TARTU ÜLIKOOL

Loodus- ja täppisteaduste valdkond

Tehnoloogiainstituut

Peeter Virk

RISC-V mikrokontrolleri simulation Logisim Evolution programmis

Bakalaureusetöö (12 EAP)

Arvutitehnika õppekava

Juhendaja:

Margus Rosin, MSc

Tartu 2025

Lühikokkuvõte

RISC-V mikrokontrolleri simulation Logisim Evolution programmis

Mikrokontrollerid on arvutisüsteemid, mis võimaldavad erinevate seadmete tööd alates koduelektroonikast kuni tööstusautomaatikani. Nende sisemise ülesehituse põhjalik mõistmine võimaldab programmeerijatel luua tõhusamat ja optimeeritumat koodi. Käesoleva bakalaureusetöö eesmärk on disainida RISC-V arhitektuuril põhinev mikrokontrolleri simulatsioon Logisim Evolution programmis – tööriistas, mis võimaldab digitaalsete loogikaskeemide loomist ja testimist. Töö ühendab teoreetilised teadmised ja praktilise rakenduse, pakkudes õppuritele ja arendajatele võimalust mõista mikrokontrolleri ehitust süvitsi. Simulatsiooni loomise kaudu antakse panus hariduslike vahendite arendamisse ning mikrokontrollerite ja teiste arvutitehnoloogiate tõhusamasse kasutusse.

CERCS: T120 Süsteemitehnoloogia, arvutitehnoloogia

Märksõnad: Mikrokontrollerid, RISC-V, simulatsioon, Hariduslikud tööriistad

Abstract

RISC-V microcontroller simulation in Logisim Evolution program

Microcontrollers are pivotal in enabling computing for devices and facilitating the existence of most modern devices, from home appliances to industrial machines. A deep understanding of the inner workings of computing devices is crucial for programmers looking to improve their code, as it significantly enhances their ability to design efficient software. This thesis explores the simulation of a RISC-V-based microcontroller using the Logisim Evolution program, a versatile and powerful tool for digital circuit design and simulation. This work aims to bridge the gap between theoretical concepts and practical implementation by simulating a possible microcontroller design, adding to the knowledge in the field and allowing for improved usage of microcontrollers and other computing technologies.

CERCS: T120 Systems engineering, computer technology

Keywords: Microcontrollers, RISC-V, Simulation, Educational tools

Sisukord

Lühikokkuvõte **2**

Abstract **3**

Lühendid, konstandid, mõisted **8**

1 Sissejuhatus **9**

2 Kirjanduse ülevaade **10**

2.1 Logisim-evolution 10

2.2 Mikrokontrollerid 10

2.2.1 Välisseadmete juhtimine 11

2.2.2 Käskude torustik 11

2.3 Mälu struktuurid 12

2.4 RISC-V 13

2.4.1 RV32I 13

2.4.2 RV32I käskude põhivormingud 13

2.4.3 RV32I käskude erivormingud 15

2.4.4 RV32M 15

3 Simulatsiooni arhitektuur ja teostus **17**

3.1 Protsessor 20

3.1.1 Programmiloendur 22

3.1.2 Käsu dekooder 23

3.1.3 Registrid 24

3.1.4 ALU 25

3.1.5 Erindid, katkestused ja lõksud 27

3.1.6 Juhtimis- ja olekuregistrid (CSR) 27

3.2 Mälu 28

3.3 Mikrokontrolleri välisliides 28

3.3.1 GPIO 29

3.3.2 Taimer 30

3.3.3 Enkoodrimoodul 31

4 Testimine **32**

4.1 RV32IM käskude programmaatiline testimine 33

4.2 Juhtimis- ja olekuregistrite testimine 34

4.3 Mälu ja mälule kaardistatud I/O testimine 34

4.4 Testide kaetavus 35
4.5 Testide tulemused 36

5 Kokkuvõte **37**

viited **38**

Lisad **40**
Lisa 1. Simulatsiooni failid 40
Lisa 2. Mikrokontrolleri registrite skeem 41
Lisa 3. Implementeeritud protsessori käskude tabelid 42

II. Litsents **45**

Joonised

1 RISC-V käskude kodeerimisvarjandid [22] 14
2 Lahenduse skeemide ja alamskeemide hierarhiline ülesehitus 17
3 Mikrokontrolleri skeemi ülevaade 18
4 Mikrokontrolleri I/O 19
5 Protsessori skeem 20
6 Protsessori skeemi komponendid – käsu dekodeerija, registrifail ja ALU 21
7 Protsessori skeemi mäluga suhtlemise alamosa. 22
8 Programmiloendur 23
9 Protsessori võrdlemissüsteem 23
10 Käsu dekodeerimismooduli näide add käsu näitel 24
11 ALU loogikaskeem 26
12 GPIO register 29
13 Enkoodrimooduli loogikaskeem 31
14 Koodi laadimine mikrokontrolleri programmimällu Logisim Evolution programmis 32
15 Mikrokontrolleri registrid 41

Tabelid

1 Mikrokontrolleri kirjutatav mäluruum 28
2 R-tüüpi käsud [22] 42
3 I-tüüpi aritmeetikakäsud [22] 43
4 CSR registritel opereerivad käsud [22] 43
5 Mälu käsud [22] 44
6 Haru- ja hüppekäsud [22] 44
7 RV32I U-tüüpi käsud [22] 44
8 Katkestuste ja lõksude käsud [22, 21] 44

Lühendid, konstandid, mõisted

MCU — *microcontroller unit*, mikrokontroller

ISA — *Instruction set architecture*, käsustik

CSR — *Control and Status Registers*, kontrolli ja oleku register

ALU — *Arithmetic logic unit*, aritmeetika-loogikaplokk

RISC — *Reduced instruction set computer*, vähendatud juhiskomplektiga arvuti

GPIO — *General-purpose input/output*, mitmeotstarbeline sisend-väljund

1 Sissejuhatus

Selle bakalaureusetöö eesmärk on luua RISC-V-põhine mikrokontrolleri simulatsioon, mis võimaldab tulevastel arvutitehnikutel ja huvitatud isikutel tutvuda selle tehnoloogiaga. Mikrokontrolleri simulatsiooni uurimine aitab paremini mõista, kuidas riistvaraline mikrokontroller töötab.

Tänapäeva tehnoloogia tugineb suuresti mikrokontrolleritele, mida leidub peaaegu igas seadmes [16]. Hoolimata laialdasest kasutusest on enamasti üldkasutatavad käsustikud ühe ettevõtte omanduses, mis piirab uute mikrokontrollerite disainimist litsentsidega, muutes selle suurettevõtete tegevuseks. Avatud käsustikud, nagu RISC-V, võimaldavad laiemal ringil tegeleda mikrokontrollerite arendusega. [1]

2 Kirjanduse ülevaade

2.1 Logisim-evolution

Logisim Evolution on Javal põhinev tarkvara digitaalse loogika disaini ja simuleerimise jaoks, mis on edasiarendus Logisim programmist. Logisim võimaldab neil, kes pole digitaalse loogikaga varem kokku puutunud, läheneda sellele visuaalsel kujul, näidates reaalajas, kuidas erinevad loogikaväravad mõjutavad skeemide käitumist. Logisimi lihtne ja kasutajasõbralik liides võimaldab keskenduda sisule, säästes aega uue tööriista õppimise arvelt. [4, 5]

2.2 Mikrokontrollerid

Mikrokontrollerid on oma olemuselt väikesed arvutisüsteemid, mis tavaliselt sisaldavad protsessorit, programmi- ja muutmälu ning mitmeid erinevaid komponente, mis on mõeldud protsessori töö kiirendamiseks või suhtluseks välismaailmaga, sealhulgas GPIO-porte, taimerite ja kommunikatsiooniliideseid. [3]

Mikrokontrollereid on võimalik kasutada erinevates situatsioonides ilma, et oleks vaja muuta integraalskeemi sisu. See võimekus tuleneb mikrokontrolleri sisseehitatud protsessorist, mis võimaldab programmeerijal määratleda seadme tegevuse vastavalt kasutusviisi vajadustele. [3]

Mikrokontrolleris sisalduvad komponendid rikastavad selle funktsionaalsust, aidates kaasa süsteemi terviklikkusele ning võimaldades protsessoritsüklite efektiivsemat kasutamist. Need komponendid võivad teostada oma tööd nii sisemiselt kui ka väliselt. Sisemiselt on võimalik kasutada taimereid, mis võimaldavad kindlate kellatsüklite tagant. Väliselt on võimalik juhtida mikrokontrolleri väljundeid kindlal viisil, mis võimaldab kontrollida või saata andmeid mikrokontrolleri välistele seadmetele. [3]

GPIO on kõige lihtsam viis, kuidas mikrokontroller saab oma väljundeid kontrollida. Protsessor saab määrata iga GPIO-jala töörežiimi — sisendiks või väljundiks. Väljundi korral saab protsessor korraga väljastada ühe kahest *boolean*-väärtusest. Sisendi korral tõlgib GPIO moodul sisendsignaali üheks *boolean*-väärtuseks ning võimaldab mikrokontrolleril seda lugeda. Taimerid on võimelised genereerima kellasignaali põhjal erinevaid signaale välisühendustele. Kommuni-

katsiooniliidesed võimaldavad mikrokontrolleri protsessoril suhelda väliste seadmetega, samal ajal vähendades protsessori koormust, kuna suhtlus toimub läbi riistvaraliste liideste, erinevalt tarkvaralisest emuleerimisest. [2, 3]

2.2.1 Välisseadmete juhtimine

Välisseadmete juhtimiseks peab protsessor olema võimeline nendega andmeid vahetama. Üheks lahenduseks on I/O seadmete kontrollregistrite kaardistamine mäluaadressiruumi. [19] Seda meetodit kasutatakse laialdaselt näiteks mikrokontrollerites, nagu ATmega328P [10].

Alternatiivina on võimalik kasutada eraldiseisvat I/O aadressiruumi, mille kaudu protsessor suhtleb välisseadmetega. Selle lähenemise eeliseks on mäluaadressiruumi kokkuhoid ning selge piirjoone tõmbamine mälu ja I/O seadmete vahel. Samas nõuab see protsessorilt spetsiaalsete käskude olemasolu, mis võimaldavad I/O aadressidele ligi pääseda ja neid juhtida. Selline kontrollskeem on kasutusel näiteks x86-arhitektuuriga seadmetes. [13]

2.2.2 Käskude torustik

Käskude torustik on süsteem, mis võimaldab protsessorituumal töötada samaaegselt mitme käsuga, jagades käsu täitmise mitmeks järjestikuseks staadiumiks. Iga staadium töötleb käsu teatud osa, mis suurendab protsessori efektiivsust. [18] Torujuhtme astmete arv sõltub konkreetse mikrokontrolleri teostusest, kuid teatud etapid on laialdaselt kasutusel mitmetes protsessorites [22, 18].

Ühesammulistes protsessorites täidetakse käsk üheainsa sammuga, mille jooksul viiakse lõpule kõik käsu täitmiseks vajalikud tegevused. See vähendab nii protsessori disaini kui ka koodi käitamise keerukust, kuna võib eeldada, et kõik andmed registrites ja mälus on käsu täitmise hetkeks ajakohased. Kuid see lähenemine on ebaefektiivne nii loogika kui ka ajakasutuse mõttes, kuna aeglaseimad käsud määravad kogu protsessori taktisageduse ning käskude täitmist ei ole võimalik väiksemateks osadeks jagada. [18]

Mitmeastmeline torustik parandab seda puudust, võimaldades mitmel käsul samaaegselt protsessori eri staadiume kasutada. Selle tulemusel võtab iga käsu täitmine kokku rohkem takte

(vastavalt torujuhtme sammude arvule), kuid tänu paremale paralleelsusele toimub programmi täitmine kiiremini kui ühesammulises protsessoris. [18]

Ideaaljuhul suudab selline torujuhe täita keskmiselt ühe käsu iga taktsükliga. Tegelikkuses ei ole see alati võimalik. Programmi käivitamisel ja peatamisel tuleb torujuhe esmalt täita ja hiljem tühjendada, mis vähendab efektiivsust. Samuti on oluline tagada, et käsu täitmiseks vajalik informatsioon oleks protsessoris saadaval. [18, 11]

Kui vajalikud andmed ei ole kohe kättesaadavad, peab protsessor ootama, kuni need muutuvad kättesaadavaks. Üheks lahenduseks on lisada torujuhtmesse NOP-käske, kuni vajalik informatsioon jõuab registrisse. Alternatiivina saab torujuhtme puhastada või kasutada edastamist (*forwarding*), mis võimaldab andmeid kasutada juba enne, kui need registrisse jõuavad, vähendades seeläbi viivitusi ja suurendades jõudlust. [18, 11]

