Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Kiril Tofiloski

Cevovodna izvedba modela CPE MiMo v2

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: viš. pred. dr. Robert Rozman

Ljubljana, 2025

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Kandidat: Kiril Tofiloski
Naslov: Cevovodna izvedba modela CPE MiMo
Vrsta naloge: Diplomska naloga na visokošolskem strokovnem študijskem programu prve stopnje Računalništvo in informatika
Mentor: viš. pred. dr. Robert Rozman

Opis:
Besedilo teme diplomskega dela študent prepiše iz študijskega informacijskega sistema, kamor ga je vnesel mentor. V nekaj stavkih bo opisal, kaj pričakuje od kandidatovega diplomskega dela. Kaj so cilji, kakšne metode naj uporabi, morda bo zapisal tudi ključno literaturo.

Title: Comprehensive Upgrade to the MiMo CPU Model

Description:
opis diplome v angleščini

Zahvaljujem se mentorju, viš. pred. dr. Robertu Rozmanu, za svetovanje in pomoč pri izdelavi diplomskega dela. Posebna zahvala gre moji družini in prijateljem za njihovo neomajno podporo skozi celotno obdobje mojega študija.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
PC	program counter	programski števec
IR	instruction register	ukazni register
RAM	random access memory	pomnilnik z naključnim dostopom
RISC	reduced instruction set computer	računalnik z zmanjšanim naborom ukazov
CISC	complex instruction set computer	računalnik s kompleksnim naborom ukazov
ARM	advanced RISC machines	napredni stroji RISC
CPI	clock cycles per instruction	število urnih period na ukaz
IF	instruction fetch	pridobivanje ukazov
ID	instruction decode	dekodiranje ukazov
EX	execute	izvedba ukazov
MA	memory access	dostop do pomnilnika
WB	write back	zapisovanje v registre
LHT	local history table	lokalna zgodovinska tabela
LPT	local prediction table	lokalna napovedna tabela
GPT	global prediction table	globalna napovedna tabela
GHR	global history register	globalni zgodovinski register
CPT	choice prediction table	izbirna napovedna tabela

Povzetek

Naslov: Cevovodna izvedba modela CPE MiMo v2

Avtor: Kiril Tofiloski

Namen diplomske naloge je nadgraditi obstoječi model procesorja MiMo, ki je pomembno orodje pri izvedbi predmeta Organizacija računalnikov v okviru strokovnega študijskega programa na Fakulteti za računalništvo in informatiko, Univerze v Ljubljani. Glavni cilj je poenostavljena, razumljiva, petstopenjska cevovodna izvedba centralne procesne enote ob uporabi osnovnih metod za povečanje učinkovitosti delovanja ob podatkovnih in kontrolnih nevarnostih. Oboje se pogosto pojavljajo v praksi in upočasnjujejo delovanje procesorja, zato smo dodali še nekaj izbranih metod za zmanjšanje škodljivega vpliva cevovodnih nevarnosti na hitrost delovanja. Dodatni cilj pa je bila tudi zasnova CPE, ki bi bila bolj podobna arhitekturi ARM procesorjev, ki se uporabljajo pri izvedbi še kar nekaj sorodnih predmetov. V nalogi so tako podrobneje opisani postopki zasnove, načrtovanja, izvedbe ter analize delovanja cevovodnega modela MiMo v simulacijskem okolju Logisim Evolution. Dodan je tudi ustrezni prevajalnik za zbirni jezik, napisan v programskem jeziku Python. Naloga je dosegla svoj glavni cilj nadgradnje modela procesorja MiMo in predstavlja dragoceno orodje pri učenju in razumevanju poteka dela cevovodnega CPU-ja.

Ključne besede: računalnik, strojna oprema, centralna procesna enota, organizacija računalnika, paralelizacija, prevajalnik, cevovod, ARM.

Abstract

Title: Pipelined implementation of the MiMo v2 CPU model

Author: Kiril Tofiloski

The objective of this diploma thesis is to enhance the existing MiMo CPU model design, which is an important tool used in the Computer Organization course within the professional study program at the Faculty of Computer and Information Science, University of Ljubljana. The main goal is a simplified, understandable, five-stage pipeline implementation of the central processing unit using basic methods to increase the efficiency of operations in the face of data and control hazards. These hazards occur frequently in practice and slow down the processor, so we have added a few more selected methods to reduce the harmful impact they cause to performance speed. An additional goal was the design of the CPU, which would be more similar to the architecture of ARM processors, which are used in the implementation of quite a few related subjects. The task thus describes in more detail the procedures for the design, planning, implementation and analysis of the operation of the MiMo pipeline model in the Logisim Evolution simulation environment. A corresponding assembly language compiler written in the Python programming language is also added. The diploma achieved it's primary goal of enhacnign the MiMo CPU model and represents a valuable tool in learning and understanding the workflow of a pipelined CPU.

Keywords: computer, hardware, central processing unit, computer organization, parallelization, compiler, pipeline, ARM.

Poglavje 1

Uvod

Računalniška organizacija je temeljna podlaga, na kateri je zgrajen svet sodobnih računalniških sistemov. Obravnava hierarhično strukturo računalniškega sistema na različnih ravneh abstrakcije, od logičnih vrat in digitalnih vezij do zapletenih hierarhij pomnilnika in cevovodov ukazov. Ta disciplina raziskuje, kako ti elementi sodelujejo pri izvajanju ukazov, upravljanju podatkov in opravljanju nalog, pri čemer upošteva tudi dejavnike, kot so optimizacija zmogljivosti, energetska učinkovitost in razširljivost.

Simulacija in uporaba modela MiMo ponujata vrsto prepričljivih prednosti kot orodji za poučevanje računalniške organizacije. Simulacija omogoča dinamično vizualno predstavitev abstraktnih konceptov, kar študentom olajša razumevanje zapletenih komponent in procesov v računalniškem sistemu. Študenti se lahko aktivno vključijo v predmet s praktičnim eksperimentiranjem v nadzorovanem okolju brez tveganja. Lahko spreminjajo parametre, izvajajo ukaze in opazujejo rezultate. Simulacije zagotavljajo takojšnje povratne informacije, kar študentom omogoča, da hitro ocenijo vpliv svojih oblikovalskih odločitev. Simulacijska okolja, kot je Logisim, se lahko prilagodijo širokemu razponu zahtevnosti, od osnovnih konceptov do naprednih tem. To poenostavi postopek osvajanja vsakega posameznega koraka in omogoča učinkovitejše učenje. Uporaba simulacij tudi odpravlja potrebo po fizični namestitvi strojne opreme, kar zmanjšuje časovne omejitve in omejenost virov.

Paralelizacija in cevovodno izvajanje sta ključna vidika računalniške organizacije in predstavljata velik del predmeta na fakulteti. Ti koncepti imajo ključno vlogo pri optimizaciji zmogljivosti, učinkovitosti in prepustnosti sodobnih računalniških sistemov. Paralelizacija vključuje hkratno izvajanje več opravil ali ukazov. Cevovodno izvajanje je domiselna tehnika, ki izvajanje ukazov razdeli na posamezne stopnje in omogoča hkratno obdelavo različnih stopenj več ukazov. To povzroči neprekinjen pretok ukazov skozi cevovod, kar znatno poveča skupno zmogljivost procesorja. Cevovodi omogočajo izkoriščanje vzporednosti na ravni ukazov, pri čemer se lahko različne stopnje različnih ukazov izvajajo sočasno.

Ta naloga se osredotoča na vključitev teh tehnik v model procesorja MiMo, kar študentom omogoča boljše razumevanje tega dela predmeta in boljši vpogled v specifične stopnje cevovoda, kot so pridobivanje ukazov, dekodiranje, izvajanje, dostop do pomnilnika in zapisovanje nazaj, ter s tem povezane nevarnosti, kot so odvisnosti podatkov in vprašanja kontrolnega toka. Drugi vidik te naloge vključuje tesnejšo uskladitev modela procesorja MiMo z modelom procesorja ARM, kar je še posebej pomembno, ker so se študenti s procesorji ARM ukvarjali že pri prejšnjih predmetih na fakulteti.

Diplomska naloga je razdeljena na šest poglavij, od katerih vsako obravnava poseben vidik. V drugem poglavju predstavimo osnovni model MiMo, njegove značilnosti in pomanjkljivosti. Nato se v tretjem in četrtem poglavju osredotočimo na prilagoditve, potrebne za tesnejšo uskladitev z arhitekturo ARM. V tem okviru predstavimo temeljni petstopenjski model cevovoda in pojasnimo njegovo vključitev v naš modelu MiMo v2. V petem poglavju sledi opis sprememb v krmilni enoti. V šestem poglavju so temeljno opisane vse stopnje našega cevovoda in njihovo delovanje. Nazadnje v sedmem poglavju raziskujemo vključitev dodatnih komponent, namenjenih zmanjševanju podatkovne in kontrolne nevarnosti cevovoda, pa opisujemo celotni postopek uporaba modela.

Poglavje 2

Predstavitev osnovnega modela MiMo v1

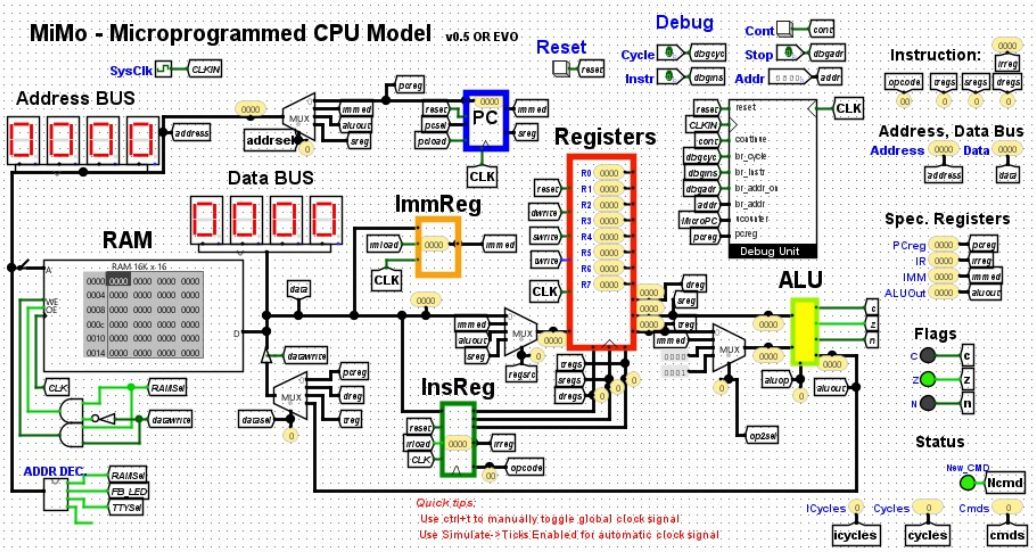
Osnovni model MiMo (sliki 2.1) temelji na že objavljenem modelu[4], v katerem je bilo nekaj stvari dodanih in prilagojenih za potrebe predmeta Organizacija računalnikov. Namen orodja je bil pomagati vizualno in interaktivno poučevati glavne dele vsake centralne procesne enote ter najpomembnejše koncepte, ki so potrebni pri njenem načrtovanju. V tem poglavju je najprej temeljito predstavljen osnovni model MiMo, v naslednjih poglavjih pa so predstavljene njegove spremembe.

V osnovnem modelu MiMo sta pomnilniška beseda in pomnilniški naslov dolga 16 bitov. Ukazi so lahko v dveh formatih: formatu 1 (16-bitni) in formatu 2 (32-bitni). Format 1 je sestavljen iz:

- operacijski kod (7 bitov), ki označuje, katera operacija obdelave operandov, skoka/vejitve ali nalaganja v pomnilniku se bo izvedla;
- ciljni register - Rd (3 biti), ki označuje register, v katerega je treba shraniti rezultat operacije;
- srednji register - Rs (3 biti), ki označuje enega od registrov, ki se uporablja pri operaciji;

- tretji register - Rt (3 biti), ki označuje drugi register, ki se uporablja pri operaciji.

Format 2 je enak prvemu formatu, skupaj s 16-bitno takojšnjo vrednostjo, ki se doda na koncu ukaza. Vsak ukaz se pridobi in izvede, preden se lahko pridobi naslednji. Izvajanje ukazov poteka v več urinih periodah oziroma elementarnih korakih. Najprej se prebere ukaz, nato se dekodira, preberejo se operandi, izvede se operacija ALE, rezultat se shrani v pomnilnik ali registre in nazadnje se posodobi programski števec.



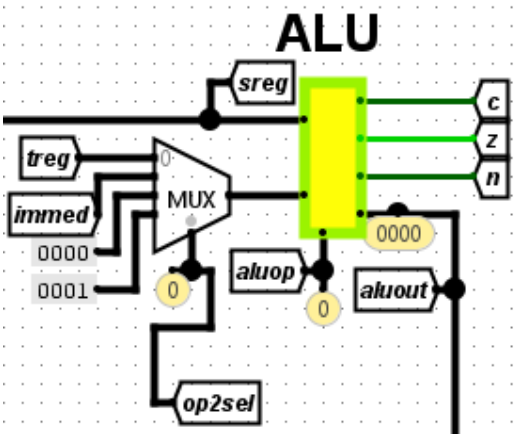
Slika 2.1: Osnovni model MiMo[5]

2.1 Aritmetično logična enota – ALE

ALE (sliki 2.2) za operacijo uporabi dva 16-bitna vhoda kot operanda. Prvi operand je vedno vrednost, shranjena v srednjem registru (`sreg`), drugi pa je izhod multiplekserja, ki ga določa krmilni signal `op2sel`. Možne vrednosti za drugi operand so vrednosti, shranjene v tretjem registru (`treg`), takojšnja vrednost (`immed`) ali konstanti 0 ali 1. Delovanje ALE določa krmilni signal

`aluop`. Izhodi ALE so rezultat izbrane operacije (`aluout`) in trije zastavice (c, z, n), ki se uporabljajo za pogojno izvajanje določenih ukazov.

- bit za prenos (c) je aktiven, ko se med seštevanjem ali odštevanjem izvede prenos;
- bit za ničlo (z) je aktiven, ko je rezultat operacije nič;
- bit za indikacijo negativnosti vrednosti števila (n) pa je aktiven, ko je rezultat operacije negativno število, tj. v običajnem načinu zapisa – dvojiškem komplementu, to pomeni, da je vrednost najpomembnejšega bita enaka 1.

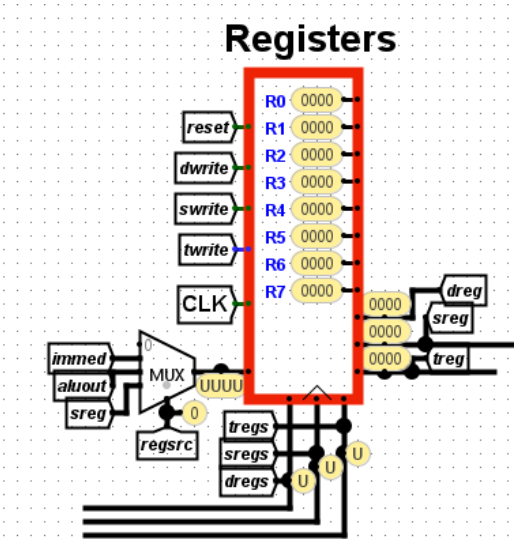


Slika 2.2: ALE v osnovnem modelu MiMo

2.2 Registrska enota – RE

Registri (sliki 2.3) so skupine hitrih pomnilniških mest v procesorju, ki igrajo ključno vlogo pri izvajanju operacij. V procesorju MiMo je vključenih osem 8-bitnih registrov za splošne namene. Izbor ustreznega registra za določen operand se upravlja s signali `dsel`, `ssel` in `tsel`. Krmilni signali `dwrite`, `swrite` in `twrite` določajo, ali se vhodni operandi zapišejo v katerega koli od teh določenih registrov.

Vsebina, ki se zapiše v registre, je določena s signalom `regsrc`. Operanda lahko prihaja iz operandnega vodila, takojšnjega registra, izhoda ALE ali drugega programsko izbranega registra (`sreg`).



Slika 2.3: Registri v osnovnem modelu MiMo

2.3 Programski števec – PC

Programski števec (PC) (sliki 2.4) ima temeljno vlogo pri izvajanju programov. Gre za register za posebne namene, v katerem se nahaja pomnilniški naslov naslednjega ukaza, ki ga je treba pridobiti in izvesti. Signal `pcload` označuje, kdaj je treba PC naložiti z novo vrednostjo, medtem ko signal

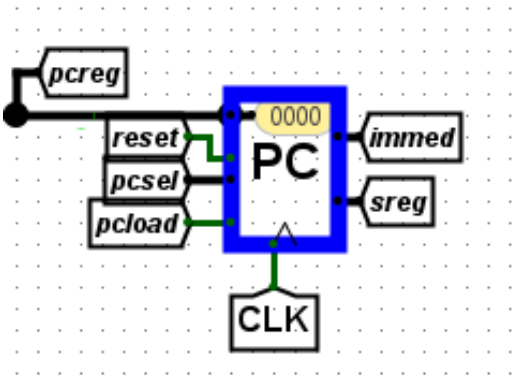
`pcsel` določa, katero vrednost je treba naložiti v PC. Možne vrednosti za signal `pcsel` so naslednje:

- `"PC + 1"` – povečanje trenutne vrednosti PC za 1 pomnilniški naslov: ta možnost se najpogosteje uporablja za zaporedno izvajanje ukazov po vrstnem redu, v katerem so shranjeni v pomnilniku;

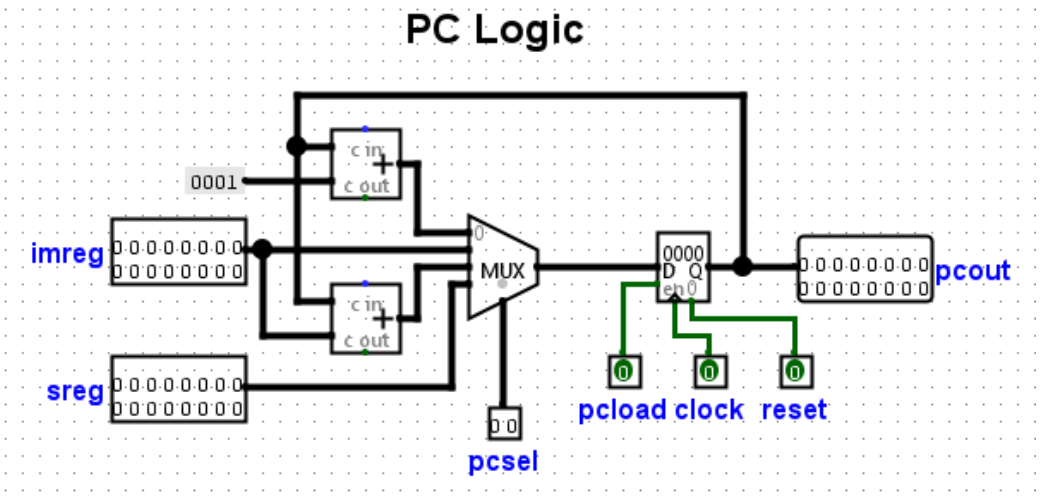
- `"immed"` – uporaba vrednosti iz takojšnjega registra: ta možnost se uporablja za absolutne skoke, ki omogočajo izvajanje pogojnih operacij, zank in klicev funkcij;

- `"PC + immed"` – inkrementiranje trenutne vrednosti PC za vrednost iz takojšnjega registra: podobno kot pri absolutnih skokih se tudi pri tej možnosti uporablja prekinitev zaporednega toka ukazov, vendar se v tem primeru uporablja vejitev, ki izračuna naslov vejitev glede na trenutni naslov PC;

- `"rx"` – uporaba vrednosti iz enega od registrov: ta možnost se običajno uporablja za klice podprogramov.



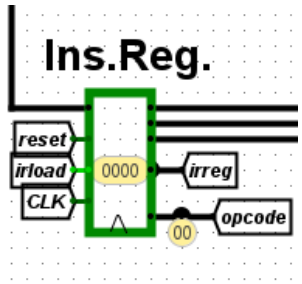
Slika 2.4: Programski števec v osnovnem modelu MiMo



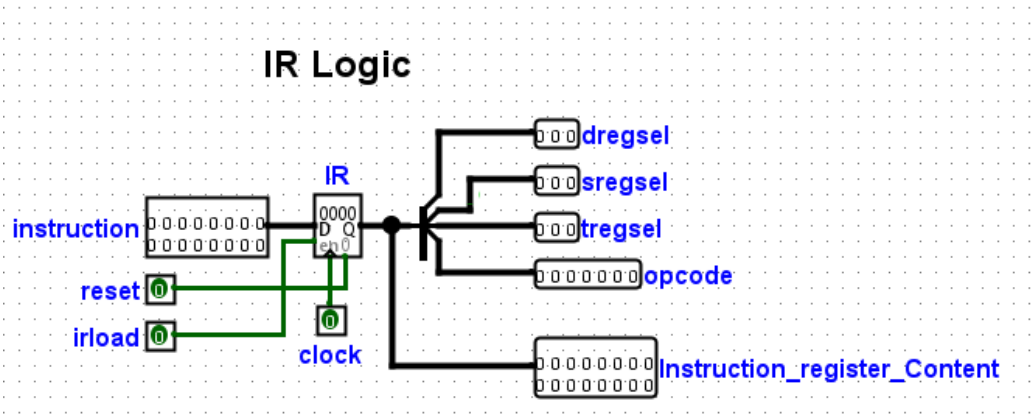
Slika 2.5: Notranje vezje programskega števca

2.4 Ukazni register – IR

Ukazni register (IR) (sliki 2.6) v procesorju služi kot začasno mesto za shranjevanje trenutnega ukaza. Uporablja se za dekodiranje in upravljanje kontrolnega toka. Pridobi ukaz, ki prihaja neposredno iz operandnega vodila, ter dekodira 3 registre, ki bodo uporabljeni za ukaz, v 3-bitne signale `dsel`, `ssel` in `tsel`, ki nato potujejo v registrsko enoto. Prav tako dekodira 7-bitno operacijsko kodo (`opcode`), ki določa, katera operacija se bo izvedla. Ta operacijska koda se nato prenese v krmilno enoto. Krmilni signal `irload` določa, kdaj naj ukazni register naloži novo vrednost iz podatkovnega vodila.



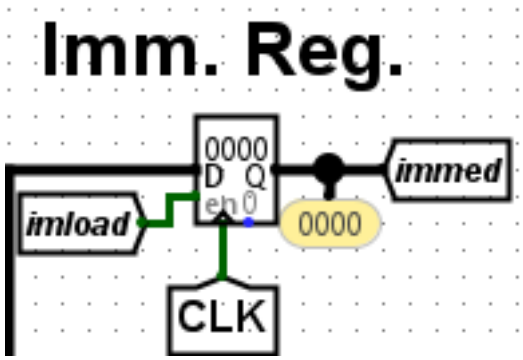
Slika 2.6: Ukazni register v osnovnem modelu MiMo



Slika 2.7: Notranje vezje ukaznega registra

2.5 Takojšnji register – IM

Takojšnji register (sliki 2.8) v modelu MiMo se uporablja za shranjevanje takojšnjega operanda ukaza med njegovo obdelavo. Signal `imload` določa, kdaj naj register naloži novo vrednost, ki prihaja iz podatkovnega vodila.



