora

Simulator MythSim napisan je u programskom jeziku Java i može se pokrenuti na svim raširenijim operacijskim sustavima; potrebno je samo imati instaliran Java Virtual Machine verzije 1.4 ili više.

Simulator se može dobaviti u obliku .zip datoteke, sa stranica navedenih u Uvodu. Nakon pohranjivanja, potrebno je dobivenu .zip datoteku raspakirati u neki prikadan direktorij, čime nastaje poddirektorij mythsim-3.1.1, koji sadrži datoteku mythsim-3.1.1.jar (ovo je sam program, koji možemo pokrenuti ako imamo instaliran Java Virtual Machine), te poddirektorij docs s dokumentacijom i primjerima.

Pokretanjem programa mythsim-3.1.1.jar otvara se glavni prozor simulatora, koji je vrlo jednostavan (slika 3).



Slika 3: **Osnovni prozor simulatora**

Da bi simulator postao funkcionalan, potrebno je najprije učitati dvije datoteke koje smo ranije pripremili u proizvoljnom editoru teksta: datoteku s makroprogramom (mem) i datoteku s mikroprogramom (ucode). Ove datoteke se otvaraju odabirom opcije File → Open u izborniku.

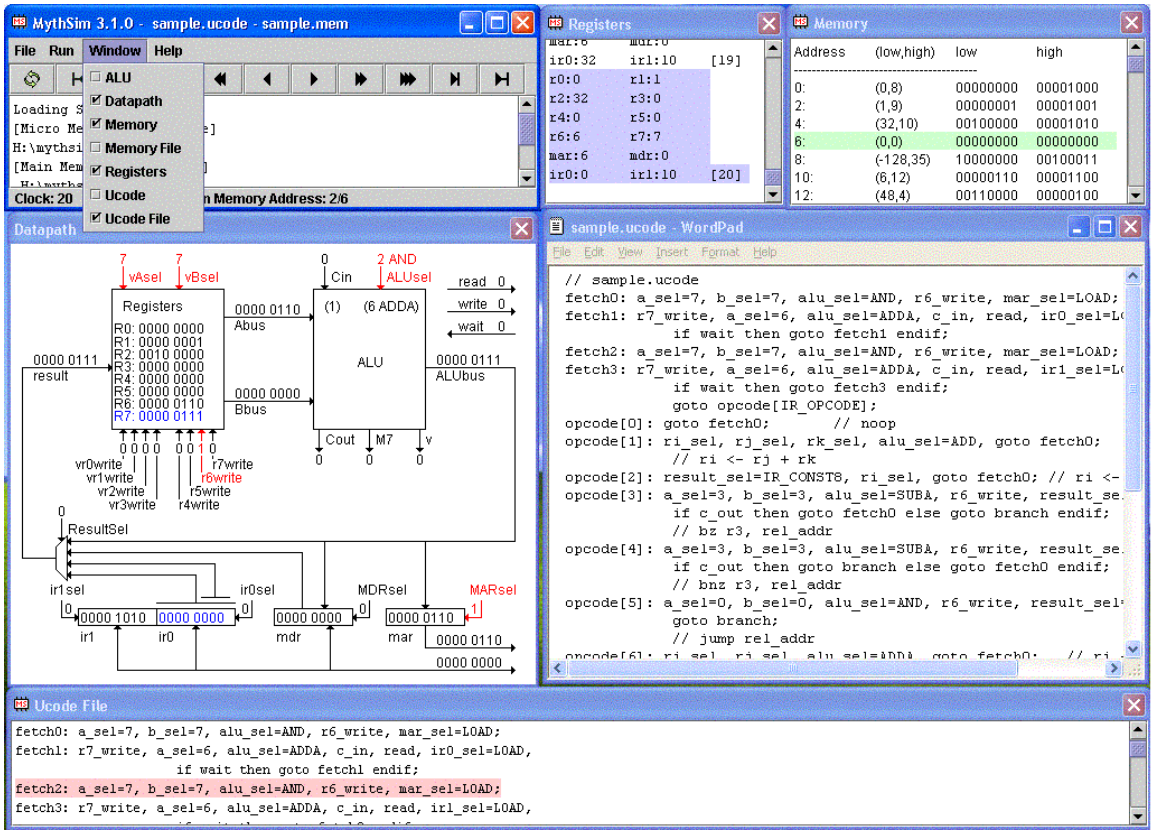
Nakon što su makroprogram i mikroprogram učitani, moguće je pratiti njihovo izvršavanje mikroinstrukciju po mikroinstrukciju, ili u grubljim koracima. No posljedice izvođenja pojedinih mikroinstrukcija nisu vidljive u osnovnom prozoru mikroprograma. Da bismo pratili utjecaj izvršavanja pojedinih mikroinstrukcija na pojedine komponente računala (registre, ALU, memoriju, ...) potrebno je aktivirati prikaz ovih komponenata u posebnim prozorima. To se postiže aktiviranjem opcija unutar izbornika Window u glavnom prozoru. Moguće je u posebnim prozorima prikazati:

- stanje aritmetičko-logičke jedinice (prozor ALU),
- grafički prikaz trenutnog stanja puta podataka (prozor Datapath),
- prikaz sadržaja glavne memorije (prozor Memory),
- prikaz tekstualne datoteke sa sadržajem glavne memorije (prozor Memory File),
- prikaz trenutnog i ranijeg sadržaja registara (prozor Registers),
- prikaz mikroprogramske memorije (prozor Ucode),
- prikaz tekstualne datoteke s mikroprogramom (prozor Ucode File).

