Verifikacijsko okruženje za oblikovanje računalnog sustava temeljenog na procesoru arhitekture RISC-V

Đurić, Vinko

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1643

VERIFIKACIJSKO OKRUŽENJE ZA OBLIKOVANJE RAČUNALNOG SUSTAVA TEMELJENOG NA PROCESORU ARHITEKTURE RISC-V

Vinko Đurić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1643

VERIFIKACIJSKO OKRUŽENJE ZA OBLIKOVANJE RAČUNALNOG SUSTAVA TEMELJENOG NA PROCESORU ARHITEKTURE RISC-V

Vinko Đurić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1643

Pristupnik:	**Vinko Đurić (0036542970)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	prof. dr. sc. Josip Knezović

Zadatak: **Verifikacijsko okruženje za oblikovanje računalnog sustava temeljenog na procesoru arhitekture RISC-V**

Opis zadatka:

Proučiti postojeća rješenja za provjeru ispravnosti procesora arhitekture RISC-V te razviti vlastito verifikacijsko okruženje za izvođenje asemblerskih programa i usporedbu s referentnim modelom sustava iz programskog paketa SSPARCSS. Programski paket SSPARCSS je razvijen na FER-u a služi za poučavanje osnovnih koncepata arhitekture procesora i računalnih sustava. SSPARCSS omogućuje simulaciju i izvođenje programa za različite arhitekture procesora, uključujući i arhitekturu RISC-V na koju se fokusira ovaj zadatak. Nakon provjere ispravnosti razvijenog verifikacijskog okruženja na osnovu usporedbe izvođenja programa u odnosu na SSPARCSS, potrebno je definirati i razviti niz ispitnih programa i osnovni ponašajni model procesora arhitekture RISC-V koji će biti podvrgnut ispitnim programima u razvijenom verifikacijskom okruženu. Po potrebi napraviti i prilagodbe referentnog modela u SSPARCSS-u s ciljem učinkovite usporedbe s razvijenim verifikacijskim okruženjem.

Rok za predaju rada: 14. lipnja 2024.

Zahvaljujem se mentoru prof. dr. sc. Josipu Knezoviću na pomoći pri izradi rada.

Sadržaj

1. Uvod

Instrukcijski skup RISC-V postaje sve popularniji odabir kod projektiranja procesora i mikrokontrolera. Njegova otvorenost olakšava razvoj novih računalnih sustava za manje kompanije. Upravo zbog toga, sustavi koji koriste instrukcijski skup RISC-V čest su odabir prilikom projektiranja sustava za internet stvari (eng. *Internet of Things*, *IoT*). Osim otvorenosti, još jedna važna značajka instrukcijskog skupa RISC-V je jednostavnost osnovnih instrukcija. Zbog toga se arhitektura skupa instrukcija RISC-V koristi za edukaciju o arhitekturi računala u sustavima visokog obrazovanja.

Osnovni skup instrukcija RISC-V (eng. *Base*) je zamrznut, odnosno sigurno se neće mijenjati u budućnosti što osigurava dugoročnu uporabivost sustava koji implementiraju navedeni osnovni skup instrukcija .

Arhitektura RISC-V osim osnovnih instrukcija nudi sve veći broj dodatnih instrukcija za specifične potrebe kao što su instrukcije za kriptografiju, vektorske procesore, bitovnu manipulaciju, atomarne operacije i sl. Ove instrukcije se obično uvode kao tzv. ekstenzije, koje mogu biti specifične ili potvrđene kao službeni skup proširenja.

Ovaj rad je dio većeg projekta FERV koji se provodi na Fakultetu elektrotehnike i računarstva Sveučilšta u Zagrebu. Cilj projekta je razvoj računalnog sustava temeljenog na procesoru arhitekture RISC-V. Oblikovani sustav koristio bi se u edukacijske svrhe, točnije za povećanje kvalitete predmeta *Arhitektura računala 1R* na Fakultetu elektrotehnike i računarstva. Sustav uključuje i proširenja na postojećem programskom paketu SSPARCSS koji uključuje simulator i zbirni prevoditelj. Zbirni prevoditelj (eng. *assembler*) služi za prevođenje zbirnih programa za RISC-V, dok simulator služi za pokretanje i simuliranje napisanog programa. Računalni sustav u simulatoru uključuje procesor arhitekture RISC-V te nekoliko vanjskih jedinica, s naglaskom na proširivost i modu-

larnost cijelog sustava. Takav sustav u cijelosti bi se preslikao na razvojnu pločicu te bi se na taj način omogućilo studentima isprobavanje njihovih asemblerskih programa na stvarnom sklopovlju pored simulacije u simulatoru programskog paketa SSPARCSS. Osim toga, sustav predviđa i razvoj automatiziranog ispitnog okruženja koje bi olakšalo izvođenje nastave jer bi se programi studenata u formi zadataka mogli automatizirano i brzo ispraviti.

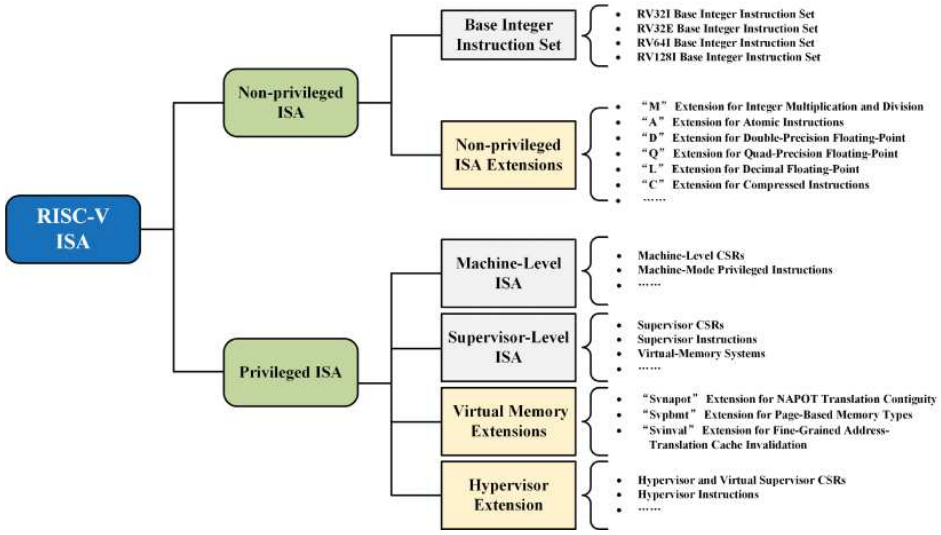
Cilj ovog rada je razvoj ispitnog okruženja za cijeli sustav kojim bi se ispitivala ispravnost razvijenog modela procesora i implementacija instrukcijskog skupa što uključuje sustav za evaluaciju i ispitne primjere s pomoću kojih će se ispitivati ispravnost rada. Osim toga, cilj rada je i nadogradnja postojećeg simulatora te modela računalnog sustava s procesorom arhitekture RISC-V.

2. Arhitektura RISC-V

RISC-V je otvoreni instrukcijski skup (eng. *instruction set architecture, ISA*) koji spada u RISC (eng. *reduced instruction set computer*) arhitekture, odnosno osnovni skup instrukcija uključuje mali broj instrukcija. Procesori koji implementiraju instrukcijski skup RISC-V spadaju u tzv. *load-store* arhitekturu [1]. Ta arhitektura memoriji pristupa samo instrukcijama za pristup memoriji (punjenje registra i spremanje registra), dok ostale instrukcije rade nad registrima i neposrednim vrijednostima. RISC-V osim osnovnog skupa instrukcija broji poveći broj dodatnih skupova instrukcija što omogućuje modularno korištenje pojedinih instrukcijskih skupova za različite domene upotrebe procesora [2]. Instrukcijske skupove možemo podijeliti u dvije velike kategorije: privilegirane i neprivilegirane. Neprivilegirani skupovi instrukcija uključuju instrukcije koji se koriste za uobičajene programe opće namjene, dok skupovi koji pripadaju privilegiranim mogu pristupati privilegiranim dijelovima sustava poput kontrolnih i statusnih registara, upravljati memorijom i sl. Slika 2.1. prikazuje pregled instrukcijskih skupova. RISC-V sve je popularniji te domene njegove primjene su sve šire iako je još uvijek u razvoju [3].

2.1. Instrukcijski skup RV32I

RV32I je osnovni instrukcijski skup kojeg bi trebao podržavati bilo koji procesor arhitekture RISC-V. On uključuje instrukcije koje koriste osnovni skup registara poput aritmetičko-logičkih instrukcija te instrukcija za promjenu tijeka izvođenja programa, odnosno instrukcija skoka, te memorijskih instrukcija. Instrukcije se odnose na 32-bitne registre i 32-bitne neposredne vrijednosti. Sve instrukcije interpretiraju memorijske vrijednosti i sadržaje registara kao cjelobrojne vrijednosti.



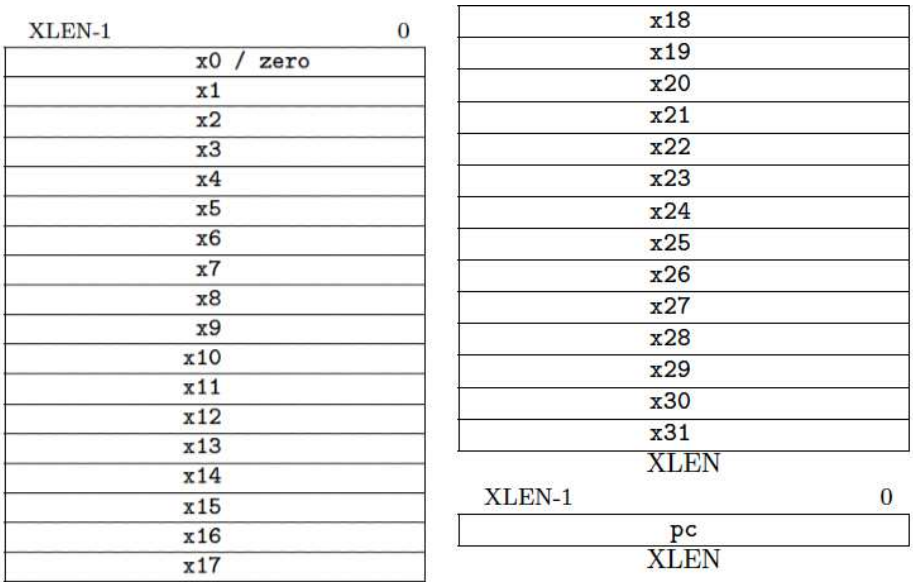
Slika 2.1. Pregled instrukcijskih skupova arhitekture RISC-V [2]

2.1.1. Registri opće namjene

Registri kao i u svim ostalim arhitekturama služe za pohranu privremenih vrijednosti unutar samog procesora, odn. puta podataka. Zbog toga, dohvat vrijednosti iz registra (čitanje registra) i spremanje vrijednosti u registre (pisanje u registar) su najbrže u odnosnu na sve ostale pristupe podacima (npr. čitanje i pisanje memorijske lokacije) Njihova brzina dolazi s nedostatkom zauzimanja nemale količine površine silicija te je stoga njihov broj ograničen.

Specifikacija arhitekture RISC-V definira 32 registra opće namjere. Slika 2.2. prikazuje registre opće namjene i programsko brojilo. Važno je uočiti da programsko brojilo nije dio standardnih registara te mu se ne može pristupiti izravno niti mu se može izravno mijenjati vrijednost. Svi registri opće namjene su 32-bitni (parametar XLEN za 32-bitni skup instrukcija RV32I je 32, vidi sliku 2.2.).

Osim prikazanog oblika imena koje se sastoji od slova 'x' i broja registra (od x0 do x31), registri se mogu adresirati i s pomoću drugog, alternativnog imena. Svaki registar u pravilu ima jedno alternativno ime. Alternativna imena posebno su korisna jer iako se svi registri mogu koristiti jednako, specifikacija definira funkciju koju bi svaki od registara trebao imati. To je također korisno pri čitanju asemblerskog koda. Tako je primjerice alternativno ime registra `x1` `ra` (*return address*), odnosno registar služi za spremanje povratne adrese. Alternativno ime `x2` je `sp` (*stack pointer*), odnosno registar obavlja funkciju pokazivača stoga. Neki od registara se koriste kao privremeni, kao



Slika 2.2. Skup registara opće namjene za instrukcijski skup RV32I [4]

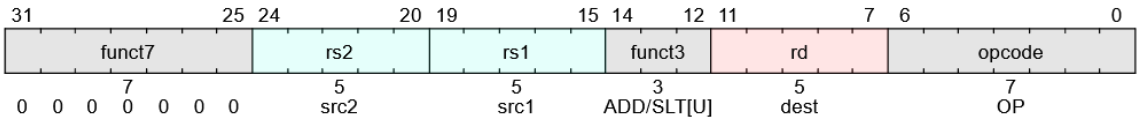
registri za prijenos parametara u potprogramima i sl.

2.1.2. Tipovi instrukcija

Sve instrukcije u skupu RV32I mogu se pridružiti jednom od 6 formata za kodiranje instrukcija: R, I, S, B, U, J. Sve instrukcije u ovom skupu su duljine 32 bita. U nastavku je ukratko opisan svaki od formata.

Format R

Format instrukcija R koristi se za aritmetičko-logičke naredbe kao što su zbrajanje, oduzimanje, logičke operacije I, ILI, posmaci i sl. Takve instrukcije kao operande imaju samo registre. Slika 2.3. prikazuje kako se odgovarajući mnemonik prevodi u strojni kod instrukcije.