2.3 Mälu struktuurid

Mikrokontrollerite mälustruktuurid saab jagada kaheks suureks arhitektuuriks. Von Neumanni arhitektuuris jagavad andme- ja programmimälu ühte ja sama mäluruumi. Harvardi arhitektuuris on need aga jagatud kaheks eraldi mäluruumiks, võimaldades samaaegset andmete kirjutamist ja käskude lugemist. [17] Kuigi tänapäevased arvutid kasutavad peamiselt Von Neumanni arhitektuuri, kasutatakse mitmetes protsessorites L1 *cache'i* tasemel Harvardi stiilis mäluarhitektuuri. [17, 23]

Enamasti kasutavad mikrokontrollerid Harvardi mäluarhitektuuri, nagu seda on kasutatud ATmega328P puhul, mis jagab mäluruumi kaheks, tehes andmete ja käskude lugemise tõhusamaks [17, 10]. Samas ei ole kõik mikrokontrollerid ehitatud Harvardi arhitektuuri peale – eksisteerivad ka Von Neumanni arhitektuuri kasutavad mikrokontrollerid nagu seda on ESP32. [14, 15]

Kuna RISC-V puhul on tegu laadimis-salvestamisarhitektuuriga (*load-store architecture*), sisaldab baaskäsustik vastavaid käske mäluga suhtlemiseks. RV32I defineerib selle jaoks kaheksa käsku, mis võimaldavad andmete lugemist või salvestamist mäluruumis 32, 16 ja 8 bitti kaupa. Iga väärtuse pikkuse salvestamiseks eksisteerib eraldi käsk. Lugemiskäske on implementeeritud viis, kuna lugemisel võib esineda nii märgiga kui ka märgita andmeid. 32-bitise väärtuse lugemisel pole oluline, millisel kujul on väärtus, kuna loetavate andmete pikkus täidab registri. Lühema märgiga väärtuse lugemisel mäluruumist tuleb see pikendada 32 bittini, kasutades mär-

gipikendust (inglise keeles: *sign extension*). Samuti kui tegu on märgita arvuga, tuleb kasutada nullpikendust (inglise keeles: *zero extension*). [22]

2.4 RISC-V

RISC-V on käsustiku arhidektuur, mis on jagatud mitmeks väiksemaks käsustikuks mida on võimalik implementeerida vastavalt seadme vajadustele. Need osad defineerivad mitmeid väiksemaid käskude komplekte, millest igaüks võimaldab teatud omadusi, kuid pole iga tuuma korral kohustuslikud. RISC-V toetab 32-, 64- ja 128-bitist arhitektuuri. Samuti eksisteerib kitsendatud versioon baaskäskude kogumist, mis vähendab registrite kogust 16-le tavapärase 32 asemel ja piirab käskude arvu, kuid samaaegselt kasutab ühe käsu jaoks 16 bitti. [22]

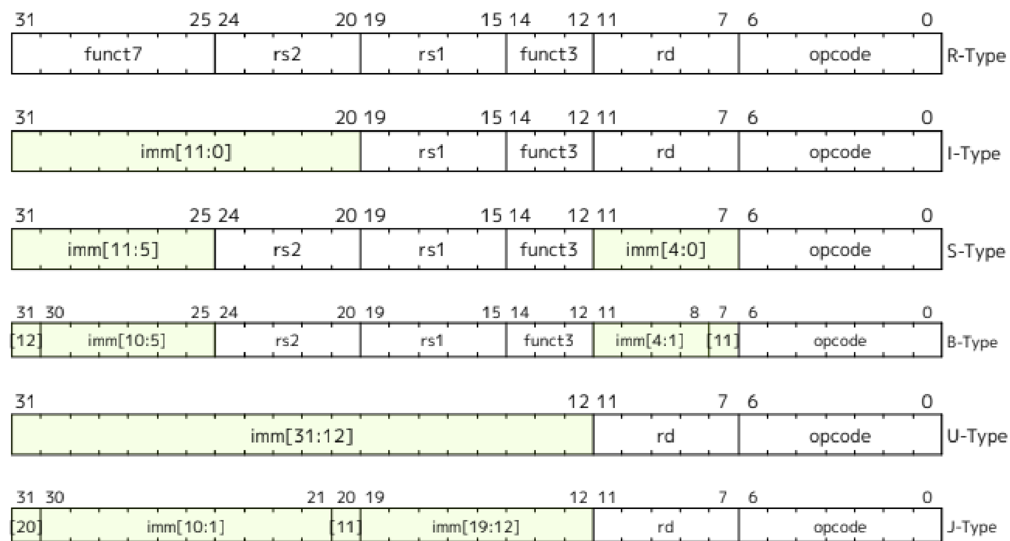
Igal RISC-V protsessoril on baaskäsustik, millele lisaks võib tuuma võimekust laiendada laiendustega, mis täiendavad protsessori funktsionaalsust, kuid nende olemasolu ei ole kohustuslik. Need laiendused võivad olla kas käsustiku juhendis defineeritud standardlaiendused või mittestandardsed laiendused, mille on loonud riistvara disainer. [22]

2.4.1 RV32I

RV32I on RISC-V baaskäsustik, mis sisaldab endas kuni 40 käsku, jaotatud kuue kodeerimisvariandi vahel. R-tüüpi kodeeritud käsud teostavad aritmeetilisi operatsioone kahe registri väärtuste vahel ilma kaasa antavat väärtust kasutamata. Joonisel 1 on näha, et I-, S-, B-, U- ja J-tüüpi kodeeringud sisaldavad kõik otsese väärtuse välja, mis võimaldab programmeerijal iga sellise käsuga kaasa anda teatud hulga eelnevalt määratud informatsiooni, mida protsessor saab käsku töödeldes kasutada. [22]

2.4.2 RV32I käskude põhivormingud

RV32I sisaldab nelja peamist käsu kodeeringu formaati mis defineerivad ära registrite aadressite, otseselt sisestatud väärtuste ning käske kodeerivate asukohad ja pikkused [22].



Joonis 1. RISC-V käskude kodeerimisvarjandid [22]

R-tüüpi (registritevahelised) käsud on ainsad käsustikus defineeritud käsud, mis kasutavad kolme registrit – kahte sisendi ja ühte väljundi jaoks. Seda tüüpi käsus on üldjuhul mõeldud aritmeetika ja loogika tehete jaoks. Siiski on sellel kodeeringul teatud piirangud. Nimelt ei ole võimalik seda tüüpi käskudega anda otsest sisendit ja seetõttu ei sobi see kõigiks otstarveteks. [22]

I-tüüpi (*immediate*, sisendiga) kodeeringuga on defineeritud enamik sisendit vajavaid käske nagu aritmeetika ja mälust laadimise käsud. Samuti on I-tüübi põhjal defineeritud süsteemikäsud nagu ECALL, EBREAK ning hüppekäsk JALR (*Jump and link register*). [22]

S-tüüpi (*store*, salvesta) käsud salvestavad informatsiooni mällu. Erinevalt I tüüpi käskudest, nagu näidatud joonisel 1, kasutab see kodeeringu tüüp sihtregistrit teise sisendregistri asemel. See võimaldab kasutada seda tüüpi kodeeringuga käske olukordades, kus ei ole vaja registritesse midagi salvestada, nagu selleks on andmemällu informatsiooni salvestamise käsud. [22]

U-tüüpi (*upper immediate*, ülemsisend) käske kasutatakse ülemiste bittide sisestamiseks koodi. Jooniselt 1 on näha, et U-tüüpi kodeering võimaldab edastada mikrokontrollerile ülemist osa muutujast ja seeläbi lubades laadida kahe käsu abil protsessori registrisse ükskõik millist 32 bittist arvu. U tüüpi kodeeringuga on defineeritud käsud LUI (*load upper immediate*) ja AUIPC (*add upper immediate to programm counter*). [22]

2.4.3 RV32I käskude erivormingud

Lisaks põhivormingutele eksisteerivad ka kaks eri tüüpi kodeeringu vormi, mis sisaldavad endas mäluaadresse ning seetõttu vajavad kaasa antava väärtuse jaoks eraldi vormingut.

B-tüüpi (*branch*, hargnemis) käsu kodeering tugineb S-tüüpi kodeeringule, säilitades kõikide käsuelementide jaoks samad positsioonid ja pikkused. Erinevus seisneb aga selles, kuidas käsuga kaasnevad väärtuse bitid on ümber paigutatud. Selline paigutus tuleneb hargnemiskäskude iseloomust: käsuga edastatav väärtus tähistab sihtaadressi, kuhu hüpata. Kuna see aadress peab alati olema paarisarv, ei ole käsus vaja eraldi ruumi kõige madalama (nullinda) biti jaoks. Tänu sellele saab käsus edastada 12-bitise märgiga väärtuse, mille nullis bitt on vaikimisi 0. [22]

RISC-V käsustiku üheks eesmärgiks on säilitada ühtne käskude formaat, mistõttu paiknevad väärtuse bitid 1–10 samadel positsioonidel nagu S-tüüpi käskudes. Käsustiku spetsifikatsioon näeb ette, et kõrgeim (12.) bitt asetatakse käsu kõige kõrgemasse positsiooni (bitt 31). Selle tulemusena paigutatakse väärtuse 11. bitt ainsale veel vabale positsioonile. Kõik nimetatud muudatused on nähtaval joonisel 1. [22]

J-tüüpi kodeeringut kasutatakse hüppekäsuks JAL (*jump and link*). Selle vormingu ülesehitus sarnaneb U-tüüpi käskudele, kuid erinevalt viimasest, sisaldab see 20-bitist hüppeaadressi. Nagu ka B-tüüpi käskude puhul, on madalaim (nullis) bitt alati väärtusega null, tagamaks paarisarvulise aadressi. Joonisel 1 on näha J-tüüpi kodeeringu ülesehitust. [22]

2.4.4 RV32M

RV32M on RISC-V käsustiku laiendus, mis lisab käsud täisarvude kiireks korrutamiseks ja jagamiseks, vähendades protsessori arvutuskoormust [22, 12]. See defineerib kaheksa R-tüüpi käsku: neli korrutamiseks ja neli jagamiseks [22]. Kuigi RV32I toetab korrutamist ja jagamist nihke- ja bittitehetega, on RV32M oluliselt efektiivsem, vältides algoritmilisi piiranguid ja protsessori ajakulu [22].

RV32M laiendus sisaldab järgmisi käske, mis tegutsevad täisarvudega [22]:

- **MUL** – kahe täisarvu korrutamine.

- **MULH** – kahe täisarvu korrutamine, salvestades ainult ülemise 32-bitise osa.
- **MULHSU** – märgiga ja märkideta täisarvu korrutamine, salvestades ülemise 32-bitise osa.
- **MULHU** – kahe märkideta täisarvu korrutamine, salvestades ülemise 32-bitise osa.
- **DIV** – kahe märgiga täisarvu jagamine.
- **DIVU** – kahe märgita täisarvu jagamine.
- **REM** – jäägi arvutamine märgiga jagamisel.
- **REMU** – jäägi arvutamine märgita jagamisel.

Kõik defineeritud käsud kasutavad R-tüüpi kodeeringut [22].