Slika 2.8: Takojšnji register v osnovnem modelu MiMo

2.6 Podatkovna in naslovna vodila

Vodilo je običajno skupina fizičnih povezav za prenose signalov med različnimi komponentami računalniškega sistema. Glede na namen poznamo več različnih

vodil. Podatkovno vodilo služi prenosu operandov, najpogosteje med pomnilnikom RAM in ostalimi enotami sistema. Omogoča prenos operandov v obe smeri, kar pomeni, da lahko operandi tečejo v pomnilnik RAM in iz njega.

Ukaz, pridobljen iz pomnilnika RAM, potuje v takojšnji register ali ukazni register in registrsko enoto. Signal `datawrite` določa, kdaj je treba operande zapisati v RAM. Signal `datasel` določa, katere operande je treba zapisati v RAM. Možni viri vključujejo:

- vrednost PC (`pcreg`), ki se uporablja za skoke v podprograme;
- vrednosti registrov `dreg` in `treg`, ki se uporabljata za shranjevanje operandov iz registrov v pomnilnik;
- rezultat, ki ga proizvede ALE (`aluout`).

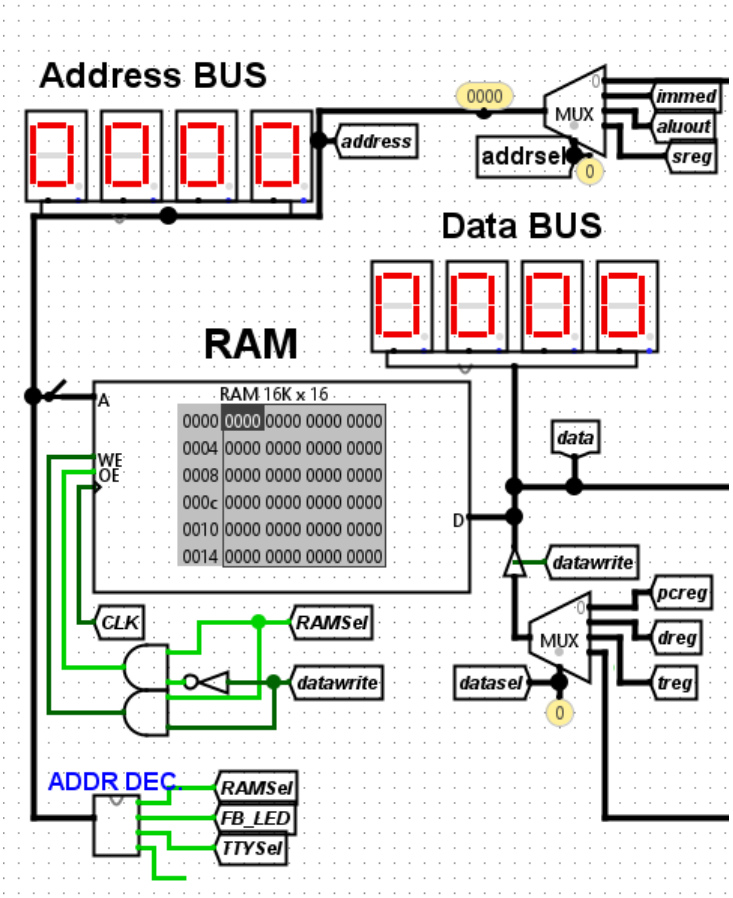
Podatkovno vodilo upravlja z operandi, ki se berejo iz pomnilnika RAM in zapisujejo vanj, medtem ko naslovno vodilo določa naslov v pomnilniku RAM za shranjevanje ali branje operandov. Signal `addrsel` določa notranji vir vsebine za naslovno vodilo, ki je lahko:

- PC: uporablja se za pridobitev naslova naslednjega ukaza, ki ga je treba izvesti po vrsti (Primer: `LI Rd, immed`);
- takojšnji register: uporablja se za neposredno pridobivanje operandov prek naslova, shranjenega v takojšnjem registru (Primer: `SW Rd, immed`);
- rezultat ALE: pogosto se uporablja za pridobivanje operandov iz dinamično izračunanega naslova, tj. posredno pridobivanje operandov z odmikom (Primer: `LWRI Rd,Rs,Rt`);
- registri (`sreg`): uporabljajo se za posredno pridobivanje operandov prek registra brez odmika (Primer: `POP Rd`).

Osnovni model MiMo uporablja tudi naslovni dekoder, ki določi napravo, v kateri se naslov nahaja in jo aktivira. Širina naslovnega vodila je 16 bitov,

medtem ko je širina naslova RAM-a 14 bitov. Dva najpomembnejša bita naslova določata, v kateri vhodno-izhodni napravi se naslov nahaja, in sicer:

- 00: naslov se nahaja v pomnilniku RAM;
- 01: naslov se nahaja v grafičnem zaslonu FB LED;
- 10: naslov se nahaja v serijskem terminalu TTY I/O;
- 11: neuporabljeno.



Slika 2.9: Vodili in pomnilnik RAM v osnovnem modelu MiMo

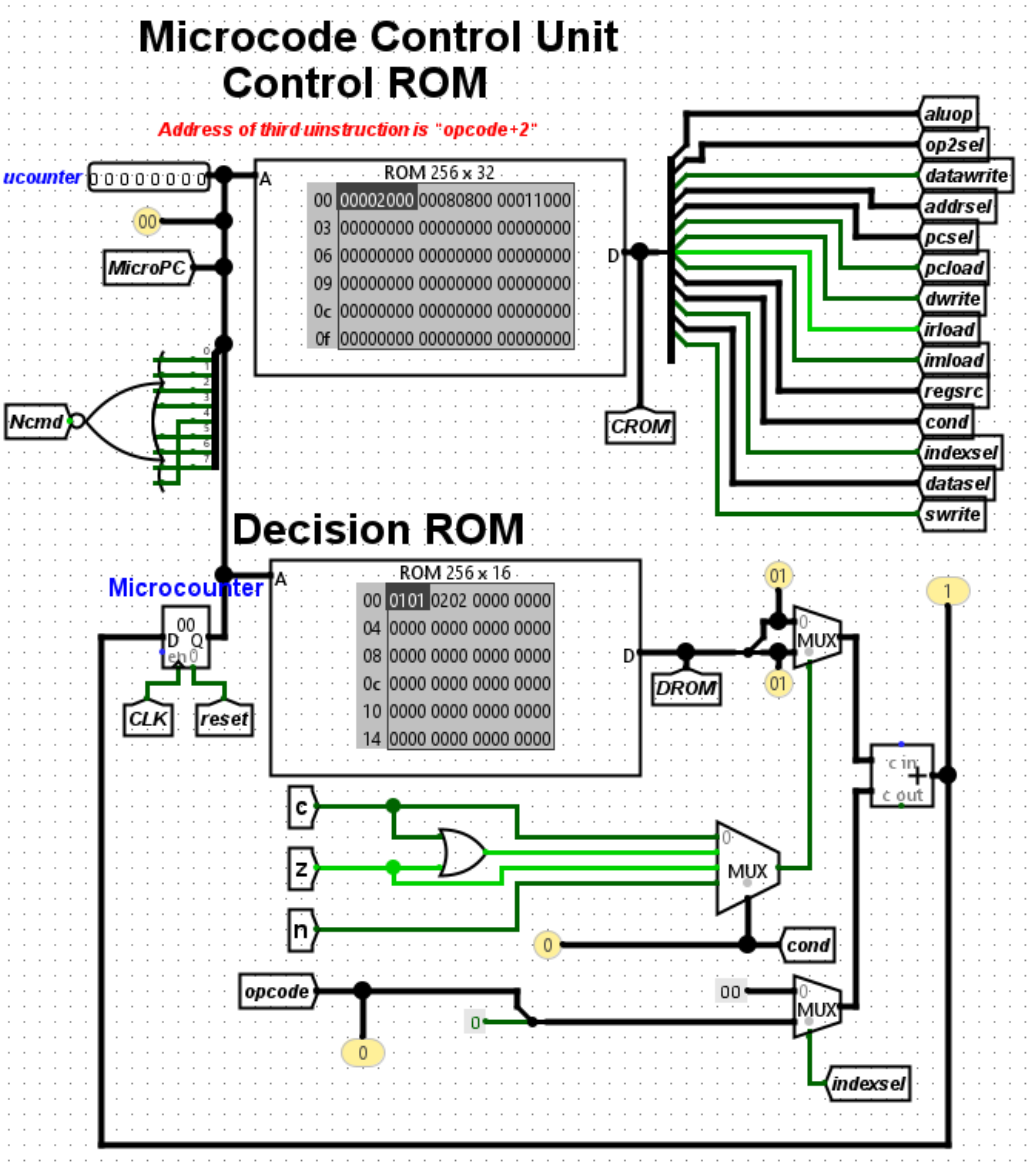
2.7 Krmilna enota – KE

Krmilna enotam (sliki 2.10) predstavlja "možgane" procesorja in je odgovorna za upravljanje izvajanja ukazov. V osnovnem procesorju MiMo je vsak ukaz sestavljen iz več mikroukazov. Vsak mikroukaz mora določiti stanje krmilnih signalov in naslednji mikroukaz, ki ga je treba izvesti. Krmilna enota kot vhod sprejme operacijsko kodo ukaza skupaj z zastavicami, ki prihajajo iz ALU. Na izhodu določi stanje za vse krmilne signale. Sestavljena je iz vezja, ki določa naslov mikroukaza, in dveh pomnilnikov ROM:

- 32-bitnega krmilnega pomnilnika ROM, ki hrani stanja vseh krmilnih signalov (1 naslov pomnilnika ROM vsebuje stanja za 1 mikroukaz);
- 16-bitnega odločitvenega ROM-a, ki določa naslov naslednjega mikroukaza.

Širina naslova krmilnega pomnilnika ROM-a je 8 bitov, zato je odločitveni pomnilnik ROM 16-bitni, saj vsaka pomnilniška beseda vsebuje dva naslova krmilnega pomnilnika ROM-a glede na to, ali je pogoj za izvedbo ukaza izpolnjen ali neizpolnjen. Prvi naslov se uporabi, ko je pogoj izpolnjen, drugi pa, ko pogoj ni izpolnjen.

Vsi ti elementi skupaj omogočajo pravilno izvajanje ukazov in učinkovito simulacijo delovanja centralne procesne enote. Čeprav je model procesorja MiMo uporabno učno orodje, se razlikuje od sodobnejših standardnih procesorjev, kot je ARM. Eden izmed glavnih izzivov trenutnega modela je pomanjkanje paralelizacije, saj se mora vsak mikroukaz izvesti v celoti, preden se pridobi naslednji. Paralelizacija in cevovodno izvajanje sta ključna koncepta pri predmetu Organizacija računalnikov, zato želimo ustvariti cevovodno izvedbo modela, ki bi olajšala razumevanje tega koncepta in se hkrati bolj približala arhitekturi procesorjev RISC oziroma ARM.



Slika 2.10: Krmilna enota v osnovnem modelu MiMo

Poglavje 3

Zasnova cevovodne izvedbe modela MiMo

3.1 Pregled arhitektur RISC in ARM

Preden se poglobimo v podrobnosti o spremembah zasnove osnovnega modela procesorja MiMo, moramo opisati izhodišča in temeljna načela, ki so privedla do teh sprememb. RISC je pristop k zasnovi, katere cilj je zagotoviti preproste, a zmogljive ukaze, ki se izvajajo v enem urinem periodu pri veliki taktni hitrosti. Načelo RISC se osredotoča na zmanjšanje števila in kompleksnosti ukazov, ki jih izvaja strojna oprema, saj je lažje zagotoviti večjo prilagodljivost in inteligenco v programski kot v strojni opremi. Zaradi tega ima zasnova RISC večje zahteve za prevajalnik. Nasprotno pa se tradicionalni računalnik s kompleksnim naborom ukazov (CISC) pri funkcionalnosti ukazov bolj zanaša na strojno opremo, zato so ukazi CISC bolj zapleteni. Načela RISC se izvaja s štirimi glavnimi pravili načrtovanja:

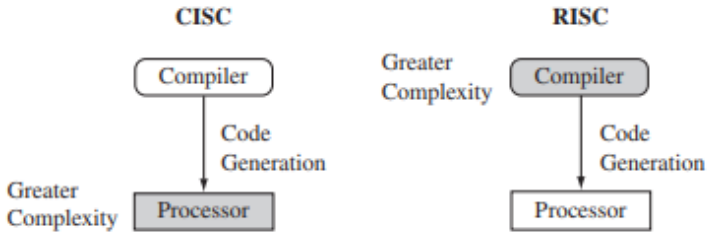
1. ukazi – procesorji RISC imajo manjše število razredov ukazov. Ti razredi omogočajo preproste operacije, ki se lahko izvedejo v enem urinem periodu. Vsak ukaz ima fiksno dolžino, da lahko cevovod pred dekodiranjem trenutnega ukaza pridobi prihodnje ukaze;

2. cevovodi – obdelava ukazov je razdeljena na manjše enote, ki se lahko izvajajo vzporedno v cevovodu. V idealnem primeru cevovod napreduje za en korak v vsakem urinem periodu, da bi zagotovil največjo prepustnost. Ukazi se lahko dekodirajo v eni stopnji cevovoda;

3. registri – računalniki RISC imajo velik nabor registrov za splošne namene. Vsak register lahko vsebuje podatke ali naslov. Registri delujejo kot hitra lokalna pomnilniška shramba za vse operacije obdelave operandov;

4. arhitektura Load-store – procesor deluje s podatki, ki so shranjeni v registrih. Ločeni ukazi za nalaganje in shranjevanje prenašajo podatke med registrsko enoto in zunanjim pomnilnikom. Dostopi do pomnilnika so časovno dragi, zato je ločitev dostopa do pomnilnika od obdelave podatkov prednost, saj lahko podatkovne elemente, shranjene v registrsko enoto, uporabite večkrat, ne da bi potrebovali večkratni dostop do pomnilnika.

Ta pravila zasnove omogočajo, da je procesor RISC preprostejši, zato lahko jedro deluje pri višjih taktnih frekvencah, lahko pa je tudi energetsko bolj učinkovit in zavzame manj prostora v elektronskem vezju.



Slika 3.1: CISC vs RISC[3]

V povezavi s načeli zasnove ARM obstaja več razlogov, ki so pripeljali do zasnove procesorjev ARM. Procesorji ARM so zasnovani za lahke, prenosne naprave, ki se napajajo iz baterije, vključno s pametnimi telefoni, prenosnimi

in tabličnimi računalniki, ter za vgrajene sisteme. Ti sistemi potrebujejo določeno obliko napajanja iz baterije, zato je bil procesor ARM posebej zasnovan tako, da je preprost in zavzame malo prostora, kar zmanjšuje porabo energije in omogoča enostavne razširitve.

Gostoto kode lahko definiramo kot skupno velikost (v bitih) vseh ukazov, potrebnih za izvedbo funkcije naprave. Velika gostota kode je še ena pomembna zahteva, saj imajo vgrajeni sistemi omejen pomnilnik zaradi stroškov in/ali omejitev fizične velikosti. Poleg tega so vgrajeni sistemi občutljivi na ceno in uporabljajo počasne in poceni pomnilniške naprave.

Druga pomembna zahteva je zmanjšanje površine vezja, ki jo zavzema vgrajeni procesor. Pri rešitvi z enim čipom velja, da čim manjšo površino zaseda vgrajeni procesor, tem več prostora je na voljo za specializirane periferne naprave. To pa zmanjša stroške načrtovanja in proizvodnje, saj je za končni rezultat potrebnih manj diskretnih čipov.

ARM je zgrajen na idejah RISC, z nekaterimi izjemami zaradi omejitev njegove primarne uporabe – vgrajenih sistemov. V današnjih vgrajenih sistemih je bolj pomembna zmogljivost glede na porabo energije in ne le absolutna zmogljivost.

Nabor ukazov ARM se od strožje definicije RISC razlikuje na več načinov, vendar moramo za naše namene in v povezavi z modelom MiMo poudariti le enega – pogojno izvajanje. Ukaz se izvede le, če je izpolnjen določen pogoj. Ta funkcija lahko izboljša zmogljivost in gostoto kode z zmanjšanjem števila ukazov za vejitev.

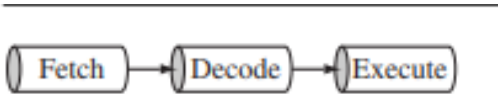
3.2 Splošne spremembe osnovnega modela

3.2.1 Cevovod in Harvardska arhitektura

Glavna in najpomembnejša sprememba osnovnega modela MiMo je cevovodna zasnova izvedbe ukazov. Cevovod je mehanizem, ki ga procesor RISC uporablja za vzporedno izvajanje ukazov. Uporaba cevovoda pospeši izvajanje ukazov, ker se hkrati izvaja pet ukazov – vsak v svoji stopnji cevovoda.

Vsaka izvaja določen elementarni korak, ki je potreben za izvedbo ukaza v celoti. Najpreprostejši cevovod, ki je bil uporabljen v prejšnjih jedrih ARM, je 3-stopenjski cevovod (sliki 3.2), ki vključuje naslednje stopnje:

1. pridobitev ukaza(Fetch): naloži ukaz iz pomnilnika;
2. dekodiranje(Decode): označuje ukaz, ki ga je treba izvesti;
3. izvedba(Execute): obdela ukaz in rezultat zapiše nazaj v register.



Slika 3.2: 3-stopenjski cevovod[3]

Čeprav je tristopenjski cevovod stroškovno učinkovit, ker je manj kompleksen in porabi manj energije, je zaradi potreb po večji zmogljivosti treba ponovno razmisliti o organizaciji procesorja. Čas, T_{prog}, ki je potreben za izvedbo določenega programa, je podan z:

$$T_{prog} = \frac{N_{inst} * CPI}{f_{clk}}$$

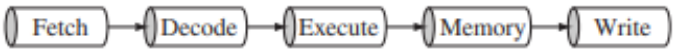
kjer je N_{inst} število ukazov ARM, izvedenih med izvajanjem programa, CPI povprečno število periodov na ukaz in f_{clk} frekvenca urina perioda procesorja. N_{inst} je za določen program konstanta, zato sta na voljo le dva načina za povečanje zmogljivosti:

- povečanje frekvence urinega signala, f_{clk}. To zahteva poenostavitev logike v vsaki stopnji cevovoda in s tem povečanje števila stopenj cevovoda;
- zmanjšanje povprečnega števila periodov na ukaz, CPI. To zahteva, da se zmanjšajo ukazi, ki zasedajo več kot eno stopnjo v cevovodu, ali da se zmanjšajo zastoji cevovoda zaradi odvisnosti med ukazi, ali pa kombinacijo obojega.

Temeljna težava pri zmanjševanju CPI v primerjavi s tristopenjskim jedrom je povezana z von Neumannovim ozkim grlom – zmogljivost vsakega računalnika s shranjenim programom z enim samim pomnilnikom ukazov in podatkov je omejena z razpoložljivo pasovno širino pomnilnika. Tristopenjsko jedro ARM dostopa do pomnilnika v (skoraj) vsakem periodu, bodisi za pridobitev ukaza bodisi za prenos podatkov.

Zato zmogljivejša jedra ARM uporabljajo večstopenjski cevovod (v našem primeru 5) ter imajo ločene predpomnilnike za ukaze in operande. Če izvajanje ukazov razdelimo na pet delov namesto na tri, se poenostavi opravilo, ki ga je treba opraviti v enem urinem periodu, in s tem omogoči uporabo višje frekvence periodov. Ločena pomnilnika ukazov in podatkov, kar je znano kot harvardska arhitektura, omogočata znatno zmanjšanje CPI jedra.

Tako smo vzpostavili standardni petstopenjski cevovod (sliki 3.3) in uvedli ločena pomnilnika RAM za ukaze in operande.



Slika 3.3: 5-stopenjski cevovod[3]

Cevovod tako ima naslednje stopnje:

1. Pridobivanje ukazov (Instruction Fetch – IF): ukaz se prikliče iz pomnilnika in postavi v cevovod ukazov;

2. Dekodiranje ukazov (Instruction Decode – ID): ukaz se dekodira in operandi registrov se preberejo iz registrske enote. V registrski enoti so tri vrata za dostop do operandov, zato lahko večina ukazov ARM v enem periodu prebere vse svoje operande;

3. Izvedba (Execute – EX): ustvari se rezultat v ALE. Če je ukaz nalaganje ali shranjevanje, se pomnilniški naslov izračuna v ALE;

4. Dostop do pomnilnika (Memory Access – MA): do operandnega pomnilnika se dostopa po potrebi. V nasprotnem primeru se rezultat

ALE preprosto shrani za ena urina perioda, da se zagotovi enak pretok cevovoda za vse ukaze;

5. Zapisovanje nazaj (Write Back – WB): rezultat, ki ga ustvari ukaz (iz ALE ali operandnega pomnilnika), se zapiše nazaj v register, vključno z vsemi podatki, naloženimi iz pomnilnika.

Posebnosti vsake stopnje in njihovo izvajanje v našem modelu CPU so pojasnjene v 6. poglavju.

3.2.2 Prehod s 16-bitnega na 32-bitni nabor ukazov

Osnovni model MiMo je 16-bitni procesor s 16-bitnim naborom ukazov in naslovnim prostorom. Pri razvoju našega cevovodnega modela smo se soočili s ključno odločitvijo o arhitekturi: ali ohraniti preprosto 16-bitno zasnovo ali preiti na 32-bitni sistem, bolj podoben standardnemu jedru ARM.

Zaradi omejitev, ki jih je postavilo simulacijsko okolje Logisim Evolution, smo obdržali 16-bitno dolžino naslova. To simulacijsko okolje vključuje module RAM z omejenim naslovnim prostorom, ki ne dopuščajo uporabe 32-bitne dolžine naslova.

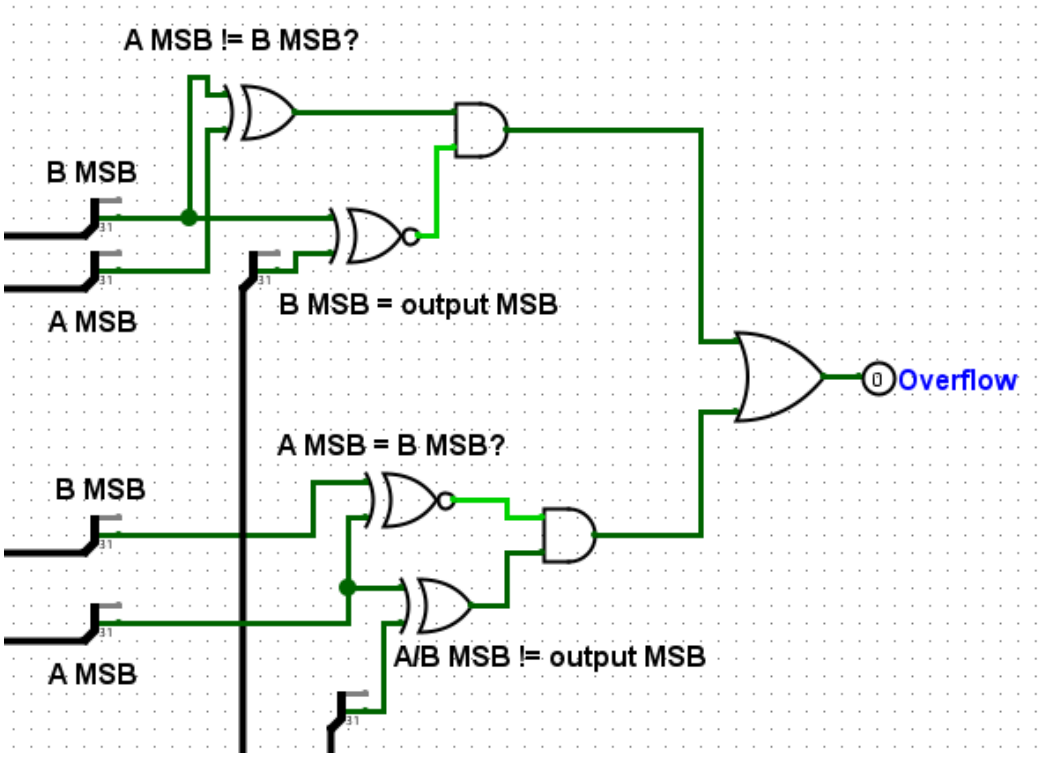
Kar zadeva zapis ukazov oziroma format, smo se zaradi približevanja arhitekturi ARM odločili za povečanje dolžine na 32 bitov. V osnovni različici modela MiMo smo imeli 2 formata različnih dolžin. Edina razlika je bila, da je daljši format vseboval še 16 bitov takojšnjega naslova. Z 32-bitnimi zapisi ukazov lahko v enotni format ukaza vključimo tudi takojšnji operand.