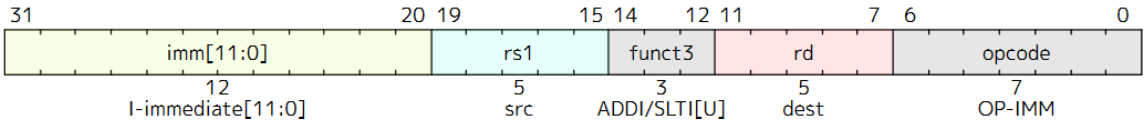
Slika 2.3. Format R instrukcija [4]

Sve instrukcije koje se prevode u ovom formatu imaju jednak operacijski kod pa se pojedina instrukcija prepoznaje prema bitovima u poljima `funct3`, ali i `funct7`. Bitovi ond. polja `rs2` i `rs1` služe za kodiranje izvornih operanada, odnosno registara opće namjene, dok je `rd` niz bitova kojim se kodira odredišni registar u koji se sprema rezultat

izvođenja instrukcije.

Format I

Instrukcije formata I su prema funkcionalnosti slične onima opisanima za format R, uz iznimku da je jedan od operanada neposredna vrijednost (eng. *immediate*) umjesto registra opće namjene. Uz to, instrukcije koje dohvaćaju podatke iz memorije (eng. *load*) također se kodiraju kao tip I.



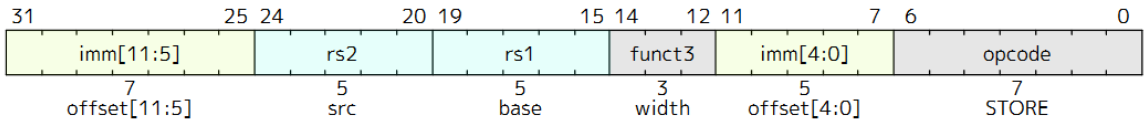
Slika 2.4. Format I instrukcija [4]

Slično kao i za tip R instrukcija, sve aritmetičko-logičke naredbe tipa I imaju isti `opcode`, a pojedine instrukcije razlikuju se prema vrijednosti polja `funct3`, a slično vrijedi i za sve naredbe za dohvat iz memorije, tip instrukcije (dohvat riječi, poluriječi s ili bez predznaka itd.) također se prepoznaje prema vrijednosti polja `funct3`. Za razliku od formata R, u ovom formatu najviši bitovi memorijske riječi su neposredna vrijednost veličine 12 bita koju korisnik ili prevoditelj može proizvoljno odabrati, dok je izostavljeno polje `rs2`.

Format S

Instrukcije formata S služe za spremanje podataka iz nekog od registra opće namjene u memoriju (eng. *store*). Slika 2.5. prikazuje kako se kodiraju instrukcije ovog tipa.

Ovaj format sličan je ranije opisanim formatima I i R. Za razliku od formata I, neposredna vrijednost je drugačije kodirana, a za razliku od formata R, ovaj tip instrukcija koristi dva registra opće namjene označena poljima `rs1` i `rs2` te s obzirom na to da je odredište instrukcija ovog tipa memorija, bitovi 7 do 11 su dio neposredne vrijednosti umjesto odredišnog registra (`rd`). Također, viši bitovi neposredne vrijednosti postavljeni su kao najviši bitovi instrukcijske riječi.

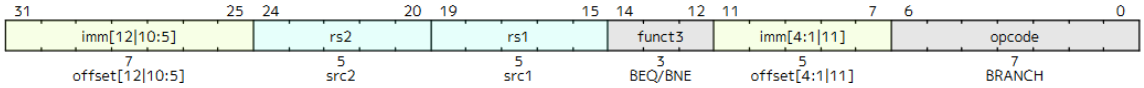


Slika 2.5. Format S instrukcija [4]

Format B

Format B jako je sličan formatu S. Jedina razlika između formata B i S je u načinu pohranjivanja neposredne vrijednosti. Slika 2.6. prikazuje kako se zapisuje neposredna vrijednost u instrukciji te se jednostavno vizualno može usporediti sa slikom 2.5. i vidjeti u čemu se razlikuju.

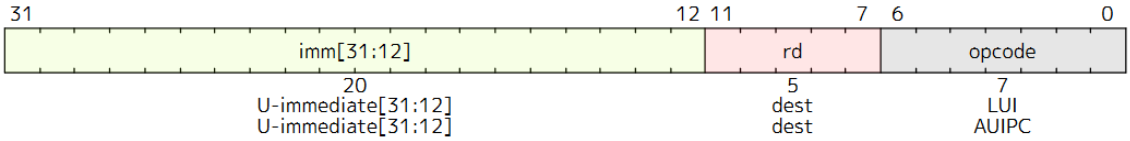
Format B koristi se za kodiranje instrukcija grananja gdje je odredišna adresa veličine 13 bitova, no zbog toga što nije moguće nastaviti program od proizvoljne adrese, već od adrese djeljive s 2, najniži bit odredišne adrese ne pohranjuje se u instrukciji.



Slika 2.6. Format B instrukcija [4]

Format U

Format U je vrlo jednostavan te služi za upis gornjih 20 bitova neposredne vrijednosti u odredišni registar `rd`. Slika 2.7. prikazuje način kodiranja za ovaj format. Instrukcije ovog tipa se u kombinaciji s instrukcijom zbrajanja s neposrednom vrijednošću `addi` koriste za upis 32-bitne konstante u neki od registara opće namjene.

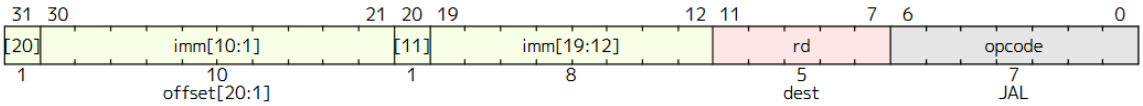


Slika 2.7. Format U instrukcija [4]

Format J

Format J je izrazito sličan formatu U te je razlika između njih jednaka kao i razlika između formata S i B, dakle u načinu zapisivanja neposredne vrijednosti unutar instrukcijske riječi. Slika 2.8. prikazuje kako se zapisuje neposredna vrijednost u instrukciji te se usporedbom sa slikom 2.7. može vidjeti u čemu je razlika.

Ovaj format koristi se za instrukciju neposrednog skoka gdje je neposredna vrijednost odmaka veličine 21 bit, no slično kao i kod instrukcija grananja, najniži bit se ignorira te se smatra da je njegova vrijednost 0.



Slika 2.8. Format J instrukcija [4]

2.1.3. Pregled instrukcija

U nastavku je dan pregled instrukcija. Instrukcije su prema funkcionalnosti razvrstane u nekoliko kategorija.

Aritmetičko-logičke instrukcije

Aritmetičko-logičke instrukcije možemo podijeliti u tri grupe: aritmetičke instrukcije koje uključuju operacije poput zbrajanja i oduzimanja, logičke naredbe koje uključuju operacije poput logičkog I i ILI, naredbe posmaka koje uključuju operacije poput aritmetičkog i logičkog posmaka ulijevo ili udesno i naredbe usporedbe koje zamjenjuju kontrolne zastavice. Konkretnije, to su instrukcije `add`, `sub`, `and`, `or`, `xor`, `sll`, `srl`, `sra`, `slt` i `sltu`.

Ove instrukcije kodiraju se prema ranije opisanom formatu R što znači da imaju tri operanda te da su svi operandi neki od 32 registra opće namjene. Dva operanda su izvorišna, a jedan operand je odredišni. Konkretno, instrukcija je oblika `[add|sub|and|or|xor|sll|srl|sra|slt|sltu] rd, rs1, rs2`, gdje su `rd`, `rs1`, `rs2` neki od registara opće namjene.

Aritmetičko-logičke instrukcije s neposrednim vrijednostima

Ove instrukcije uključuju sve ranije opisane aritmetičko-logičke instrukcije izuzev naredbe `sub`. To su instrukcije `addi`, `andi`, `ori`, `xori`, `slli`, `srli`, `srai`, `slti` i `sltiu`. One se od običnih aritmetičko-logičkih instrukcija razliku samo u tome što je drugi operand umjesto registra `rs2` 12-bitna neposredna vrijednost. Konretno, ove instrukcije su oblika `[addi|andi|ori|xori|slli|srli|srai|slti|sltiu] rd, rs1, imm`, gdje je `imm` neposredna vrijednost veličine 12 bitova, a `rd`, `rs1` neki od registara opće namjene.

Instrukcije za dohvat podataka iz memorije

Ove instrukcije koje, kako im i ime kaže, služe da dohvat podataka iz memorije jedine su takve u instrukcijskom setu. Za razliku od primjerice instrukcijskog seta x86, gdje dohvat operanda može biti kodiran unutar različitih tipova instrukcije, procesor arhitekture RISC-V podatke iz memorije dohvaća isključivo ovim instrukcijama. Sve instrukcije ove vrste kodiraju se prema formatu I. Postoji nekoliko verzija ove instrukcija, ovisno o veličini podatka koji se dohvaća, no sve su oblika `l[b|bu|h|hu|w] rd, imm(rs1)`, gdje je `imm` 12-bitna neposredna vrijednost, a `rd, rs1` neki od registara opće namjene.

Iz memorije je moguće dohvatiti jedan bajt s proizvoljne adrese instrukcijom `lb` (eng. *load byte*). S obzirom na to da su svi registri u koji se pročitan podatak može spremiti veličine 32 bita, podatak će se predznačno proširiti. Ako pak predznačno proširenje podatka nije poželjno, za dohvat okteta iz memorije koristi se instrukcija `lbu` (eng. *load byte unsigned*).

Slično, za čitanje poluriječi koriste se instrukcije `lh` (eng. *load halfword*) i `lhu` (eng. *load halfword unsigned*). Adrese s kojih se čita poluriječ mora biti višekratnik broja 2.

Za čitanje memorijske riječi, koristi se instrukcija `lw` (eng. *load word*). S obzirom na to da se čita 32-bitni podatak, on se ne može predznačno proširiti pa za čitanje riječi postoji samo jedna instrukcija. Adresa s koje se čita podatak instrukcijom `lw` mora biti višekratnik broja 4.

Instrukcije za spremanje podataka u memoriju

Ove instrukcije su jedine koje omogućuju spremanje podataka u memoriju. Drugim riječima, spremanje podatka koji se trenutno nalazi unutar procesora u registru nije moguće izvesti kao dio izvođenja neke druge instrukcije. Za ove instrukcije koristi se format S. Slično kao i kod instrukcija za dohvat podataka iz memorije, razlikujemo tri instrukcije ovisno o veličini podatka, a koje su sve oblika `s[b|h|w] rs2, imm(rs1)`, gdje je `imm` 12-bitna neposredna vrijednost, a `rs1, rs2` neki od registara opće namjene.

Za spremanje jednog bajta podataka na proizvoljnu adresu koristi se instrukcija `sb` (eng. *store byte*). Za spremanje 16-bitnog podatka, odnosno poluriječi koristi se instrukcija `sh` (eng. *store halfword*). Analogno instrukciji `lh`, adresa na koju se sprema podatak

mora biti višekratnik broja 2. Za spremanje memorijske riječi, odnosno 32-bitnog podatka koristi se instrukcija `sw` (eng. *store word*). Odredišna adresa podatka mora biti višekratnik broja 4.

Instrukcije grananja i skoka

Instrukcija grananja i skoka služe za promjenu uobičajenog programskog toka. Razlikujemo instrukcije grananja koje ovisno o ispunjenju uvjeta mijenjaju vrijednost programskog brojila i instrukcije skoka koje to rade bezuvjetno. Odredišna adresa mora biti višekratnik broja 2 što je osigurano tako da se najniži bit relativnog pomaka ignorira te se niti ne nalazi u kodiranoj instrukciji.

Instrukcije grananje kodiraju se prema formatu B i oblika su `b[eq|ne|lt|ge|ltu|geu] rs1, rs2, label`. S obzirom na to da ne postoji statusni registar u kojem bi se nalazile zastavice poput preljeva i predznaka, instrukcije grananja uvjet provjeravaju izvršavanjem usporedbe dva registra `rs1` i `rs2` prilikom izvođenja. Instrukcijski set podržava 6 uvjeta. U slučaju da je uvjet ispunjen, vrijednost programskog brojila mijenja se na zadanu vrijednost `label`. S obzirom na to da se ove instrukcije kodiraju prema formatu B, odredište grananja može biti 13-bitna vrijednost u dvojnom komplementu.

Ovisno o načinu računanja relativnog odmaka, razlikujemo dvije instrukcije skoka.

Instrukcija `jal` (eng. *jump and link*) kodira se prema formatu J te je oblika Instrukcija je oblika `jal rd, label`. Instrukcija radi na način da se u navedeni registar `rd` sprema trenutna vrijednost programskog brojila uvećana za 4, dok se u programsko brojilo dodaje navedeni 21-bitni odmak `label` te se često koristi za poziv potprograma.

Druga instrukcija skoka je `jalr` (eng. *jump and link register*) koja se kodira prema formatu I te često koristi za povratak iz potprograma. Oblika je `jalr rd, imm(rs1)`. Ona slično kao i `jal` u registar `rd` upisuje vrijednost programskog brojila uvećanu za 4, no odmak računa tako da na 12-bitnu neposrednu vrijednost `imm` dodaje vrijednost zapisanu u registru `rs1`.

Posebne aritmetičke instrukcije

Posebne aritmetičke instrukcije uključuju dvije instrukcije koje se kodiraju prema formatu U. Ove instrukcije služe za učitavanje najviših 20 bitova u neki od registara opće namjene.

Instrukcija `lui` (eng. *load upper immediate*) služi da izravan upis najviših 20 bitova neposredne vrijednosti u jedan od 32 registra opće namjene te je oblika `lui rd, %hi(konstanta)`. Ova instrukcija u kombinaciji s instrukcijom `addi` omogućuje upis bilo koje 32-bitne vrijednosti u registar.

Instrukcija `auipc` (eng. *add upper immediate program counter*) kao i `lui` služi za učitavanje 20-bitne konstante, ali ne izravno već u zboju s trenutnom vrijednošću programskog brojila. Primjerice instrukcija `auipc t1, %pcrel(0x100)` će u registar `t1` zapisati vrijednost programskog brojila uvećanog za 0x100 posmaknutog za 12 mjesta jer se upisuju samo najviših 20 bitova.