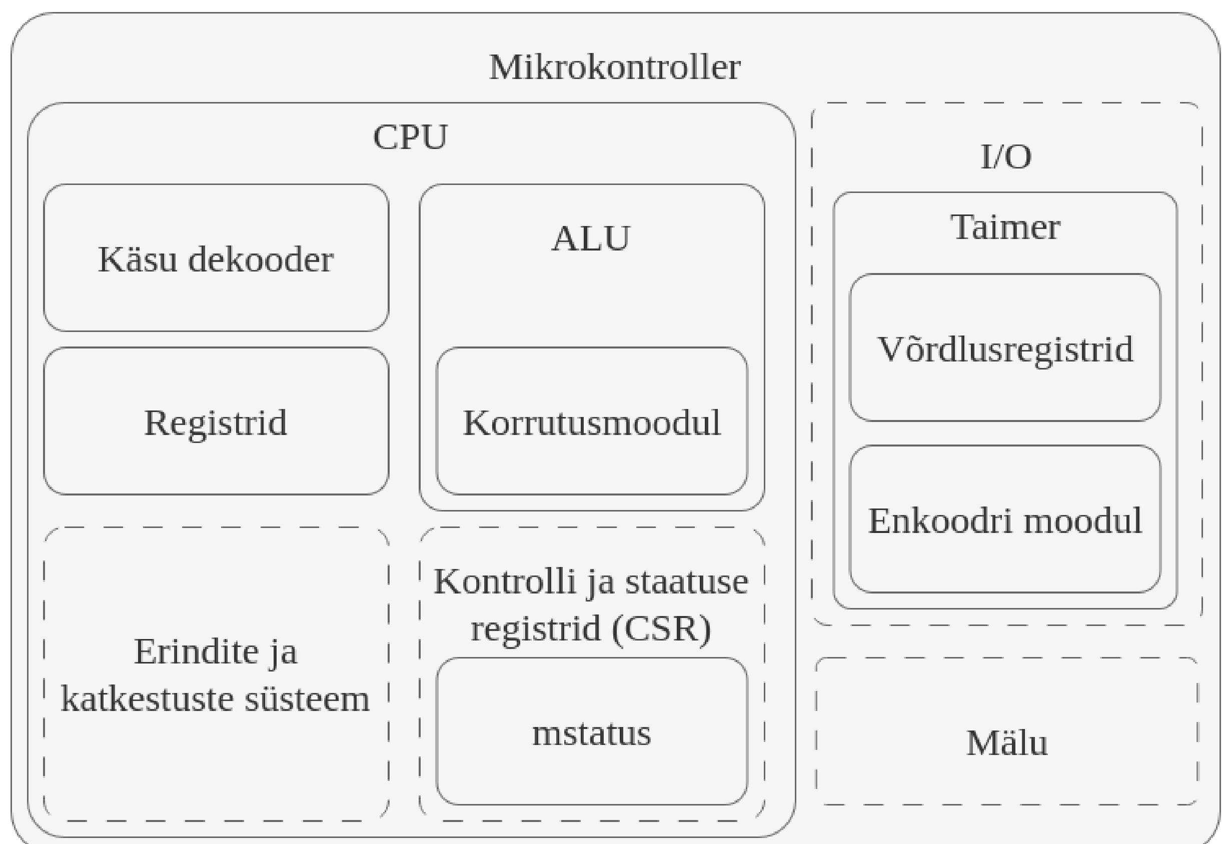
2.4.4.1 Implementatsioon ja jõudlus

RV32M laiendus võib olla implementatsioonis realiseeritud kas riistvaraliselt spetsiaalsete ALU moodulite kaudu või katkestuse abil. Riistvaralise implementatsiooni korral toimub korrutamine ja jagamine kiiremini, kuid see nõuab rohkem loogikalülitusi ja võib suurendada energiatarvet. Tarkvarapõhine implementatsioon võib kasutada iteratiivseid meetodeid, vähendades ALU loogikaskeemi keerukust, kuid kulutades rohkem protsessoriaega. [22]

3 Simulatsiooni arhitektuur ja teostus

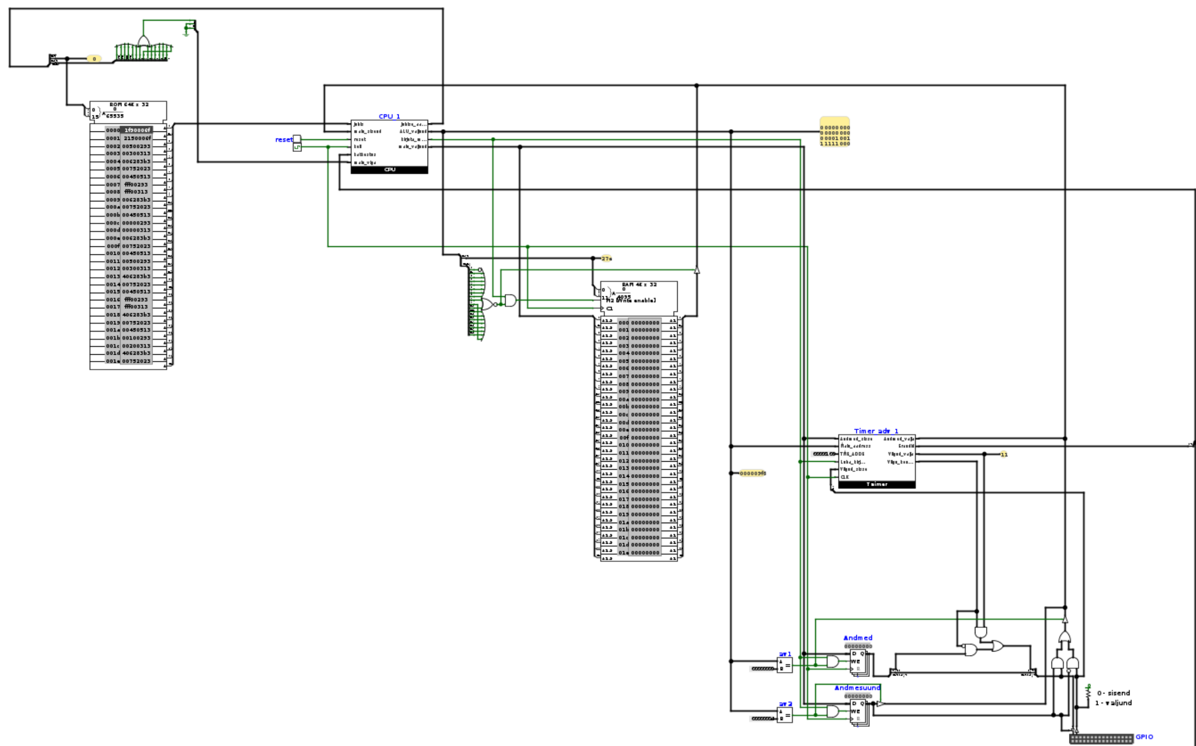
Selle bakalaureusetöö eesmärk on mikrokontrolleri simulatsiooni loomine Logisim Evolution tarkvaras. Töö raames valmiv simulatsioon on suunatud lihtsasti mõistetava, ning muudetava lahenduse pakkumisele, mis võimaldab efektiivselt illustreerida mikrokontrolleri sisemist toimimist. Simuleeritava mikrokontrolleri keskseks osaks on RV32IM käsustikule vastav protsessor, mis juhib kogu süsteemi tööd. Lahenduse failid koos vastava testikomplektiga on esitatud lisas 1.

Simulatsiooni selguse ja hallatavuse suurendamiseks on mikrokontroller jaotatud mitmeks alamskeemiks. Joonisel 2 on kujutatud skeemide hierarhiline ülesehitus ning nende omavaheline seotus. Lisaks on märgitud skeemide sees implementeeritud olulisemad komponendid. Kõrgeima taseme skeemiks on mikrokontrolleri skeem, mis sisaldab nii protsessori, kui ka taimeri alamskeeme. Samuti on selles realiseeritud sisend-väljund loogika, ning programmi- ja andmemälu.



Joonis 2. Lahenduse skeemide ja alamskeemide hierarhiline ülesehitus

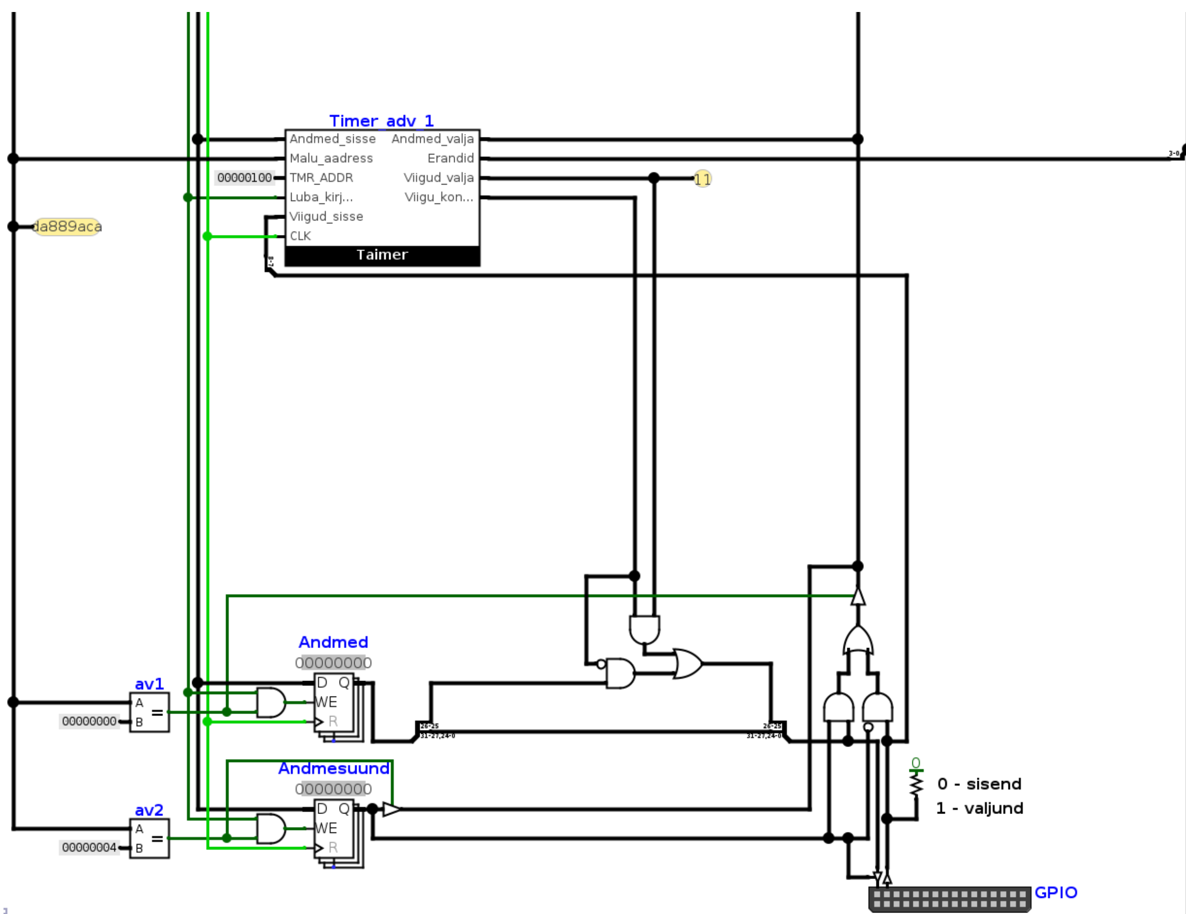
Joonisel 3 on esitatud mikrokontrolleri kõrgtaseme plokkskeem. Skeem sisaldab järgmisi komponente, alustades vasakult: programmimälu, protsessor, andmemälu ning sisend- ja väljundliidesed. Sisend- ja väljundosa jaguneb kaheks alamosaks: ülemine neist on taimer ja alumine üldotstarbe-



Joonis 3. Mikrokontrolleri skeemi ülevaade

line sisend-/väljundport (GPIO). Mõlemad osad on üksikasjalikult kujutatud joonisel 4. Kasutaja saab mikrokontrolleri tööd mõjutada, muutes programmimälu või andmemälu sisu, lähtestades seadme algolekusse, andes sellele kellasignaali või kasutades suhtlemiseks GPIO porti.

Mikrokontroller edastab protsessorile uue käsu, tuginedes protsessori poolt väljastatud käsu aadressile. Andmemäluga suhtlemine toimub samal põhimõttel: mikrokontroller loeb protsessori esitatud aadressi alusel mälust vastava väärtuse ning edastab selle protsessori skeemile.