V cevovodnih izvedbah je mogoče učinkovito realizirati le ukazni nabor enake dolžine. Tudi zato smo se odločili za prehod na enotno dolžino zapisa ukazov – 32 bitov. Znotraj tega zapisa pa bo lahko v našem primeru takojšnja vrednost zasedla 10 bitov. Ta pristop je v skladu s standardno arhitekturo jedra ARM. Slabost tega pristopa je seveda ta, da smo bolj omejeni pri velikosti naših takojšnjih vrednosti.

3.2.3 Dodatni statusni bit preliva (V)

Druga komponenta standardnih arhitektur ARM, ki je osnovni model MiMo ni vključeval, je bil bit preliva kot del zastavic ALE. To je majhen, a pomemben dodatek k našemu modelu.

V aritmetično-logični enoti (ALE) procesorja ima bit preliva ključno vlogo pri obvladovanju številskega preliva, ki se pojavi, ko rezultat aritmetične operacije preseže številski obseg, ki ga predstavi zapis v določenem številu bitov. Bit preliva v bistvu signalizira, da je izračunan rezultat prevelik ali premajhen, da bi ga natančno predstavili v dani dolžini bitov. To je pomembno pri operacijah s celimi števili s predznakom, saj lahko prelivanje označi nepričakovane napake ali napačno interpretacijo pri numeričnih izračunih.



Slika 3.4: Notranje vezje za določitev preliva

V zgornji figuri je prikazano vezje v cevovodnem modelu, ki določa preliv. Preliv se lahko zgodi v dveh primerih:

1. pri operaciji odštevanja, ko najpomembnejši bit prvega operanda ni enak najpomembnejšemu bitu drugega operanda, vendar je najpomembnejši bit drugega operanda enak najpomembnejšemu bitu izhoda operacije; takrat se zgodi preliv;

2. pri operaciji seštevanja se preliv zgodi, ko imata oba operanda isti najpomembnejši bit in ta bit ni enak najpomembnejšemu bitu izhodnega operanda.

3.2.4 Dodatek register povratnih naslovov Link

Pogosto je treba izvesti skoke za klic podprograma, kjer se mora po koncu podprograma izvesti naslednji ukaz, ki je bil na vrsti pred klicem podprograma. Takšne klice imenujemo klici podprogramov in so dovolj pogosti, da večina arhitektur vključuje posebna ukaza za njihovo učinkovitost.

Osnovni model MiMo ne vsebuje posebnega registra za klice podprogramov – za to je bil predviden kar `r7`. V tem registru se ob klicu podprograma shrani naslov za vrnitev, ki pravzaprav predstavlja naslov naslednjega ukaza (vsebina PC). V cevovodni nadgradnji smo dodali poseben register povratnih naslovov, imenovan kot Link register, da bomo lahko realizirali klice podprogramov z ukazom `bl` (branch with link).

Uporaba enega od običajnih registrov v našem cevovodnem procesorju ni smiselna, saj se vrednosti registrov v cevovodnem procesorju posodobijo v zadnji stopnji, Write-back. Pri ukazih, ki spremenijo tok izvrševanja zaporednih ukazov, kot so skoki ali vejitve, se po dekodiranju naslova v ID takoj posodobi sprememba zaporedja ukazov, da se cevovod ne napolni z napačnimi ukazi. Če bi uporabili splošni register, ki je del registrske enote za klice podprogramov, spremembe zaporedja ne bi mogle biti takoj obdelane, saj bi register vseboval pravilno vrednost šele tri stopnje pozneje, na koncu stopnje WB.

Uporaba posebnega registra nam omogoča, da v register Link shranimo povratni naslov podprograma takoj, ko dekodiramo ukaz v stopnje ID.

Poglavje 4

Ukazna arhitektura MiMo v2

V tem poglavju je opisana ukazna arhitektura našega cevovodnega modela in njene posebnosti.

4.1 Nabor ukazov

Celotni nabor ukazov modela MiMo v2 je podan v tabeli 4.1. V novem naboru ukazov smo uporabili večino standardnih ukazov ARM.

Iz standardnega nabora ukazov ARM smo izključili ukaze, specifične za arhitekturo ARM, katerih ni bilo smiselno dodati v naš procesor ali bi bilo potrebnega preveč dodatnega vezja in sprememb arhitekture (kot je primer z ukazi load/store multiple), da bi omogočili vključitev teh ukazov. Zaradi tega smo izključili naslednje ukaze:

- bolj zapletene ukaze za množenje – `mla, umull, umlal, smull, smlal`;
- ukaze, ki spreminjajo CPSR in SPSR (v MiMo v2 ne obstajata kot posebna registra) – `mrs, msr`;
- ukaze iz tipa Load/Store Multiple – `ldm, stm`;
- seštevanje/odštevanje s prenosom (v MiMo v2 se prenos samodejno upošteva pri ukazih `add, sub`) – `adc, sbc`;

- ostale – `bx, adr`.

Iz osnovnega modela MiMo v1 smo vključili ukaza `j` in `rts`. V naboru ukazov ARM ne obstaja ukaz za vrnitev iz podprograma, kot je `rts`; v ARM kodi mora uporabnik neposredno spremeniti vrednost programskega števca preko ukaza `mov pc, lr`. Ta ukaz pomembno pomaga pri poenostavitvi te operacije za potrebe učenja, kar je glavni cilj modela MiMo v2.

4.2 Format ukaza

Celotni ukaz modela MiMo v2 lahko razdelimo na naslednje dele:

- operacijska koda (`opcode`) – 7 bitov;
- pogojna koda (`Cond`) – 4 biti;
- `imload` – 1 bit;
- nastavitev zastavic (Set flags - `S`) – 1 bit;
- `Rd` – 3 biti;
- `Rs` – 3 biti;
- `Rt` – 3 biti;
- takojšnja vrednost (`immed`) – 10 bitov.

Celotni format ukaza je podan v sliki 4.1. V nadaljevanju bomo podrobneje opisali dodane dele vsakega ukaza.

opcode	imload	S	Cond	Rt	Rs	Rd	immed
31 - 25	24	23	22 - 19	18 - 16	15 - 13	12 - 10	09 - 00

Slika 4.1: Format ukaza MiMo v2

Instruction Code	Arguments	Function
mov	Rd, Rs/Immediate	Rd ← Rs/Immediate
mvn	Rd, Rs/Immediate	Rd ← NOT Rs/Immediate
add	Rd, Rs, Rt/Immediate	Rd ← Rs + Rt/Immediate
sub	Rd, Rs, Rt/Immediate	Rd ← Rs - Rt/Immediate
rsb	Rd, Rs, Rt/Immediate	Rd ← Rt/Immediate - Rs
mul	Rd, Rs, Rt/Immediate	Rd ← Rs * Rt/Immediate
div	Rd, Rs, Rt/Immediate	Rd ← Rs / Rt/Immediate
rem	Rd, Rs, Rt/Immediate	Rd ← Rs % Rt/Immediate
and	Rd, Rs, Rt/Immediate	Rd ← Rs AND Rt/Immediate
orr	Rd, Rs, Rt/Immediate	Rd ← Rs OR Rt/Immediate
eor	Rd, Rs, Rt/Immediate	Rd ← Rs XOR Rt/Immediate
nand	Rd, Rs, Rt/Immediate	Rd ← Rs NAND Rt/Immediate
nor	Rd, Rs, Rt/Immediate	Rd ← Rs NOR Rt/Immediate
bic	Rd, Rs, Rt/Immediate	Rd ← Rs AND NOT(Rt/Immediate)
cmp	Rs, Rt/Immediate	Rs - Rt, Set flags
cmn	Rs, Rt/Immediate	Rs + Rt, Set flags
tst	Rs, Rt/Immediate	Rs XOR Rt, Set flags
teq	Rs, Rt/Immediate	Rs AND Rt, Set flags
lsl	Rd, Rs, Rt/Immediate	Rd ← Rs << Rt/Immediate
lsr	Rd, Rs, Rt/Immediate	Rd ← Rs >> Rt/Immediate
asr	Rd, Rs, Rt/Immediate	Rd ← Rs >> Rt/Immediate Filled bits are sign bit
ror	Rd, Rs, Rt/Immediate	Rd ← Rs ROLL RIGHT Rt/Immediate
rol	Rd, Rs, Rt/Immediate	Rd ← Rs ROLL LEFT Rt/Immediate
j	Rs/Immediate/Label	PC ← Rs/Immediate/Label
b	Rs/Immediate/Label	PC ← PC + Rs/Immediate/Label
bl	Label	Jump to subroutine Load Link Register with return address
rts	None	Return from subroutine Always unconditional
ldr	Rd, Immediate Rd, [Rs] Rd, [Rs, Immediate/Rt]	Rd ← M[Immediate] Rd ← M[Rs] Rd ← M[Rs + Immediate/Rt]
str	Rd, Immediate Rs, [Rd] Rd, [Rs, Immediate/Rt]	M[Immediate] ← Rd M[Rd] ← Rs M[Rs + Immediate/Rt] ← Rd
nop	None	No operation

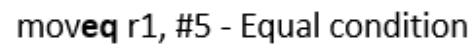
Tabela 4.1: Nabor ukazov v MiMo v2

4.3 Pogojno izvajanje ukazov

Prva pomembna sprememba se nanaša na obravnavo pogojnega izvajanja, kjer se ukaz izvede le, če je izpolnjen določen pogoj.

V osnovni različici modelu MiMo v1 je bilo za vsako pogojno operacijo potrebno izvesti ločen ukaz v mikrozbirniku, kar je povzročilo nepotrebne dodatne mikroukaze.

V cevovodni različici MiMo v2 smo sprejeli pristop, ki je bolj v skladu z običajnim jedrom ARM, tako da smo v vsak ukaz dodali neobvezno polje pogoja. To polje predstavlja dvočrkovno oznako, ki je dodana imenu ukaza in v našem primeru zavzema 4 bite prostora za ukaz. Primer je podan na sliki 4.2.



Slika 4.2: Primer ukaza s pogojnim izvajanjem

Pogojno izvajanje je odvisno od dveh komponent: polja pogojev in zastavic pogojev. Pogojno polje se nahaja v ukazu, medtem ko so zastavice pogojev prisotne v stopnji izvajanja (EX) našega modela. Pogoj se preveri v stopnji ID v vezju `check_condition`. Vsi pogoji temeljijo na pogojnih oznakah ARM ISA. Navedene so v tabeli 4.2.

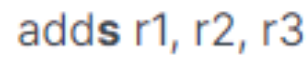
4.4 Dodatni bit za nastavitev vpliva na zastavice (S)

V osnovni različici MiMo so vsi ukazi spreminjali zastavice ALE. Zdaj pa ima vsak ukaz rezerviran dodaten bit za nastavitev vpliva na zastavice in spreminja stanje zastavic samo, če je ta bit nastavljen.

Samo primerjalna ukaza in ukaza za obdelavo podatkov, označena z dodano oznako 'S', posodabljata stanje zastavic. Primer je podan na sliki 4.3

Mnemonic	**Description**	**Flags State**
EQ	Equal	Z set
NE	Not equal	Z clear
CS	Carry set/unsigned higher or same	C set
CC	Carry clear/unsigned lower	C clear
MI	Minus/negative	N set
PL	Plus/positive	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
GE	Signed greater than or equal	N = V
LT	Signed less than	N != V
GT	Signed greater than	Z clear and N = V
LE	Signed less than or equal	Z set and N != V
AL	Always/unconditional	Irrelevant

Tabela 4.2: Pogoji za izvedbo ukaza ARM



Slika 4.3: Primer ukaza, ki vpliva na stanje zastavic

To je še ena sprememba, ki cevovodni model približuje modelom procesorja ARM. Poleg tega prispeva k zmanjšanju tveganj v cevovodnem sistemu, saj, v primeru, da prejšnji ukaz ni vplival na zastavice, pogojna operacija že v stopnji dekodiranja ukazov (ID) pozna rezultat svojega pogoja.

4.5 Takojšnja vrednost kot del ukaza

Kot je bilo že razloženo v prejšnjem poglavju, je v našem cevovodnem modelu takojšnja vrednost vključena kot del 32-bitnega ukaza, namesto da bi bila v ločenem naslednjem ukazu.

Za označevanje prisotnosti takojšnje vrednosti v našem modelu rezerviramo 1 bit v ukazu, ki je označen kot bit `imload`. Ko je ta bit aktiven, procesor ve, da ukaz vsebuje predvideno takojšnjo vrednost. Na ta način ni potrebe po ločenih ukazih za uporabo takojšnjih vrednosti, kot je to bilo v osnovnem modelu MiMo.

4.6 Sintaksa ukazov v zbirniku

Za sintakso ukazov smo uporabili poenostavljeno sintakso ARM Assembly prikazano v tabeli 4.3.

Kombinacije v obliki [r1, r2] niso bile dodane za noben ukaz razen ukaza `ldr` in `str`.

Oddelek `.data` izpiše drugo datoteko ram za operand RAM, če se uporablja. Programer mora najprej deklarirati `.data`, nato pa sledijo podatki, ki jih želi dodati. Primer je podan na sliki 4.4. Nato mora programer pred deklariranjem ukazov deklarirati `.text`, podobno kot pri standardnem ARM.

Simbol	Namen	Primer
/* */	Več vrstični komentar	/*not read by assembler*/
@	Komentar v eni vrstici	@also not read
#	Označuje takojšnjo vrednost	mov r1, #1
.text	Označuje začetek sekcije z ukazi	add r1,r2,r3 @this instruction would not be read .text add r1,r2,r3 @this one would
.data	Označuje začetek sekcije z podatki	.word 0xff, 0x23 @this data would not be read .data .asciiz "Hello World" @this data would

Tabela 4.3: MiMo v2 sintaksa

Direktiva `.word` določa pomnilniške besede, ki se dodajo v operandni RAM. Zapisane so lahko v desetiški ali šestnajstiški obliki.

Direktiva `.space` določa količino prostora (bajtov), ki ostane prazen v operandnem pomnilniku RAM.

Direktivi `.ascii` in `.asciiz` določata niza ASCII, ki ju je treba dodati operandnemu pomnilniku RAM. Nista poravnana in zavzameta toliko prostora, kolikor je treba. Edina razlika je, da .asciiz vsakemu nizu doda zaključni ničelni znak (prazen bajt).

4.7 Prevajalnika zbirnik in mikrozbirnik

Da bi omogočili te spremembe nabora ukazov, sta bila potrebna nova prevajalnika: zbirnik in mikrozbirnik. Oba sta bila za ta projekt razvita v Pythonu.

Zbirnik je temeljna komponenta, ki je odgovorna za prevajanje človeku

```
.data

.word 1,2, 3 , 4
.word 0x1d, 0xff,0x23

.space 12

.ascii "Hello World", "aaa","bbb"
.asciiz "testing", "ccc"
```

Slika 4.4: vsebina `.data` sekcije

berljive kode jezika zbirnik v strojne ukaze, ki jih lahko neposredno izvede procesor. Med postopkom prevajanja zbirnik ustvari zaporedje strojnih ukazov, ki predstavijo operacijski kodi, opredeljeni v mikrokodi.

Mikrozbirnik je odgovoren za prevajanje teh operacij na višji ravni v mikroukaze na nižji ravni, ki določajo stanje krmilnih signalov (podrobneje opisano v naslednjem poglavju) v procesorju.

Mikrozbirnik pomembno prispeva k zmožnosti procesorja za izvajanje širokega nabora ukazov, saj ponuja kompromis med strojno zapletenostjo in vsestranskostjo nabora ukazov.

Poglavje 5

Krmilni signali MiMo v2

Krmilni signali so ključni del procesorja in omogočajo izvajanje različnih ukazov, usmerjanje pretoka podatkov in uravnavanje celotnega delovanja centralne procesne enote. V tem poglavju bomo pregledali, kako so ti signali drugače izvedeni v našem cevovodnem modelu procesorja, da bi se bolje uskladili z našim novim formatom ukazov.

5.1 Pogojno izvajanje in odstranitev odločitvenega pomnilnika ROM

V osnovni različici modela MiMo ni obstajal zapis pogoja kot del formata ukaza, zato je bilo potrebno izvedbo ukazov s pogojem sprogramirati s pomočjo mikroprograma in odločitvenega ROM pomnilnika.

```
# JNEZ Rs,Immed  ; If Rs != 0, PC <- immed else  PC <- PC + 2
40:     addrsel=pc  imload=1
        aluop=sub  op2sel=const0, if z then pcincr else jump
```

Slika 5.1: Primer mikroprogramiranega pogojnega ukaza JNEZ v MiMo v1

V cevovodnem modelu MiMo vsak zapis ukaza vsebuje posebno polje za zapis pogoja, zato se zdi programiranje vsakega ukaza posebej nepotrebno.

Ugotovili smo, da je elegantnejša rešitev odstranitev odločitvenega ROM-a in izvajanje pogojnega preverjanja v strojni opremi. Novo pogojno izvajanje poteka na naslednji način:

1. signal `condition_met` skupaj z `opcode` (polje operacijsko kodo v ukazu) pride iz stopnje ID. Če je v stopnji EX ukaz, ki spreminja zastavice, mora uporabnik v kodo dodati ukaz `nop` pred trenutni ukaz. V različicah 2.1, 2.2 in 2.3 modela MiMo, CPE samodejno doda mehurček ali pa premošča zastavice v tem primeru;

2. prvi naslov v našem krmilnem pomnilniku ROM je rezerviran za ukaz `nop` in se uporabi v primerih, ko pogoj ni izpolnjen. Ukaz `nop` prestavi stanje mirovanja trenutne stopnje in se uporabi kot ukaz, ki ne vpliva na stanje nobenega kontrolnega signala. To stanje mirovanja imenujemo tudi mehurček;

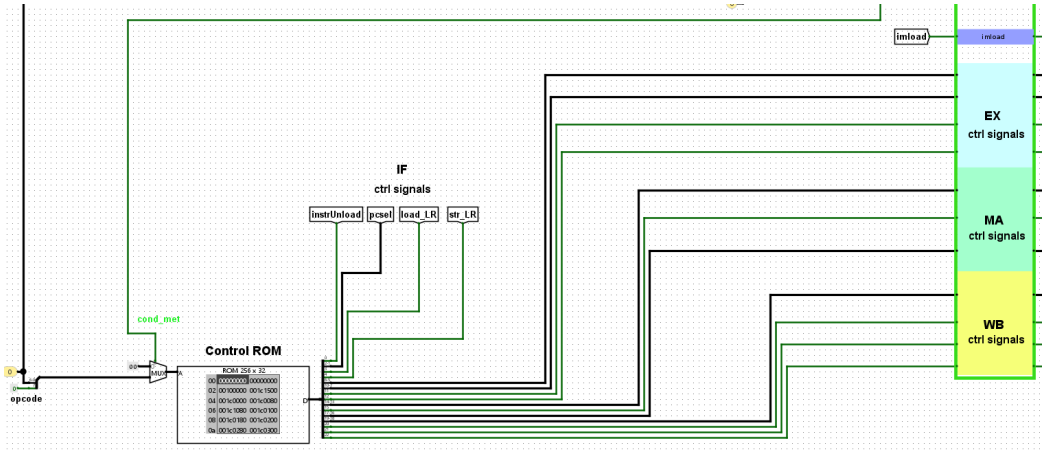
3. signal `condition_met` nam sporoča, kdaj je pogoj izpolnjen, zato lahko izvedemo ukaz. Ko je aktiven, krmilni (Control) ROM kot naslov prevzame `opcode`, ko pa ni aktiven, prevzame prvi naslov krmilni ROM – `nop`, tj. mehurček, ker pogoj ni bil izpolnjen;

4. vsak naslov ROM vsebuje vrednosti za vse krmilne signala v trenutnem mikroukazu.

Ta zasnova omogoča lažje razumljivo izvedbo mikroprogramirane krmilne enote. Z odstranitvijo odločitvenega ROM-a se zmanjšata tudi zapletenost in poenostavi delo, ki ga mora opraviti mikrozbirnik.

5.2 Razporeditev krmilnih signalov

Ker gre seveda za cevovodni procesor, bodo nekateri signali potrebni v kasnejših stopnjah kot drugi. Da bi se temu prilagodili, signali potujejo skozi cevovod preko vmesnih stopenj, kot so ID->EX, EX->MA.



Slika 5.2: Razporeditev krmilnih signalov

Signali, ki so potrebni za stopnjo EX, na primer, potujejo iz krmilnega ROM-a v stopnjo EX preko vmesne stopnje ID->EX. Signali, ki so potrebni za stopnjo MA, potujejo preko vmesnih stopenj ID->EX in EX->MA, medtem ko signali, ki so potrebni za stopnjo WB, potujejo preko treh vmesnih stopenj: ID->EX, EX->MA in MA->WB.

Edina izjema so signali, ki so potrebni za stopnjo IF. Ti se takoj obdelajo, saj so vedno signali skoka ali vejitve, kar zahteva takojšnjo spremembo poteka izvajanja.

Celotni nabor mikroukazov modela MiMo v2 je podan v tabeli 5.1.