Ostale instrukcije

Instrukcijski skup RV32I osim svih navedenih i opisanih instrukcija sadrži još 3 instrukcije. Ove instrukcije kodirane su prema formatu I.

Instrukcija `fence` služi kao memorijska ograda za procesor. Moderni prevoditelji često mijenjaju redoslijed kojim se program izvodi zbog povećanja performansi, a isto tako i moderni procesori neke instrukcije izvode izvan redoslijeda (eng. *out of order execution*). Ovakvom instrukcijom naglašava se kako je redoslijed kojim se neki podaci dohvaćaju ili spremaju u memoriju bitan te kako se on ne smije mijenjati.

Instrukcija `ecall` (eng. *environment call*) služi za obavljanje sistemskih poziva što primjerice može biti poziv jezgre operacijskog sustava za koji je potreba veća razina ovlasti [4].

Instrukcija `ebreak` (eng. *environment break*) daje kontrolu za izvođenje tzv. *debugging* okruženju te može biti korisna pri oblikovanju programske podrške [4].

2.2. Načini rada

Specifikacija arhitekture RISC-V [4][5] definira 3 načina rada. Različiti načini rada omogućuju zaštitu računalnog sustava tako što ovisno o tome kakav se program izvodi, procesor ima ograničenu mogućnost pristupa i mijenjanja drugih programa, sklopovlja i raznih drugih resursa kao što su vanjske jedinice. Kratki opis i namjena svakog načina rada dani su u nastavku.

Način rada U

Korisnički (eng. *user*) ili aplikacijski (eng. *application*) način rada ima najniže ovlasti i kodira se nizom `00`. Ovaj način rada koristi se za izvođenje uobičajenih korisničkih programa te stoga programi u ovom načinu rada moraju imati ograničen pristup memorijskom prostoru, pojedinim registrima kao što su kontrolni i statusni registri i sl.

Način rada S

Nadgledni (eng. *supervisor*) način rada ima višu razinu ovlasti od korisničkog i kodira se nizom `01`. Ovaj način rada koristi se pri izvođenju procedura jezgre operacijskog sustava ili upravljača virtualnim strojevima (hipervizora). Iako ovaj način rada ima više razine ovlasti, programi u nadglednom načinu rada nemaju pristup svim dijelovima procesora kao što su neki kontrolni i statusni registri.

Način rada M

Strojni (eng. *machine*) način rada ima najvišu razinu ovlasti i kodira se nizom `11`. Način rada M koristi se za programe koji daju pristup svim dijelovima procesora. Programi koji se izvode u ovom načinu rada smatraju se u potpunosti povjerljivima. Primjer programa koji se može izvoditi u načinu rada M je ugrađen program za inicijalizaciju (eng. *bootloader*).

Način rada M obavezan je za implementaciju, dok se ostali načini rada implementiraju ovisno o aplikaciji procesora. Uobičajeno je da procesori za ugradbene računalne sustave implementiraju i način rada U [5].

2.3. Kontrolni i statusni registri - CSR

Osim 32 registra opće namjene, procesori arhitekture RISC-V uobičajeno sadrže i neke kontrolne i statusne registre. Ti registri razlikuju se od registara opće namjene tako što njihovom sadržaju nije moguće pristupiti instrukcijama is osnovnog skupa RV32I, već instrukcijama iz skupa "Zicsr". Kontrolni i statusni registri se razlikuju i svojom namjenom. U te registre se ne spremaju podaci programa, već servisne informacije koje daju procesoru dodatne upute o tome kako program kojeg izvršava treba raditi ili informacije o stanju procesora kroz parametre poput oznake trenutne dretve u izvođenju ili broja ciklusa.

Način na koji su registri alocirani u memoriji ovisi o tome u kojem načinu rada im je moguće pristupiti. U nastavku je dan detaljniji opis nekih kontrolnih i statusnih registara za način rada M. Prikazani su samo statusni i kontrolni registri za način rada M s obzirom na to kako je on obavezan za implementaciju te kako se iznimke i prekidi obrađuju u tom načinu rada.

2.3.1. Strojni informacijski registri

Ovi registri sadrže informacije o procesoru i programu koji se izvodi. Tako ovaj podskup sadrži sljedeće registre:

- `mvendorid` - oznaka proizvođača
- `marchid` - oznaka arhitekture procesora
- `mimpid` - oznaka verzije procesora
- `mhartid` - oznaka trenutne dretve u izvođenju
- `mconfigptr` - pokazivač na konfiguracijsku strukturu podataka

Ovi registri se mogu samo čitati, ali nisu obavezni za implementaciju. U slučaju da procesor ne implementira ove registre ili ih ne koristi, pokušaj čitanja ovih registara može vratiti vrijednost 0 [5].

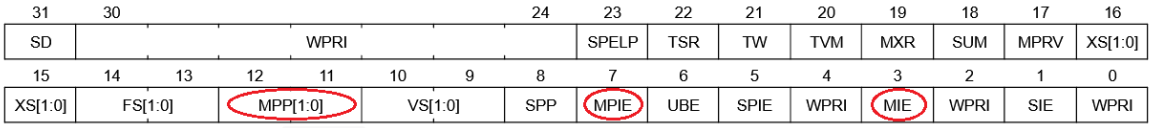
2.3.2. Strojni registri za iznimke

U nastavku je dan pregled i kratak opis registara za iznimke u strojnom načinu rada:

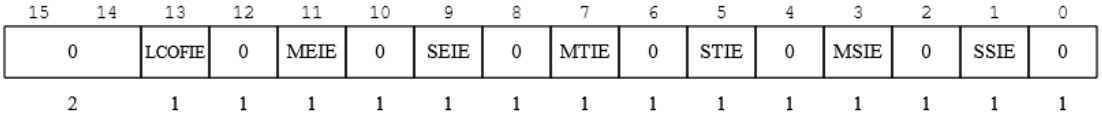
- `mstatus` - sadrži informacije o trenutnom stanju procesora kao što su omogućenost prekida, prethodni način rada, konfiguraciju virtualizacije, virtualne memorije i sl.
- `misa` - sadrži informacije o implementiranih instrukcijskim skupovima u procesoru
- `medeleg` - registar služi za prosljeđivanje iznimke u način rada nižeg prioriteta
- `mideleg` - registar služi za prosljeđivanje prekida u način rada nižeg prioriteta
- `mie` - registar koji određuje koje vrste prekida u načinu rada M su omogućene
- `mtvec` - postavlja baznu adresu prekidnih vektora te određuje izvode li se prekidi u zasebnim prekidnim rutinama (*vectored mode*) ili u jednoj prekidnoj rutini za sve vrste prekida (*direct mode*)
- `mcounteren` - registar za omogućavanje registara za praćenje performanse procesora
- `mstatush` - dodatni registar koji sadrži dodatnih (viših) 32 bita registra mstatus u kojima je opisan način zapisa podataka u memoriji (eng. *little/big endian*)
- `medelegh` - gornjih 32 bita registra medeleg
- `mscratch` - registar bez specifikacijom definirane uloge, a u prekidima se uobičajeno koristi za spremanje pokazivača stoga u prekidnom potprogramu
- `mepc` - registar u koji se prihvaćanjem prekida sprema trenutna vrijednost programskog brojila
- `mcause` - registar koji sadrži dodatne informacije što je izazvalo prekid (i koji prekid je prihvaćen ako postoji više prekida u čekanju)
- `mtval` - sadrži dodatne informacije o iznimci poput virtualne adrese stranice koja je izazvala grešku (eng. *page fault*)
- `mip` - sadrži zastavice koje označavaju koji prekidi su u stanju čekanja na prihvat (eng. *pending*)
- `mtinst` - sadrži informaciju koja instrukcija je izazvala pogrešku
- `mtval2` - sadrži istu informaciju kao i mtval, ali za virtuali stroj

Ovi registri služe za implementaciju prekidnog sustava koji uključuje posluživanje prekida (od primjerice vanjskih jedinica) i iznimki (greške u izvođenju procesora što pri-

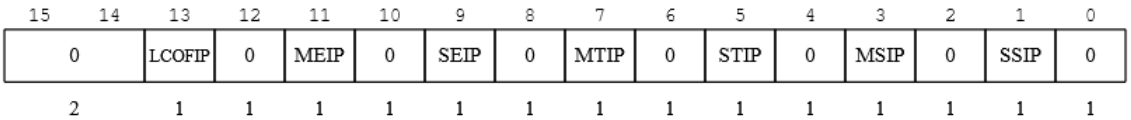
mjerice može biti dohvaćanje instrukcije s nepostojeće adrese). Ako procesor nema implementiran prekidni sustavi, tada niti ovi registri ne moraju biti implementirani. U neke od navedenih registara program u izvođenju može izravno pisati (npr. `mscratch`), dok neke od registara samo čita (npr. `mcause`). U nastavku su opisani najvažniji registri za obradu prekida.



Slika 2.9. Registar `mstatus` [5]. Označene su zastavice važne za prekid, a to su `MPP` u kojoj je spremljen prethodni način rada, `MPIE` u koju se prilikom ulaska u prekidni potprogram sprema zastavica `MIE` koja označava jesu li prekidni globalno omogućeni.



Slika 2.10. Registar `mie` [5]. Pojedine zastavice označavaju je li omogućen konkretan prekid. Primjerice zastavica `MTIE` označava je li omogućen vremenski prekid u strojnom načinu rada.



Slika 2.11. Registar `mip` [5]. Pojedine zastavice označavaju je li stigao zahtjev za konkretan prekid. Primjerice zastavica `MTIP` označava je li stigao zahtjev za vremenski prekid u strojnom načinu rada.

2.3.3. Strojni registri brojila

Ovi registri služe za mjerenje performanse procesora (eng. *profiling*) kroz broj ciklusa za što služi `mcycle(h)`, broj izdanih (eng. *retired*) instrukcija `minstret(h)`. Razni drugi parametri, neki od kojih su broja promašaja priručne memorije, broj instrukcija grananja, broj pogrešnih predviđanja grananja, mogu se proizvoljno definirati u registrima `mhpmcounter{3-31}(h)`. Oznaka `h` označava registar u koji se sprema gornjih 32 bita, odnosno svaki registar je efektivno 64-bitan.

2.4. Ostali instrukcijski skupovi

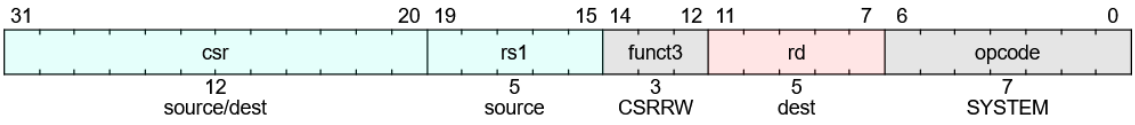
Osim osnovnog instrukcijskog skupa RV32I, specifikacija definira puno veći broj instrukcija te se njihov broj iz dana u dan povećava s obzirom na to da razvoj ukupnog skupa instrukcija još uvijek traje. Neki od ostalih skupova instrukcija prikazani su u sljedećim potpoglavljima.

2.4.1. Instrukcijski skup "Zicsr"

Kako je već rečeno, čitanje i pisanje u kontrolne i statusne registre nije moguće dosad opisanim instrukcijama skupa RV32I. Zbog toga, definirano je 6 instrukcija koje omogućuju čitanje, pisanje i manipulaciju pojedinim bitovima u kontrolnim i statusnim registrima.

Ove instrukcije kodiraju se slično formatu I. Slika 2.12. prikazuje kako se prevode instrukcije za pristup kontrolnim i statusnim registrima. Jedina razlika u odnosu na format I je u tome što se najviših 12 bitova, umjesto za smještanje neposredne vrijednosti, koristi za spremanje adrese kontrolnog i statusnog registra. Svaki kontrolni i statusni registar adresira se s 3 heksadecimalne znamenke odnosno 12 bitova. Polje `rs1` može biti registar (npr. za instrukciju csrrw) ili neposredna vrijednost duljine 5 bitova (npr. za instrukciju csrrwi). Vrijednost polja `opcode` za sve instrukcije iz ovog skupa je ista dok se svaka od pojedinih instrukcija razaznaje (dekodira) prema polju `funct3`.

U nastavku su opisane instrukcije ovog skupa.



Slika 2.12. Format instrukcija skupa Zicsr [4]

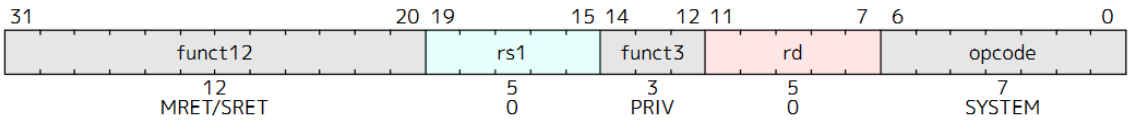
- csrrw - omogućuje čitanje kontrolnog i statusnog registra u registar opće namjene te upis (drugog) registra opće namjene u kontrolni i statusni registar
- csrrs - omogućuje čitanje kontrolnog i statusnog registra u registar opće namjene te postavljanje bitova u istom kontrolnom i statusnom registru prema (drugom) registru opće namjene
- csrrc - omogućuje čitanje kontrolnog i statusnog registra u registar opće namjene te brisanje bitova u istom kontrolnom i statusnom registru prema (drugom) registru

opće namjene

- csrrwi - omogućuje čitanje kontrolnog i statusnog registra u registar opće namjene te upis neposredne konstante u kontrolni i statusni registar
- csrrsi - omogućuje čitanje kontrolnog i statusnog registra u registar opće namjene te postavljanje bitova u istom kontrolnom i statusnom registru prema neposrednoj konstanti
- csrrci - omogućuje čitanje kontrolnog i statusnog registra u registar opće namjene te brisanje bitova u istom kontrolnom i statusnom registru prema neposrednoj konstanti

2.4.2. Privilegirane instrukcije strojnog načina rada

Način rada M ima dodatne instrukcije svojstvene za privilegiran način rada. Te instrukcije prevode se slično kao i instrukcije skupa "Zicsr", kao format I. Slika 2.13. prikazuje kako se kodiraju ovakve instrukcije. Valja napomenuti kako su polja `rs1` i `rd` kodirana nizom nula. Polje `opcode` isto je kao i za instrukcije skupa "Zicsr". U nastavku su opisane instrukcije ovog skupa.