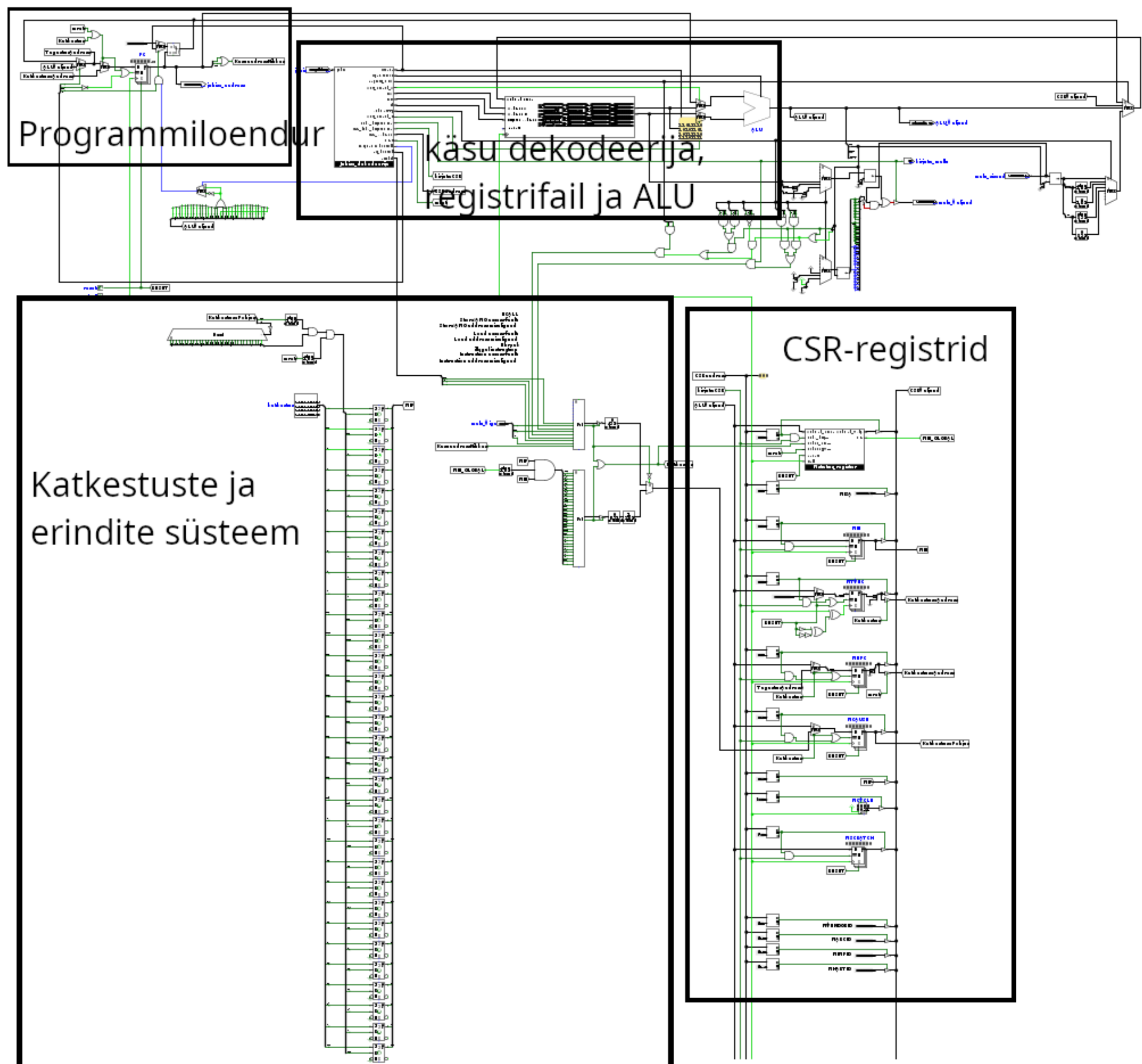


Joonis 4. Mikrokontrolleri I/O

3.1 Protsessor

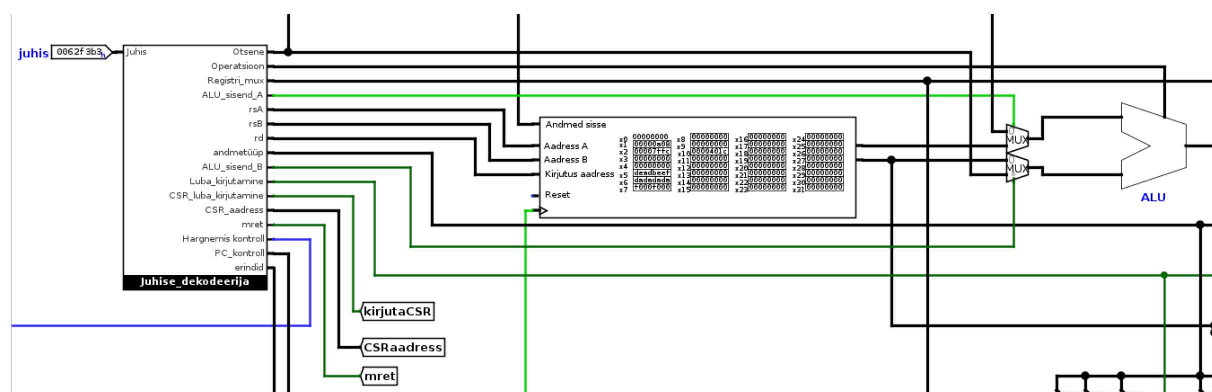
Mikrokontrolleri juhtimiseks on rakendatud üheastmeline protsessor, mille ülesehitus on kujutatud joonisel 5. Protsessor põhineb RV32IM-käsustandardi arhitektuuril. Selle põhifunktsiooniks on programmimälus paiknevate käskude täitmine, kasutades selleks sisemisi komponente ja alamskeeme.

Protsessorile edastatakse sisendina käsud ja andmed mälust, erindite signaalid (mis viitavad tavapärasest täitmisvoost kõrvalekaldumisele), lähtestus- ning kellasignaalid ning teave mälus ilmnenud vigade kohta. Protsessor genereerib väljundina käsuaadressi, mäluaadressi, andmed mäluruumi kirjutamiseks ning vastava kontrollsignaali andmemällu kirjutamiseks.



Joonis 5. Protsessori skeem

Käsu täitmisprotsess algab programmiloendurist (vt joonis 8), mis sisaldab hetkel täidetava käsu aadressi. Selle aadressi alusel laaditakse programmimälust vastav käsk, mis seejärel dekodeeritakse kontrollsignaalideks. Kontrollsignaalid juhivad protsessori tööd, määrates muu hulgas, millised registrid edastavad oma väärtused protsessori kasutusse. Samal ajal väljastatakse kaks registriväärtust, mida valitakse samuti kontrollsignaalide abil. Lisaks määravad kontrollsignaalid ALU sisendite (A ja B) valiku ning operatsiooni, mida ALU teostab. Kõnealused protsessi etapid käsu laadimisest kuni ALU sisenditeni toimuvad joonisel 6 kujutatud komponentides.

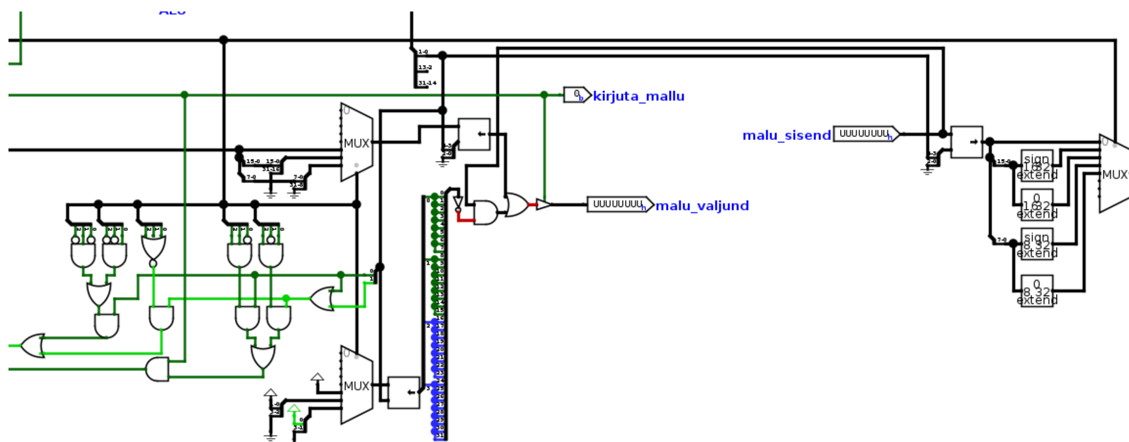


Joonis 6. Protsessori skeemi komponendid – käsu dekodeerija, registrifail ja ALU

ALU (aritmeetika-loogikaseade) arvutab tulemuse, mis suunatakse kontrollsignaalide abil sobivasse sihtkohta. ALU väljundit kasutatakse ka mäluaadressina andmemäluga suhtlemisel — nii lugemisel kui kirjutamisel. Vajalikud juhtsignaalid, sealhulgas kirjutusluba, genereeritakse protsessori loogikaskeemist. Antud protsessi realiseerivad komponendid on kujutatud joonisel 7, kus protsessor kasutab registrifaili teist väljundit andmeväärtusena, mis salvestatakse mällu. ALU väljund täidab mäluaadressi rolli ning lisaks kasutatakse seda ka andmete paigutamiseks õigetesse baitidesse juhul, kui toimub 16- või 8-bitine andmete lugemine.

Kuna Logisimi keskkonnas kasutatav mälukomponent võimaldab korraga lugeda ja kirjutada vaid 32-bitiseid väärtusi, toimub väiksema bitilaiusega kirjutamise korral esmalt olemasoleva 32-bitise väärtuse lugemine, seejärel asendatakse sihtbait(id) vastavalt käsule ning lõpuks kirjutatakse muudetud väärtus tagasi mällu. Väärtuse lugemisel valitakse 32-bitisest mälu väljundist ainult vajalikud baidid, mis seejärel nihutatakse vastavalt vajadusele madalama järgu positsioonidele, et moodustada korrektselt vormindatud tulemus.

Registrifaili kirjutatav väärtus võib pärineda neljast allikast: programmiloendurist, ALU väljundist, CSR-registerist või andmemälust loetud väärtusest. Salvestatav väärtus valitakse kontrollsignaalide abil.



Joonis 7. Protsessori skeemi mäluga suhtlemise alamosa.

3.1.1 Programmiloendur

Programmiloendur on register, mis hoiab hetkel täidetava käsu aadressi. Joonisel 8 on see register tähistatud numbriga 1. Üldjuhul suureneb programmiloenduri väärtus nelja võrra, võimaldades laadida järgmise käsu.

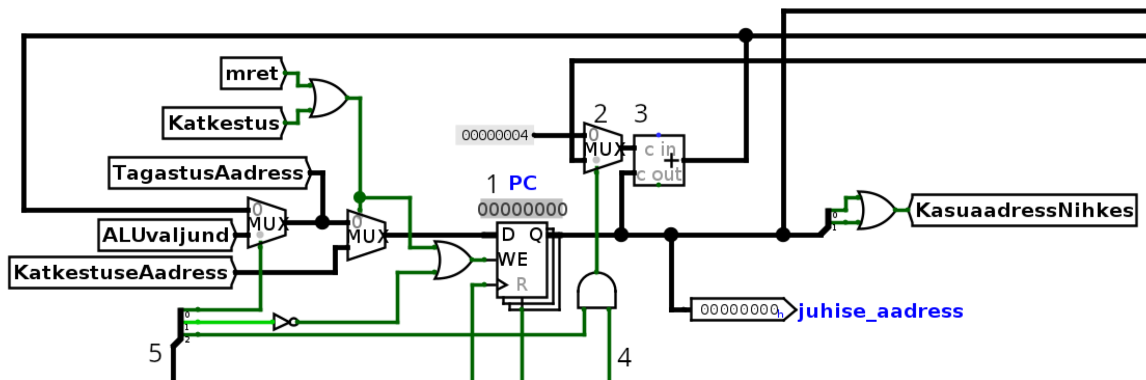
Seda funktsionaalsust toetavad komponendid 2 ja 3: komponent 2 on multiplekser, mis valib konstantse väärtuse 4 ja käsu dekoodrist tuleneva nihke vahel, ning komponent 3 liidab valitud väärtuse programmiloenduri hetkeväärtusele. Multiplekseri tööd juhib hargnemisotsuse signaal (tähistatud numbriga 4), mis pärineb joonisel 9 kujutatud võrdlusskeemist.

Programmiloenduri väärtust muutvate käskude realiseerimiseks kasutatakse veel kahte multiplekserit, mis võimaldavad väärtust muuta vastavalt programmi loogikale. Neid kontrollivad signaalid on joonisel 8 tähistatud numbriga 5.

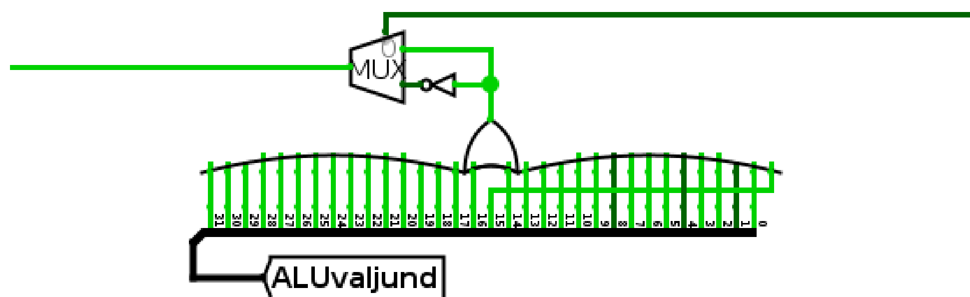
Hargnemisotsuseid võimaldab teha skeem, mis on esitatud joonisel 9. Hargnemine võib toimuda sõltuvalt sellest, kas ALU väljundiks on null või mõni nullist erinev arv. Signaal hargnemisotsuse kohta saadetakse seejärel edasi programmiloendurile, kujutatud joonisel 8, milles valitakse hargnemisel järgmine programmiloenduri aadress, liites programmiloenduri hetkeväärtusele käsust kaasaantud nihe.

Lisaks tavapärasele käsutäitmisele võib programmiloenduri väärtust mõjutada katkestuste tekkimine, mis võib toimuda kahe käsu täitmise vahel. Katkestuse korral salvestatakse programmiloenduri hetkeseis ning juhtimine suunatakse katkestuse käsitlusaadressile, mille määrab

vastav CSR-register `mpie`. Vastupidises olukorras, kui täidetakse MRET käsk, taastatakse eelnev programmiloenduri väärtus lugedes see `mpie` registrist. Mõlemal juhul toimub aadressi valik ümbersuunamisloogika abil, mille toimimist illustreerib joonis 8.