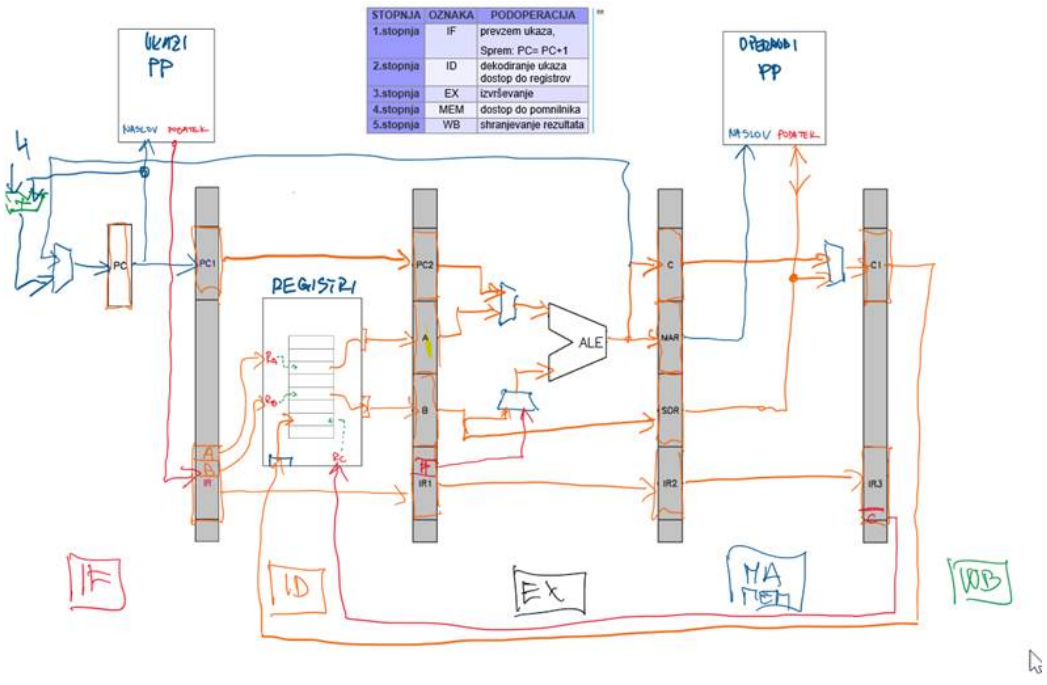
Mikroukaz	**Možne vrednosti**	**Stopnja**	**Namen**
instrUnload	0/1	IF	Zamenjaj ukaz, ki gre v IF->ID, z mehurček. Se ne uporablja v različici 2.3 modela.
pcsel	pc/immed/pcimmed	IF	Izvor nasljednje vrednosti PC-ja.
loadlink	0/1	IF	Shrani vrednost iz PC v register Link.
strlink	0/1	IF	Shrani vrednost iz registra Link v PC. Se ne uporablja v različici 2.3 modela.
op2sel	op2/const0/const1	EX	Izvor drugega operanda v ALE operaciji. Ko je vrednost `op2`, se vzame `immed`, če obstaja, ali `treg`, če ne.
negOp2	0/1	EX	Negira drugi operand v ALE operaciji.
rvrsOps	0/1	EX	Obrni operande v ALE operaciji.
aluop	add/sub/mul/div/ rem/and/or/xor/ nand/nor/not/lsl/ lsr/asr/ror/rol	EX	Izbir ALE operaciji.
datasel	sreg/treg/aluout/dreg	MA	Podatek, ki se bo pisal v operandni pomnilnik.
datawrite	0/1	MA	Označi pisanje v operandni pomnilnik.
addrsel	immed/aluout/sreg	MA	Podatek, ki se uporabi za naslavljanje operandnega pomnilnika.
regsrc	op2/operand/aluout	WB	Podatek, ki se bo pisal nazaj v register. Ko je vrednost `op2`, se vzame `immed`, če obstaja, ali `sreg`, če ne.
dwrite	0/1	WB	Označi pisanje v register Rd.
swrite	0/1	WB	Označi pisanje v register Rs.
twrite	0/1	WB	Označi pisanje v register Rt.

Tabela 5.1: Nabor mikroukazov v MiMo v2

Poglavje 6

Opis stopenj cevovoda

V tem poglavju podrobno predstavljamo zasnovo našega cevovodnega modela procesorja ter kako vsaka stopnja deluje za pravilno in učinkovito izvajanje ukazov.

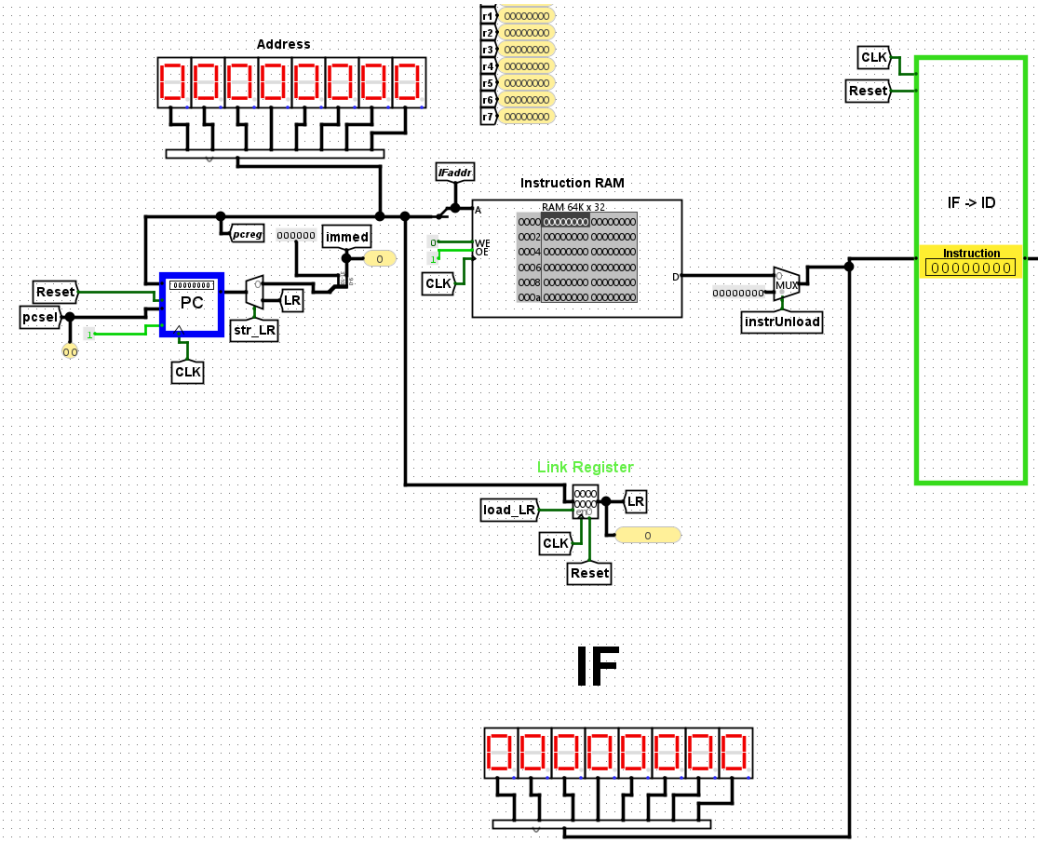


Slika 6.1: Skica modela cevovodnega procesorja iz predavanj pri predmetu Organizacija računalnikov[5]

Splošno vezje procesorja in njegovih stopenj je zasnovano tako, da posnema shemo iz predmeta Organizacija računalnikov (OR). V glavnem vezju so prikazane vse stopnje in njihovi deli, dodane pa so tudi vmesne stopnje med njimi (kot so IF->ID, ID->EX), ki shranjujejo podatke, ki vstopajo v stopnjo in izstopajo iz nje.

6.1 Pridobitev ukaza (Instruction Fetch)

Stopnja pridobivanja ukazov (Instruction Fetch) ima nalogo, da vsaka urina perioda pridobi nov ukaz.



Slika 6.2: Stopnja za pridobivanje ukazov (IF)

Vhodi:

- krmilni signali za stopnjo IF: `immed`, `Rs`, `pcsel`, `instrUnload`;
- posebna krmilna signala za register Link: `str_LR`, `load_LR`.

Izhodi:

- vsebina ukaza

Programski števec je najpomembnejše vezje v tej stopnji, ker določa naslov naslednjega ukaza. Signal `pcsel` določa, ali naj se programski števec (PC) poveča, spremeni na takojšnji naslov (`immed`) ali spremeni na PC + takojšnji. Vrednost `immed` se vzame iz ukaza, ki je trenutno v stopnji ID.

Po pridobitvi pravilnega naslova nam ukazni RAM posreduje ustrezni ukaz. Signal `instrUnload` odloča o tem, ali bo ukaz prešel v stopnjo ID. Njegov namen je preprečiti prehod ukaza, ki sledi ukazu za skok/vejitev. Dokler druga stopnja (ID) dekodira ukaz za skok/vejitev, prva stopnja (IF) pridobiva nov ukaz (PC + 1).

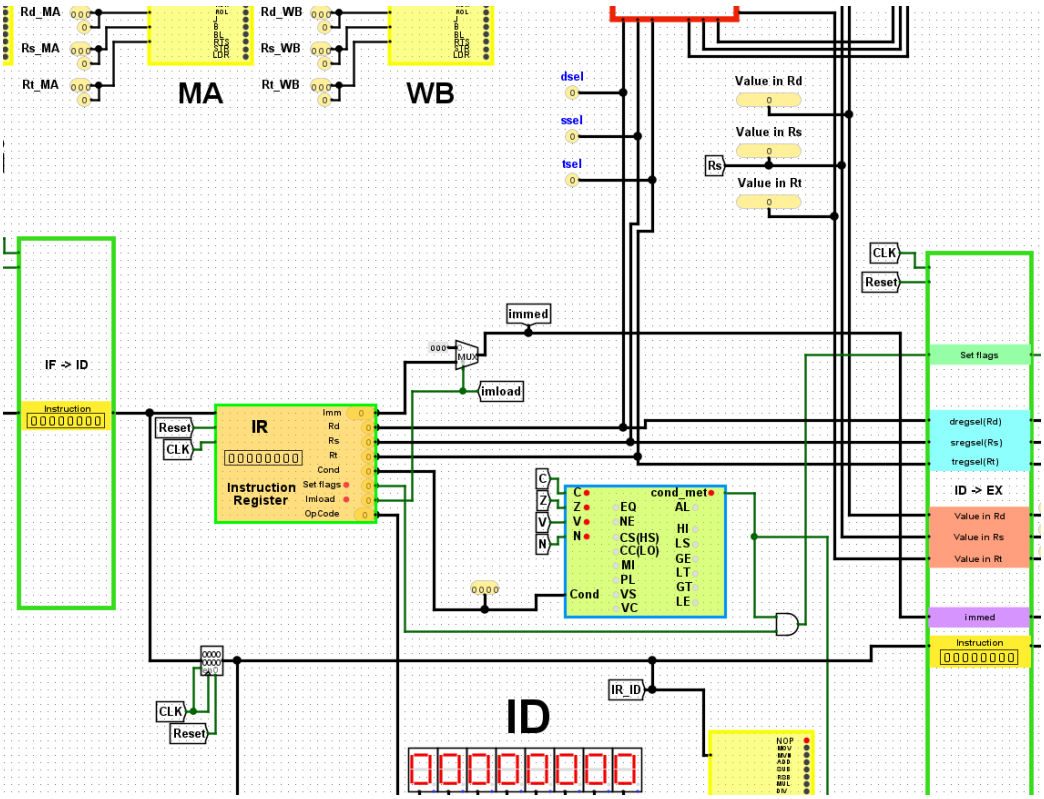
Če po dekodiranju ukaza naše vezje za preverjanje pogoja v stopnji ID odloči, da naj bi se programski števec spremenil na drugačen način kot PC + 1, potem je ukaz, ki je trenutno v stopnji IF, neuporaben in ne sme iti skozi preostali del cevovoda. V tem primeru, ko je pogoj za skok izpolnjen, signal `instrUnload` prepreči, da bi ukaz šel v stopnjo ID.

V MiMo v1 se zastavice nastavijo v stopnji EX, pogoj ukaza pa se preveri v isti stopnji. V MiMo v2 se pogoj ukaza preveri v stopnji ID. V večini primerov je ta pristop bolj učinkovit, saj eno urino periodo prej vemo, ali je pogoj izpolnjen. V primeru, da imamo v programu takoj po nastavitvi zastavic pogojni ukaz, mora uporabnik dodati ukaz `nop` med prejšnja dva ukaza, da počaka eno urino periodo, da se zastavice nastavijo. V različicah modela 2.1, 2.2 in 2.3 se ta problem samodejno reši z zaklenitev cevovoda ali premoščanjem zastavic.

Link register deluje, kot je opisano v poglavju 3.2.4.

6.2 Dekodiranje ukaza (Instruction Decode)

Namen te stopnje je dekodirati dani ukaz, se odločiti, ali ga je treba izvesti, in dostopati do potrebnih vrednosti registrov.



Slika 6.3: Stopnja za dekodiranje ukazov (ID)

Vhodi:

- vsebina ukaza;
- zastavice (c, z, v, n).

Izhodi:

- `immed`, `imload`;
- izbrani registri v ukazu: s`dregsel`, `sregsel`, `tregsel`;

- `cond_met`, `set_flags`, `opcode`.

Ukaz vstopi v ukazni register(`Instruction register`), iz katerega prejmemo naslednje krmilne signale:

- Takojšnjo vrednost (`immed`), ki se naloži le, če je aktiven tudi bit `imload`. Ta takojšnja vrednost se nato posreduje naslednji stopnji.
- `dsel`, `ssel` in `tsel`, ki označujejo, do katerih registrov je treba dostopati.
- Operacijsko kodo (`opcode`), ki bo poslana v krmilni ROM.
- Bit `set_flags`, ki označuje, ali naj ukaz vpliva na zastavice ali ne.
- Bite pogojev, ki označujejo pogoj, za katerega se mora ta ukaz izvesti.

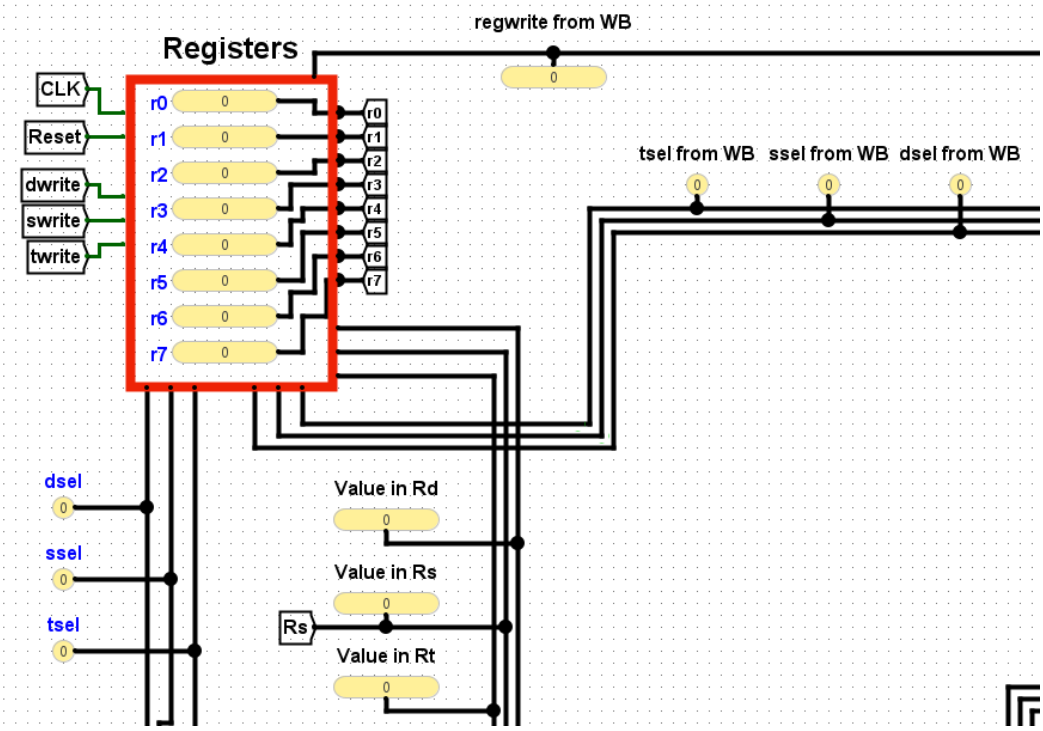
Po dekodiranju ukaza preverimo, ali je njegov pogoj izpolnjen. Biti pogojev vstopijo v vezje `check_condition` skupaj z zastavicami (n, z, c, v). Izhodni signal je signal `condition_met`, ki kaže, ali ta ukaz izpolnjuje dani pogoj.

Ta signal `cond_met` se nato skupaj z `opcode` posreduje krmilnemu ROMu. Če je pogoj izpolnjen, krmilni ROM določa stanja vseh krmilnih signalov prek naslova `opcode`. Če pogoj ni izpolnjen, se uporabi privzeti naslov 0, ki ustreza ukazu `nop`.

Signal `set_flags` iz našega ukaznega registra gre skozi vrata AND s signalom `condition_met` in se nato prenese v naslednjo stopnjo (Execute).

Registrska enota je morala biti spremenjena, da bi omogočala branje vrednosti registrov v stopnji ID in pisanje v registre v stopnji WB.

Signali `dsel`, `ssel` in `tsel` so bili spremenjeni v `dselin`, `sselin` in `tselin` za stopnjo WB ter `dselout`, `sselout` in `tselout` za stopnjo ID. Vhodi `dselin`, `sselin` in `tselin` registrske enote se preberejo v stopnji WB, saj se takrat izvede zapisovanje v registre. Vrednosti signalov `dsel`, `ssel` in `tsel` trenutnega ukaza se posredujejo vsaki nadaljnji stopnji, tako da se lahko iste vrednosti uporabijo v stopnji WB čez 3 urine periode.



Slika 6.4: Registri v modelu MiMo v2

6.3 Izvedba ukaza (Instruction Execute)

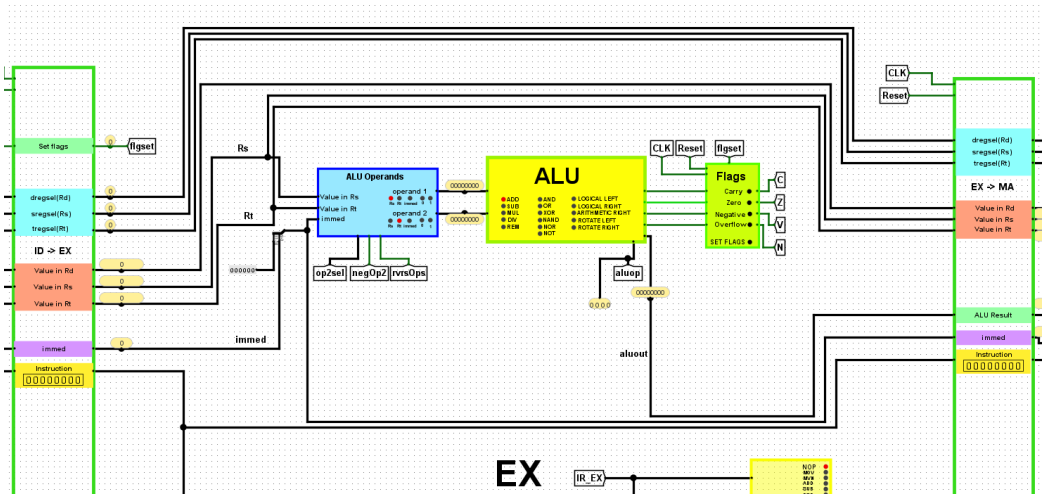
Namen stopnje Execute je po potrebi preoblikovati podatke z uporabo operacije ALE in spremeniti zastavice (C, Z, V, N), če je to potrebno.

Vhodi:

- vrednosti iz izbranih registrov v ukazu: `Rd`, `Rs`, `Rt`;
- takojšnja vrednost: `immed`;
- signal, ki označuje, ali naj bi se zastavice spremenile: `flags_set`;
- krmilni signali za stopnjo EX: `op2sel`, `aluop`, `negOp2`, `rvrsOps`;
- izbrani registri v ukazu: `dregsel`, `sregsel`, `tregsel`.

Izhodi:

- rezultat ALE: `aluot`;
- zastavice: c, z, v, n;
- vrednosti iz izbranih registrov v ukazu: `Rd`, `Rs`, `Rt`;
- takojšnja vrednost: `immed`;
- izbrani registri v ukazu: `dregsel`, `sregsel`, `tregsel`.



Slika 6.5: Stopnja za izvedbo ukazov (EX)

Execute za izvedbo operacije uporabi vrednosti registrov in takojšnje vrednosti, pridobljene v stopnji ID. Vezje `ALU Operands` določi, katere vrednosti naj bi se uporabljale.

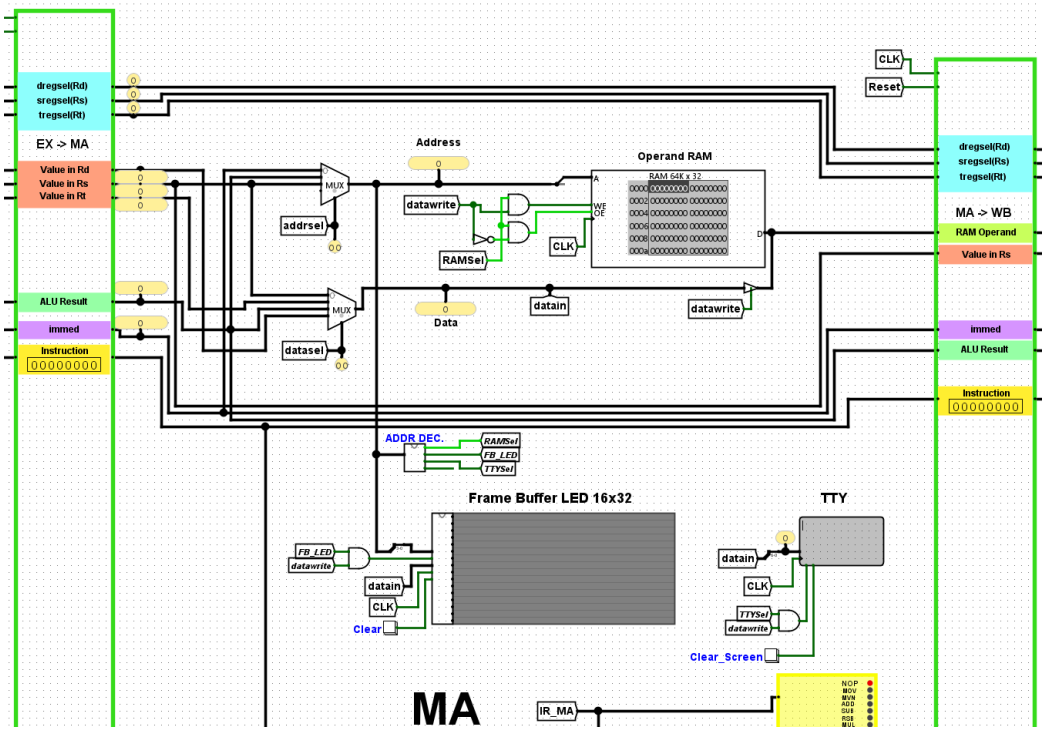
`Rs` je vedno prvi operand. Signal `op2sel` določa, katere vrednosti naj se vzamejo kot drugi operand: `Rt`, takojšnja vrednost, konstanta 0 ali konstanta 1. Signal `negOp2` določa, ali želimo, da se drugi operand negira v logičnih vratih NOT. Signal `rvrsOps` narekuje, ali naj v ALE zamenjamo prvi in drugi operand (uporabljeno za ukaz `rsb`).

Signal `aluop` določa, katero operacijo ALE želimo izvesti. Vsaka od zastavic se spremeni le, če je aktiven signal `flags_set`.

Rezultat operacije v ALE se skupaj z vrednostmi registrov in takojšnjih vrednosti za ta ukaz prenese v naslednjo stopnjo.

6.4 Dostop do pomnilnika (Memory Access)

Namen stopnje dostopa do pomnilnika je izvesti operacijo dostopa do operandnega pomnilnika. Če se operacija izvede, je potrebno določiti naslov in vrsto operacije. V primeru pisanja pa še vsebino, ki se bo zapisala v pomnilnik.