Slika 2.13. Format instrukcija za privilegirano izvođenje [5]

- mret - služi za povratak iz prekidnog potprograma, točnije instrukcija vraća staro stanje zastavice `MIE` u registru mstatus te u programsko brojilo pc upisuje sadržaj registra mepc
- wfi (wait for interrupt) - zaustavlja rad procesora dok se ne dogodi prekid ili iznimka

2.4.3. Direktive

Direktive služe za davanje više informacija u programu. Direktive nisu instrukcije, već se one prevode u prvom prolazu asemblera, dok se nakon toga prevode instrukcije programa. Tako se može definirati sadržaj memorijske riječi, poluriječi ili okteta na nekoj

adresi s pomoću direktiva `.word`, `.half` i `.byte`, rezervirati dio memorijskog prostora s pomoću direktive `.zero`. Jednako tako se može definirati konstanta direktivom `.equ` ili za neki niz instrukcija definirati od koje adrese se one nalaze direktivnom `.org`. Uz navedene, postoji još niz drugih direktiva poput smještanja imena u globalni doseg (`.global`) i sl. [6]

2.4.4. Pseudoinstrukcije

Pseudoinstrukcije se za razliku od direktiva smatraju instrukcijama. One uključuju instrukcije koje nisu dio niti jednog instrukcijskog skupa, ali se prevode u neku od instrukcija koje definiraju specifikacije, primjerice pseudoinstrukcija nop (eng. *no operation*) prevodi se u instrukciju `addi x0, x0, 0`. Tako za arhitekturu RISC-V postoje instrukcije poput negacije u 1-komplementu (not), negacije u 2-komplementu (neg), razne instrukcije grananja s ostalim mogućim uvjetima koji nisu pokriveni standardnih skupom instrukcija grananja poput `ble` (eng. *branch less or equal*), `beqz` (eng. *branch equal zero*) itd. Osim njih, postoje pseudoinstrukcije poput upisa sadržaja jednog registra opće namjene u drugi (mv) itd. [6]

3. Verifikacijsko okruženje

Kako modeli računalnog sustava postaju sve kompleksniji i povećavaju brzinu i broj vanjskih jedinica, sve je veća vjerojatnost za pogrešku u oblikovanju sustava. Iz tog razloga za ispitivanje ispravnosti modela koristi se verifikacijsko okruženje. U ovom poglavlju opisano je verifikacijsko okruženje razvijeno u sklopu ovog rada. Verifikacijsko okruženje kao referentni model koristi programski paket SSPARCSS pa je stoga najprije dan njegov kratak opis te je zatim opisano samo okruženje i niz programa korištenih za provjeru ispravnosti rada.

3.1. Programski paket SSPARCSS

SSPARCSS (eng. *Software Suite for Processor ARChitectue Simulation and Study*) je programski paket koji služi za simulaciju rada procesora i računalnog sustava u cjelini, odnosno procesora povezanog s vanjskim jedinicama. SSPARCSS je razvijen na Fakultetu elektrotehnike i računarstva te je njegova svrha upotreba u nastavi, točnije poučavanju osnovnih koncepata organizacije i arhitekture računalnih sustava. Simulacija je izvedena tako da model procesora, odnosno računalnog sustava izvodi program napisan u asemblerskom jeziku (eng. *assembly*) za taj model procesora. Programski paket sastoji se od asemblera i simulatora. Asembler služi za prevođenje programa iz niza mnemonika u strojni kod, odnosno niz bitova [7]. Simulator služi za simuliranje izvođenja strojnog koda na sklopovlju [8]. Programski paket koristi se na način da se u simulatoru učita željeni model računalnog sustava koji sadrži pored procesora (različitih arhitektura), i ostale komponente sustava kao što su memorija, sabirnica, vanjske jedinice. Za izvođenje programa potrebno je prevesti program napisan u zbirnom jeziku procesora za ranije odabrani model. Kada korisnik završi pisanje svog programa, pokreće prevođenje u strojni kod te ako je program ispravno napisan, asembler generira izvršnu datoteku

koja se učitava u simulator te korisnik može pokrenuti simulaciju programa na modelu računalnog sustava.

3.1.1. Asembler

Princip rada

Asembler je dio programskog paketa SSPARCSS kojim se niz mnemonika prevodi u niz bitova, odnosno u strojni kod. Program nudi mogućnost pisanja, uređivanja i prevođenja programa u izvršni kod. Ulazna datoteka programa u koju se piše program ima ekstenziju **.a*, dok je izlaz iz asemblera izvršna datoteka koja ima ekstenziju **.e* [7].

S obzirom na to da je cilj programskog paketa SSPARCSS simulacija različitih računalnih sustava s procesorima različitih arhitektura, odnosno instrukcijskih skupova, asembler se može konfigurirati za prevođenje programa pisanih za različite modele procesora. Ovime je omogućena jednostavna nadogradnja i dodavanje novih računalnih sustava i instrukcijskih skupova. Konfiguracija asemblera nalazi se u zasebnoj datoteci ekstenzije **.assebler* koja je pisana u jeziku ADEL.

ADEL

ADEL (eng. *Assembler DEscription Language*) je jezik kojim se opisuje konfiguracija asemblera za programski paket SSPARCSS. Glavni dijelovi **.assembler* datoteke su init-blok i blokovi znakovnih i brojevnih zamjena [9].

Init-blok služi za definiranje početnih postavki asemblerskog prevoditelja među kojima su ime procesora, širina adresne sabirnice, poravnanje memorijskih riječi i sl.

Blok znakovne ili brojevne zamjene služi za zamjenu niza znakova ili broja odgovarajućim nizom bitova. Ovim blokovima ostvaruje se najvažnija funkcionalnost asemblerskog prevoditelja, a to je prevođenje mnemonika u niz bitova, odn. u strojni kod. Brojevne zamjene služe za smještanje neposredne vrijednosti u instrukcijsku riječ.

Osim navedenih dijelova, **.assembler* datoteka može imati i blokove za zamjenu direktiva koje asemblerski prevoditelj obrađuje prije prevođenja instrukcija i pseudoinstrukcija, blok u kojem se definiraju funkcije itd. Osim toga, program asembler omogućuje makrozamjene [9].

3.1.2. Simulator

Princip rada

Simulator je dio programskog paketa SSPARCSS kojim se simulira izvođenje izvršne datoteke na modelu procesora [8]. Program osim učitavanja i izvođenja programa pruža i mogućnost izvođenja instrukciju po instrukciju te praćenje simulacije kroz animaciju računalnog sustava. Kroz animaciju je moguće očitavati stanje vanjskih jedinica i registara unutar procesora.

Slično kao i asembler, simulator je moguće konfigurirati za različite računalne sustave i različite procesore. Konfiguracija simulatora, odn. računalnog sustava koj se simulira, nalazi se u zasebnoj datoteci ekstenzije **.system* koja je pisana u programskom jeziku COMDEL.

Programski jezik COMDEL

COMDEL (eng. *COMponent DEscription Language*) je jezik koji služi za modeliranje računalnog sustava koji se zatim može simulirati u programu simulator [10]. Svojom strukturom sličan je poznatim jezicima za opis sklopovlja (eng. *hardware description language*) VHDL i Verilog, no COMDEL služi samo za simuliranje modela te nudi grafičku podršku za animacije.

Model računalnog sustava sastoji se od komponenti koje čine procesor, memorija, vanjske jedinice i sl. Glavna komponenta koja objedinjuje ostale, prethodno navedene komponente naziva se System i zapisana je u datoteci s ekstenzijom **.system* [10]. Unutar središnje komponente uključuju se podkomponente koje su svaka za sebe definirane u zasebnim datotekama s ekstenzijom **.comdel*. Ovakav način modeliranja analogan je strukturnom modelu u jezicima za opis sklopovlja. Uz uključivanje podkomponenti, COMDEL pruža mogućnost prikaza komponenti u simulatoru. Stoga se za pojedinu komponentu može odabrati položaj prikaza u simulatoru unutar bloka `display`.

Središnja komponenta System ne sadrži nikakvu funkcionalnost. Sva funkcionalnost modela računalnog sustava definirana je u ostalim komponentama opisanim u **.comdel* datotekama. Te komponente poput procesora strukturom se razlikuju od središnje komponente System u **.system* datoteci. One sadrže 3 glavne vrste blokova koji opisuju pona-

šanje komponente: blokove za vanjski i unutarnji prikaz, inicijalizacijski blok i procesni blok.

Inicijalizacijski blok služi za dovođenje svih unutarnjih registara i signala komponente u odgovarajuće početno stanje. Procesni blok opisuje funkcionalnost komponente, analogno procesnom bloku u VHDL-u. Blok za vanjski prikaz komponente (`external display`) opisuje smještaj komponente u cijelom računalnom sustavu, dok blok za unutarnji prikaz (`display`) opisuje prikaz unutarnjih dijelova komponente, npr. registara u sklopu GPIO ili procesoru. Uz to, komponenta može imati funkcije, slično kao i u jezicima za opis sklopovlja.

3.2. Ispitni programi

Kako bi se provela verifikacija modela, potrebno je imati dovoljan broj programa s pomoću kojih će se testirati što veći skup funkcionalnosti. Ovisno o tome koje funkcionalnosti se testiraju u modelu, bit će potrebno razviti drugačiji skup testnih programa.

Ako je cilj verifikacije ispitati ispravnost rada modela procesora, tada će ispitni programi biti generički te će primarno ispitivati ispravnost implementacije instrukcijskog skupa. S druge strane, ako je cilj ispitati ispravnost modela u cjelini, primjerice nakon nadogradnje ili promjene modela, tada će ispitni programi biti složeniji od generičkih te će se svaki od njih oblikovati tako da ispituje jednu ili više funkcionalnosti kao što je rad s vanjskim jedinicama, obrada skupa podataka, obrada prekida i sl.

Ispitni programi za rad procesora

Za ispitivanje rada procesora napravljen je niz generičkih ispitnih programa. Ti programi, odnosno ispitni primjeri, napravljeni su s pomoću skripte u programskom jeziku Python.

Svaki od primjera radi na sličan način. S obzirom na to da su ovi primjeri isprobani na simulatoru programskog paketa SSPARCSS, za početak je u registre potrebno upisati neke vrijednosti što se radi nizom instrukcija `lui` i `addi`. Nakon toga, izvodi se niz instrukcija kojima se testira ispravnost rada modela procesora za određenu instrukciju. Korektnost takvog ispitivanja postiže se korištenjem različitih registara kao odredišnih

te korištenjem čim raznovrsnijih vrijednosti operanada.

Svaki ispitni program pohranjuju se u zaseban direktorij. Ako direktorij ne postoji, novi će biti stvoren. Programsko rješenje nudi više opcija izgradnje ispitnih primjera.

Program nudi opciju kreiranja asemblerskih programa za jednu od niza podržanih instrukcija ili kreiranja programa za svaku podržanu instrukciju ovisno o tome je li naveden argument `-i`. Za svaku instrukciju program generira dva asemblerska programa. Jedan program sadrži po jedan primjer instrukcije za svaki od registara opće namjene te je njegovo ime oblika *[instrukcija]_basic.a*. Drugi program sadrži slučajno odabrane primjere registara te je ime tog programa oblika *[instrukcija]_random.a*.

Kako je ranije rečeno, svaki od registara opće namjene ima jedno dodatno ime koje nije oblika `x[0..31]` (izuzev `x8` koji ima dva alternativna imena). Iz tog razloga, programsko rješenje za generiranje ispitnih programa ima opciju generirati niz instrukcija samo za jednu od dvije nomenklature navođenjem parametra `-v` pri pozivu programa. Moguće vrijednosti parametra su "X", što označava generiranje asemblerskih programa uz općenitiju nomenklaturu prikazanu na slici 2.2., i "Y", što označava generiranje asemblerskih programa uz alternativnu nomenklaturu. Ako parametar nije naveden, generiraju se programi za obje nomenklature.

Trenutno podržane instrukcije su aritmetičko-logičke instrukcije bez instrukcija posmaka, memorijske naredbe i posebne aritmetičke instrukcije.

Ispitni programi za model sustava

Ako se pak želi ispitati model u cjelini, tada se ispitivanje može provesti na nizu općenitih programa kakvi bi se stvarno mogli izvoditi na modelu.

U okviru kolegija *Arhitektura računala 1R* napravljena je zbirka zadataka za posebno izrađen procesor *FRISC*, a kasnije i procesor arhitekture *ARM*. Zbirka se sastoji od dvije velike grupe zadataka, programiranja procesora i programiranja vanjskih jedinica.

Za potrebe ovog rada, uzeti su zadaci iz grupe programiranja procesora. Ti zadaci mogu se podijeliti u nekoliko podgrupa, dio koji radi manipulaciju bitova u registrima, dio koji radi pretvorbe podataka iz jednog zapisa u drugi, dio koji radi razne operacije

i obrade nad podacima, dio koji je posvećen pisanju potprograma i dio koji objedinjuje prethodne.

Nakon ispravljanja programa i usklađivanja s tekstom zadataka, ovaj niz zadataka poslužit će za testiranje rada verifikacijskog okruženja.

3.3. Verifikacijsko okruženje

U nastavku je prikazano razvijeno verifikacijsko okruženje koje se oslanja na programski paket SSPARCSS kao referentni model.