Joonis 8. Programmiloendur



Joonis 9. Protsessori võrdlemissüsteem

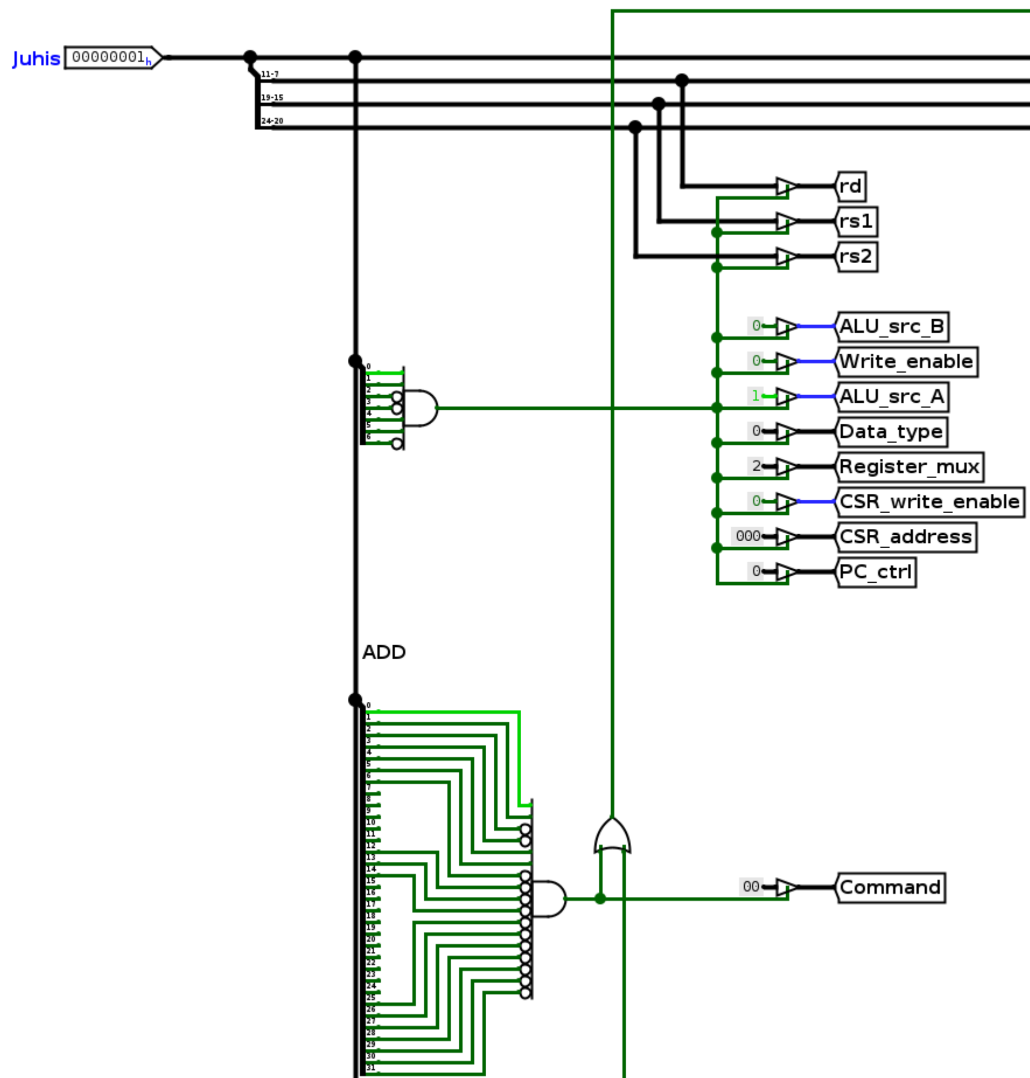
3.1.2 Käsu dekooder

Käsu dekoodri ülesanne on tõlkida programmimälus paiknevad käsud vastavateks juhtsignaalideks. Joonisel 10 on toodud näide `add`-käsku dekodeerivast skeemiosast. Dekodeerimisprotsess on jaotatud kahte etappi: esmalt määratakse käsu tüüp ja seadistatakse universaalsed kontrollsignaalid, seejärel tuvastatakse konkreetne käsk ning määratakse sellele vastavad täiendavad juhtsignaalid. Antud käsu puhul on ainus eraldi määratav kontrollsignaal ALU operatsioon.

Valminud dekoodri poolt toetatud käsude loetelu on esitatud tabelites 2, 3, 4, 5, 6, 7 ja 8.

Lisaks käsu dekodeerimisele kontrollitakse käesolevas moodulis ka käsu kehtivust, väljastades vastava signaali, ning tagades juhtsignaalide turvalise oleku ebaõigete käskude korral. Tava-

pärastele käskudele lisaks tunneb dekooder ära ka käsud ECALL, EBREAK ja MRET. Neist esimesed kaks kutsuvad esile erindi, viimane aga võimaldab naasta erindist. Kuigi RISC-V privileegimata arhitektuur ei nõua MRET käsu implementeerimist [22], otsustas autor selle siiski lisada, võimaldamaks kiiret tagasitulekut lõksudest.



Joonis 10. Käsu dekodeerimismooduli näide add käsu näitel

3.1.3 Registrid

Välja töötatud süsteemis on implementeeritud 31 üldotstarbelist registrit, mille põhieesmärk on võimaldada andmete ajutist säilitamist ning nende edastamist aritmeetika-loogikaseadmele (ALU). Registrite toimimispõhimõte on illustreeritud joonisel 15. Igal taktil on võimalik lugeda andmeid kahest registrist ning kirjutada andmeid ühte valitud registrisse.

Lisaks muudetavatele registritele on defineeritud ka register `x0`, mille väärtus on kontstantselt null. Lugemisel tagastab see register alati väärtuse null ning kirjutusoperatsioonid sellesse registrisse ei muuda selle sisu.

Andmete lugemine registrifailist on realiseeritud kahe multiplekseri abil, mis suunavad valitud registrite väljundid ALU sisenditeks või muudeks otstarveteks. Kõigi registrite sisendid on ühendatud mooduli ühise andmesisendiga, mille kaudu määratakse registrisse kirjutatav väärtus. Kirjutusoperatsiooni sihtregister valitakse dekoodri abil, mis aktiveerib vastava registri salvestusmehhanismi.

Parema ülevaate pakkumiseks protsessori sisetoimingutest kuvab registrifaili skeemikujundus kõigi registrite väärtused, võimaldades nende sisu jälgida skeemi avamata.

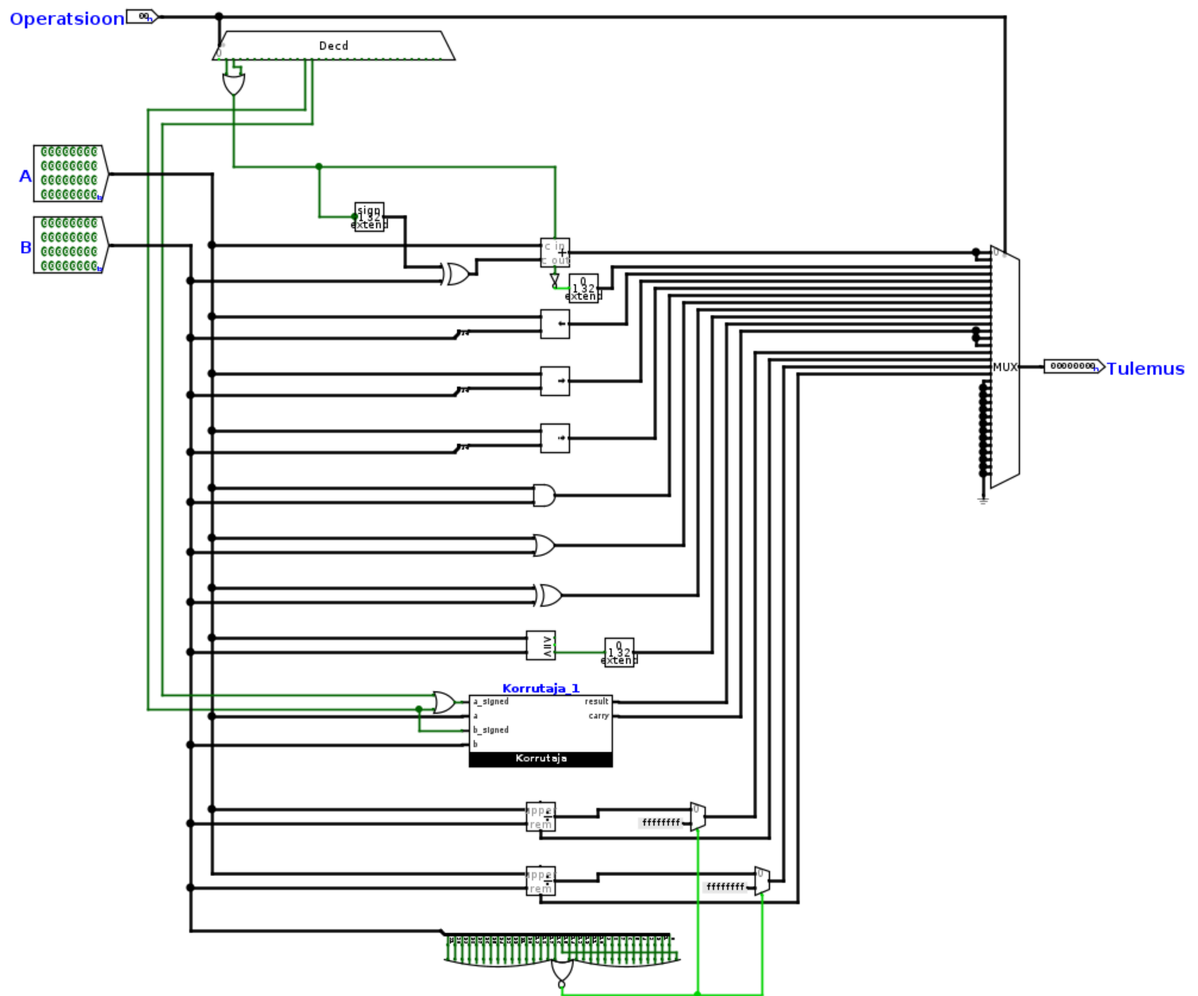
3.1.4 ALU

Protsessori aritmeetika-loogikaseade (ALU) vastutab käskude dekodeerimise järel kahe registrist loetud operandi vaheliste aritmeetiliste või loogiliste operatsioonide teostamise eest. Implementeeritud ALU ehitus on kujutatud joonisel 11. ALU saab sisendiks kaks operandi ning operatsiooni tüübi määrava juhtsignaali, mille alusel valitakse teostatav tehe.

ALU toetab järgmisi põhitehteid: liitmine, lahutamine, vasak- ja parempoolne nihutamine (nii aritmeetiline kui loogiline), loogikatehted (`AND`, `OR`, `XOR`), täisarvuline korrutamine ja jagamine ning võrdlus. ALU väljund valitakse multiplekseri abil.

Kuna ALU juhtsignaalide ruum sisaldab rohkem koode kui implementeeritud operatsioonide arv, on määramata või defineerimata operatsioonikoodi korral seadme väljundiks null. See välistab määramatu käitumise, näiteks juhul kui programmimälus asub vigane või mittealgatatud kood.

Korrutustehted on realiseeritud spetsiaalse mooduli abil, mis toetab nii märgiga kui ka märgita ja segatüüpi korrutustehteid. Segatüübi korral, kus üks operand on märgiga ja teine märgita, tagastatakse tulemus märgiga kujul. ALU tasemel saab salvestada kas korrutustulemuse alumise või ülemise 32-bitise osa: alumine saadakse märgita korrutusega, ülemise osa jaoks on realiseeritud kolm varianti – märgiga, märgita ja segatüüpi korrutus.



Joonis 11. ALU loogikaskeem

3.1.5 Erindid, katkestused ja lõksud

Mõningatel juhtudel peab protsessor katkestama tavapärase käsujada täitmise, et reageerida ootamatule olukorrale. Selleks on implementeeritud erindite ja katkestuste käsitlemise mehhanismid, mis suunavad protsessori spetsiaalsesse lõksurežiimi. Lõks (trap) on seisund, milles protsessor täidab vastava erindi- või katkestusekäsitluse.

Erindid on protsessori sisemised sündmused, näiteks keelatud käsu täitmine või lubamatu mälupöördumine, samas kui katkestused on välised sündmused, nagu näiteks taimeri poolt genereeritud katkestus. Protsessor reageerib erinditele alati, sõltumata katkestuste lubatuse olekust. Katkestuste puhul peab nende käivitumiseks olema täidetud kaks tingimust: globaalse katkestuslipu ja vastava individuaalse katkestusallika lubamine. Käesolevas süsteemis on katkestuste ainsaks allikaks taimer.

Erinditest käsitletakse korraga vaid kõige prioriteetsem, katkestused seevastu töödeldakse määratud prioriteedi alusel. Lõksurežiimi sisenemisel salvestatakse katkestuse või erindi hetkeprogrammiloendur (PC) registrisse `MEPC`, ning lõksu põhjus registrisse `MCAUSE`. Seejärel laetakse programmiloendurisse `MTVEC` registri väärtus ja protsessor jätkab tööd vastavas käsitlusrutiinis. MRET käsu täitmisel taastatakse programmiloenduri varasem väärtus MEPC registrist, võimaldades protsessoril jätkata täitmist täpselt sealt, kus töö enne lõksu sisenemist katkes.