Slika 6.6: Stopnja za dostop do pomnilnika (MA)

Vhodi:

- rezultat ALE: `aluot`;
- vrednosti iz izbranih registrov v ukazu: `Rd`, `Rs`, `Rt`;
- takojšnja vrednost: `immed`;
- krmilni signali za stopnjo MA: `addrsel`, `datasel`, `datawrite`;
- izbrani registri v ukazu: `dregsel`, `sregsel`, `tregsel`.

Izhodi:

- rezultat ALE: `aluot`;
- `Rs`;
- takojšnja vrednost: `immed`;
- izbrani registri v ukazu: `dregsel`, `sregsel`, `tregsel`.

Signal `addrsel` določa, s katerega naslova v operandnem pomnilniku (Operand RAM) se bo bralo ali pisalo. Možnosti so `immed`, `aluout` in `Rs`.

Signal `datasel` odloča o tem, katere podatke je treba zapisati v operandni pomnilnik v primeru ukaza za pisanje. Možnosti so `Rd`, `Rt` in `aluout`.

Če je signal `datawrite` aktiven, potem gre za pisanje in se podatki (`datain`) zapišejo v operandni pomnilnik na izbrani naslov. V primeru branja, pa se vsebina iz izbranega naslova prenese v naslednjo stopnjo.

Podatke lahko na podlagi sheme dekodiranja naslovov osnovnega modela MiMo zapišemo tudi v katero koli od dveh drugih vhodno-izhodnih naprav, in sicer v grafični zaslon FB LED ali serijski terminal TTY I/O.

6.5 Zapisovanje nazaj (Write Back)

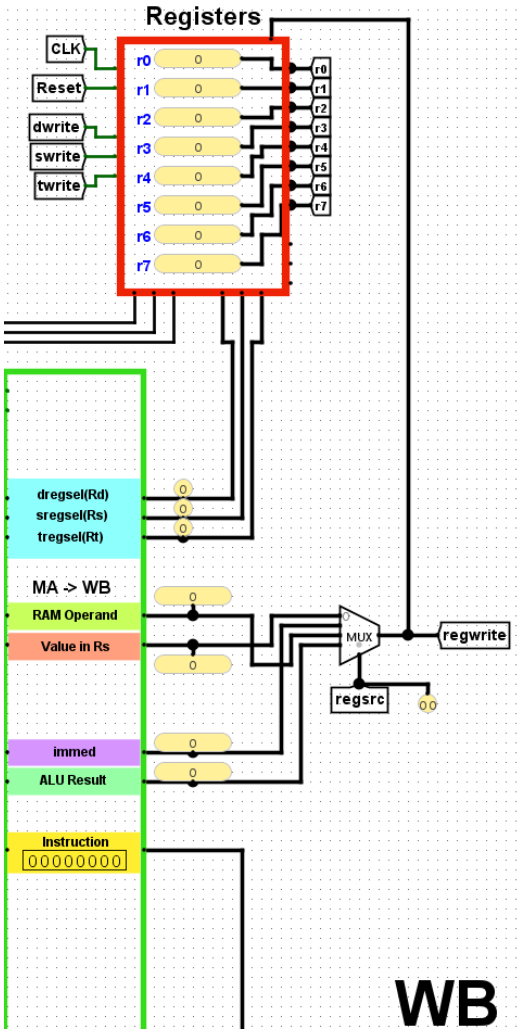
Namen stopnje za zapisovanje nazaj (Write Back) je zapisati vse preoblikovane ali pridobljene rezultate iz prejšnjih stopenj v registre (če je potrebno).

Vhodi:

- rezultat ALE: `aluot`;
- `Rs`, `RAM operand`;
- takojšnja vrednost: `immed`;
- krmilni signali za stopnjo WB: `regsrc`, `dwrite`, `swrite`, `twrite`;
- izbrani registri v ukazu: `dregsel`, `sregsel`, `tregsel`.

Izhodi:

- podatek, ki se zapisuje v register: `regwrite`.



Slika 6.7: Stopnja za zapisovanje nazaj (WB)

Signal `regsrc` določa, kateri podatki se zapišejo v register. Njegove možnosti so `Rs`, `immed`, `operand` (vzet iz operandnega pomnilnika) in `aluout`.

Rezultat je izhod `regwrite`, ki vstopi v registrsko enoto skupaj s signali `dwrite`, `swrite` in `twrite`, ki označujejo, ali naj se zapiše ali ne.

Signali `dregsel`, `sregsel` in `tregsel` iz prejšnjih treh periodov zdaj zapustijo cevovod in označujejo, v katere registre je treba pisati.

Procesor je zdaj v celoti končal izvajanje danega ukaza in je zapustil cevovod v petih stopnjah, pri čemer je v vsaki od prejšnjih stopenj sočasno obdelal štiri druge ukaze. To prikazuje učinkovitost cevovodnega modela v primerjavi z osnovnim modelom MiMo.

Poglavje 7

Preizkus in analiza delovanja cevovoda MiMo v2 in dodatne modele

7.1 Cevovodne nevarnosti

Cevovodne nevarnosti lahko preprečijo nemoten prehod ukazov skozi različne stopnje cevovoda. Nastanejo zaradi odvisnosti med ukazi, kar lahko privede do zastojev ali nepravilnega izvajanja.

Čeprav je naš cevovodni model učinkovitejši od osnovne različice, še vedno naleti na cevovodne nevarnosti, ki ob pojavu upočasnjujejo procesor. V tem poglavju se ukvarjamo s cevovodnimi nevarnostmi v okviru zasnove cevovodnega procesorja. Pogledali bomo vrste nevarnosti, njihov vpliv na učinkovitost in tehnike, uporabljene za obvladovanje teh nevarnosti.

Cevovodne nevarnosti lahko na splošno razvrstimo v tri glavne vrste:

1. Podatkovne nevarnosti

   Podatkovne nevarnosti se pojavijo, kadar obstaja odvisnost med ukazi, kar bi lahko privedlo do napak pri obravnavi vrednosti podatkov oziroma operandov. Obstajajo tri vrste podatkovnih nevarnosti:

(a) Read-After-Write (RAW) podatkovna nevarnost. V naš primer ukaz B sledi ukaz A v cevovodu. Ta vrsta nevarnosti nastane, kadar je ukaz B odvisen od rezultata prejšnjega ukaza A, ki se še vedno izvaja v cevovodu. Tako lahko pride do situacije, ko ukaz A še ni shranil rezultate v register, od koder ukaz B lahko prehitro prebere napačno (prejšnjo) vrednost.

(b) Write-After-Read (WAR) podatkovna nevarnost. Ta nevarnost nastane, ko kasnejši ukaz B zapiše v register, preden predhodni ukaz A prebere vrednost pred zapisom. V petstopenjskem cevovodu, kot je naš, se ta nevarnost nikoli ne pojavi.

(c) Write-After-Write (WAW) podatkovna nevarnost. Nevarnosti WAW se pojavijo, ko dva ukaza poskušata pisati v isti register. Tudi to za naš petstopenjski cevovod ne predstavlja težav, saj se pisanje v registre zgodi le v eni stopnji, WB.

2. Kontrolne nevarnosti

   Kontrolne nevarnosti nastanejo zaradi negotovosti pri določanju naslova naslednjega ukaza pri skočnih ukazih. Najbolj pogosto imamo t.i. pogojne skoke, za katere običajno šele v drugi izvemo, ali pogoj velja ali ne. S tem je pa povezana tudi določitev naslova naslednjega ukaza, ki se mora izvesti.

3. Strukturne nevarnosti

   Strukturne nevarnosti nastanejo zaradi omejitev strojnih virov, ko mora več stopenj v cevovodu dostopati do istega vira oziroma naprave. V primeru enotnega predpomnilnika v cevovodu pride do najpogostejše strukturne nevarnosti, ko več stopenj (običajno stopnji IF in MA) poskuša dostopati do predpomnilnika, kar lahko zelo upočasni delovanje, ker zahteve morajo počakati na sprostitev vira.

   Z uporabo Harvardske arhitekture smo v našem modelu procesorja odpravili možnost te nevarnosti na nivoju predpomnilnika. Za razliko od

pogostejše Von Neumannove (oziroma tudi Princetonske) arhitekture, ki uporablja en pomnilniški prostor za podatke in ukaze, ima Harvardska arhitektura različne in ločene pomnilniške prostore za ukaze in podatke. Ta zagotavlja, da stopnji pridobivanja ukazov in dostopa do pomnilnika nikoli ne dostopata do iste pomnilniške enote.

Za nadaljnjo optimizacijo cevovodnega modela procesorja in prikaz tehnik, s katerimi zmanjšamo vpliv teh nevarnosti, smo razvili tri dodatne različice našega modela procesorja: MiMo v2.1, MiMo v2.2 in MiMo v2.3, pri čemer je vsaka nadgradnja prejšnje. Te različice so podrobno opisane v nadaljevanju v poglavjih 7.2, 7.3 in 7.4.

7.2 MiMo v2.1 - Zaklenitev

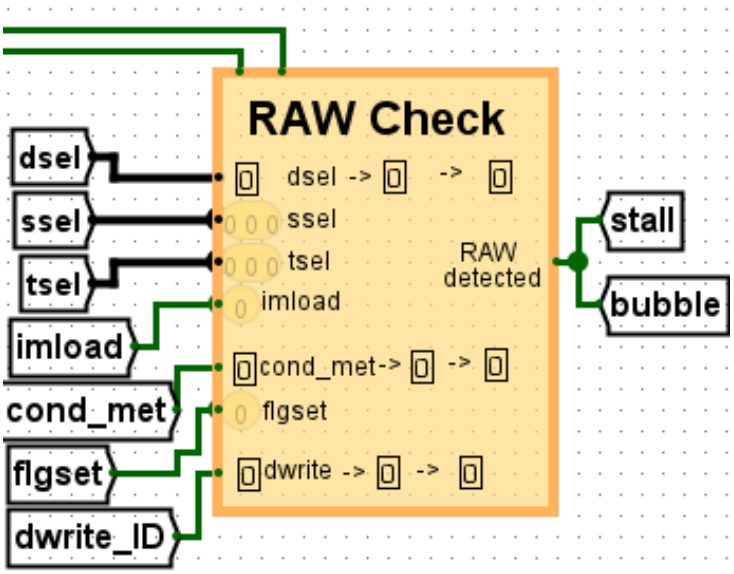
Najprej bomo raziskali nevarnost branja po zapisu (RAW) in možne rešitve zanjo. Kot smo že opisali, se ta nevarnost pojavi, ker ukaz B poskuša prebrati vrednost registra, preden je predhodni ukaz A to vrednost shranil. Kot smo prej omenili, ukaz B sledi ukaz A v tem primeru.

V prvi nadgradnji modela, MiMo v2.1, uporabimo najpreprostejšo rešitev za reševanje nevarnosti RAW: dodajanje zaklenitve cevovoda. Ko se v stopnji ID naloži ukaz B, zaznamo prihajajočo nevarnost RAW, zato za eno urino periodo ustavimo nalaganje novih ukazov v cevovod, dokler prejšnji ukaz A v registrsko enoto ne zapiše ustrezne podatkovne vrednosti.

Naš primer bomo razširili na štiri ukaze: A, B, C in D. Ukaz B sledi ukazu A, ukaz C sledi ukazu B, in ukaz D sledi ukazu C. Trenutno je ukaz A v stopnji WB, ukaz B v stopnji MA, ukaz C v stopnji EX in ukaz D v stopnji ID.

Zaznavanje nevarnosti RAW se izvaja v stopnji ID z novim vezjem `RAW Check`, prikazano na sliki 7.1. To vezje shrani tri predhodne ukaze (A, B in C v našem primeru), ki so vstopili v cevovod, saj traja tri urine periode oziroma tri stopnje cevovoda, da ukaz preide od stopnje ID, kjer se določijo vrednosti operandov, do stopnje WB, kjer se registrska enota posodobi z

novimi vrednostmi.



Slika 7.1: `RAW Check` vezje

Vezje nato preveri, ali so v katerem koli od predhodnjih treh ukazov (A, B in C) izpolnjeni vsi naslednji pogoji:

- so ukazi, ki morajo pisati v registrsko enoto (signal `dwrite` je enak 1);
- njihov pogoj je bil izpolnjen.

  Če pogoj ukaza ni izpolnjen, ne more zapisati v registrsko enoto in zato ne more priti do nevarnosti RAW;
- register, ki je bil uporabljen kot ciljni register (Rd) v katerem koli od predhodnih treh ukazov A, B ali C, se uporabi za izračun (Rs ali Rt) v trenutnem ukazu D.

V ukazih se tretji register, Rt, uporabi samo v primerih, ko v ukazu ni prisotna takojšnja vrednost. Prisotnost takojšnje vrednosti lahko razberemo iz bita `imload` v ukazu. Ko je ta bit aktiven za ukaz D, v vezju `RAW Check` ne preverjamo, ali je tretji register ukaza D (Rt) bil uporabljen kot ciljni register (Rd) v katerem koli od predhodnih treh ukazov A, B ali C.

Signali `dsel`, `ssel` in `tsel` v stopnji ID označijo, katere registre (Rd, Rs in Rt) naj se uporabijo za ukaz. Programsko preverimo, ali je signal `dsel` katerega koli od predhodnih ukazov A, B ali C enak signalu `ssel` trenutnega ukaza D ali signalu `tsel` trenutnega ukaza D, medtem ko je `imload` enak 0.

Vezje ob zaznavi vhodne nevarnosti RAW sproži signale za zaklenitev (`stall`) in mehurček (`bubble`). Signal `stall` ustavi signala `pcload` in `irload`, tako da se v stopnjo IF ali ID ne naloži nobenega novega ukaza.

Signal mehurčka (`bubble`) nadomesti vse signale znotraj vmesnega stopnja ID -> EX z ničlami (mehurčki), tako da lahko predhodni ukazi, ki so povzročali RAW, potujejo po cevovodu in se izvajajo, dokler niso več problematični.

Vezje `RAW Check` tudi preveri, če ukaz C nastavi zastavice v stopnji EX, medtem ko je ukaz D pogojni ukaz v stopnji ID. Ker se pogoj ukaza preveri v stopnji ID, v tem primeru zastavice še niso zapisane, zato moramo počakati eno urino periodo. Vezje `RAW Check` sproži signale `stall` in `bubble` tudi v tem primeru.

Na sliki 7.2 je podan primer, ki prikazuje razliko, ki jo nova različica prinaša v naš prevajalnik. Na levi strani je navedena koda, ki bi jo uporabili v naši osnovni različici MiMo v2, kjer je bilo treba ukaze `nop` ročno dodati programu, na desni strani pa je koda, ki bi jo uporabili v naši različici z zaklenitvijo cevovoda, MiMo v2.1.

```
mov r1, #3      @cycle 5          mov r1, #3      @cycle 5
mov r2, #3      @cycle 6          mov r2, #3      @cycle 6
nop             @cycle 7          cmp r1, r2      @cycle 10
nop             @cycle 8          streq r1, #3    @cycle 12
nop             @cycle 9
cmp r1, r2      @cycle 10
nop             @cycle 11
streq r1, #3    @cycle 12
```

Slika 7.2: Razlika v programski kodi za samodejno ali ročno dodajanje mehurčkov

Oglejmo si primer programa, ki prikazuje, kako deluje model procesorja

MiMo v2.1.

```
.data

.word 1

.text

loop:           @ stall
mov r3, #3      @ 5, 22
ldr r1, [r2]    @ 6, 23
add r1, r1, #1  @ 10, 27
add r7, r7, #1  @ 11, 28
str r2, r1      @ 14 (written to operand memory on cycle 13, but left pipeline on 14), 31
subs r4, r3, r1 @ 15, 32
add r5, r5, #1  @ 17, 34
add r7, r7, #1  @ 18, 35
add r6, r1, r4  @ 19, 36
jne loop        @ 20, 37
```

Slika 7.3: Primer kode v MiMo v2.1 - Zaklenitev

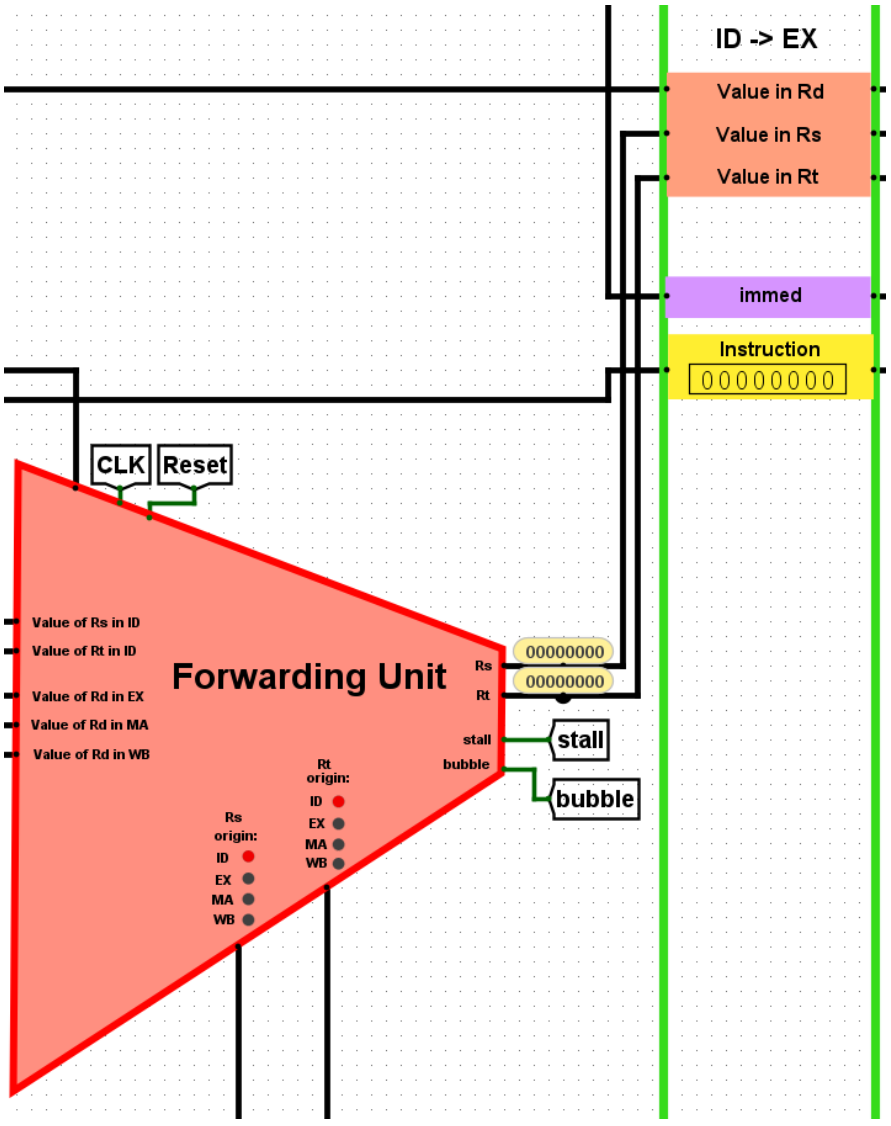
V primeru podan na sliki 7.3 je za vsakim ukazom v komentarju zapisano urino periodo, v katerem je ukaz zapustil cevovod. Uporabljamo različico v2.1 našega procesorja z zaklenitvijo. Pri tretjem ukazu, `add r1, r1, #1`, naletimo na nevarnost RAW, saj je bil register `r1`, potreben za ta ukaz, spremenjen v drugem ukazu, `ldr r1, [r2]`. Zaradi tega so izgubljeni trije urine periode, da se v register `r1` zapiše ustrezna vrednost

Enaka nevarnost RAW se ponovno pojavi z registrom `r1` v petem ukazu, `str r2, r1`. Pri tem izgubimo dva urina perioda, saj je do RAW prišlo pred dvema ukazoma. Pri devetem ukazu, `add r6, r1, r4`, izgubimo le eno urino periodo, ker se je RAW zgodil pred tremi ukazi. Pri drugem kroženju skozi zanko izgubimo dodatnih šest periodov pri istih ukazih.

Za dokončanje celotnega programa potrebujemo 37 urinih periodov.

7.3 MiMo v2.2 - Premoščanje

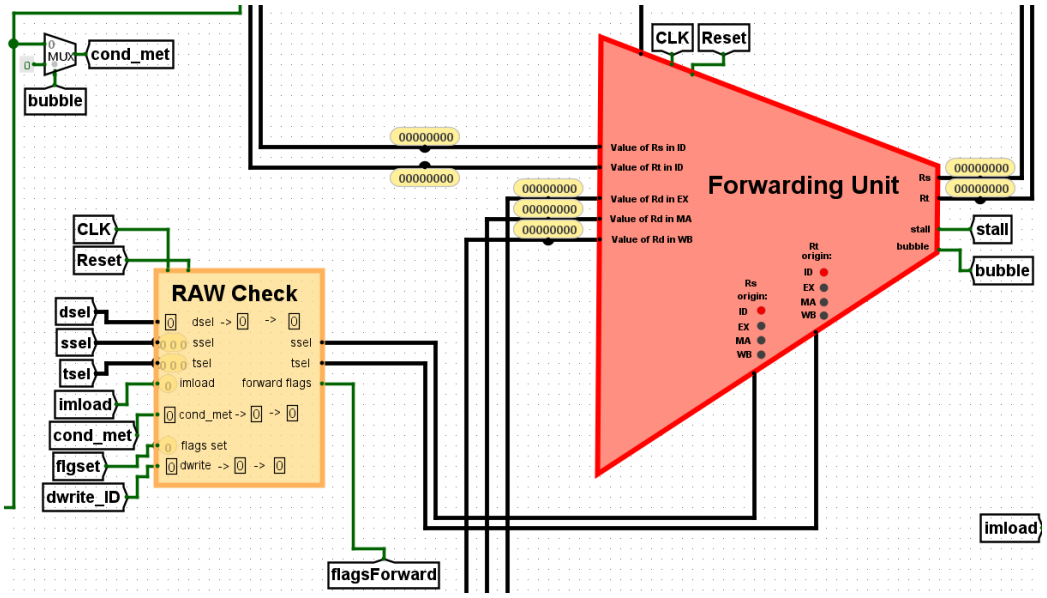
V različici MiMo v2.2 uporabimo učinkovitejši pristop za izogibanje nevarnostim RAW v cevovodnem procesorju. Ob zaznavi nevarnosti, namesto generiranja signalov za zaklenitev, posredujemo operande neposredno iz stopenj EX, MA ali WB v stopnjo ID, da jih uporabimo za prihajajoči ukaz.



Slika 7.4: Vezje `Forwarding Unit`

Uporabimo novo vezje `Forwarding Unit`, prikazano na sliki 7.4, za premoščanje

operandov. Znotraj vezja shranimo vrednosti, shranjene v drugem in tretjem registru v stopnji ID (`Value of Rs in ID, Value of Rt in ID`), ter vrednosti, shranjene v ciljnem registru v stopnjah EX, MA in WB (`Value of Rd in EX, Value of Rd in MA, Value of Rd in WB`). Signali `Rs origin` in `Rt origin` označijo, iz katere od prej navedenih vrednosti (`Value of Rs/Rt in ID, Value of Rd in EX/MA/WB`) moramo posredovati vrednosti v Rs in Rt vhode v vmesne stopnje ID -> EX.