3.3.1. Specifikacija

Cilj verifikacijskog okruženja je zadovoljavanje sljedećih zahtjeva:

- usporedba nekoliko rješenja istog zadatka s referentnim rješenjem
- usporedba jednog zadatka s referentnim rješenjem
- usporedba niza različitih zadataka s odgovarajućim referentnim rješenjem
- izrada referentnih rješenja

Verifikacijsko okruženje namijenjeno je korištenju u edukaciji za evaluaciju rješenja zadataka. Uz to, verifikacijsko okruženje oblikovano je na način da se njegovi dijelovi mogu zasebno koristiti, primjerice za kontinuiranu integraciju (eng. *continuous integration*).

Provjera s referentnim modelom i generiranje referentnog rješenja napravljeno je korištenjem programskog paketa SSPARCSS bez pokretanja grafičkog korisničkog sučelja (eng. *Graphical User Interface, GUI*) [11].

Rezultat simulacije asemblerskog programa prikazan je datotekom tipa JSON. Datoteka se generira na temelju ulazne datoteke JSON. Primjer ulazne datoteke JSON koja ispituje sadržaj memorijske lokacije 256 prikazan je na slici 3.1. Ta datoteka sadrži put do **.system* datoteke računalnog sustava nad kojim se izvodi simulacija, tip procesora, polje sa sadržajem željenih memorijskih lokacija te polje u kojem su zapisane vrijednosti svih registara opće namjene.

Slika 3.1. Prikaz ulazne datoteke JSON

Generiranje referentnih rješenja ili usporedba niza različitih zadataka s odgovarajućim referentnim rješenjem na velikom broju zadataka može biti izrazito nepregledno te se iz tog razloga može koristiti program za preoblikovanje direktorija koji je implementiran za potrebe ovog rada.

3.3.2. Implementacija

Verifikacijsko okruženje implementirano je u programskom jeziku Python. Slika 3.3. prikazuje strukturu programa.

Središnje funkcije u programu su `evaluate_different_tasks`, `evaluate_same_tasks`, `evaluate_single_task` i `generate_ref_json`. Te funkcije implementiraju redom usporedbu nekoliko rješenja istog zadatka s referentnim rješenjem, usporedbu jednog zadatka s referentnim rješenjem, usporedbu niza različitih zadataka s odgovarajućim referentnim rješenjem te izradu referentnih rješenja.

Uz to, u programu postoje pomoćne funkcije koje služe pokretanju simulacije, usporedbi datoteka JSON te provjeri ispravnosti argumenata. Slika 3.2. prikazuje primjer referentne datoteke JSON za program koji provjerava memorijsku lokaciju 256 te registar `x3`.

Slika 3.4. prikazuje način pokretanja verifikacijskog okruženja. Program kao ulazne argumente prima vrstu zadatka. Postoje 4 vrste zadatka koje program obavlja te su vidljive na slici. Svaka vrsta zadataka ima analogno ime jednoj od ranije navedenih središnjih funkcija. Uz to, pri pozivu programa obavezno je navesti putanju do izvršne datoteke simulatora te put do asemblerskih programa što može biti put do datoteke ili direktorija

```json
{
    "cpu": {
        "cpu_type": "RISCV",
        "system": "C:\\Users\\vinko\\OneDrive\\Dokumenti\\SSPARCSS_usporedba\\stabilni\\models\\RV32I.system"
    },
    "states": [
        {
            "memory": [
                {
                    "address": 256,
                    "data": [
                        6
                    ],
                    "length": 1
                }
            ],
            "registers": {
                "x3": 6
            }
        }
    ]
}
```


Slika 3.2. Prikaz referentne datoteke JSON

Slika 3.3. Pregled funkcija okruženja

ovisno o argumentu `--type`. Pri pozivu se za neke vrste izvršavanja, uz ranije navedene argumente, navodi i putanja do ulaznih i referentnih datoteka JSON. Konkretan prikaz za svaki tip pokretanja prikazan je u dodatku A.

```
usage: verif_env.py [-h] --type {gen,eval_same,eval_diff,eval_single} [--input INPUT] [--ref REF] --simulator
                    SIMULATOR --task TASK

options:
  -h, --help            show this help message and exit
  --type {gen,eval_same,eval_diff,eval_single}
                        type of testing
  --input INPUT         input json
  --ref REF             referent json
  --simulator SIMULATOR
                        simulator executable file
  --task TASK           task file/directory
```

Slika 3.4. Način pokretanja verifikacijskog okruženja

Slika 3.5. prikazuje sekvencijski dijagram za verifikacijsko okruženje. Dijagram pri-

kazuje način rada verifikacijskog okruženja.

Nakon što korisnik pokrene program s ulaznim argumentima kao na slici 3.4., program provjerava ispravnost ulaznih argumenata. Ulazni argumenti razlikuju se ovisno o vrsti izvršavanja koje je odabrano. Primjerice ako je pokrenuta evaluacija samo jednog zadatka, argumenti `--input`, `--task` i `--ref` moraju biti putanja do jedne datoteke. S druge strane, pri pokretanju generiranja referentnih datoteka JSON, isti argumenti moraju biti putanja do direktorija. Ako su ulazni argumenti u neispravnom formatu, ili navedene datoteke ili direktoriji ne postoje, program će korisniku javiti pogrešku te prekinuti svoj rad.

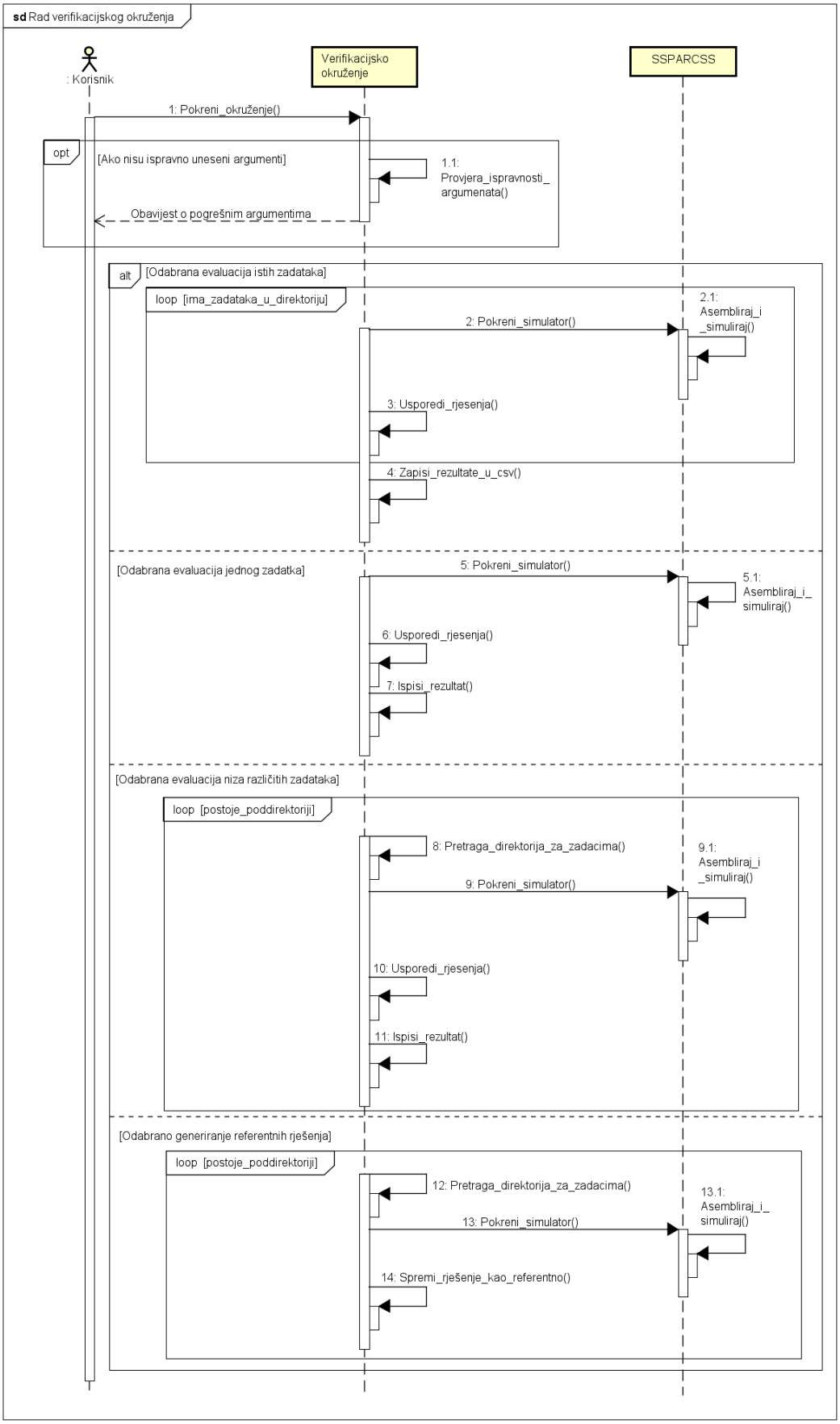
Dijagram nadalje za svaku vrstu izvršavanja prikazuje interakciju između verifikacijskog okruženja i programskog paketa SSPARCSS. Princip rada za svaku vrstu je donekle sličan.

Usporedba nekoliko rješenja istog zadatka s referentnim rješenjem primarno je namijenjena za evaluaciju studentskih rješenja, primjerice u okviru laboratorijskih vježbi. Evaluacija radi na način da verifikacijsko okruženje za svaki zadatak u direktoriju pokreće SSPARCSS bez GUI-ja.

Nakon što novostvoreni proces, odnosno simulacija, završi, u istom direktoriju kreirana je izlazna datoteka JSON u kojoj su navedene vrijednosti registara i željenih memorijskih lokacija. Verifikacijsko okruženje zatim uspoređuje izlaznu s referentnom datotekom JSON te sprema podudarnost. Nakon što je za sve programe u direktoriju završena simulacija, podudarnosti se zapisuju u datoteku s ekstenzijom **.csv*. Nakon toga, korisnik dobiva poruku kako je evaluacija završena.

Usporedba jednog zadatka s referentnim rješenjem radi vrlo slično. Jedina razlika u odnosu na usporedbu nekoliko rješenja istog zadatka s referentnim rješenjem je u tome što se simulacija i usporedba datoteka JSON radi samo jednom. Uz to, rezultat se ispisuje korisniku kao povratna informacija i kao obavijest da je završeno izvođenje verifikacijskog okruženja umjesto zapisivanja u drugu datoteku.

Usporedba niza različitih zadataka s odgovarajućim referentnim rješenjem radi donekle slično, ali ipak različito u odnosu na prethodna dva načina izvršavanja. S obzirom



Slika 3.5. Sekvencijski dijagram koji prikazuje način rada verifikacijskog okruženja

na to da je potrebno pokretati i usporediti rješenja različitih asemblerskih programa, za svaki asemblerski program potrebno je pronaći odgovarajući ulazni JSON, program s ekstenzijom **.a* te referentni JSON. Za svaku takvu pronađenu uređenu trojku pokreće se simulacija. Nakon što simulator generira izlazni JSON, verifikacijsko okruženje će ga usporediti s referentnim te rezultat usporedbe ispisati korisniku.

Izrada referentnih rješenja radi analogno prethodnom načinu. S obzirom na to da je zadatak verifikacijskog okruženje izraditi referentne datoteke JSON, ono za svaki pronađeni par (ulazni JSON, program s ekstenzijom **.a*) pokreće simulaciju, no nakon toga ne radi nikakve usporedbe, već izlazni JSON označuje kao referentni.

Okruženje osim programa za evaluaciju uključuje program koji služi za preoblikovanje direktorija. Program na temelju niza asemblerskih programa kreira novi poddirektorij koji sadrži asemblerski program, ulaznu datoteku tipa JSON te datoteku tipa *.assembler. Ovakav poddirektorij je pregledan jer sadrži samo jedan program te olakšava rad verifikacijskom okruženju.

3.4. Ostala postojeća rješenja

Prethodno je prikazano rješenje verifikacije s pomoću programskog paketa SSPARCSS kao zlatnog modela. Takva praksa nije neuobičajena [12].

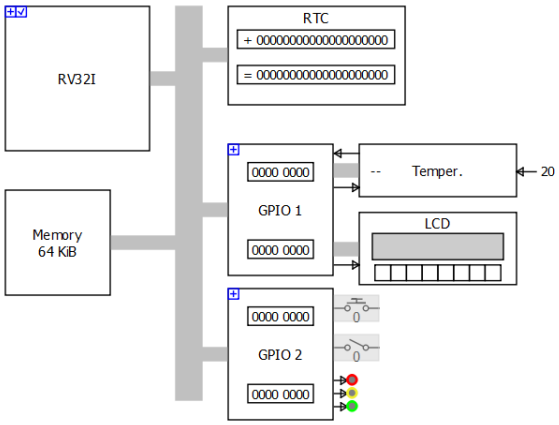
Jedna od prednosti RISC-V arhitekture je njena zajednica [13]. Tako je nastalo verifikacijsko okruženje *riscv-formal* [14] koje preko sučelja RVFI komunicira s procesorom. Ovaj radni okvir je primjer formalne verifikacije digitalnog sustava koja se oslanja na matematičke metode za provjeru ispravnosti. Uz to, postoji i službeni skup ispitnih programa za ispitivanje ispravnosti implementacije pojedine instrukcije i usklađenosti procesora sa specifikacijom [15].

S druge strane, osim formalne verifikacije se može, kao u našem rješenju, koristiti funkcijska verifikacija koja za određene ulaze provjerava podudarnost izlaza [13].

4. Model sustava s procesorom arhitekture RV32I

4.1. Pregled

Slika 4.1. prikazuje izgled polazišnog, prethodno razvijenog, modela računalnog sustava s procesorom RISC-V u programskom paketu SSPARCSS koji se sastoji od jednostavne sabirnice, procesora, memorije, sklopa RTC (eng. *Real Time Clock*) i dva sklopa GPIO (eng. *General Purpose Input Output*) na koje su spojeni temperaturni sklop, zaslon LCD (eng. *Liquid Crystal Display*) te svjetleće diode i tipkala. U nastavku rada opisana je nadogradnja polazišnog modela sustava.