Lahendus tuvastab ja käsitleb erindeid järgmistel juhtudel: vigane käsk, ECALL, EBREAK, käsuaadressi joondusviga ning joondusviga mälupöördustel (nii lugemisel kui kirjutamisel).

3.1.6 Juhtimis- ja olekuregistrid (CSR)

CSR (*Control and Status Registers*) ehk juhtimis- ja olekuregistrid on eraldi viiebitisel aadressiruumil paiknev registrihulk, mis võimaldab protsessori tööd juhtida ja selle olekut jälgida. Käesolevas töös on implementeeritud 13 CSR-registrit, millest seitse on ainult loetavad ning ülejäänud on vähemalt osaliselt kirjutatavad. Registrid `mstatus`, `mtvec`, `mepc`, `mcause` ja `mip` on seotud katkestuste käsitlemisega. `mcycle` loendab protsessoritsükleid, `mscratch` on mõeldud ajutiste väärtuste hoidmiseks ning `misa` sisaldab infot protsessori funktsionaalsuse kohta.

Lisaks on implementeeritud neli ID-registrit: `mvendorid`, `marchid`, `mimpid` ja `mhartid`, mis kõik tagastavad konstantselt nulli. Esimesed kolm on mõeldud protsessori identifitseerimiseks ning viimane erinevate tuumade eristamiseks mitmetuumalistes süsteemides.

3.2 Mälu

Protsessori implementatsioon on üles ehitatud nii, et mäluruum ja mälumaht ei sõltu otseselt protsessorist, vaid on määratud välise keskkonna poolt. See lähenemine võimaldab kasutada ühte protsessorituuma mitme erineva mikrokontrolleri simulatsioonis, millel on erinevad mälukonfiguratsioonid.

Mikrokontrolleri mäluruum jaguneb kolmeks osaks: programmimälu, mälupeale kaardistatud I/O ja andmemälu. Mikrokontrolleri simulatsioonis määratud andmemälu ja mälupeale kaardistatud I/O ruumid on esitatud tabelis 1. Programmimälu sisaldab käivitatavat koodi, millele antud lahenduses ei ole võimalik kirjutada. Mälupeale kaardistatud I/O sisaldab mikrokontrolleri väliseid I/O-seadmeid, nagu registrid, kommunikatsioonisüsteemid, taimerid jne.

Lisaks mäluruumile eksisteerivad ka kontrolli- ja olekuregistrid, mis juhivad protsessori tööd ja sisaldavad infot protsessori oleku kohta. RISC-V ISA määratleb kuni 4096 sellist registrit, kuid protsessorid ei pea kõiki CSR-registreid implementeerima ja seetõttu on implementeeritud vaid üksikud.

	Kaardistatud I/O	Programmimälu
Andmemälu mäluruum	0x0 - 0x3FFF	0x4000 - 0x7FFF

Tabel 1. Mikrokontrolleri kirjutatav mäluruum

3.3 Mikrokontrolleri välisliides

Mikrokontrolleri välisliidesed võimaldavad vahetada andmeid ja juhtida väliseid seadmeid vastavalt süsteemi erinevatele liideste omadustele [2]. Antud lahenduses on implementeeritud väljundport (GPIO) ja taimer, mis võimaldavad ühenduda välisseadmete ja -süsteemidega.

Kuigi käesolevas lahenduses on implementeeritud vaid kaks liidest, on mikrokontrollerile võimalik lisada mitmesuguseid erinevaid I/O-seadmeid. Lisaks taimerile ja üldotstarbelistele sisend-

/väljundliidestele (GPIO) leidub mikrokontrollerites sageli ka suhtlusliideseid, nagu UART, SPI ja I²C, samuti analoogsignaale töötlevaid seadmeid – näiteks PWM-väljundeid, analoog-digitaalmuundureid (ADC) ja digitaal-analoogmuundureid (DAC) [10].

Antud töös lähtus autor I/O-seadmete valikul Logisim Evolutioni keskkonna piirangutest. Tegemist on digitaalloogika simulaatoriga, mis ei toeta analoogsignaalide simuleerimist [4]. Seetõttu valiti implementeeritavateks seadmeteks taimer ja väljundport, kuna need sobituvad Logisimi funktsionaalsusega ning on üldiselt kasulikud mikrokontrollerite tavapäraste ülesannete täitmisel.

3.3.1 GPIO

GPIO port on realiseeritud kahe registriga. Esimene register on andmete register, mis on mõeldud andmete väljastamiseks. Andmete kirjutamine sellesse registrisse kajastub väljundis. Teine register on andmesuuna register, mille abil saab määrata, kas port toimib sisendina või väljundina. Kui andmesuuna register sisaldab väärtust 1, on port määratud väljundiks, ja kui väärtus on 0, siis sisendiks.

Joonisel 12 on kujutatud loodud GPIO port, mida kontrollivad kaks registrit aadressidega 0 ja 4. Loogikaskeem valib sobiva registri, kontrollides sissetuleva aadressi kõrgema 30 bitti vastavust, võimaldades registrite valikut nihkega (kui salvestatakse 16 või 8 bitti).

Kui andmesuuna registrit loetakse, tagastatakse selle väärtus. Kui andmesuuna register on seadistatud väljundiks, tagastatakse andmeregistri väärtus. Kui register on sisendiks, siis tagastatakse pordi sisendväärtus. Kuna iga bitt võib olla väärtuses 1, 0 või määramata, viivad määramatud väärtused pordi madalseisundisse, et vältida vigade tekkimist mikrokontrolleris.

Joonis 12. GPIO register

3.3.2 Taimer

Mikrokontrolleris on realiseeritud taimer, mis suudab loendada kellatakte, mõõta sisendsignaale ning genereerida katkestusi. Taimeril on kahte tüüpi katkestusi:

- Ülevoolu katkestus – aktiveerub, kui loenduri väärtus ületab maksimaalse 32-bitise väärtuse ja läheb ümber nulli.
- Võrdluskatkestus – tekib, kui taimeri väärtus saavutab või ületab ühe kahest määratud võrdlusväärtusest (CR0 või CR1).

Taimeri sisendkella allikaks saab valida süsteemikella, enkoodri dekooderimooduli või ühe kahest määratud sisendviigust. Valitud kellasignaal suunatakse jagurisse, mille abil saab taktsagedust vähendada teguriga 1, 2, 4 või 8. Jagatud signaal saadetakse edasi 32-bitisele loendurregistrile, mis suurendab või vähendab oma väärtust vastavalt seadistusele.

Taimerit juhitakse kontrollregistri kaudu, mille aadress määratakse alamskeemist väljaspool. Lisaks kontrollregistrile on taimeril:

- loendurregister (aadress: kontrollregistri aadress + 4),
- esimene võrdlusregister CR0 (aadress + 8),
- teine võrdlusregister CR1 (aadress + 12).

Kontrollregistri kaudu on võimalik:

- käivitada või peatada taimerit,
- valida kella jagamise astet ja sisendsignaali allikat,
- määrata, kas loendamine toimub kasvavalt või kahanevalt,
- määrata, kas taimer lähtestatakse CR0 või CR1 võrdluse korral,
- lubada automaatne peatumine CR0/CR1 võrdluse või ülevoolu korral,

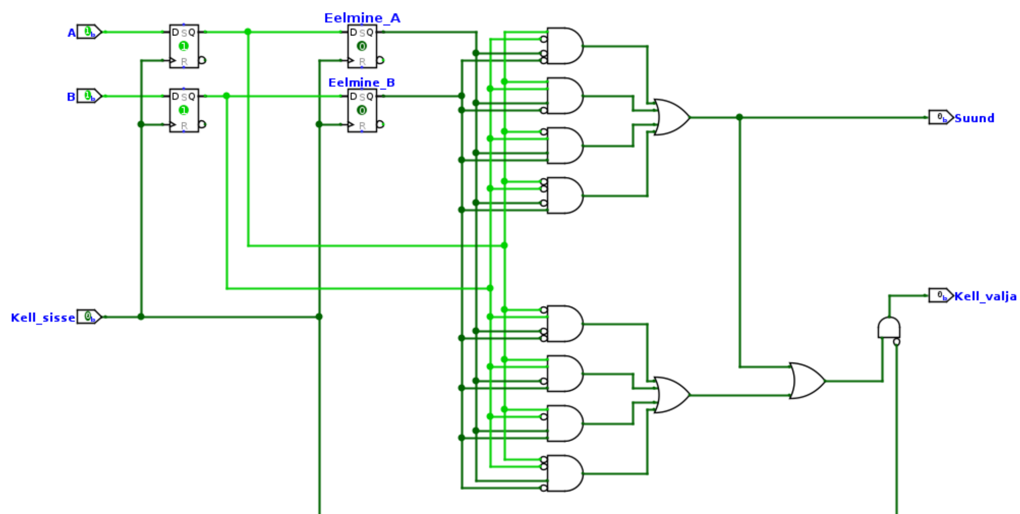
- lubada enkoodri dekooderil kontrollida loendussuunda,
- seadistada taimeri väljundit kontrollivaid viike.

Selline paindlikkus võimaldab taimerit kasutada nii ajastusrakendustes kui ka signaalide mõõtmise nagu mootorite enkoodrite lugemisel ja väljutamise rollides.

3.3.3 Enkoodrimoodul

Enkoodri moodul on realiseeritud vastavalt joonisel 13 toodud skeemile. Sisendsignaalid A ja B salvestatakse D-trigerisse, koos nende varasemate väärtustega, kasutades ajastuseks ühist kellasignaali. Nii saadakse üheaegselt kätte signaalide praegused ja eelnevad olekud, mida seejärel võrreldakse, et määrata liikumise suund ja vajadusel genereerida väljundkellasignaal.

Kui signaalide olek ei ole muutunud või mõlemad signaalid muutuvad samaaegselt, ei genereerita väljundit ning suund jääb määramata. Päripäeva liikumise korral (A=0, B=0 → A=1, B=0 → A=1, B=1 → A=0, B=1) määratakse suunaväljundiks 1, vastupäeva liikumisel aga 0. Kellasignaal väljastatakse ainult juhul, kui toimub korrektne üheetapiline muutus, ning see toimub sisendi kellasignaali langeval frondil.

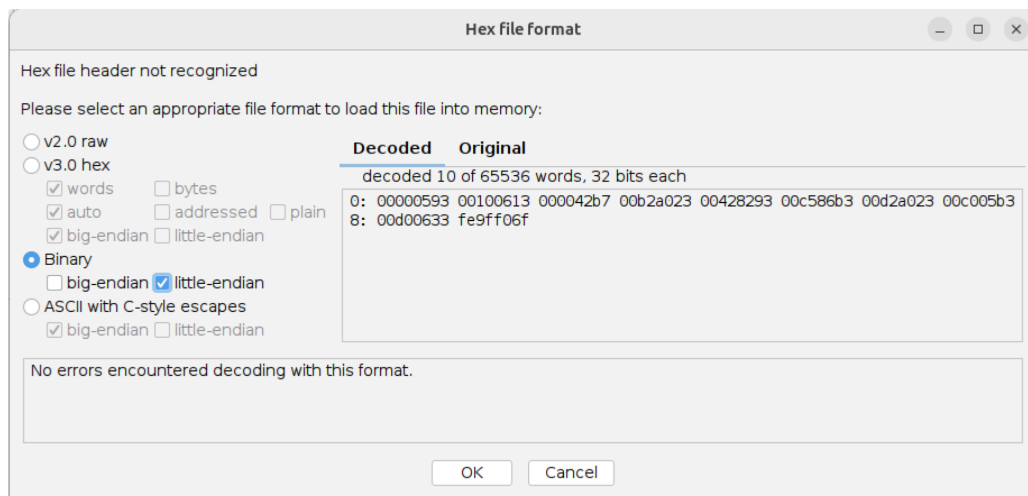


Joonis 13. Enkoodrimooduli loogikaskeem

4 Testimine

Mikrokontrolleri programmeerimiseks on saadaval mitmesuguseid tööriistu, millest autor kasutas GNU RISC-V tööriistakomplekti, et kompileerida mikrokontrolleri kood assembli-, C- ja C++-keeltest [8]. Nende tööriistade abil loodud binaarfailid laadis autor mikrokontrolleri ROM-i ja RAM-i mäludesse.

Lisas 1 toodud Makefile kasutab RISC-V assemblerit ja *objcopy* käske, et kompileerida kood ning teisendada see Logisimis kasutatavasse formaati. Väljundiks olevad binaarfailid saab laadida mikrokontrollerisse, kasutades "*Binary*"formaati ning valides "*little-endian*"režiimi, nagu on näidatud joonisel 14. "*Binary*"formaat ütleb logisimile, et tegemist on binaarfailiga. "*Little-endian*"kirjeldab siinkohal bittide järjestust baidi sisemiselt [5].