Slika 7.5: Vezje `RAW Check` v MiMo v2.2

Vezje `RAW Check`, prikazano na sliki 7.5, je bilo v tej novi različici spremenjeno tako, da proizvaja vrednosti za signali `Rs origin` in `Rt origin` v vezju `Forwarding Unit`. Signal `Rs origin` določa, iz katere stopnje cevovoda je treba posredovati vrednost drugega registra (`Rs`). Signal `Rt origin` določa, iz katere stopnje cevovoda je treba posredovati vrednost tretjega registra (`Rt`). Vezje `RAW Check` proizvaja še en signal, `flagsForward`, ki določa, ali naj se posredujejo zastavice.

Uporabimo isti primer kot prej, imamo 4 ukaze: ukaz B sledi ukazu A, ukaz C sledi ukazu B, in ukaz D sledi ukazu C. Ukaz A je trenutno v stopnji WB, ukaz B v stopnji MA, ukaz C v stopnji EX in ukaz D v stopnji ID. Vezje

`RAW Check` v stopnji ID proizvaja svoje signale po naslednjem principu:

1. če je v ukazu C zaznana nevarnost RAW, to pomeni, da je pravilna vrednost operanda zdaj v stopnji EX, zato operand od tam posredujemo naprej;

2. če v ukazu C ni zaznana RAW, v ukazu B pa je, to pomeni, da pravilna vrednost operanda zdaj v stopnjo MA, zato ta operand posredujemo naprej;

3. če v ukazu C in v ukazu B ni zaznana RAW, v ukazu A pa je, to pomeni, da je pravilna vrednost operanda zdaj v stopnjo WB, zato ta operand posredujemo naprej;

4. če RAW sploh ni zaznan, se posredovanje operandov ne izvede in program se nadaljuje kot običajno.

V modelu MiMo v2.2, če ukaz C nastavi zastavice v stopnji EX, medtem ko je ukaz D pogojni ukaz v stopnji ID, `RAW Check` aktivira signal `flagsForward`, ki posreduje zastavice iz stopnje EX v stopnjo ID.

Vezje `Forwarding Unit` proizvaja signala `stall` in `bubble` za situacijo, ko je ukaz C tipa `ldr` in je zaznana RAW nevarnost. Ker se dostop do pomnilnika operandov zgodi v 4. stopnji (MA), pravilna vrednost operanda v stopnji EX še ni znana, zato moramo za pravilno izvajanje programa počakati eno urino periodo.

Opisali smo, kako vezje `Forwarding Unit` določi, iz katere stopnje je treba posredovati vrednost operanda prek signalov `Rs origin` in `Rt origin`. Vendar pa mora procesor na podlagi predhodnega ukaza (ukaz A, B ali C) določiti tudi pravilen vir operanda za vhode `Value of Rd in EX`, `Value of Rd in MA` in `Value of Rd in WB`. V tem kontekstu vir pomeni, ali je vrednost prišla iz enote ALE, registra, operandnega pomnilnika ali takojšnje vrednosti.

Katero vrednost (`operand`, `immed`, `sreg` ali `aluout`) je treba posredovati naprej, določimo tako, da pogledamo signal `regsrc` tega ukaza. Ta krmilni signal določa, katera vrednost se na koncu zapiše v registrsko enoto. S pomočjo

pomikalnega registra in dveh novih tunelov, `prevRegsrc` in `prevRegsrc1`, shranimo ustrezne vrednosti signalov v vhode `Value of Rd in EX`, `Value of Rd in MA` in `Value of Rd in WB`.

```
.data

.word 1

.text

loop:           @ forwarding
mov r3, #3      @ 5, 17
ldr r1, [r2]    @ 6, 18
add r1, r1, #1  @ 8, 20 (here one mandatory stall is used to get the value from the MA stage)
add r7, r7, #1  @ 9, 21
str r2, r1      @ 10, 22
subs r4, r3, r1 @ 11, 23
add r5, r5, #1  @ 12, 24
add r7, r7, #1  @ 13, 25
add r6, r1, r4  @ 14, 26
jne loop        @ 15, 27
```

Slika 7.6: Primer kode v različici MiMo v2.2 - Premoščanje

Na sliki 7.6 je enak primer kot na sliki 7.3, ki je zdaj izveden na našem procesorju z uporabo premoščanja. Povečana zmogljivost je takoj vidna. Za naš peti ukaz, `str r2, r1`, niso potrebne nobene zaklenitve, saj je spremenjena vrednost registra `r1` takoj posredovana iz stopnje MA. Prav tako ni težav za naš deveti ukaz, saj je vrednost registra `r4` posredovana iz stopnje WB.

Pred našim tretjim ukazom, `add r1, r1, #1`, je potrebna ena urna perioda zaklenitve, saj prejšnji ukaz, `ldr r1, [r2]` naloži vrednost, ki jo je treba shraniti v `r1`, v pomnilnik v stopnji MA. Ukaz `ldr r1, [r2]` v trenutni fazi še ni dosegel stopnje MA, saj sta oddaljena le eno stopnjo. Za dokončanje celotnega programa je potrebnih le 27 periodov, kar pomeni 17-odstotno povečanje učinkovitosti.

Zadnja neučinkovitost, ki je še nismo obravnavali, je ena urina perioda zastoja po ukazu `jne loop`. Do tega pride, ker se v stopnje ID dekodira skočni ukaz, v stopnji IF pa se že pridobiva naslednji ukaz. Ko je pogoj za skočni ukaz izpolnjen, je treba vsebino stopnje IF izprazniti in pridobiti naslednji ukaz, kar povzroči izgubo ene urne periode. To je kontrolna nevarnost, ki jo

bomo obravnavali v poglavju 7.4, z uporabo naše tretje nadgradnje modela, MiMo v2.3.

7.4 MiMo v2.3 - Napovedi

V različici MiMo v2.3 smo spremenili model, da bi zmanjšali vpliv kontrolnih nevarnosti. Kontrolne nevarnosti se pojavijo zaradi negotovosti v kontrolnem toku programa. Pojavijo se pri vseh ukazih, ki spreminjajo programski števec (PC) drugače, kot se običajno posodablja (PC <- PC + 1). To se zgodi pri vseh pogojnih in nepogojnih skočnih ukazih.

V naši prejšnji različici procesorja se skočni ukaz dekodira v stopnji ID in možna sta dva scenarija:

- pogoj skočnega ukaza ni izpolnjen, zato sprememba v računalniku ni potrebna. Ukaz, ki je trenutno v stopnji IF, je pravilni ukaz in v cevovodu ne pride do zastoja;

- pogoj skočnega ukaza je izpolnjen ali pa je nepogojen, zato je potrebna sprememba v PC. Ukaz, ki je trenutno v stopnji IF, je treba izprazniti in cevovod se ustavi za eno urino periodo, da se pridobi pravilni ukaz v stopnji IF.

Najboljši način za zmanjšanje zamud, ki jih povzročajo te nevarnosti, je napovedovanje, ali se bo vsak skočni ukaz izvedel na podlagi vzorcev prejšnjih skočnih ukazov v programu. To so tako imenovane dinamične napovedi, ki so podrobneje obravnavane v nadaljevanju poglavja.

Najprej moramo pogledati mehanizme in spremembe, ki so potrebni za izvedbo napovedi. Vsaka napoved se izvede v treh korakih:

1. Ugotavljanje, ali je treba narediti napoved, je prvi korak. V stopnji IF, po dostopu do pomnilnika RAM, preverimo, ali gre za ukaz skočni ukaz, kar odloči, ali moramo za ta ukaz inicializirati napovedi. V različici MiMo v2.3 smo naredili novo vezje, `Instruction Prediction Unit`,

ki izvede to preverbo in oddaja signale, potrebne za naslednja 2 koraka. Podrobnejši opis vezja je podan v podpoglavju 7.4.1.

2. Spreminjanje kontrolnega toka na podlagi naše napovedi je naslednji korak. Če smo ugotovili, da je treba izvediti napoved, izvedemo napoved nad ukazom z želeno metodo napovedovanja (metode so opisane v podpoglavju 7.4.5). Spreminjanje kontrolnega toka izvedemo s spremembo v vezju PC in novem vezju, `pcsel & immed Prediction Unit`, podrobneje opisanem v podpoglavju 7.4.2.

3. Preverjanje, ali je bila naša napoved pravilna, in popravljanje sprememb, če ni bila, je naslednji korak. Ugotovimo, ali je bil za ta ukaz izpolnjen pogoj in ali smo predhodno napovedali izpolnjen pogoj. Če sta rezultata enaka, je bila naša napoved pravilna. Podrobnejši opis mehanizma glede spremembe cevovoda v primeru napačne napovedi je podan v podpoglavju 7.4.3.

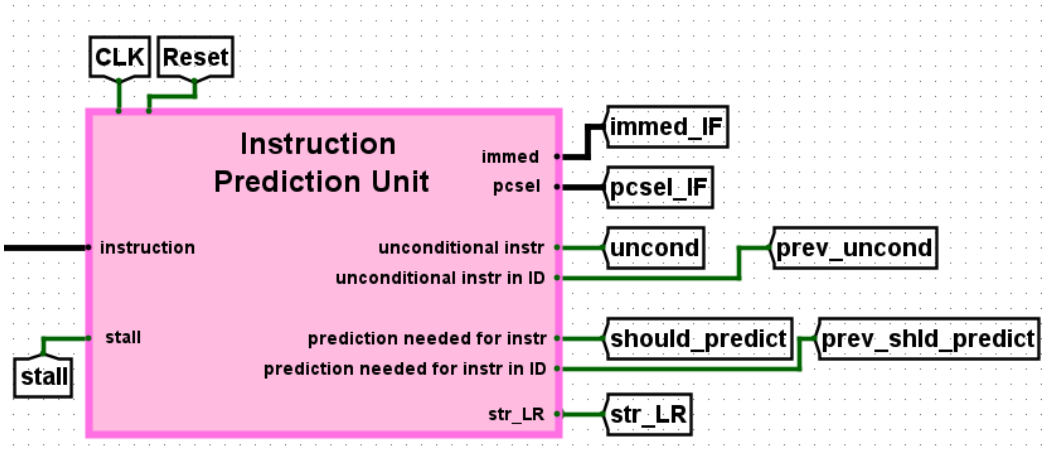
S tem pristopom je negativna stran zgrešene napovedi le ena izgubljena urina perioda, saj rezultat napovedi prepoznamo v stopnji ID in moramo vsebino stopnje IF izprazniti le, če je naša napoved napačna. Dobra stran pa je, da ko je naša napoved pravilna, ni potrebna nobena zamuda in naš program teče nemoteno. Splošno povečanje učinkovitosti je popolnoma odvisno od tega, kako natančna je naša metoda napovedovanja.

7.4.1 Vezje "Instruction Prediction Unit"

Po prevzemu ukaza iz ukaznega pomnilnika v stopnji IF, preko vsebine ukaza in vezja `Instruction Prediction Unit` (prikazano na sliki 7.7) preverjamo če je potrebno napovedati rezultat pogoja za dani ukaz.

Izhodi vezja `Instruction Prediction Unit` in njihov pomen so navedeni spodaj:

- `should_predict`: označuje, ali je potrebno napovedati rezultat pogoja za dani ukaz ozorima preverja, ali gre za skočni ukaz;



Slika 7.7: Vezje `Instruction Prediction Unit`

- `prev_shld_predict`: predstavlja vrednost, ki je bila v signalu `should_predict` eno periodo prej, oziroma označuje, ali je bilo potrebno napovedati rezultat pogoja za prejšnji ukaz, ki je trenutno v stopnji ID. To je pomembno, saj moramo v stopnji ID preveriti, ali je bila naša napoved pravilna, če smo rezultat pogoja napovedali;

- `uncond`: označuje, ali je dani ukaz nepogojen. Če je dani skočni ukaz nepogojen, se naslov naslednjega ukaza neposredno spremeni v PC;

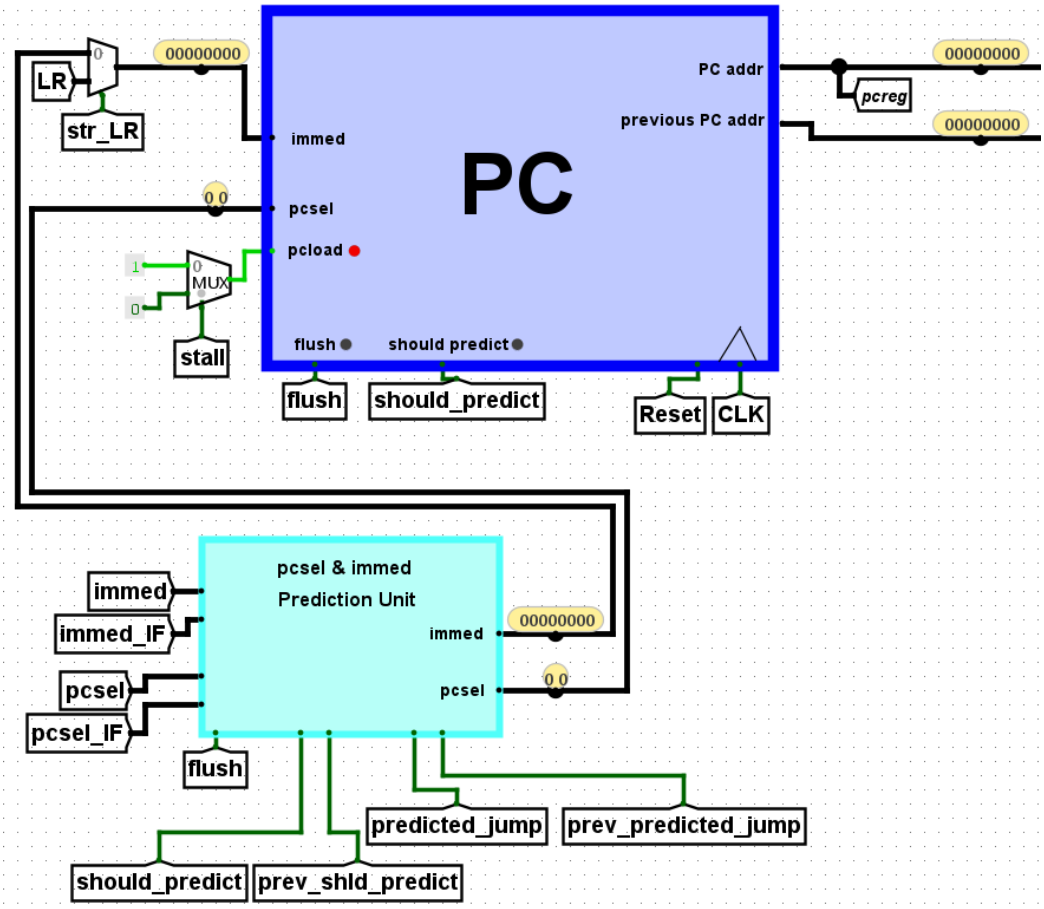
- `prev_uncond`: predstavlja vrednost, ki je bila v signalu `uncond` eno periodo prej, oziroma označuje, ali je bil prejšnji ukaz (ki je trenutno v stopnji ID) nepogojni;

- signala `immed_IF` in `pcsel_IF`: označujeta takojšnjo vrednost in vrednost krmilnega signala `pcsel` za dani ukaz. Njihov namen je razložen v podpoglavju 7.4.2;

- `str_LR`: označuje, ali je ukaz iz tipa `rts`. Njegov namen je razložen v podpoglavju 7.4.4.

7.4.2 Vezje "pcsel & immed Prediction Unit"

V prejšnji različici modela MiMo v2 se spreminjanje kontrolnega toka programa v primeru skočnega ukaza izvede v stopnji ID po dekodiranju ukaza. Uporabita se takojšnja vrednost ukaza in krmilni signal `pcsel` za določitev, kako se bo spremenil PC števec, s katerim se določi naslov naslednjega ukaza.



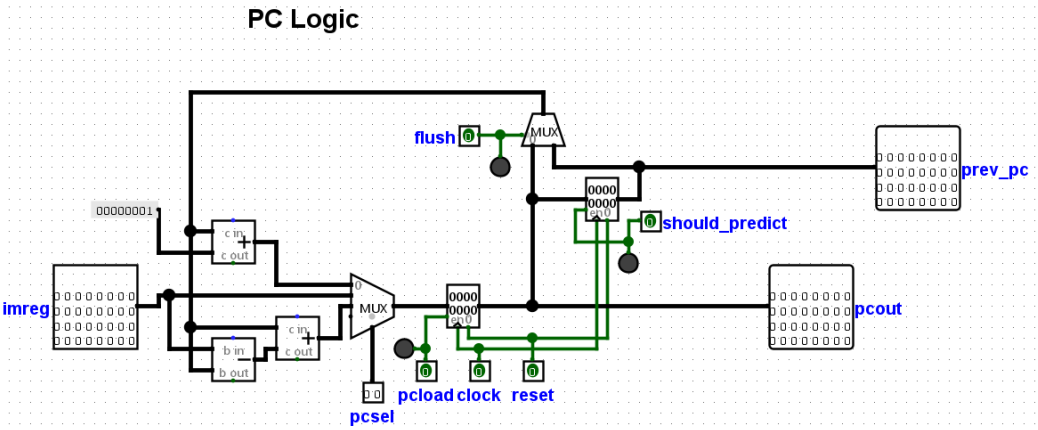
Slika 7.8: Vezje `pcsel & immed Prediction Unit` in PC števec v MiMo v2.3

V različici MiMo v2.3, ko napovemo, da bo pogoj izpolnjen, moramo kontrolni tok programa spremeniti v stopnji IF. Zato moramo v PC naložiti takojšnjo vrednost iz ukaza v stopnji IF (`immed_IF`) in krmilni signal `pcsel` iz ukaza v stopnji IF (`pcsel_IF`). To naredimo preko vezje `pcsel & immed`

`Prediction Unit`, prikazano na sliki 7.8.

Signal `predicted_jump` označuje, ali je napovedan izpolnjen pogoj za dani skočni ukaz v stopnji IF. Signal `prev_predicted_jump` predstavlja vrednost, ki je bila v signalu `predicted_jump` eno periodo prej oziroma označuje, ali je bil napovedan izpolnjen pogoj za prejšnji ukaz, ki je trenutno v stopnji ID.

Signal `flush` je pomemben v primeru napačne napovedi in je podrobneje razložen v podpoglavju 7.4.3.



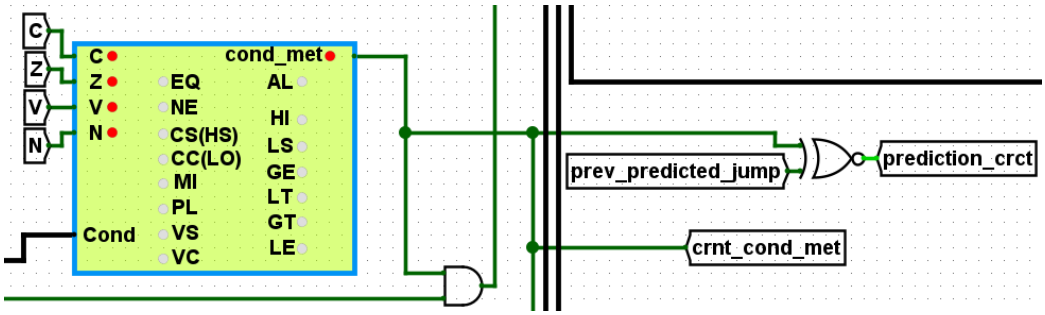
Slika 7.9: Notranjo vezje PC števec v MiMo v2.3

Izhodni signal `previous PC addr` (na sliki 7.8) predstavlja vrednost v PC števec pri zadnji napovedi v modelu. Kot je prikazano na sliki 7.9, register, ki shrani rezultat izhoda `previous PC addr` (`prev_pc` v notranjem vezju na sliki), se posodobi le, ko je signal `should_predict` aktiven. Ta signal se uporabi za posodobitev napovedne tabele z dejanskim rezultatom pogoja. To je podrobneje opisano v podpoglavju 7.4.5.

7.4.3 Spremembe cevovoda v primeru napačne napovedi

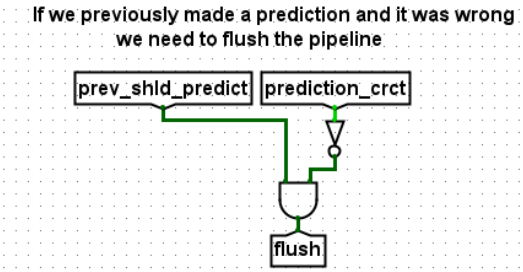
Naj uporabimo primer, kjer imamo tri ukaze: A, B in C. A je prvi v vrsti, B sledi A, in C sledi B. A je pogojni skočni ukaz, ki spremeni naslov na ukaz C v primeru izpolnjenega pogoja. Če smo za ukaz A napovedali izpolnjen

pogoj v stopnji IF, smo spremenili PC števec, da kaže na ukaz C. Zdaj, po eni urini periodi, je ukaz A v stopnji ID in ukaz C v stopnji IF.



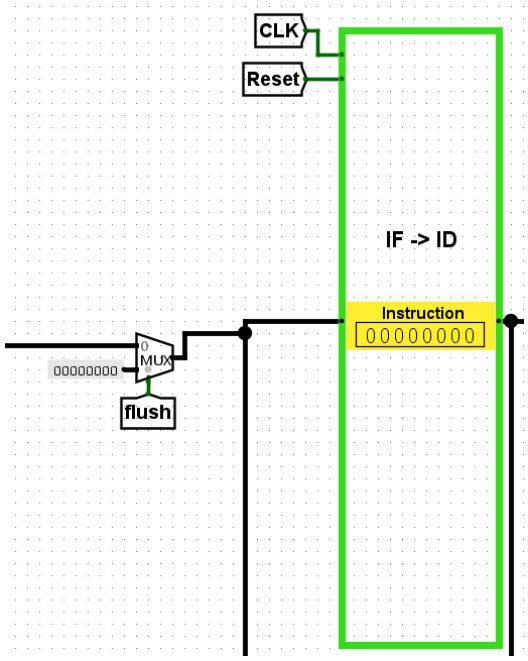
Slika 7.10: Signal `prediction_crct` v stopnji ID

V stopnji ID moramo preveriti, ali je bila naša napoved za ukaz A pravilna. Kot je prikazano na sliki 7.10, primerjamo signala `prev_predicted_jump` (ki označuje, ali smo napovedali izpolnjen pogoj za ukaz A) in `cond_met` (ki označuje dejanski rezultat pogoja za ukaz A) z logičnim vrati XNOR. Če sta enaka, je naša napoved pravilna in ukaz C lahko nadaljuje po cevovodu. Če nista enaka, moramo sprazniti ukaz C iz stopnje IF in napolniti stopnjo IF z ukazom B.



Slika 7.11: Signal `flush` v stopnji IF

Signal `flush` (prikazan na sliki 7.11) označuje, da je potrebno izprazniti stopnjo IF v cevovodu. Z istim signalom onemogočimo prehod na ukaz C v vmesni stopnji IF $->$ ID in ga zamenjamo z mehurčkom, prikazano na sliki 7.12.