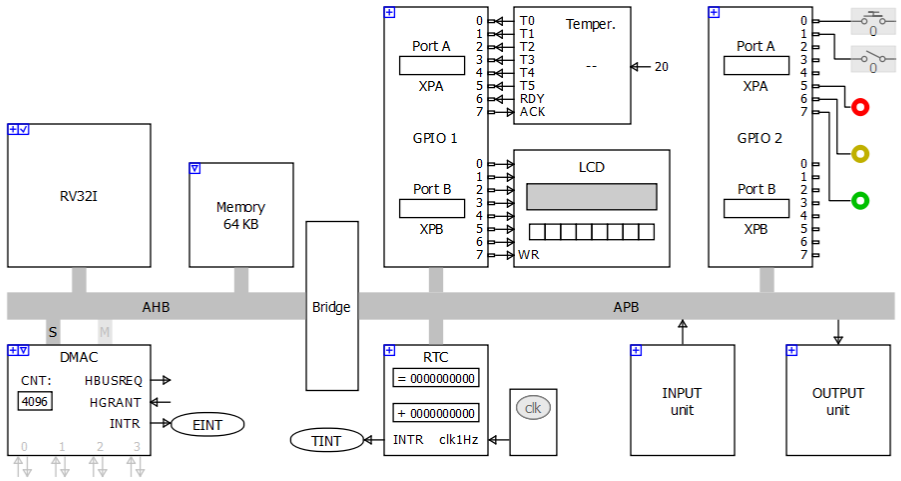


Slika 4.1. Stari model računalnog sustava s procesorom RISC-V

4.2. Nadogradnja modela

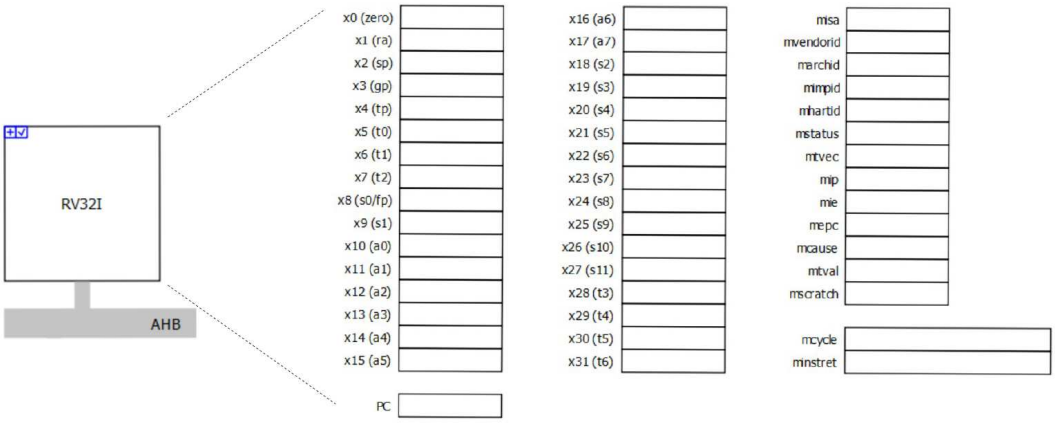
Slika 4.2. prikazuje izgled računalnog sustava nakon nadogradnje [16]. Računalni sustav, za razliku od prethodne verzije, sadrži sabirnicu AMBA (eng. *Advanced Microcontroller Bus Architecture*) pa je stoga periferni dio koji je sporiji odvojen na sabirnicu APB (eng.

Advanced Peripheral Bus), dok su glavni dijelovi računalnog sustava koji rade na višoj frekvenciji signala takta spojeni na sabirnicu AHB (eng. *Adavanced High-Performance Bus*). Ova dva dijela sustava razdvaja most koji upravlja pristupom između njih, omogućuje prijenos podataka i adresiranje vanjskih jedinica. Osim toga, model je proširen dodavanjem sklopa s neposrednim pristupom memoriji (eng. *direct memory access, DMA*) i ulaznom, odnosno izlaznom jedinicom. Uz to, jedinica za mjerenje vremena je promijenjena na način da je sada konfigurabilna.



Slika 4.2. Novi model računalnog sustava s procesorom RISC-V

Slika 4.3. prikazuje sve registre procesora. Uz 32 registra opće namjene i programsko brojilo, novi model procesora proširen je tako da sadrži i niz kontrolnih i statusnih registara prikazanih u desnom stupcu slike.



Slika 4.3. Registri procesora arhitekture RISC-V

Iako je ime komponente procesora "RV32I" što upućuje na to da procesor implementira samo osnovni skup 32-bitnih instrukcija, ovaj procesor uz njih implementira i instrukcije skupa Zicsr (`csrrw`, `csrrs`, `csrrc`, `csrrwi`, `csrrci`, `csrrsi`) i privilegirane instrukcije strojnog načina rada (`mret`) opisanih u poglavlju 2.

S obzirom na to da je strojni način rada M obavezan za implementaciju, isti je implementiran i u prikazanom procesoru.

Komponenta procesora, kao i cijeli model, napisani su u jeziku COMDEL. Svaka komponenta može definirati svoj unutarnji izgled i vanjski izgled. Slika 4.3. prikazuje vanjski i unutarnji prikaz komponente gdje je vanjski prikaz lijevi dio slike dok je unutarnji prikaz desni dio slike.

U novom modelu unutarnji izgled komponente procesora, točnije registri izmijenjeni su na način da je svaki od njih umjesto prezentacijskog elementa *rectangle* sada prezentacijski element *value*. Ta promjena omogućuje da je svaki upis vrijednosti u registar sada vidljiv označavanjem registra crvenom bojom te pruža označavanje nekorištenih registara sivom bojom što korisniku pruža zorniji prikaz o stanju i promjenama u modelu procesora. Ove promjene vidljive su i na slikama 4.5.-4.12.

Uz standardne instrukcije, prikazani procesor implementira i niz drugih instrukcija. Za zaustavljanje programa koristi se nestandardna pseudoinstrukcija `halt` koja se prevodi u skok na adresu 0x10000. S obzirom na to da je memorija veličine 64 kB, skok na adresu 0x10000 u uobičajenom izvođenju nije dozvoljen. Razlog ovakve implementacije je taj što će u sklopovskoj implementaciji modela na toj lokaciji biti potprogram koji će služiti za slanje vrijednosti svih registara opće namjene natrag u simulator, dok se za izvođenje na modelu u simulatoru ova konkretna instrukcija izvodi kao kraj izvođenja programa. Ta instrukcija omogućuje zaustavljanje rada simulacije. Uz nju, sustav implementira standardne RISC-V direktive poput `.word` ili `.org`, ali i njihove ekvivalente specifične za programski paket SSPARCSS. Primjer takve direktive je `dw` koja je ekvivalent standardne direktive `.word`. Procesor implementira i nekoliko desetaka pseudoinstrukcija, a neke od njih su `nop`, `neg`, `ble`, `j`, `csrr`, `ret` itd. [6].

Prilikom izvođenja programa u simulatoru, korisnik može pokrenuti izvođenje različitim brzinama, s ili bez animacije. Osim toga, korisnik kao i u svakom integriranom

radnom okruženju (eng. *Integrated Development Environment*, *IDE*) može izvoditi program liniju po liniju što u slučaju simulatora znači instrukciju po instrukciju. U dosadašnjem modelu bila je moguća jedino opcija *Step Into* čime nije bilo moguće preskočiti ili prijevremeno izaći iz potprograma. Stoga su opcije *Step Out* i *Step Over* dodane u model procesora. Ove opcije zbog prevođenja nekih od pseudoinstrukcija moguće je koristiti samo prilikom poziva i povratka iz potprograma s pomoću registra x1 koji prema specifikaciji služi za spremanje povratne adrese.

4.3. Prekidni sustav

Računalni sustav osim procesora i memorije ima niz vanjskih jedinica. Svaka vanjska jedinica koju koristi procesor odradit će neki posao poput prijenosa podataka, brojanja impulsa signala takta, A/D ili D/A pretvorbe te će završetak posla vanjska jedinica dojaviti procesoru. Procesor zatim rekonfigurira vanjsku jedinicu, obrađuje podatke iz vanjskog svijeta i sl. Procesor može poslužiti vanjsku jedinicu na dva načina, prozivanjem (eng. *polling*) ili prekidima (eng. *interrupt*).

Prozivanje funkcionira na način da procesor u petlji provjerava status vanjske jedinice te kada je ona završila svoj posao, procesor ju poslužuje. Lako je vidjeti kako je ovakav način izrazito neučinkovit jer procesor umjesto da obavlja neki drugi koristan posao, on provjerava kroz velik broj iteracija spremnost vanjske jedinice.

Rješenje za prethodni problem je prekidni način rada. On funkcionira tako da vanjska jedinica postavi prekid što će prekinuti normalan način rada procesora te će procesor započeti obradu prekida skokom na odgovarajući prekidni potprogram (eng. *interrupt service routine, ISR*). Procesor dok čeka na dojavu prekida može obavljati neki drugi koristan posao. Prekidi koje procesor prima mogu biti izazvani od vanjske jedinice, no mogu biti i programski uzrokovani, a primjer za to je sistemski poziv.

Prekidi s procesorom RISC-V

Procesori arhitekture RISC-V gotovo uvijek će implementirati neki oblik prekidnog sustava pa stoga postoji velika podrška za razne vrste prekidnih sustava, od najjednostavnijih bez prekidnog kontrolera do složenih s prekidnim kontrolerima za višeprocesorske i višedretvene sustave [17]. Na razini cjelokupng ekosustava procesora RISC-V podrška za

prekide još uvijek se proširuje, a dokaz za to je novi instrukcijski skup za nemaskirajuće prekide koji ne zaustavljaju rad procesora (eng. *Resumable Non-Maskable Interrupts*) Smrnmi [5].

U nastavku se razmatra osnovni prekidni sustav u strojnom načinu rada koji je implementiran u sklopu ovog rada za model u programskom paketu SSPARCSS.

Službena specifikacija prekidnog sustava procesora RISC-V definira potrebne registre i instrukcije za posluživanje prekida koji su opisani u ranijim poglavljima. Najjednostavniji prekidni sustav u strojnom načinu rada podrazumijeva tri prekida, programski (eng. *software interrupt*) kojeg može izazvati procesor, vremenski (eng. *timer interrupt*) kojeg može izazvati sklop kao što je RTC i vanjski (eng. *external interrupt*) kojeg može izazvati neka vanjska jedinica.

Slika 4.4. prikazuje sve podržane vrste prekida i iznimki (jednom riječju eng. *trap*) koje procesor arhitekture RISC-V može implementirati. Svi prekidi ne moraju biti jednako važni pa se tako za prekide definira prioritet. Prioritet prekida za procesore arhitekture RISC-V određen je kodom iznimke (eng. *Exception Code*) tako da je veći broj jednak većem prioritetu. Iznimka tog pravila je vremenski prekid koji ima manji prioritet od programskog prekida iako mu je kod iznimke veći.

Table 14. Machine cause register (mcause) values after trap.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12	*Reserved*
1	13	Counter-overflow interrupt
1	14-15	*Reserved*
1	≥16	*Designated for platform use*

0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16-17	*Reserved*
0	18	Software check
0	19	Hardware error
0	20-23	*Reserved*
0	24-31	*Designated for custom use*
	32-47	*Reserved*
	48-63	*Designated for custom use*
	≥64	*Reserved*

Slika 4.4. Vrste prekida i iznimki [5]

Obrada prekida

Pri kraju svake instrukcije procesor provjerava je li postavljen zahtjev na nekom od prekidnih signala. U algoritmu 1 danom u nastavku opisan je postupak prepoznavanja i obrade iznimke za strojni način rada. Najprije se provjerava jesu li prekidi globalno omogućeni nakon čega se provjerava je li postavljen zahtjev na nekom prekidnom signalu (zastavice `MTIP`, `MEIP`, `MSIP` u registru `mip`). Ako su svi uvjeti zadovoljeni, onemogućuju se daljnji prekidi te se programsko brojilo sprema u za to previđeni registar `mepc`. Na temelju zastavice `mtvec.MODE` određuje se adresa prekidnog potprograma. Varijabla *vrste_prekida* označava listu prekida koje podržava procesor sortiranu silazno po prioritetu. Izraz *izlaz_iz_petlje* označava da je pronađen prekid koji je postavio zahtjev, a koji je najvišeg prioriteta pa stoga nije potrebno provjeravati ostale vrste prekide.

ako $mstatus.MIE == 1$ **onda**
  za svaki $x \in vrste\_prekida$ **čini**
    ako $mip.MxIP == 1 \land mip.MxIE == 1$ **onda**
      $mstatus.MPIE \leftarrow mstatus.MIE$
      $mstatus.MIE \leftarrow 0$
      $mepc \leftarrow pc$
      $mcause \leftarrow intterupt \mid exception\_code$
      ako $mtvec.MODE == 1$ **onda**
        $pc \leftarrow mtvec.BASE + 4 * exception\_code$
      inače
        $pc \leftarrow mtvec.BASE$
      kraj
      $izlaz\_iz\_petlje$
    kraj
  kraj
kraj

Algoritam 1: Obrada prekida

Prekidna rutina

Prekidna rutina je potprogram u kojem se poslužuje prekid. S obzirom na to da specifikacija definira dva načina određivanja adrese prekidne rutine, prekidne rutine se ovisno o načinu određivanja adrese razlikuju. Dva načina su izravni i vektorski i prikazani su u algoritmu 1 (način se odabire ovisno o zastavici `mtvec.MODE`, ako je ona 1 tada se odabire

vektorski, a u suprotnom izravni način izračuna adrese).