Joonis 14. Koodi laadimine mikrokontrolleri programmimällu Logisim Evolution programmis

RISC-V käsustiku arhitektuur on modulaarne ja võimaldab selle laiendamist mitmete standardsete või kasutaja poolt defineeritud laiendustega [22], mistõttu ei pruugi olemasolevad üldotstarbelised testikomplektid alati sobida konkreetsete riistvaraliste realiseeringute valideerimiseks. Eelnevast tulenevalt tuleb arendajatel sageli koostada kohandatud testid, mis võtavad arvesse konkreetse mikrokontrolleri arhitektuurilisi ja funktsionaalseid iseärasusi [7, 6]. Kuigi on olemas mitmeid avalikult kättesaadavaid testikomplekte, näiteks riscv-tests GitHubi repositoorium [9], otsustas autor valideerida enda loodud mikrokontrollerit spetsiaalselt selle tarbeks väljatöötatud testide abil, lähtudes peamiselt RISC-V priviligeerimata arhitektuuri manuaali versioonist 20240411 [22], et tagada vastavus ametlikele arhitektuurinõuetele.

Autor otsustas kasutada spetsiaalselt loodud testikomplekti, kuna olemasolevad üldised RISC-V testikomplektid ei kata kõiki konkreetse mikrokontrolleri funktsioone, täpsemini sisend-/väljundseadmete (I/O) ja mällu kaardistatud seadmete testimist. Sellised valmis testikomplektid keskenduvad enamasti käsustiku ja registrifaili korrektsele toimimisele, kuid ei arvesta konkreetse süsteemi arhitektuuri eripärasid.

Kuna testitaval mikrokontrolleril on unikaalne I/O-lahendus, taimeriloogika ning kindlale aadressiruumi osale kaardistatud seadmed, oli vajalik koostada kohandatud testid, mis kontrollivad just nende komponentide töökindlust ja vastavust spetsifikatsioonile.

4.1 RV32IM käskude programmaatiline testimine

Autor koostas väljundit andvate käskude testimiseks koodi, mis laadis ette määratud väärtused, seejärel käivitas testitava käsu, salvestades tulemuse esmalt registrisse ja seejärel RAM-i. Testide lõppedes võrreldi RAM-i sisu eeldatavate väärtustega. Eeldati, et kogu ülejäänud mikrokontrolleri funktsionaalsus, sealhulgas mäluga suhtlemise käsud, töötab korrektselt. Kui mikrokontrolleri teised komponendid ei töötanud ootuspäraselt, loeti kogu test ebaõnnestunuks. Kui RAM-i sisu vastas oodatule, loeti test edukalt läbituks, vastasel juhul ebaõnnestunuks.

Mäluga suhtlemise käskude testimiseks kasutati sarnast testimistaktikat, kuid seda viidi läbi mitmes etapis. Esmalt testiti 4-baidise sõne kirjutamise ja lugemise käske, kirjutades kindlale mäluaadressile väärtus. Seejärel loeti samalt aadressilt väärtus ning lõpuks kirjutati saadud see tagasi. Koos eelnevalt kirjeldatud testidega kinnitasid tulemused andmete korrektset lugemist ja kirjutamist. Seejärel testiti 2- ja 1-baidi suuruseid mälukäske, kasutades eelnevalt testitud 4-baidiseid käske kindlate väärtuste lugemiseks ja kirjutamiseks. Nimetatud mälutestide käigus ei kontrollitud aadresse, millele lugemine või kirjutamine ei ole võimalik, ega aadresse, mis võiksid põhjustada erindite tekkimist.

Erindite ja katkestuste testimiseks põhjustati testides erind või katkestus, kasutades käsku, mis pidi vastava situatsiooni esile kutsuma. Erindi tekkimisel suunatakse programm lõksu, mille aadress määratakse *mepc* CSR-registri kaudu. Lõksus loetakse *mcause* CSR-registri väärtus ning salvestatakse see kindlale mäluaadressile. Seejärel kirjutatakse *mtvec* CSR-registrisse soovitud tagastusaadress ning lõksust väljutakse käsuga MRET. Pärast seda loetakse lipuväärtus, mis salvestatakse samuti mällu, et kinnitada MRET käsu korrektne toimimine.

Hüppe- ja hargnemiskäskude testimiseks defineeriti protseduurid, kuhu hüpatakse või millele hargnetakse. Eelnevalt mainitud protseduurides, salvestati mällu kindel väärtus, mis tähistas protseduuri jõudmist. Hüppekäskude puhul pidi mikrokontroller protseduurini jõudma. Hargnemise korral eksisteerisid nii harud, mille korral eeldati protseduuri läbimist, kui ka harud, mille korral seda ei eeldatud. Testiti nii toimima pidavaid hargnemisi kui ka neid, mis ei pidanud toimuma.

CSR-registeritega suhtlemise jaoks ei koostatud eraldi testikomplekti, kuid nende käsud said kaetud teiste testide käigus.

4.2 Juhtimis- ja olekuregistrite testimine

Mikrokontrolleri juhtimis- ja olekuregistrite (CSR) testimiseks koostati testikomplekt, mis kirjutas registritesse andmeid ning luges need seejärel tagasi, kontrollides, kas saadud väärtus vastas ootustele vastavalt registri tüübile. Registrite puhul, kuhu kirjutamine ei ole lubatud, ei tohtinud ükski käsk nende sisu muuta. Osaliselt kirjutatavate registrite puhul tohtisid muutuda ainult need bitid, kuhu kirjutamine on lubatud. Täielikult kirjutatavad ja loetavad registrid pidid vastu võtma kõik muudatused ning väljastama täpselt sisestatud väärtuse.

Kuna RISC-V CSR-käsud võimaldavad üheaegselt registri sisu lugeda ja muuta [22], kasutati ühe testi jooksul mitut erinevat käsku ja mäluaadressi. Seetõttu võis mõningate testide puhul pidada testi osaliselt läbituks, kui vähemalt osa käsust toimis ootuspäraselt. Ootuspärane käitumine sõltus konkreetsest registrist. Registrid, mis on määratud ainult loetavateks, ei tohi kirjutamisel oma väärtust muuta. Seega, kui ainult loetav register salvestas kirjutamisel uue väärtuse, võis lugemist pidada toimivaks, kuid kirjutamist mitte.

4.3 Mälu ja mälule kaardistatud I/O testimine

Mäluruumi testimiseks koostatud testikomplektis kontrolliti, kas andmete kirjutamine ja lugemine toimib ootuspäraselt. Lisaks tavapärasele mälule, mille töötamist kontrollivad tulemuste mällu kirjutamised, testiti ka suhtlust mäluaadressidele kaardistatud sisend-/väljundseadmete (I/O) kontrollregistritega. Valminud mikrokontrolleris olid I/O-seadmeteks taimer ning 32-bitine andmesideport.

Erinevalt tavalisest mälust võivad mälule kaardistatud I/O-seadmete registrite väärtused muutuda iseseisvalt – sõltumata protsessori tegevusest [20]. Seda arvestati ka testimisel, kontrollides registrite korrektsust nii otsese kirjutamise käigus kui ka seadmete normaalse töö jooksul toimuvaid muutusi.

4.4 Testide kaetavus

Testide eesmärk oli hinnata mikrokontrolleri vastavust RISC-V RV32IM spetsifikatsioonile. Katsetes kontrolliti kõigi spetsifikatsiooni kuuluvate käskude korrektset täitmist. Kuigi kõiki käske testiti vähemalt ühe sisendkombinatsiooniga, ei hõlmanud testid kogu võimalikku sisendspektrit. Seetõttu võib käsukaetuse hinnata täielikuks, kuid sisendite kaetuse osas jäi katvus osaliseks.

Tingimuslike harukäskude puhul testiti mõlema haru – nii tingimuse täitumist kui mittetäitumist. Iga harukäsu korral kontrolliti, et programmikäik järgiks õigesti nii haru võtmise kui ka haru mittetootmise loogikat. See võimaldas veenduda, et juhtloogika töötab kõigis loogilistes olukordades ootuspäraselt.

Registrifaili ja aritmeetika-loogikaploki (ALU) toimimine testiti põhjalikult. Kontrolliti andmete korrektset liikumist registritesse ja sealt välja, samuti ALU poolt sooritatavate operatsioonide (sh liitmine, lahutamine, nihked, bitilised operatsioonid, korrutamine ja jagamine) tulemusi.

Mäluga seotud testides kontrolliti lugemis- ja kirjutamisoperatsioone mälu eri aadressidele. Lisaks sellele testiti osaliselt mällu kaardistatud välisseadmete tööd. I/O portide korral kontrolliti nende mäluaadressidele kirjutamist ja sealt lugemist. Taimeri korral testiti samuti selle registritele juurdepääsu, kuid mitte kõiki võimalikku töörežiime ega kõikide katkestuse genereerimist.

Lisaks automatiseeritud testidele kasutati visuaalseks kontrolliks ka *probe*-komponente, mis võimaldasid jälgida signaalide kulgemist mikrokontrolleri sees. Need vahendid võimaldasid visuaalselt kinnitada, et käsud dekodeeriti õigesti, juhtsignaalid tekkisid ootuspäraselt ning andmeedastus ALU, registrifaili ja mälu vahel toimis korrektselt.

4.5 Testide tulemused

Mikrokontroller läbis kõik loodud testid edukalt. Kontrolliti ALU toimimist, registrifaili lugemis- ja kirjutamisoperatsioone, programmiloenduri loogikat, mäluga suhtlemist ning sisend-/väljundportide töökindlust. Samuti testiti juhtloogikat, sealhulgas käsu dekodeerimist, katkestuste käsitlemist ning tingimuslikke harusid.

Testide käigus salvestati tulemused andmemällu eelnevalt määratud aadressidele. Lipubittide ning eelnevalt teadaolevate väärtuste alusel kontrolliti, kas toimingud vastasid ootustele. Näiteks ALU pidi andma õiged tulemused eri aritmeetiliste ja loogiliste operatsioonide korral, registrite väärtused pidid muutuma vastavalt käsule ning ainult loetavad registrid ei tohtinud kirjutamisel muutuda.

Testide korrektsust ja läbivust kontrolliti käsitsi: protsessor täitis esmalt testi koodi, mille järel peatati selle töö ning andmemälu sisu loeti ja võrreldi ootustega. Kuna testid olid jaotatud mitmeks eraldiseisvaks komplektiks, käivitati iga testikomplekt eraldi, et vältida omavahelisi konflikte ning hoida testkeskkond võimalikult puhtana.

Lisaks automatiseeritud testidele viidi läbi ka reaalajas visuaalne kontroll, kasutades *probe*-komponente mikrokontrolleri sisemiste signaalide jälgimiseks. Need võimaldasid mõõta juhtmetel toimuvaid olekumuutusi ja teisendada need loetavateks arvväärtusteks. Visuaalse jälgimise käigus veenduti, et käsud dekodeeriti õigesti, juhtsignaalid tekkisid ootuspäraselt ning andmeedastus ALU, registrifaili ja mälu vahel toimis korrektselt.

Lisaks testprogrammidele koostati ka C-keeles programm Fibonacci jada arvutamiseks. See programm genereeris järjestikuseid Fibonacci arve ning salvestas need andmemällu. Programmi edukas täitmine demonstreerib protsessori võimekust täita üldotstarbelisi ülesandeid ja hallata mälutoiminguid.

Kõik testid andsid oodatud tulemused. Mikrokontroller täitis käsud vastavalt RISC-V RV32IM spetsifikatsioonile ning kõrvalekaldeid ei täheldatud. Testitulemuste põhjal võib järeldada, et mikrokontroller töötab korrektselt ja suudab usaldusväärselt täita RV32IM käsustiku funktsionaalsust.

5 Kokkuvõte

Käesoleva bakalaurusetöö eesmärk oli realiseerida mikrokontrolleri simulatsioon Logisim Evolution tarkvaras, et illustreerida mikrokontrolleri sisemist tööpõhimõtet. Väljatöötatud simulatsioonis rakendati RV32IM käsustikul põhinevat protsessorituuma, mis juhib mikrokontrolleri üldist toimimist. Lisaks hõlmas simulatsioon eraldiseisvat mälustruktuuri ning mitmeid mäluruumile kaardistatud sisend-/väljundseadmeid.

Loodud süsteemi funktsionaalsust hinnati iseloodud testprogrammi abil, mille tulemusi võrreldi ootuspärase väljundiga. Kõik testid kinnitasid protsessorituuma korrektsust ja selle vastavust RISC-V standardile.

Tulevikusuunana nähakse võimalusi mikrokontrolleri arhitektuuri täiendamiseks, näiteks erineva pikkusega käskude toe lisamise, protsessorituuma torujuhtme (pipeline) arhitektuuri rakendamise või täiendavate funktsionaalsete moodulite, nagu graafilise liidese või krüptograafilise kiirendi, integreerimise kaudu.

viited

[1] Krste Asanović ja David A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Tehniline raport UCB/EECS-2014-146. August 2014. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html` (vaadatud 21.04.2024).