Slika 7.12: Spraznitev stopnji IF v primeru napačne napovedi

Z uporabo signala `flush` spremenimo vrednost PC števca na vrednost v PC števcu pri zadnji napovedi v modelu (`prev_pc`), kot jeprikazano na sliki 7.9. V primeru napačne napovedi z uporabo vezja `pcsel & immed Prediction Unit` spremenimo vrednost signala `pcsel` v nasprotno vrednost od naše napovedi:

- če smo napovedali izpolnjen pogoj za ukaz A, spremenimo vrednost `pcsel` na `pc + 1`;

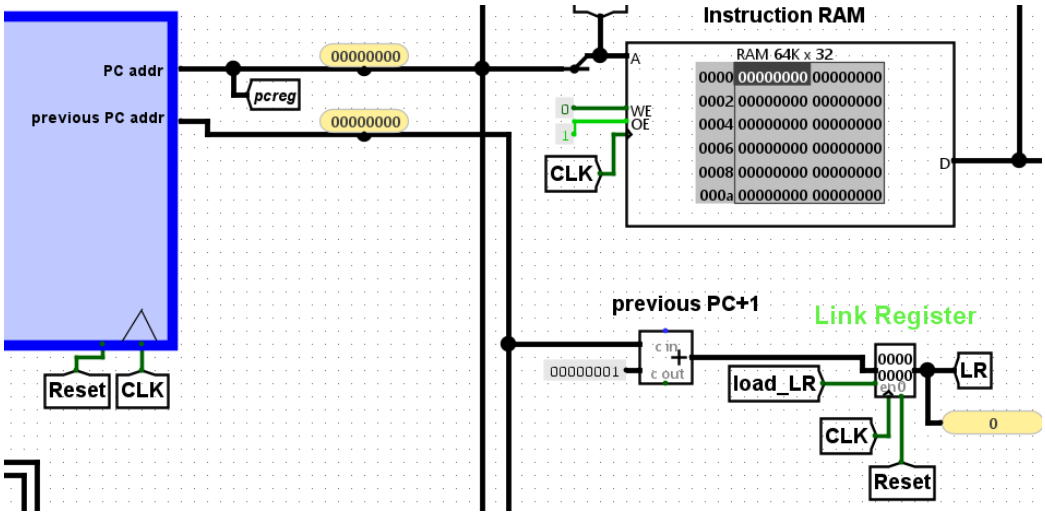
- če smo napovedali neizpolnjen pogoj za ukaz A, spremenimo vrednost `pcsel` na `immed` ali `pc + immed` (odvisno od tega, ali je ukaz A skok ali vejitev).

S tem, z izgubo ene urine periode, smo popravili kontrolni tok programa v primeru napačne napovedi.

7.4.4 Register povratnih naslovov Link v MiMo v2.3

Pri ukazu `bl` za klic podprograma se po koncu podprograma mora izvesti naslednji ukaz, ki je bil na vrsti pred klicem podprograma.

V prejšnjih različicah modela, med dekodiranjem ukaza `bl` v stopnji ID, se sproži signal `load_LR`, ki shrani naslov, ki je trenutno v PC števcu, v register povratnih naslovov Link.



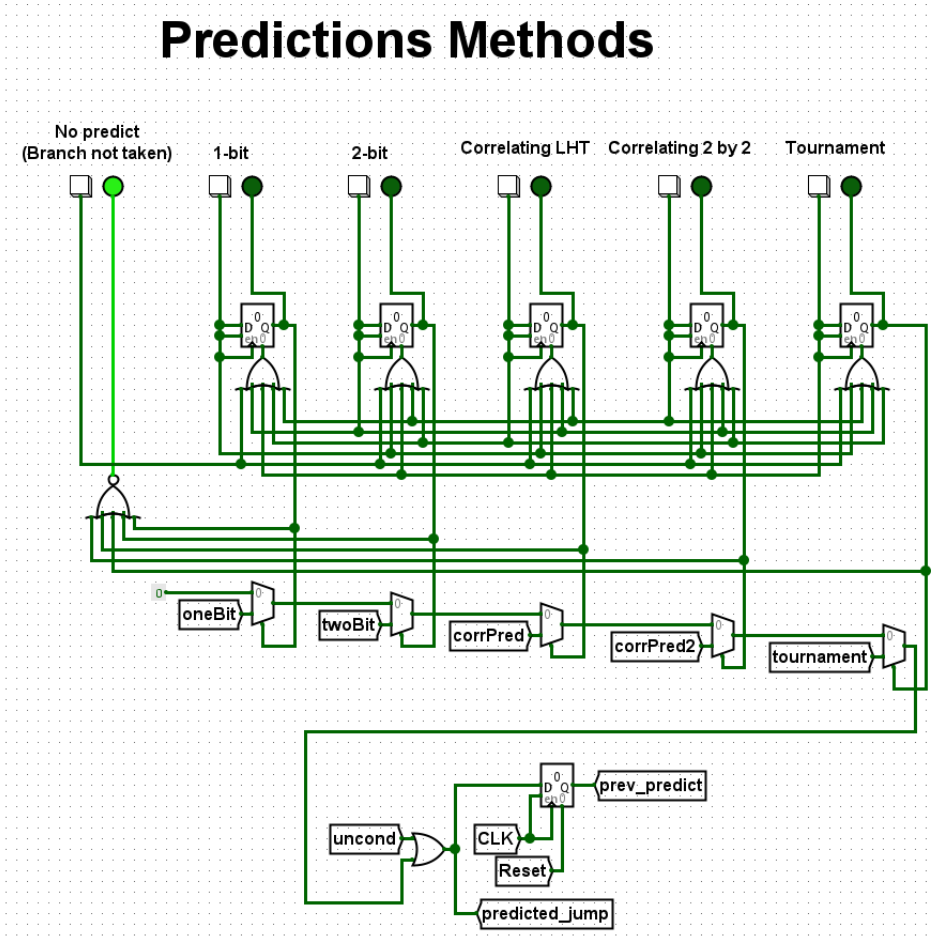
Slika 7.13: Register Link v MiMo v2.3

V različici MiMo v2.3 se pri napovedovanju izpolnjenega pogoja za ukaz `bl` v stopnji IF vrednost v PC spremeni že v stopnji IF. Po eni periodi, ko je ukaz `bl` v stopnji ID in se sproži signal `load_LR`, naslov, ki je trenutno v PC števcu, ni več naslednji ukaz, ki bi bil na vrsti pred klicem podprograma, ampak je prvi naslov v podprogramu. Zaradi tega v MiMo v2.3 v register Link shranimo vrednost v PC števcu pri zadnji napovedi v modelu (`previous PC addr`) plus ena, da pridobimo naslednji ukaz na vrsti pred klicem podprograma (prikazano na sliki 7.13).

Ukaz `rts` je nepogojni skočni ukaz, ki se uporabi za vrnitev iz podprograma. Ker je to vedno nepogojni ukaz, v različici MiMo v2.3 neposredno spremenimo naslov naslednjega ukaza v stopnji IF preko izhodnega signala `str_LR` iz vezja `Instruction Prediction Unit`.

7.4.5 Primerjava metod napovedovanja

Uporabljamo šest različnih metod napovedovanja, ki se razlikujejo po uspešnosti. Izbrano metodo lahko preklapljamo z gumbi, prikazanimi na sliki 7.14.



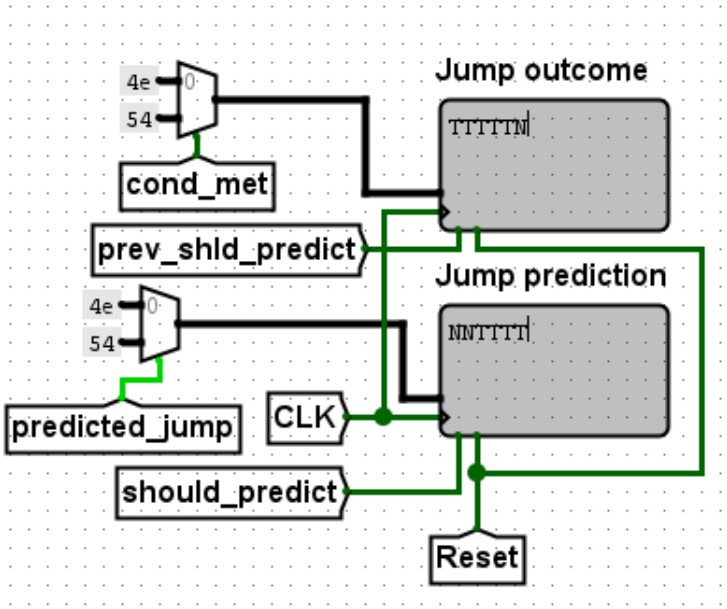
Slika 7.14: Vezje za preklapljanje med metodami napovedovanja

Uporabimo tudi dva serijska terminala TTY I/O (prikazana na sliki 7.15), da vizualno predstavimo izpolnjene (T) in neizpolnjene (N) pogoje za skočne ukaze ter prikažemo, kako se napovedi razlikujejo od dejanskega izida.

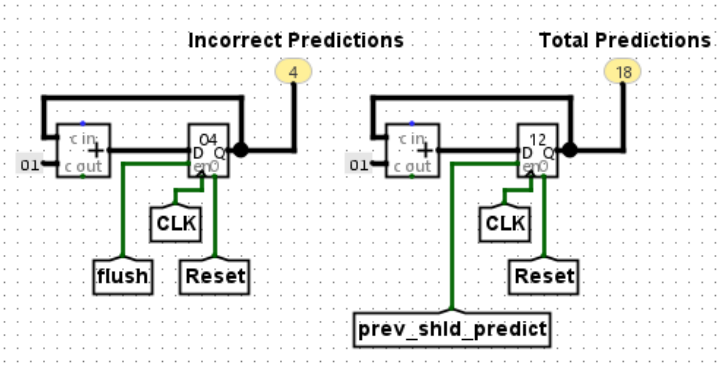
Uporabimo dva števca za beleženje števila napačnih in vseh napovedi (prikazana na sliki 7.16), da prikažemo stopnjo uspešnosti posamezne metode.

Za vsako metodo napovedovanja je za naslavljanje potreben naslov skočnega

ukaza. Prek naslova ukaza v napovedne tabele shranjujemo zgodovino izpolnjenosti pogoja, ki jo uporabimo za napoved pogoja istega ukaza.



Slika 7.15: Vizualni prikaz uspešnosti napovedi

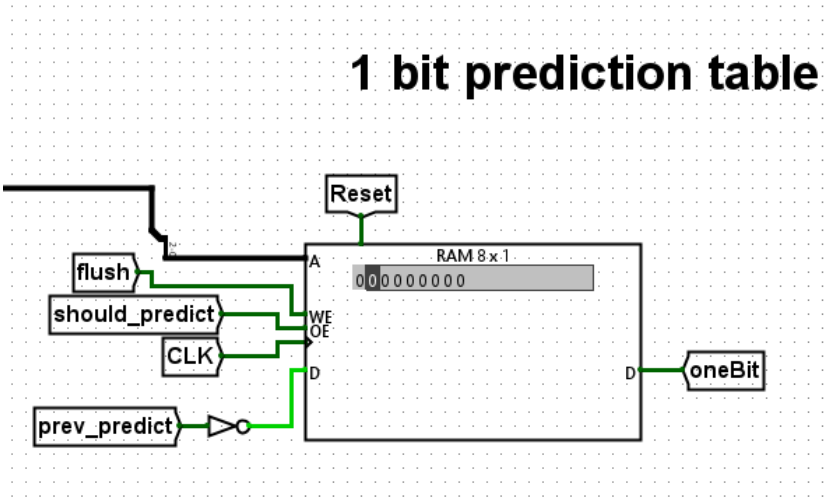


Slika 7.16: Števec napačnih in vseh napovedi

1-bitna napovedna tabela

1-bitna napovedna tabela (prikazana na sliki 7.17) je najpreprostejša metoda napovedovanja. Sestoji iz pomnilnika z 8 naslovi, ki ga naslavljamo z

zadnjimi tremi biti našega naslova. Vsaka pomnilniška lokacija lahko vsebuje vrednost 0 ali 1. Vrednost 0 pomeni napoved neizpolnjenega pogoja, vrednost 1 pa izpolnjenega pogoja. Po vsaki napovedi v naslednji stopnji posodobimo pomnilnik glede na dejansko izpolnjenost pogoja. Tako shranimo zgodovino izpolnjenosti pogoja, iz katere napovemo naslednji pogoj istega ukaza.



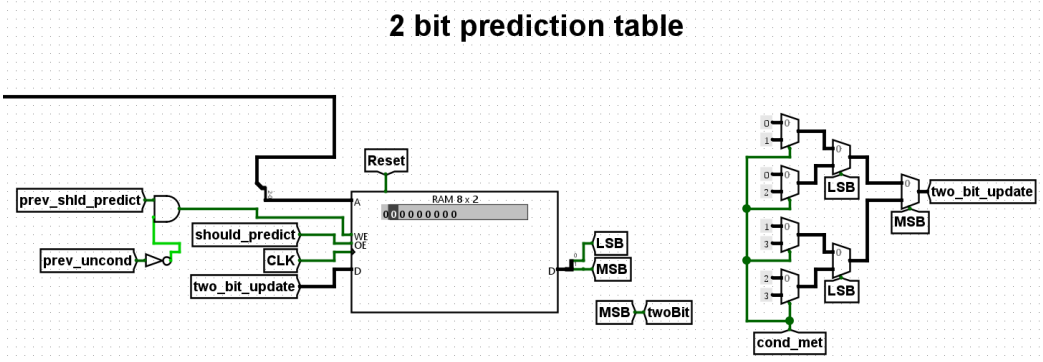
Slika 7.17: 1-bitna napovedna tabela

Čeprav je to najpreprostejša metoda, je tudi najmanj točna. Za vsako zanko sta dve obvezni zgrešeni napovedi; prva ob vstopu v zanko in druga ob izstopu iz zanke.

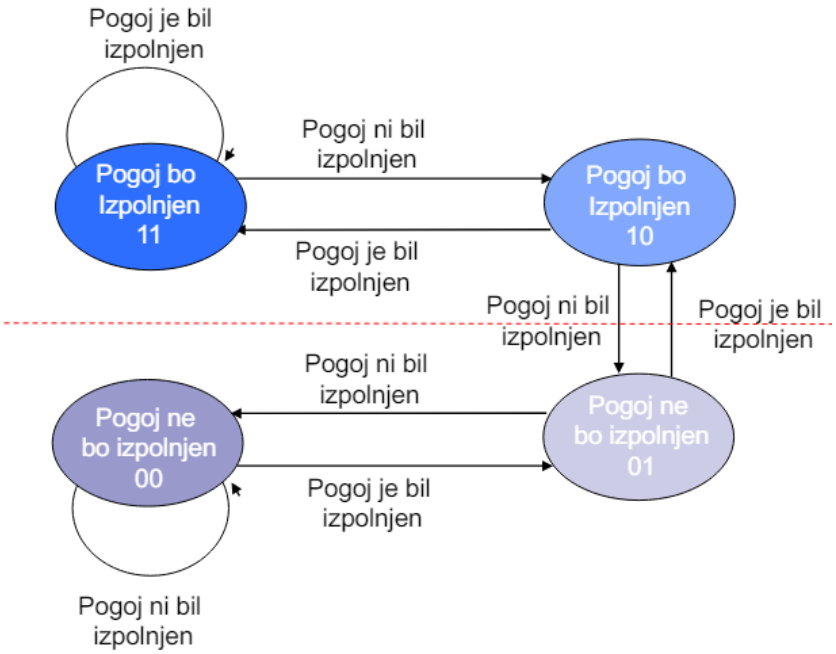
2-bitna napovedna tabela

2-bitna napovedna tabela (prikazana na sliki 7.18) deluje na podoben način kot 1-bitna, vendar je vsak pomnilniški naslov sestavljen iz dveh bitov, kar omogoča štiri možna stanja:

- 00 – Strong not taken;
- 01 – Weak not taken;
- 10 – Weak taken;



Slika 7.18: 2-bitna napovedna tabela



Slika 7.19: Prehodi med stanji v 2-bitni napovedni tabeli[5]

- 11 – Strong taken.

Najpomembnejši bit pomeni, ali napovedujemo, da bo pogoj izpolnjen, ali ne bo izpolnjen. Prehodi med stanji so prikazani na sliki 7.19. To je točnejše od 1-bitnih napovedi, saj je pri zankah samo ena obvezna zgrešena napoved, pri ponovnih vstopih v zanko pa pravilno napoveduje ponovni vstop.

Na primeru programa `program1` (prikazan na sliki 7.20) bomo preverili razlike v učinkovitosti med 1-bitnim in 2-bitnim napovednikom. Primer `program1` je preprost izmenični skok, pri katerem je v enem periodu napoved pravilna, v naslednjem pa ne.

```
/*
while (true){
        if(a % 2 == 0){jump1}   TNTNTNTNTNT
        a++
}
a => r0, r1 => used for counting jumps for a
*/

.text

loop:
rem r0, r0, #2
cmp r0, #0
jeq jump1
add r0, r0, #1
j loop

jump1:
add r0, r0, #1
add r1, r1, #1
j loop
```

Slika 7.20: Program `program1` za primerjavo med 1-bitno in 2-bitno napovedno tabelo

Z 1-bitnim napovednikom je 50/100 napovedi nepravilnih, z 2-bitnim napovednikom pa le 25/100 napovedi, kar v tem primeru pomeni 50 % večjo točnost.

Primer `program2` (prikazan na sliki 7.21) predstavlja dvojno vgnezdeno zanko. V tem primeru, z 1-bitnim napovednikom je 23/100 napovedi nepravilnih, z 2-bitnim napovednikom pa le 13/100 napovedi nepravilnih, kar pomeni približno 13 % večjo točnost.

```
.text

mov r0, #8

loop:
mov r1, #8
nop
nop
nop

repeat:
subs r1, r1, #1
nop
nop
nop
bne repeat
nop
nop
nop
subs r0, r0, #1
b  loop
```

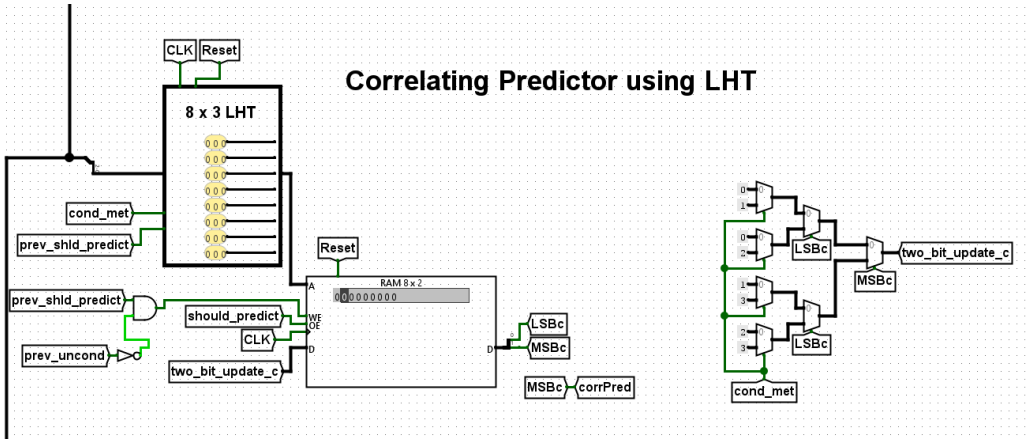
Slika 7.21: Program `program2`, ki predstavlja dvojno vgnezdeno zanko

Korelacijski prediktor z uporaba LHT

V prejšnjih metodah napovedovanja smo naše napovedne tabele naslavljali po zadnjih treh bitih naslova ukaza. Težava pri tem je premajhen pomnilniški prostor v primeru velikih programov, kjer imajo nekateri ukazi enake zadnje tri bite naslova.

V primeru, da imata ukaz A in ukaz B enake zadnje tri bite naslova, se bo zgodovina dejanske izpolnjenosti pogoja ukaza A spremenila glede na zgodovino ukaza B in obratno. Za oba ukaza izgubimo pravo zgodovino izpolnjenosti pogoja, ki je ključna za napovedovanje naslednjega pogoja ukaza.

V korelacijskem prediktorju (prikazanem na sliki 7.22) povečamo pomnilniški prostor z uporabo dveh pomnilnikov: 8 x 3 lokalno zgodovinske tabele (LHT) in 2-bitne napovedne tabele (ki jo v tem primeru imenujemo



Slika 7.22: Korelacijski prediktor z LHT

lokalna napovedna tabela - LPT).

Najprej naslavljamo tabelo LHT z naslovom skočnega ukaza, nato z vrednostjo v tabeli LHT naslavljamo 2-bitno napovedno tabelo LPT. Po izvedeni napovedi se naslov LHT posodobi, tako da se na naslovljenem mestu LHT premakne 1 ali 0 (Branch Taken ali Not Taken) glede na dejansko izpolnjenost pogoja. To nato kaže na drug naslov v tabeli LPT, do katerega se bo dostopalo pri naslednjem dostopu do tabele LHT. Torej, naslednji ukaz, ki ima iste zadnje tri bite, ne bo dostopal do istega naslova v 2-bitni napovedni tabeli kot tisti pred njim.

V tem primeru bosta ukaza A in B imela ločene zgodovine izpolnjenosti pogoja, ker bosta imela različne naslove v 2-bitni napovedni tabeli LPT.

Za boljšo predstavitev delovanja si lahko ogledamo primer programa `program3`, prikazanega na sliki 7.23.

```
while (true){
        if(a % 2 == 0){jump1}    TNTNTNTNTNT
        a++
        if(b == 1){jump2}        NNNNNNNNNNN
}
```

Slika 7.23: Primer programa `program3`

```
.text

loop:
rem r0, r0, #2
cmp r0, #0
jeq jump1       @2nd instruction

afterjump1:
add r0, r0, #1
add r5, r5, #1
add r5, r5, #1
add r5, r5, #1
add r5, r5, #1
add r5, r5, #1
cmp r1, #1
jeq jump2       @10th instruction, last 3 bits are 010 so it'll have the same LPT spot as the 2nd instruction
add r4, r4, #1
j loop

jump1:
add r3, r3, #1
j afterjump1

jump2:
mov r6, #15
```

Slika 7.24: Koda zbirnika za program `program3`

V navedeni kodi se ukaz `jump1` izvede vsako drugo zanko, medtem ko se ukaz `jump2` ne izvede nikoli. Recimo, da je naslov ukaza `jump1` 0010, naslov ukaza `jump2` pa 1010. Oba naslova imata enake zadnje tri bite, 010.

Pri uporabi 2-bitne napovedne tabele, kjer naslavljamo po zadnjih treh bitih, bi si ta dva ukaza morala deliti naslovni prostor v naši napovedni tabeli, zato bi se njune napovedi med seboj motile.

```
Branch predictions when using only 2 bit table:
jump1   NNNNNN
jump2   NNNNNN
```

Slika 7.25: Napovedi z uporabo 2-bitne tabele

Na koncu dobimo napačno napoved za ukaz `jump1` v vsaki drugi zanki.

Če bi uporabili korelacijski prediktor, bi naša dva ukaza za skok zasedla različni mesti v naši tabeli LPT. Z uporabo primera kode iz `program3`:

1. Naslov 010 v tabeli LHT bi imel sprva vrednost 000, kar kaže na naslov 000 v tabeli LPT. Naslov v tabeli LPT ima privzeto vrednost 00 (Strong Not Taken).

2. Za ukaz `jump1` je bilo predvideno, da pogoj ne bo izpolnjen, vendar je bil dejansko izpolnjen. Naša napoved je bila napačna.