U izravnom načinu prekidna rutina je uvijek na istoj, baznoj, adresi. Nakon skoka na prekidnu rutinu, unutar te rutine određuje se adresa potprograma za obradu specifičnog prekida na temelju sadržaja registra mcause. To je potrebno napraviti programski u prekidnoj rutini. U nastavku je dan popis dijelova odn. koraka prekidne rutine u vektorskom načinu koje bi svaki prekidni potprogram trebao imati. Rutina za izravni način bi na početku prema uzroku prekida skočila na odgovarajuću adresu u kojoj bi se posluživao prekid.

Prekidni potprogram:

1. Ako se koriste različiti pokazivači stoga, zamijeniti sp i mscratch
2. Spremiti kontekst
3. U slučaju gniježđenja prekida, spremiti registre mcause, mepc, mie
4. U slučaju gniježđenja promijeniti mie i mstatus.MIE tako da su omogućeni željeni prekidi postavljanjem odgovarajućih bitova
5. Posluživanje VJ
6. Vraćanje konteksta (i potencijalna zamjena sp i mscratch)
7. Vraćanje iz prekidnog potprograma (instrukcija mret)

Zadnja instrukcija svake prekidne rutine je mret. Ta instrukcija je već prikazana u poglavlju 2., a ovdje je dan pseudokod njene interne izvedbe.

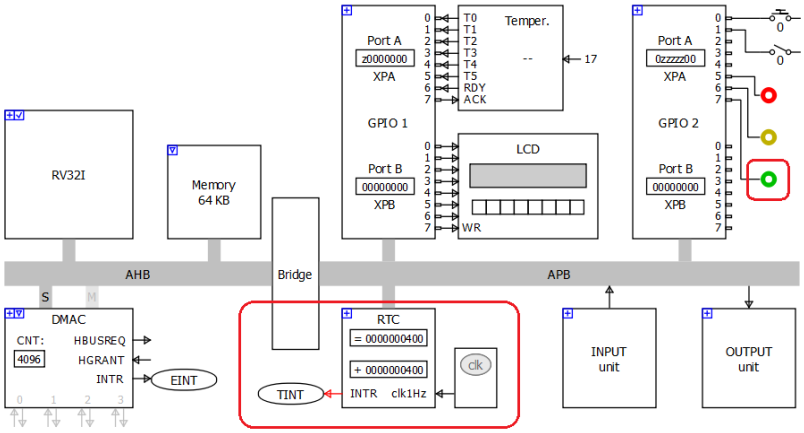
mret :
$mstatus.MIE \leftarrow mstatus.MPIE$
$pc \leftarrow mepc$

Implementacija

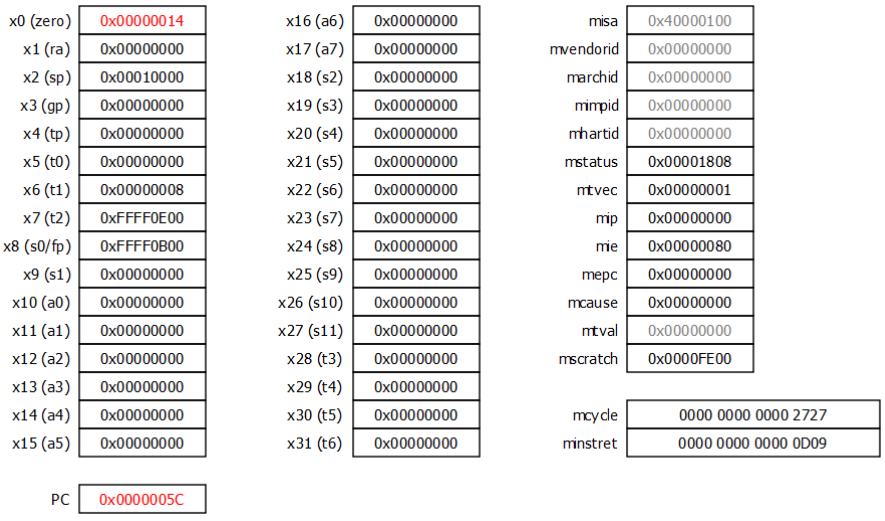
Prikazani prekidni sustav implementiran je u modelu računalnog sustava s procesorom arhitekture RISC-V za programski paket SSPARCSS. Prethodni model prikazan na slici 4.1. nije imao niti jednu vanjsku jedinicu koja bi mogla generirati zahtjev za prekid. U novom modelu sa slike 4.2. sklopovi DMA i RTC mogu generirati zahtjev za prekid.

Implementirani prekidni sustav podržava vanjski i vremenski prekid u strojnom načinu rada. Kao što je ranije opisano, vanjski prekid ima veći prioritet. Vrsta prekida koju vanjska jedinica generira prikazana je na slici 4.2. elipsama koje su spojene na komponente. Tako je sklop RTC spojen na vremenski, a sklop DMA na vanjski prekid.

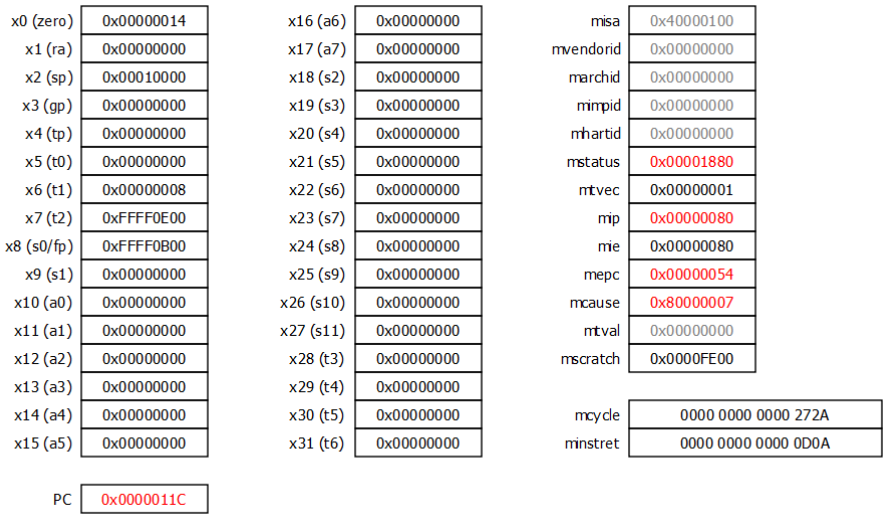
U privitku B je dan primjer programa koji koristi prekidni sustav, točnije vremenski prekid. U nastavku je prikazano izvođenje tog primjera u simulatoru SSPARCSS.



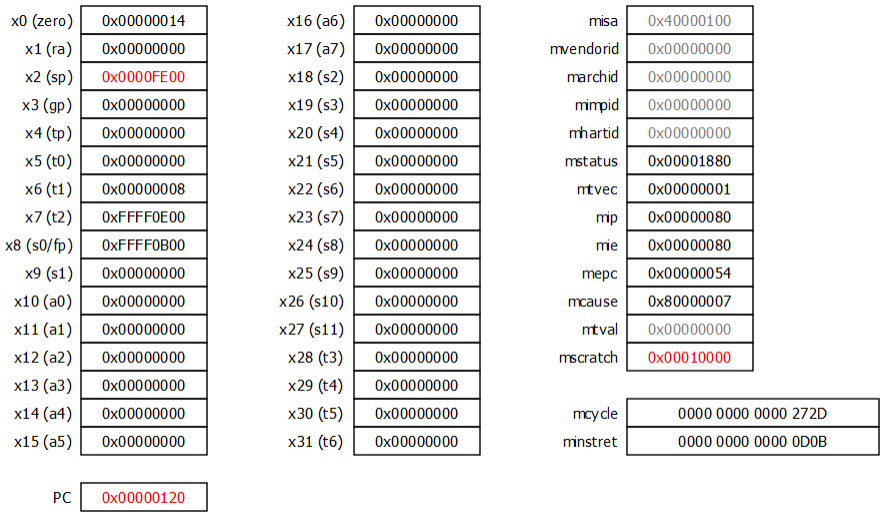
Slika 4.5. Sklop RTC je odbrojao do zadanog broja te postavlja zahtjev za prekid. Zelena svjetleća dioda je ugašena.



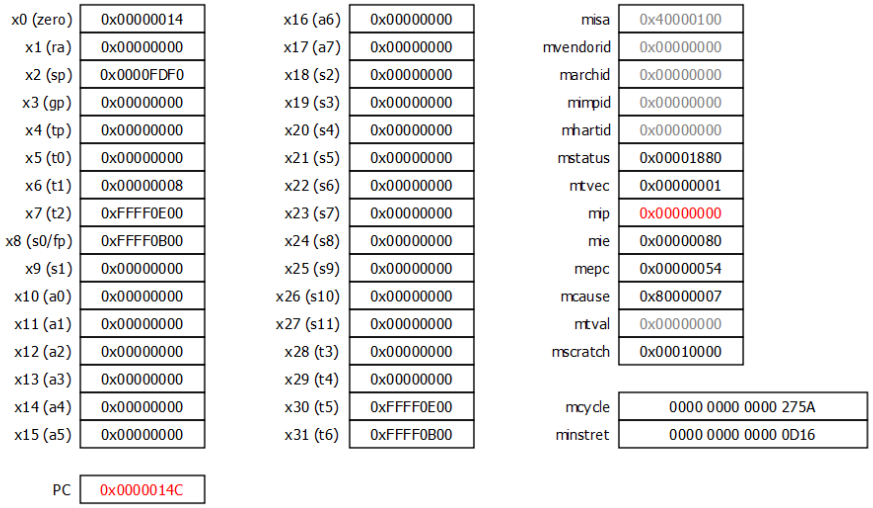
Slika 4.6. Prikaz stanja registara u trenutku prije postavljanja zahtjeva za prekid. Programsko brojilo ima vrijednost instrukcije u glavnom progamu.



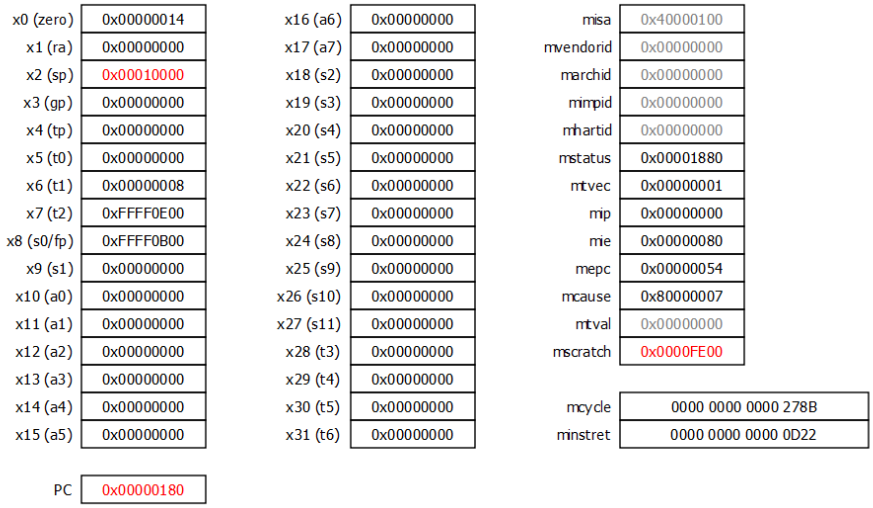
Slika 4.7. Prikaz stanja registara nakon ulaska u prekidnu rutinu. Sadržaj programskog brojila je promijenjen na adresu prekidnog potprograma, a staro stanje je upisano u registar mepc. U registru mip se vidi kako je vanjska jedinica postavila zahtjev za prekid. U registru mstatus zastavica MIE je prepisana u zastavicu MPIE. Uzrok prekida prema tablici na slici 4.4 vidljiv je u registru mcause.



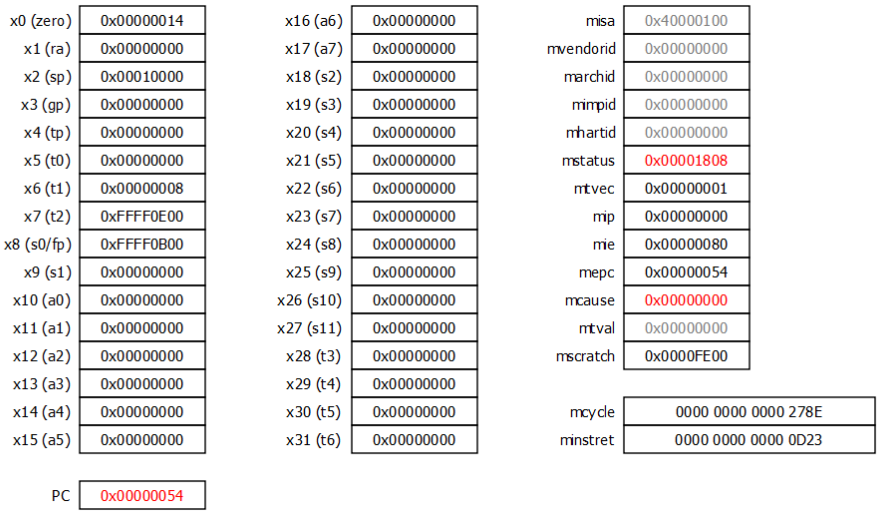
Slika 4.8. Stari pokazivač stoga je zamijenjen sadržajem registra mscratch.



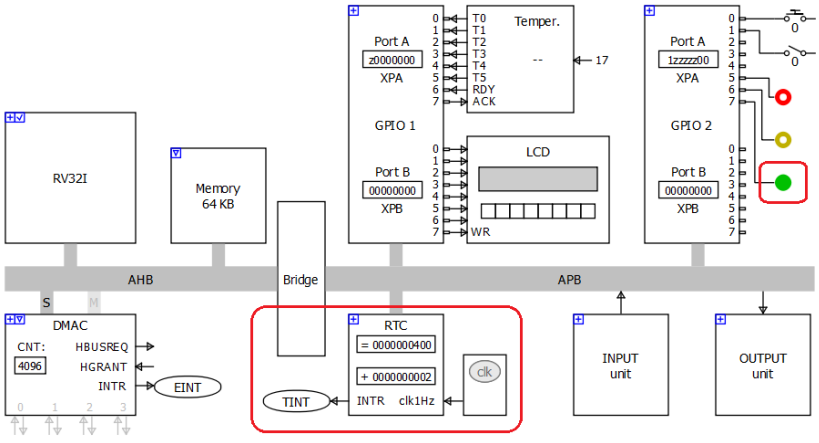
Slika 4.9. Nakon što se u odgovarajućem registru u sklopu RTC upiše podatak, zahtjev za prekid se ukida što je vidljivo u registru mip.