[2] Sasang Balachandran. "General purpose input output (gpio)". *Michigan State University College of Engineering. Published* (2009), lk. 08–11.

[3] R. Bannatyne ja G. Viot. "Introduction to microcontrollers". Teoses: *WESCON/97 Conference Proceedings*. 1997, lk. 564–574. DOI: `10.1109/WESCON.1997.632384`.

[4] Carl Bruch *et al. Logisim-evolution*. Oktoober 2022. URL: `https://github.com/logisim-evolution/logisim-evolution`.

[5] Carl Burch. "Logisim: A graphical system for logic circuit design and simulation". *Journal on Educational Resources in Computing (JERIC)* 2.1 (2002), lk. 5–16.

[6] Mikhail Chupilko, Alexander Kamkin ja Alexandr Protsenko. "Open-Source Validation Suite for RISC-V". Teoses: detsember 2019, lk. 7–12. DOI: `10.1109/MTV48867.2019.00010`.

[7] Mikhail Chupilko *et al.* "Test Program Generator MicroTESK for RISC-V". Teoses: *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. 2018, lk. 6–11. DOI: `10.1109/MTV.2018.00011`.

[8] RISC-V Collaboration. *RISC-V GNU Toolchain*. 2025. URL: `https://github.com/riscv-collab/riscv-gnu-toolchain` (vaadatud 28.04.2025).

[9] RISC-V Software Contributors. *riscv-tests: ISA Tests for RISC-V Architectures*. `https://github.com/riscv-software-src/riscv-tests`. Accessed: 2025-04-30. 2024.

[10] Atmel Coorporation. *Atmel ATmega328P datasheet*. 2015. URL: `https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf` (vaadatud 15.03.2025).

[11] Roberto Giorgi ja Gianfranco Mariotti. "WebRISC-V: a Web-Based Education-Oriented RISC-V Pipeline Simulation Environment". Teoses: *Proceedings of the Workshop on Computer Architecture Education*. WCAE'19. Phoenix, AZ, USA: Association for Computing Machinery, 2019. ISBN: 9781450368421. DOI: `10.1145/3338698.3338894`. URL: `https://doi.org/10.1145/3338698.3338894`.

[12] Microchip Technology Inc. *Hardware vs. Software Multiplication in Microcontrollers*. 2025. URL: https://onlinedocs.microchip.com/oxy/GUID-D364745C-8255-47AE-9528-9EB17646EE19-en-US-18/GUID-4CDA8BD0-8C39-47CE-8297-FB61A0890E23.html (vaadatud 18.02.2025).

[13] Abraham Kcholi. "Device Driver I/O and Interrupts". Teoses: *Pro Windows Embedded Compact 7: Producing Device Drivers*. Springer, 2011, lk. 127–144.

[14] Dragi Kimovski *et al.* "Beyond Von Neumann in the Computing Continuum: Architectures, Applications, and Future Directions". *IEEE Internet Computing* 28.3 (2024), lk. 6–16. DOI: 10.1109/MIC.2023.3301010.

[15] Marek Matej. *ESP32's family Memory Map 101*. 2024. URL: https://developer.espressif.com/blog/esp32-memory-map-101/ (vaadatud 16.03.2025).

[16] Eduardo Montañez ja Andrew Mastronardi. "Microcontrollers in education: Embedded control–everywhere and everyday". Teoses: *2005 Annual Conference*. 2005, lk. 10–938.

[17] Tammy Noergaard. *Embedded systems architecture: a comprehensive guide for engineers and programmers*. Newnes, 2012.

[18] C. V. Ramamoorthy ja H. F. Li. "Pipeline Architecture". *ACM Comput. Surv.* 9.1 (märts 1977), lk. 61–102. ISSN: 0360-0300. DOI: 10.1145/356683.356687. URL: https://doi.org/10.1145/356683.356687.

[19] Edwin D. Reilly. "Memory-mapped I/O". Teoses: *Encyclopedia of Computer Science*. GBR: John Wiley ja Sons Ltd., 2003, lk. 1152. ISBN: 0470864125.

[20] Ioan Susnea ja Marian Mitescu. *Microcontrollers in practice*. Köide 18. Springer Science & Business Media, 2005.

[21] Andrew Waterman, Krste Asanović ja John Hauser, toim. *The RISC-V instruction set manual, volume II: Privileged architecture*. Aprill 2024. URL: https://drive.google.com/file/d/17GeetSnT5wW3xNuAHI95-SI1gPGd5sJ_/view (vaadatud 12.05.2025).

[22] Andrew Waterman *et al.*, toim. *"The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20240411*. Aprill 2024. URL: https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view?pli=1 (vaadatud 12.05.2025).

[23] x86 Guide. *AMD Ryzen 9 7950X Specifications and Details*. 2025. URL: https://www.x86-guide.net/en/cpu/AMD%2BRyzen%2B9%2B7950X-cpu-no8361.html (vaadatud 14.03.2025).

Lisad

Lisa 1. Simulatsiooni failid

Töö käigus valminud simulatsiooni skeem, testimiseks loodud kood ja kompileerimist toetavad failid mille leiab aadreesilt: `https://github.com/peeter-virk/RISC-V_mikrokontrolleri_simulatsioon`

Lisa 2. Mikrokontrolleri registrite skeem

Joonis 15. Mikrokontrolleri registrid

Lisa 3. Implementeeritud protsessori käskude tabelid

käsu kodeering	käsk	parameetrid	kirjeldus
R	ADD	rs1, rs2, rd	liitmine rs1 ja rs2 registrite vahel
R	SUB	rs1, rs2, rd	rs2 lahutamine rs1-st
R	SLL	rs1, rs2, rd	rs1 vasaknihe rs2 võrra
R	SRL	rs1, rs2, rd	rs1 paremnihe rs2 võrra
R	SRA	rs1, rs2, rd	rs1 aritmeetiline paremnihe rs2 võrra
R	AND	rs1, rs2, rd	bittiülene AND tehe rs1 ja rs2 vahel
R	OR	rs1, rs2, rd	bittiülene OR tehe rs1 ja rs2 vahel
R	XOR	rs1, rs2, rd	bittiülene XOR tehe rs1 ja rs2 vahel
R	SLT	rs1, rs2, rd	salvestab 1 kui rs1 < rs2 märgiga kujul
R	SLTU	rs1, rs2, rd	salvestab 1 kui rs1 < rs2 märgita kujul
R	MUL	rs1, rs2, rd	rs1 korrutamine rs2-ga, salvestades alumised 32 bitti registrisse rd
R	MULH	rs1, rs2, rd	rs1 märgiga korrutamine rs2-ga, salvestades ülemised 32 bitti registrisse rd
R	MULHU	rs1, rs2, rd	rs1 märgita korrutamine rs2-ga, salvestades ülemised 32 bitti registrisse rd
R	MULHSU	rs1, rs2, rd	rs1 märgiga korrutamine märgita rs2-ga, salvestades ülemised 32 bitti registrisse rd
R	DIVU	rs1, rs2, rd	rs1 märgita täisarvuline jagamine rs2-ga, salvestades tulemuse rd registrisse
R	REMU	rs1, rs2, rd	rs1 märgita täisarvuline jagamine rs2-ga, salvestades jäägi rd registrisse
R	DIV	rs1, rs2, rd	rs1 märgiga täisarvuline jagamine rs2-ga, salvestades tulemuse rd registrisse
R	REM	rs1, rs2, rd	rs1 märgiga täisarvuline jagamine rs2-ga, salvestades jäägi rd registrisse

Tabel 2. R-tüüpi käsud [22]

käsu kodeering	käsk	parameetrid	kirjeldus
I	ADDI	rd, rs1, imm	liitmine rs1 ja imm väärtuse vahel
I	SUBI	rd, rs1, imm	imm väärtuse lahutamine rs1-st
I	SSLI	rd, rs1, imm	rs1 vasaknihe imm väärtuse võrra
I	SRLI	rd, rs1, imm	rs1 paremnihe imm väärtuse võrra
I	SRAI	rd, rs1, imm	rs1 aritmeetiline paremnihe imm väärtuse võrra
I	ANDI	rd, rs1, imm	bittiülene AND tehe rs1 ja imm väärtuse vahel
I	ORI	rd, rs1, imm	bittiülene OR tehe rs1 ja imm väärtuse vahel
I	XORI	rd, rs1, imm	bittiülene XOR tehe rs1 ja imm väärtuse vahel
I	SLTI	rd, rs1, imm	salvestab 1 kui rs1 < imm märgiga kujul
I	SLTUI	rd, rs1, imm	salvestab 1 kui rs1 < imm märgita kujul

Tabel 3. I-tüüpi aritmeetikakäsud [22]

käsu kodeering	käsk	parameetrid	kirjeldus
I	CSRRW	rd, csr, rs1	Vahetab CSR registri ja protsessori registrite väärtused, kasutades lähteregistriks registrit rs1 ja sihtregestriks registrit rd
I	CSRRS	rd, csr, rs1	Salvestab CSR registri registrisse rd ja teostab CSR registril loogilise liitmistehte registri rs1 väärtusega
I	CSRRC	rd, csr, rs1	Salvestab CSR registri registrisse rd ja teostab CSR registril loogilise korrutustehte registri rs1 väärtusega
I	CSRRWI	rd, csr, zimm	Teostab CSRRW operatsiooni kasutades nullpikendatud rs1 välja kohese väärtusena rs1 registri asemel
I	CSRRSI	rd, csr, zimm	Teostab CSRRS operatsiooni kasutades nullpikendatud rs1 välja kohese väärtusena rs1 registri asemel
I	CSRRCI	rd, csr, zimm	Teostab CSRRC operatsiooni kasutades nullpikendatud rs1 välja kohese väärtusena rs1 registri asemel

Tabel 4. CSR registritel opereerivad käsud [22]

käsu kodeering	käsk	parameetrid	kirjeldus
I	LB	rd, nihe(rs1)	loeb mälust märgiga baidi aadressilt nihe+rs1
I	LH	rd, nihe(rs1)	loeb mälust märgiga poolsõna aadressilt nihe+rs1
I	LW	rd, nihe(rs1)	loeb mälust märgiga sõna aadressilt nihe+rs1
I	LBU	rd, nihe(rs1)	loeb mälust märgita baidi aadressilt nihe+rs1
I	LHU	rd, nihe(rs1)	loeb mälust märgita poolsõna aadressilt nihe+rs1
S	SB	rs2, nihe(rs1)	salvestab baidi mällu aadressil nihe+rs1
S	SH	rs2, nihe(rs1)	salvestab poolsõna mällu aadressil nihe+rs1
S	SW	rs2, nihe(rs1)	salvestab sõna mällu aadressil nihe+rs1

Tabel 5. Mälu käsud [22]

käsu kodeering	käsk	parameetrid	kirjeldus
B	BEQ	rs1, rs2, nihe	kui rs1 = rs2, hüppab aadressile PC+nihe
B	BNE	rs1, rs2, nihe	kui rs1 != rs2, hüppab aadressile PC+nihe
B	BLT	rs1, rs2, nihe	kui rs1 < rs2, kus arvud on märgiga, hüppab aadressile PC+nihe
B	BGE	rs1, rs2, nihe	kui rs1 >= rs2, kus arvud on märgiga, hüppab aadressile PC+nihe
B	BLTU	rs1, rs2, nihe	kui rs1 < rs2, kus arvud on märgita, hüppab aadressile PC+nihe
B	BGEU	rs1, rs2, nihe	kui rs1 >= rs2, kus arvud on märgita, hüppab aadressile PC+nihe
J	JAL	rd, nihe	Hüppab aadressile PC + nihe ja salvestab PC + 4 registrisse rd
I	JALR	rd, rs1, nihe	Hüppab aadressile rs1 + nihe ja salvestab PC + 4 registrisse rd

Tabel 6. Haru- ja hüppekäsud [22]

käsu kodeering	käsk	parameetrid	kirjeldus
U	LUI	rd, imm	Salvestab väärtuse [imm « 12] registrisse rd
U	AUIPC	rd, imm	Liidab väärtuse [imm « 12] programmiloenduri väärtusele ja salvestab registrisse rd

Tabel 7. RV32I U-tüüpi käsud [22]

Käsu kodeering	Käsk	Parameetrid	Kirjeldus
I	`EBREAK`	-	Põhjustab EBREAK erindi
I	`ECALL`	-	Põhjustab ECALL erindi
-	`MRET`	-	Naaseb lõksust

Tabel 8. Katkestuste ja lõksude käsud [22, 21]

II. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Peeter Virk**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose

   RISC-V mikrokontrolleri simulation Logisim Evolution programmis,

   mille juhendaja(d) on Margus Rosin,

   reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.

3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.

4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Peeter Virk
21.05.2025