Izgled simulatora s otvorenim glavnim prozorom te većim brojem otvorenih prozora za prikaz pojedinih komponenata može se vidjeti na slici 4.

Iz glavog prozora sada možemo izvoditi mikroinstrukciju po mikroinstrukciju (tipka s jednim trokutićem usmjerenim u desno), vraćati se unazad za po jednu mikroinstrukciju (tipka s jednim trokutićem usmjerenim u lijevo), izvoditi po jednu makroinstrukciju (krajnje desna tipka s trokutićem u desno i crtom) te vraćati se unazad za jednu makroinstrukciju (trokutić u lijevo i crta).

Tu je također i tipka za resetiranje simulacije, odnosno vraćanje procesora u početno stanje u kojem je bio prije pokretanja programa (krajnje lijeva tipka s dvije kružno usmjerene strelice), te tipke koje omogućuju izvođenje 10, odnosno 100 mikroinstrukcija unaprijed ili unatrag u jednom koraku (dva, odnosno tri trokutića usmjerena desno, odnosno lijevo).



Slika 4: **Izgled simulatora: glavni prozor s ostalim prozorima koji su odabrani u izborniku**

8 Zadaci

Korištenjem simulatora MythSim realizirajte procesor koji, osim četiriju makroinstrukcija prikazanih u poglavlju 5, podržava još i sljedeće makroinstrukcije:

Instrukcija	**Opis**
`MOVE ri, rj`	prijenos sadržaja između dvaju registara: ri ← rj
`LOAD ri, addr`	punjenje registra ri sadržajem izravno zadane memorijske adrese: ri ← MEM(addr)
`STORE rj, (rk)`	spremanje sadržaja registra rj na memorijsku lokaciju čija je adresa u rk: MEM(rk) ← rj
`JMP addr`	grananje na apsolutno zadanu adresu: PC ← addr
`JZ rj, rk`	uvjetno grananje: ako je rj=0 tada PC ← rk
`SUB ri, rj, rk`	oduzimanje: ri ← rj − rk
`SHL ri, rj`	posmak sadržaja registra rj u lijevo za 1 mjesto i spremanje rezultata u registar ri
`LDSP konst`	inicijalizacija kazala stoga (r5) na vrijednost konst
`PUSH rj`	stavljanje sadržaja registra rj na stog
`POP ri`	skidanje podatka sa stoga u registar ri
`CALL addr`	poziv potprograma na adresi addr
`RET`	povratak iz potprograma

Potrebno je napisati mikroprogram (ucode datoteku) koji ostvaruje navadene instrukcije te makroprogram (mem datoteku) koji ispituje ispravnost ostvarenih instrukcija.

Kao programsko brojilo (PC) koristiti registar r7. Kao statusni registar (SR) koristiti registar r6. Kao kazalo stoga (SP) koristiti registar r5. Registar r4 može se koristiti kao pomoćni privremeni registar, dok registri r0 – r3 trebaju biti registri opće namjene, dostupni iz makroinstrukcija.

9 MythSim Microcode Language Quick Reference

Register Set		
`a_sel=n`		Place the value in register n (0-7) on the **a_bus**.
`b_sel=n`		Place the value in register n (0-7) on the **b_bus**.
`ri_sel`		Store result in the register defined in **ir_ri**.
`rj_sel`		Place the value in the register defined by **ir_rj** on the **a_bus**.
`rk_sel`		Place the value in the register defined by **ir_rk** on the **b_bus**.
`rn_write`		Store the **result_bus** value into register n (0-7).
Arithmetic Logic Unit		
`alu_sel=`	**NOT**	**NOT(a_bus)** → **alu_bus**
	OR	**OR(a_bus, b_bus)** → **alu_bus**
	AND	**AND(a_bus, b_bus)** → **alu_bus**
	XOR	**XOR(a_bus, b_bus)** → **alu_bus**
	ADD	**a_bus + b_bus + c_in** → **alu_bus**
	SUB	**a_bus + NOT(b_bus) + c_in** → **alu_bus**
	ADDA	**a_bus + c_in** → **alu_bus**
	SUBA	**a_bus - 1 + c_in** → **alu_bus**
`c_in`		Set the **c_in** to 1 for use by ALU operations (4-7).
Memory Interface		
`result_sel=`	**ALU**	Place the **alu_bus** value on the **result_bus**.
	MDR	Place the mdr value on the **result_bus**.
	IR_CONST4	Place **ir_const4** on the **result_bus**. (**ir_const4** = last 4 bits of ir0)
	IR_CONST8	Place **ir_const8** on the **result_bus**. (**ir_const8** = all 8 bits of ir0)
`ir0_sel=`	**LOAD**	Load the value on the **memory_bus** into ir0 (Instruction Register 0).
`ir1_sel=`	**LOAD**	Load the value on the **memory_bus** into ir1 (Instruction Register 1).
`mar_sel=`	**LOAD**	Load the value on the **alu_bus** into mar (Memory Address Register).
`mdr_sel=`	**LOAD_ALU**	Load the value on the **alu_bus** into mdr (Memory Data Register).
	LOAD_MEM	Load the value on the **memory_bus** into mdr (Memory Data Register).
`read`		Place the value at the main memory address mar on the **memory_bus**.
`write`		Write the value of mdr to the main memory location mar.
Control Structures		
`goto label`		Jump to the given label
`if status_line then goto label_1 (else goto label_2) endif`		If the value of status_line=1 then goto label_1 else goto label_2

status_line	Description
m_7	most significant bit from **alu_bus**
c_out	carry bit from the **alu_bus**
v	overflow bit from the ALU
wait	wait signal from Main Memory