Slika 4.10. Pokazivači stoga su ponovno zamijenjeni.



Slika 4.11. Prikaz stanja registara pri izlasku iz prekidne rutine. Vrijednost programskog brojila je vraćena iz registra mepc. Zahtjev za prekid u registru mip je obrisan kao i uzrok prekida. Zastavica MPIE u registru mstatus je vraćena u zastavicu MIE.



Slika 4.12. Prikaz računalnog sustava nakon prekidne rutine. Svjetleća dioda je upaljena, a sklop RTC ponovno broji do zadane vrijednosti i zahtjev za prekid više nije postavljen.

5. Zaključak

U ovom radu predstavljen je model računalnog sustava s procesorom arhitekture RISC-V o programskom paketu SSPARCSS. Opisani su osnovni instrukcijski skup RV32I, instrukcijski skup za rad s kontrolnim i statusnim registrima Zicsr, instrukcije karakteristične za strojni način rada te dio direktiva i pseudoinstrukcija karakterističnih za arhitektutu RISC-V. Uz to, opisani su načini rada i dio kontrolnih i statusnih registara..

Ukratko je opisan programski paket SSPARCSS te je napravljeno verifikacijsko okruženje. Verifikacijsko okruženje kao referentni model ima programski paket SSPARCSS u kojem programi asembler i simulator rade bez korištenje grafičkog korisničkog sučelja. Za ispitivanje ispravnosti rada verifikacijskog okruženja napravljen je niz testnih programa.

Trenutni model računalnog sustava temeljenog na arhitekturi RISC-V korišten za verifikacijsko okruženje nadograđen je te sada procesor pokriva veći skup instrukcija i pseudoinstrukcija te je reprezentacija procesora, točnije njegovih registara sada vizualno pristupačnija krajnjem korisniku. Model uz to sada u potpunosti implementira prekidni sustav te je dan primjer programa te prikaz izvođenja.

U budućnosti verifikacijsko okruženje može biti prošireno te metoda usporedbe rješenja može biti naprednija. Model sustava za SSPARCSS može se proširiti dodavanjem novih vanjskih jedinica, protočne strukture, implementiranjem drugim instrukcijskih skupova i sl.

Literatura

[1] J. L. Hennessy i D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6. izd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[2] E. Cui, T. Li, i Q. Wei, "Risc-v instruction set architecture extensions: A survey", *IEEE Access*, sv. 11, str. 24 696–24 711, 2023. https://doi.org/10.1109/ACCESS.2023.3246491

[3] *All Specifications Under Development*. [Mrežno]. Adresa: https://wiki.riscv.org/display/HOME/All+Specifications+Under+Development

[4] *RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*, 2024. [Mrežno]. Adresa: https://github.com/riscv/riscv-isa-manual

[5] *RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, 2024. [Mrežno]. Adresa: https://github.com/riscv/riscv-isa-manual

[6] *RISC-V Assembly Programmer's Manual*, 2024. [Mrežno]. Adresa: https://github.com/riscv-non-isa/riscv-asm-manual

[7] prof. dr. sc. Danko Basch, *Korisnička dokumentacija SSPARCSS simulatora, v2.0*, 2022. [Mrežno]. Adresa: https://www.fer.unizg.hr/_download/repository/SIMULATOR_-_USER_MANUAL_-_2022.pdf

[8] ——, *Korisnička dokumentacija SSPARCSS asemblera, v1.0*, 2022. [Mrežno]. Adresa: https://www.fer.unizg.hr/_download/repository/ASEMBLER_-_manual_v.1.0.pdf

[9] ——, *Konfigurabilni asemblerski prevoditelj i metajezik ADEL v.3.1 namijenjen definiranju mnemoničkog jezika*, 2024.

[10] ——, *Modelacijski jezik COMDEL2*, 2024.

[11] S. Šarić, *Unaprjeđenje edukacijskog simulacijskog paketa SSPARCSS za automatizirano izvođenje simulacija bez korištenja grafičkog korisničkog sučelja*, Diplomski rad, Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva, Zagreb, 2024.

[12] S. J. Davidmann, L. Moore, R. Ho, T. Liu, D. Letcher, i A. Sutton, "Rolling the dice with random instructions is the safe bet on risc-v verification", 2020. [Mrežno]. Adresa: https://api.semanticscholar.org/CorpusID:250724831

[13] A. Oleksiak, S. Cieślak, K. Marcinek, i W. A. Pleskacz, "Design and verification environment for risc-v processor cores", u *2019 MIXDES - 26th International Conference "Mixed Design of Integrated Circuits and Systems"*, 2019., str. 206–209. https://doi.org/10.23919/MIXDES.2019.8787108

[14] *RISC-V Formal Verification Framework*. [Mrežno]. Adresa: https://github.com/YosysHQ/riscv-formal

[15] *RISC-V Architecture Test SIG*, 2024. [Mrežno]. Adresa: https://github.com/riscv-non-isa/riscv-arch-test

[16] L. Grubišin, *Verifikacijsko okruženje za oblikovanje računalnog sustava temeljenog na procesoru arhitekture RISC-V*, Završni rad, Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva, Zagreb, 2024.

[17] R. Balas, A. Ottaviano, i L. Benini, "Cv32rt: Enabling fast interrupt and context switching for risc-v microcontrollers", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, sv. 32, br. 06, str. 1032–1044, jun 2024. https://doi.org/10.1109/TVLSI.2024.3377130

Sažetak

Verifikacijsko okruženje za oblikovanje računalnog sustava temeljenog na procesoru arhitekture RISC-V

Vinko Đurić

Pri razvijanju i nadogradnji računalnog sustava važno je ispitati ispravnost rada i usklađenost s očekivanim ponašanjem. Iz tog razloga se za računalni sustav provodi verifikacija. U ovom radu razvijeno je verifikacijsko okruženje temeljeno na programskom paketu SSPARCSS kao referentnom modelu. Postojeći model računalnog sustava s procesorom arhitekture RISC-V vizualno je unaprijeđen te je proširen dodavanjem i ispravljanjem postojećih instrukcija, pseudoinstrukcija i direktiva te implementacijom prekidnog sustava. U radu su opisani načini rada i instrukcijski skupovi arhitekture RISC-V.

Ključne riječi: RISC-V, verifikacija, SSPARCSS, COMDEL, ADEL, arhitektura računala

Abstract

Development of verification environment for the development of computing system based on RISC-V ISA

Vinko Đurić

When developing and upgrading a computer system, it is important to check its correctness and compliance with the expected behavior. For this reason, verification is performed for the computer system. In this work, a verification environment based on the SSPARCSS software package as a reference model was developed. The existing model of a computer system with a RISC-V processor was visually enhanced, extended by adding and correcting existing instructions, pseudo-instructions and directives and by implementing an interrupt system. Thesis describes privilege modes and instruction sets of the RISC-V architecture.

Keywords: RISC-V, verification, SSPARCSS, COMDEL, ADEL, computer architecture

Privitak A:
Pokretanje verifikacijskog okruženja

S obzirom na to da verifikacijsko okruženje radi s verzijom programskog paketa SSPARCSS bez GUI-ja, najprije je potrebno podesiti takav SSPARCSS. U najnovijoj instalaciji Qt-a više ne postoji potrebna verzija za izvođenje simulatora i asemblera. Stoga je potrebno preuzeti offline instalaciju (i odabrati instalaciju za odgovarajući OS) te u sklopu nje skinuti Qt 5.12.2 i MinGW 7.3.0-32bit. Nakon instalacije potrebno je pronaći odgovarajuće .pro datoteke (Assembly_GUI.pro i atlas.pro) i ubaciti ih u Qt Creator. Qt Creator koji se dobije instalacijom je starije verzije, no radi ispravno. Najprije je potrebno buildati Assembly_GUI. Zatim je u projektu atlas potrebno pronaći WithoutGUI.cpp i na 138. liniji upisati mjesto datoteke Assembly_GUI.exe koja je dobivena iz projekta Assembly_GUI. Za pokretanje simulatora bez GUI-ja potrebno je pokrenuti *build* projekta atlas i u Projects > Run dodati argumente: „[SYSTEM file]„ „[A file]„ „[TIP PROCESORA]„ „[INPUT JSON]„. Pokretanjem sve bi trebalo ispravno raditi i u direktoriju u kojem se nalazi input json trebao bi se generirati imena „[A file ime].json“.

Za rad verifikacijskog okruženja potrebno je omogućiti pokretanje simulatora iz naredbenog retka. To je moguće dodavanjem direktorija s datotekama ekstenzije **.dll* (oblika C:/Qt/5.12.2/mingw73_32/bin) u varijable okruženja.

U nastavku je dan primjer lokalnog pokretanja verifikacijskog okruženja:

```
python verif_env.py --type eval_single --simulator C:/
    FERV/build-atlas-Desktop_Qt_5_12_2_MinGW_32_bit-Debug/
    simulator_gui/debug/atlas.exe --task C:\Users\vinko\
    OneDrive\Dokumenti\SSPARCSS_testiranje\zbirka_p2\
    zadatak-2-3-13\zadatak-2-3-13.a --input C:\Users\vinko\
    OneDrive\Dokumenti\SSPARCSS_testiranje\zbirka_p2\
    zadatak-2-3-13\zadatak-2-3-13-input.json --ref C:\Users
    \vinko\OneDrive\Dokumenti\SSPARCSS_testiranje\zbirka_p2
    \zadatak-2-3-13\zadatak-2-3-13-ref.json

python verif_env.py --type eval_same --input C:\Users\
    vinko\OneDrive\Dokumenti\SSPARCSS_testiranje\test-
    lcd_puno\test-lcd-input.json --ref C:\Users\vinko\
    OneDrive\Dokumenti\SSPARCSS_testiranje\test-lcd_puno\
    test-lcd.json --simulator C:\FERV\build-atlas-
    Desktop_Qt_5_12_2_MinGW_32_bit-Debug\simulator_gui\
    debug\atlas.exe --task C:\Users\vinko\OneDrive\
    Dokumenti\SSPARCSS_testiranje\test-lcd_puno

python verif_env.py --type eval_diff --simulator C:/FERV/
    build-atlas-Desktop_Qt_5_12_2_MinGW_32_bit-Debug/
    simulator_gui/debug/atlas.exe --task C:\Users\vinko\
    OneDrive\Dokumenti\SSPARCSS_testiranje\zbirka_p2

python verif_env.py --type gen --simulator C:/FERV/build-
    atlas-Desktop_Qt_5_12_2_MinGW_32_bit-Debug/
    simulator_gui/debug/atlas.exe --task C:\Users\vinko\
    OneDrive\Dokumenti\SSPARCSS_testiranje\zbirka_p2
```

Privitak B:
Primjer programa s prekidom

```
; program u beskonacnoj petlji koji nakon sto rtc odbroji do 400
; i aktivira timer prekid promijeni (toggle) stanje zelene led lampice
RTC     equ     0xFFFF0E00
GPIO2   equ     0xFFFF0B00

        org     0
GLAVNI  lui     sp, %hi(0xFE00)
        addi    sp, sp, %lo(0xFE00)
        csrrw   x0, mscratch, sp        ; spremanje adrese sp za prekide
        addi    sp, sp, 0x200

        lui     x7, %hi(RTC)
        addi    x7, x7, %lo(RTC)
        lui     x8, %hi(GPIO2)
        addi    x8, x8, %lo(GPIO2)

        addi    x6, x0, 1
        sw      x6, 16(x7)              ; omogucavanje prekida
        addi    x6, x0, 400
        sw      x6, 4(x7)               ; postavljanje MR
        sw      x0, 12(x7)
        addi    x6, x0, 0x80
        sw      x6, 8(x8)               ; postavljanje smjera gpio2 vrata A
```

```
        addi    x6, x0, 0b01            ; maska za vectored mode
        csrrw   x0, mtvec, x6           ; odabir vectored moda prekida
        addi    x6, x0, 0b10000000      ; maska za omogucavanje timer prekida
        csrrw   x0, mie, x6             ; omogucavanje timer prekida
        addi    x6, x0, 0b1000          ; maska za omogucavanje globalnih prekida
        csrrw   x0, mstatus, x6         ; omogucavanje globalnih prekida

POSAO   addi    x0, x0, 5
        addi    x0, x0, 20
        beq     x0, x0, POSAO

        halt

        org     0x11C
        csrrw   sp, mscratch, sp        ; zamjena pokazivaca stoga
        addi    sp, sp, -16
        sw      x28, 0(sp)
        sw      x29, 4(sp)
        sw      x30, 8(sp)
        sw      x31, 12(sp)

        lui     x30, %hi(RTC)
        addi    x30, x30, %lo(RTC)
        lui     x31, %hi(GPIO2)
        addi    x31, x31, %lo(GPIO2)

        sw      x0, 8(x30)
        sw      x0, 12(x30)

        lw      x29, 0(x31)
        sltiu   x30, x29, 0x80
        beq     x30, x0, UGASI
```

```
        addi    x28, x0, 0x80
        beq     x0, x0, SPREMI
UGASI   addi    x28, x0, 0x00
SPREMI  sw      x28, 0(x31)

        lw      x28, 0(sp)
        lw      x29, 4(sp)
        lw      x30, 8(sp)
        lw      x31, 12(sp)
        addi    sp, sp, 16
        csrrw   sp, mscratch, sp        ; zamjena pokazivaca stoga
        mret                            ; povratak iz prekidne rutine
                                        ; u glavni program
```