3. Naslov 010 v tabeli LHT se posodobi z dejanskim izidom pogoja ukaza. Ker je bil pogoj izpolnjen, se vrednost spremeni iz 000 na 100. Naslov 010 v tabeli LHT zdaj kaže na naslov 100 v tabeli LPT, ki je privzeto 00 (Strong Not Taken). Vrednost na naslovu 000 v tabeli LPT se za prihodnost posodobi na 01 (Weak Not Taken).

4. Napoved za ukaz `jump2` je, da ne bo pogoj izpolnjen, in dejansko ni. Naša napoved je bila pravilna.

5. Naslov 010 v tabeli LHT se ponovno posodobi z dejanskim izidom pogoja ukaza. Ker ukaz ni bil izveden, se vrednost spremeni iz 100 v 010. Zdaj naslov 010 v tabeli LHT kaže na naslov 010 v tabeli LPT.

```
Branch predictions when using correlating predictor:
jump1: NNNNTNTNTNTN
jump2: NNNNNNNNNNNN
```

Slika 7.26: Napovedi z uporabo korelacijskega prediktorja z LHT

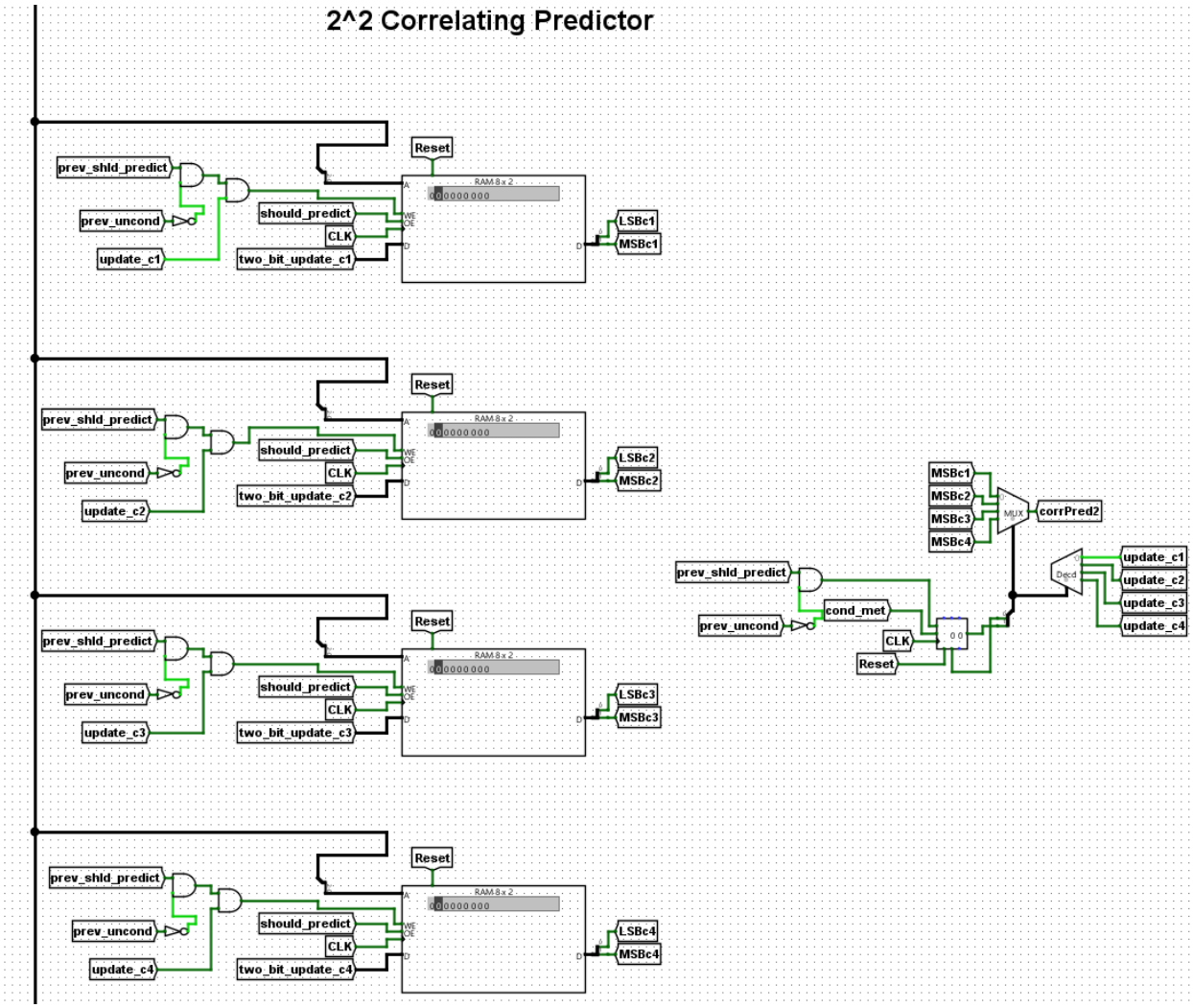
Po četrti iteraciji v našem primeru za vsakega od naših skočnih ukazov predvidimo pravilne izide.

Kot je prikazano na sliki 7.26, napovedi so v povprečju točnejše, če se uporabi korelacijski prediktor, kadar dva skočna ukaza zasedeta iste zadnje tri bite naslova.

Korelacijski prediktor (m,n)

Drug način izvedbe korelacijskega prediktorja je vključitev več n-bitnih napovednih tabel, ki shranijo lokalne zgodovine izpolnjenosti pogoja za skočne ukaze, in register globalne zgodovine, ki shrani dejanske izide zadnjih m skočnih ukazov..

To se imenuje n^m korelacijski prediktor. V našem primeru uporabljamo 2^2 korelacijski prediktor (prikazan na sliki **??**), saj smo postavili štiri dvobitne napovedne tabele in upoštevamo rezultate zadnjih dveh skokov.



Slika 7.27: 2^2 korelacijski prediktor

Ob vsaki napovedi se na podlagi rezultatov zadnjih dveh skočnih ukazov izbere ena od štirih 2-bitnih napovednih tabel. Njena vrednost se uporabi kot napoved. V naslednjem periodu se prej uporabljeni 2-bitni napovedni tabeli ustrezno posodobi na podlagi istih pravil, kot so navedena na sliki 7.19. Register globalne zgodovine se posodobi z dejanskim izidom ukaza.

Če primerjamo to vrsto korelacijskega prediktorja z zgornjo vrsto, ki uporablja tabelo LHT v istem primeru (program `program 3` prikazan na sliki 7.23), je natančnost korelacijskega prediktorja, ki uporablja LHT, veliko večja, saj ima le 2/100 napačnih napovedi, medtem ko naš prediktor 2^2 ima 15/100 napak. Vendar to ni vedno tako, saj lahko včasih nepovezani skoki, ki si ne delijo istega naslova v tabeli LHT, negativno vplivajo in poslabšajo napovedi.

Za boljše razumevanje si oglejmo razširjeno obliko primera `program3`, `program4`, prikazan na sliki 7.28.

```
/*
while (true){
        if(a % 2 == 0){jump1}   TNTNTNTNTNT
        a++
        if(b == 1){jump2}       NNNNNNNNNN

        if(c % 2 == 0){jump3}   TNTNTNTNTNT
        c++
}
a => r0, b => r1, c => r2, r5=>stalling command, r3,r4,r6 => counting loops for a,b and c
*/
```

Slika 7.28: Primer programa `program4`

V kodo dodamo še en skočni ukaz, `jump3`, ki deluje enako kot ukaz `jump1`, vendar si z ukazom `jump1` in ukazom `jump2` ne deli zadnjih treh bitov naslova.

V tem primeru je naš korelacijski prediktor 2^2 veliko učinkovitejši, saj ima le 4/100 napačnih napovedi. Po drugi strani pa ima naš korelacijski prediktor, ki uporablja LHT, v tem primeru zdaj 14/100 napačnih napovedi.

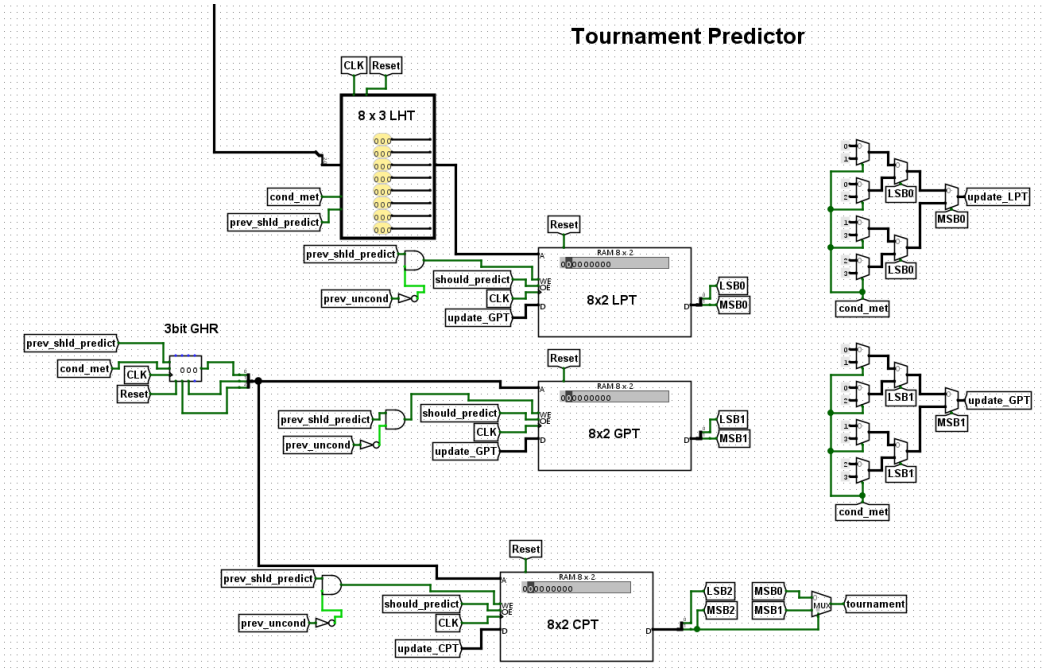
Turnirski prediktor

Kot je razvidno iz zgornjih primerov, je lahko včasih en prediktor natančnejši od drugega in obratno, odvisno od posameznega programa. Turnirski prediktor (prikazan na sliki 7.30) to odpravi tako, da dinamično izbere, ali bo napovedoval na podlagi lokalne ali globalne zgodovine.

Imamo lokalno zgodovinsko tabelo (LHT) in lokalno napovedno tabelo (LPT), ki sta enaki tistima pri korelacijskem prediktorju. Ti dva predsavljata

```
loop:
rem r0, r0, #2
cmp r0, #0
jeq jump1       @2nd instruction

afterjump1:
add r0, r0, #1
add r5, r5, #1
add r5, r5, #1
add r5, r5, #1
add r5, r5, #1
add r5, r5, #1
add r5, r5, #1
cmp r1, #1
jeq jump2       @10th instruction, last 3 bits are 010 so it'll have the same LPT spot as the 2nd instruction
add r4, r4, #1
rem r2, r2, #2
cmp r2, #0
jeq jump3       @another jump, independent to the last 2
afterjump3:
add r2, r2, #1
j loop

jump1:
add r3, r3, #1
j afterjump1

jump2:
mov r7, #15

jump3:
add r6, r6, #1
j afterjump3
```

Slika 7.29: Koda zbirnika za program `program4`



Slika 7.30: Turnirski prediktor

lokalno zgodovino posameznih ukazov.

Globalni zgodovinski register (GHR) se po vsaki napovedi posodobi z dejanskim izidom skočnega ukaza. Naslov iz GHR se nato uporabi za naslovitev globalne napovedne tabele (GPT), ki deluje kot standardna 2-bitna napovedna tabela. Ta tabela predstavlja globalno zgodovino v Turnirskem prediktorju.

Register GHR se prav tako uporablja za naslovitev izbirne napovedne tabele (CPT), ki se posodobi na enak način kot 2-bitna napovedna tabela. Rezultat iz tabeli CPT določa, ali bomo uporabili napoved iz tabeli LPT ali tabeli GPT. Tako tabela CPT določi, katera zgodovina (lokalna ali globalna) je bolj primerna za napoved pogoja posameznega ukaza.

Poglejmo, kako bi naš prediktor deloval v primeru `program4` (prikazan na sliki 7.28).

1. Na začetku imajo vse naše napovedne tabele in registri (LHT, LPT, GPT, CPT, GHR) privzete vrednosti 0. Tako bo prvi pogoj skočnega

ukaza privzeto napovedan kot neizpolnjen, saj bo CPT kazal na vrednost v LPT, ki prav tako znaša 0.

2. Ko se izvede ukaz `jump1`, je pogoj izpolnjen, vendar je naša prvotna napoved napačna.

3. Tabeli LHT in LPT se posodobita na enak način kot pri korelacijskem prediktorju (opisanem v poglavju 7.4.5). Naslov 010 v LHT se spremeni na vrednost 100, medtem ko se naslov 000 v LPT spremeni na 01 (Weak Not Taken).

4. Naslov 000 v globalni napovedni tabeli (GPT) se prav tako posodobi iz 00 na 01. Ta deluje kot standardna 2-bitna napovedna tabela.

5. Tudi naslov 000 v CPT se posodobi z 00 na 01. To bo v naslednji iteraciji spet kazalo na vrednost v tabeli LPT.

6. Register globalne zgodovine (GHR) se posodobi z dejanskim rezultatom izida, kar pomeni, da se v tem primeru spremeni vrednost iz 000 v 100. Ta nova vrednost zdaj kaže na drug naslov v naših tabelah GPT in CPT.

7. Zdaj se izvede ukaz `jump2`. Vrednost v našem registru GHR je 100, kar pomeni, da kaže na naslov 100 v naših CPT in GPT. Naslov 100 v tabeli CPT vsebuje vrednost 00, zato ponovno kaže na vrednost v našem LPT.

8. Vrednost na naslovu 010 v tabeli LHT je spremenjena na 100 in zdaj kaže na naslov 100 v tabeli LPT, ki vsebuje vrednost 00.

9. Pogoj za ukaz `jump2` je bil predviden kot neizpolnjen in dejansko ni bil. Naša napoved je bila pravilna.

10. Naslov 010 v tabeli LHT se spremeni na vrednost 010, medtem ko vrednosti na naslovih 100 v LPT, GPT in CPT ostanejo 00. Register globalne zgodovine (GHR) se posodobi na vrednost 010.

Stopnja neuspešnosti turnirskega prediktorja pri programu `program 4` je 5/100 napačnih napovedi.

V prvem primeru, `program3` (na sliki 7.23), kjer ni bil dodan ukaz `jump3`, je stopnja neuspešnosti turnirskega prediktorja znašala 3/100 napačnih napovedi. To prikazuje učinkovitost turnirskega prediktorja ter dokazuje, da je najbolj kompleksna in tudi najbolj točna metoda napovedovanja.

Na sliki 7.31 je prikazan `program5`, realni primer programa, ki ureja vrsto številk z uporabo algoritma - mehurčnega urejanja. Vsi programi in njihova uspešnost pri posameznih metodah napovedovanja so podani v tabeli 7.1.

	`program1` (7.20)	`program2` (7.21	`program3` (7.23)	`program4` (7.28)	`program5` (X)
1-bitna napovedna tabela	50%	23%	29%	40%	32%
2-bitna napovedna tabela	25%	13%	15%	20%	20%
Korelacijski (LHT)	3%	21%	2%	14%	22%
Korelacijski (m,n)	3%	18%	15%	4%	14%
Turnirski	3%	19%	3%	5%	21%

Tabela 7.1: Primerjava uspešnosti med metodami napovedovanja (odstotek število napačnih napovedi / število napovedi)

7.4.6 Navodila za uporabo modela MiMo v2

V tem podpoglavju bomo na kratko opisali postopek ustvarjanja programa, njegovega prevajanja v strojno kodo, prenosa v pomnilnik in pregleda izvedbe programa v modelu. Opis je enak za vse štiri modele (MiMo v2, MiMo v2.1, MiMo v2.2, MiMo v2.3).

Najprej mora uporabnik dodati mikroukaze v krmilni pomnilnik. V datoteki `microcode.txt` so zapisani vsi mikroukazi za model MiMo v2. Uporabnik mora te ukaze prevesti v strojno kodo z aplikacijo `microassembler.exe`.

V mapi, kjer se nahajata `microcode.txt` in `microassembler.exe`, mora uporabnik odpreti terminal operacijskega sistema in izvesti naslednji ukaz: `./microassembler.exe microcode.txt`

V isti mapi se bo ustvarila datoteka `microcode.rom`, ki jo mora uporabnik naložiti v krmilni pomnilnik modela.

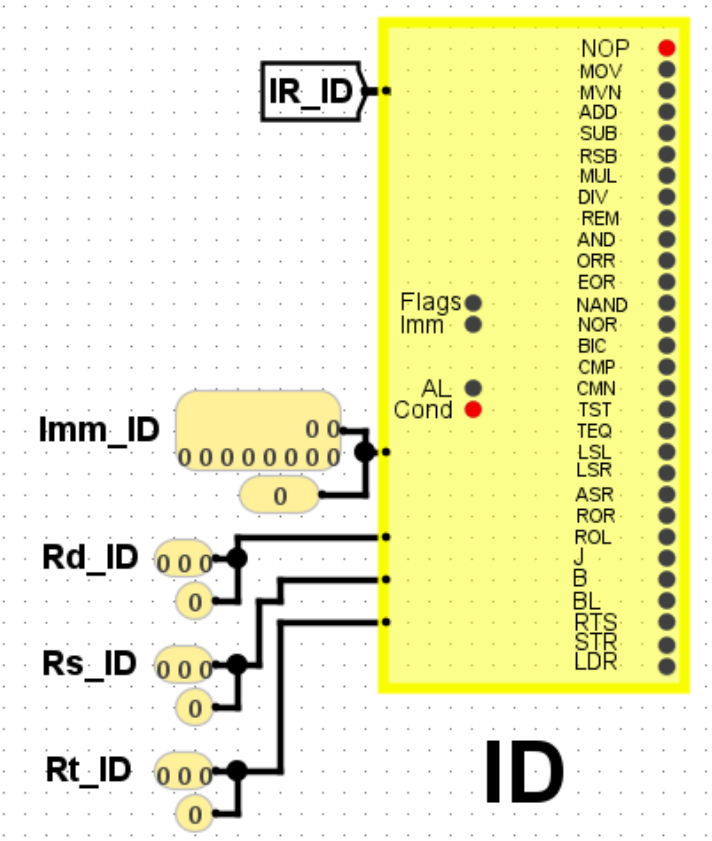
Nato mora uporabnik svojo tekstovno datoteko za programo `primerProgram.txt` prevesti v strojno kodo z aplikacijo `assembler.exe`. Enako kot prej mora uporabnik v terminalu operacijskega sistema izvesti naslednji ukaz: `./assembler.exe primerProgram.txt`

V mapi se bo ustvarila datoteka `primerProgram.iram`, ki vsebuje strojno kodo za ukaze. To datoteko je treba naložiti v ukazni pomnilnik v fazi ID modela. Če obstaja sekcija `.data`, se bo ustvarila tudi datoteka `primerProgram.oram`, ki vsebuje strojno kodo za operande. To datoteko je treba naložiti v operandni pomnilnik v fazi MA modela.

Nato lahko uporabnik izvede program v modelu. Preko vezja, prikazanega na sliki 7.32, ima uporabnik pregled ukaza znotraj stopnje ID skupaj z njegovimi podrobnostmi. Enako vezje obstaja tudi za stopnje Ex, MA in WB.

```
@ Bubble sort algorithm


.data
.word 10 @array size
.word 42, 17, 88, 23, 54, 75, 12, 67, 31, 99 @array elements

.text

ldr r0, #0              @ r0 -> size and outer loop counter
sub r0, r0, #1          @ size = size - 1

sort_outer:
cmp r0, #0              @ Check if outer loop is done
beq sort_done           @ Exit if size == 0
mov r1, r0              @ r1 is inner loop counter
mov r2, #1              @ r2 points to array

sort_inner:
cmp r1, #0              @ Check if inner loop is done
beq sort_outer_dec      @ Exit inner loop if r1 == 0
ldr r3, [r2]            @ Load array[j]
ldr r4, [r2, #1]        @ Load array[j+1]
cmp r3, r4              @ Compare array[j] and array[j+1]
ble sort_inner_next     @ Skip swap if array[j] <= array[j+1]
str r4, [r2]            @ Swap array[j] and array[j+1]
str r3, [r2, #1]

sort_inner_next:
add r2, r2, #1          @ Move on to next pair in array
sub r1, r1, #1          @ Decrement inner loop counter
b sort_inner


sort_outer_dec:
sub r0, r0, #1          @ Decrement outer loop counter
b sort_outer

sort_done:
mov r7, #1
```

Slika 7.31: Primer programa `program5` - Mehurčno urejanje



Slika 7.32: Vezje za pregled ukaz v stopnji

Poglavje 8

Zaključek

Model MiMo služi kot učinkovito izobraževalno orodje, ki premošča vrzel med teoretičnimi koncepti procesorjev in praktično izvedbo. S soočanjem študentov z realnimi izzivi pri načrtovanju procesorjev jih pripravlja na prihodnje kariere na področju računalniške arhitekture. Projekt je pokazal uspešno ustvarjanje poenostavljenega petstopenjskega cevovodnega procesorja, ki učinkovito zmanjšuje pogoste podatkovne in kontrolne nevarnosti, ki so prisotne v cevovodnih arhitekturah.

Nadaljnje izboljšave cevovodnega modela MiMo so vključevale implementacijo zaklenitev cevovoda, posredovanje operandov in tehnike napovedovanja vejitev za zmanjšanje teh nevarnosti. Te optimizacije so znatno izboljšale zmogljivost in učinkovitost procesorja ter ga bolj približale modelom procesorjev ARM.

Na splošno ta diplomska naloga ni le dosegla svojega primarnega cilja nadgradnje modela MiMo procesorja, temveč je ponudila tudi dragocene vpoglede v napredne tehnike načrtovanja procesorjev. Rezultirajoči model dokazuje potencial izobraževalnih orodij pri spodbujanju globljega razumevanja računalniške arhitekture, kar na koncu prispeva k napredku tega področja.

V prihodnosti bi lahko procesor še dodatno izboljšali, da bi se še bolj približal standardnemu procesorju ARM. Format ukazov in vezje za dekodiranje ukazov bi lahko spremenili tako, da v celoti posnemata format ukazov

ARM. To bi nam omogočilo neposredno nalaganje ARM strojne kode v naš cevovodni procesor. Prav tako bi lahko izboljšali simulacijsko okolje za boljši vizualni prikaz ukazov, ko potujejo skozi cevovod.

Literatura

[1] John L. Hennessy David A. Patterson. *Computer Organization and Design - ARM Edition*. Morgan Kaufmann, 2016.

[2] Steve Furber. *ARM System-on-Chip Architecture, 2nd Edition*. Addison Wesley Longman Limited, 2000.

[3] Andrew N. Sloss, Dominic Symes in Chris Wright. *ARM System Developer's Guide: Designing And Optimizing System Software*. Elsevier Inc., 2004.

[4] Warren Toomey. *Warren's Microcoded CPU*. 2012. URL: `https://minnie.tuhs.org/Programs/UcodeCPU/index.html`.

[5] *Univerza v Ljubljani, Fakulteta za računalništvo in informatiko*. URL: `https://fri.uni-lj.si/sl`.

[6] Priti Shankar Y.N Srikant. *The Compiler Design Handbook: Optimizations and Machine Code Generation, 2nd Edition*. CRC Press, 2007.