



UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA
KATEDRA ZA
ELEKTRONIKU



Implementacija vektorskog procesora baziranog na RISC-V setu instrukcija

Kandidat
Nikola Kovačević

Mentor
Vuk Vranjković

Septembar 2020



UNIVERZITET U NOVOM SADU • FAKULTET TEHNIČKIH NAUKA 21000
NOVI SAD, Trg Dositeja Obradovića 6

KLJUČNA DOKUMENTACIJSKA INFORMACIJA

Redni broj, RBR:		
Identifikacioni broj, IBR:		
Tip dokumentacije, TD:	Monografska dokumentacija	
Tip zapisa, TZ:	Tekstualni štampani materijal	
Vrsta rada, VR:	Master rad	
Autor, AU:	Nikola Kovačević	
Mentor, MN:	Dr Vuk Vranjković	
Naslov rada, NR:	Implementacija vektorskog procesora baziranog na RISC-V setu instrukcija	
Jezik publikacije, JP:	Srpski	
Jezik izvoda, JI:	Srpski	
Zemlja publikovanja, ZP:	Republika Srbija	
Uže geografsko područje, UGP:	Vojvodina	
Godina, GO:	2020.	
Izdavač, IZ:	Autorski reprint	
Mesto i adresa, MA:	21000 Novi Sad, Trg Dositeja Obradovića 6	
Fizički opis rada, FO: (poglavlja/strana/citata/tabela/slika/grafika/priloga)	7/64/12/9/26/0/0	
Naučna oblast, NO:	Elektrotehnika i računarstvo	
Naučna disciplina, ND:	Embedded sistemi i algoritmi	
Predmetna odrednica/Ključne reči, PO:	RISC-V arhitektura, vektorski procesor, uvm metodologija	
UDK		
Čuva se, ČU:	Biblioteka Fakulteta Tehničkih Nauka 21000 Novi Sad, Trg Dositeja Obradovića 6	
Važna napomena, VN:	Nema	
Izvod, IZ:	U ovom radu prezentovan je 32-bitni vektorski procesor zasnovan na verziji 0.8 RISC-V nacrtu vektorske ekstenzije, implementiran na Zybo razvojnoj ploči.	
Datum prihvatanja teme, DP:	23.09.2020.	
Datum odbrane, DO:	25.09.2020.	
Članovi komisije, KO: Predsednik:	Dr Rastislav Struharik	
Član:	Dr Nebojša Pjevalica	Potpis mentora
Član, Mentor	Dr Vuk Vranjković	



UNIVERSITY OF NOVI SAD • FACULTY OF TECHNICAL SCIENCES 21000
NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, ANO:		
Identification number, INO:		
Document type, DT:	Monographic publication	
Type of record, TR:	Textual printed material	
Contents code, CC:	Master thesis	
Autor, AU:	Nikola Kovačević	
Mentor, MN:	PhD Vuk Vranjković	
Title, TI:	Implementation of a vector processor based on a RISC-V instruction set	
Language of text, LT:	Serbian	
Language of abstract, LA	Serbian	
Country of publication, CP:	Republic of Serbia	
Locality of publication, LP:	Vojvodina	
Publication year, PY	2020.	
Publisher, PB:	Author's reprint	
Publication place, PP:	21000 Novi Sad, Trg Dositeja Obradovića 6	
Physical description, PD: (chapters/pages/ref./tables/pictures/graphs/appendixes))	7/64/12/9/26/0/0	
Scientific field, SF:	Electrical Engineering	
Scientific discipline, SD:	Embedded systems and algorithms	
Subject/Key words, S/KW:	RISC-V architecture, vector processor, UVM methodology	
UC		
Holding data, HD:	The Library of Faculty of Technical Sciences, Novi Sad, Serbia	
Note, VN:		
Abstract, AB:	This paper presents 32 bit vector processor based on the version 0.8 draft of RISC-V vector extension, implemented on a Zybo development board.	
Accepted by the Scientific Board on, ASB:	23.09.2020.	
Defended on, DE:	25.09.2020.	
Defended Board, KO: President:	PhD Rastislav Struharik	
Member:	PhD Nebojša Pjevalica	Mentors sign
Member, Mentor	PhD Vuk Vranjković	

Izjava o akademskoj čestitosti

Student/kinja:

Broj indeksa:

Student/kinja: osnovnih ili master akademskih studija


Autor/ka rada pod nazivom:

Potpisivanjem izjavljujem:

- da je rad isključivo rezultat mog sopstvenog istraživačkog rada;
- da sam rad i mišljenja drugih autora koje sam koristio/la u ovom radu naznačio/la ili citirao/la i navedeni u spisku literature/referenci koji su sastavni deo ovog rada;
- Da sam dobio/la sve dozvole za korišćenje autorskog dela koji se u potpunosti/ celosti unose u predati rad i da sam to jasno naveo/la;
- Da sam svestan/na da je plagijat korišćenje tuđih radova u bilo kom obliku (kao citata, parafraza, slika, tabela, dijagrama, dizajna, planova, fotografija, filma, muzike, formula, veb sajtova, kompjuterskih programa i sl.) bez navođenja autora ili predstavljanje tuđih autorskih dela kao mojih, kažnjivo po zakonu (Zakon o autorskom i srodnim pravima, Službeni glasnik Republike Srbije, br. 104/2009, 99/2011, 119/2012), kao i drugih zakona i odgovarajućih akata Univerziteta u Novom Sadu;
- Da sam svestan/na da plagijat uključuje i predstavljanje, upotrebu i distribuiranje rada predavača ili drugih studenata kao sopstvenih;
- Da sam svestan/na posledica koje kod dokazanog plagijata mogu prouzrokovati na predati rad i moj status;
- da je elektronska verzija rada identična štampanom primerku i pristajem na njegovo objavljivanje pod uslovima propisanim aktima Univerziteta.

Novi sad, _____
studenta/kinje

Potpis

	UNIVERZITET U NOVOM SADU • FAKULTET TEHNIČKIH NAUKA 21000 NOVI SAD, Trg Dositeja Obradovića 6	Datum:
	ZADATAK ZA IZRADU Master RADA	List/Listova
		1/1

(Podatke unosi predmeti nastavnik - mentor)

Vrsta studija	Master studije
Studijski program:	Energetika, elektronika i telekomunikacije
Rukovodilac studijskog programa	Dr Stevan Grabić

Student:	Nikola Kovačević	Broj Indeksa	E1 6/2018
Oblast	Projektovanje digitalnih sistema		
Mentor	Dr Vuk Vranjković		

NA OSNOVU PODNETE PRIJAVA, PRILOŽENE DOKUMENTACIJE I ODREDBI STATUTA FAKULTETA IZDAJE SE ZADATAK ZA DIPLOMSKI(Bachelor) RAD, SA SLEDEĆIM ELEMENTIMA: - Problem-tema rada - Način rešavanja problema i način praktične provere rezultata rada, ako je takva provera neophodna; - Literatura

NASLOV MASTER RADA

Implementacija vektorskog procesora baziranog na RISC-V setu instrukcija

TEKST ZADATKA:

Implementirati vektorski procesor prateći verziju 0.8 RISC-V nacrtu vektorske ekstenzije

Rukovodilac studijskog programa:	Mentor rada:
Dr Stevan Grabić	Dr Vuk Vranjković

Primerak za: ☐ Studenta; ☐ Mentora;

Sadržaj

1	Uvod.....	10
1.1	Pregled rada po poglavljima.....	10
2	Uvod u vektorske procesore.....	11
2.1	Vektorsko procesiranje.....	12
2.2	Prednosti vektorskog seta instrukcija.....	14
2.3	RISC-V ISA (<i>eng. Instruction Set Architecture</i>).....	16
2.4	FPGA platforma i motivacija za njeno korišćenje.....	17
3	Implementacija vektorskog procesora RV32IV.....	20
3.1	Opis skalarnog jezgra.....	21
3.2	RISC-V vektorska ekstenzija i karakteristike (<i>eng. features</i>) vektorskog procesora.....	25
3.2.1	Programski model vektorske ekstenzije.....	26
3.2.2	Skup vektorskih instrukcija koje ekstenzija podržava.....	28
3.2.3	Vektorsko maskiranje.....	30
3.3	Mikroarhitektura vektorskog jezgra.....	31
3.3.1	Vektorske linije (<i>eng. Vector lanes</i>).....	33
3.3.2	Vektorska registarska banka (VRF).....	36
3.3.3	Vektorska kontrolna jedinica (V_CU).....	42
3.3.4	Arbiter.....	43
3.3.5	M_CU modul i memorijski podsistem.....	48
4	Funkcionalna verifikacija vektorskog jezgra.....	53
4.1	Opis verifikacionog okruženja.....	55
5	Performanse i iskorišćenost resursa.....	59
5.1	Iskorišćenost resursa.....	61
5.2	Performanse.....	63
6	Zaključak.....	64
7	Bibliografija.....	65

Slike

<i>Slika 1: Tri vrste paralelizacije. Kvadrat, krug, trougao i plus, predstavljaju različite vrste instrukcija, dok svaki okvir oko njih predstavlja jednu instrukciju.....</i>	<i>11</i>
<i>Slika 2: Pojednostavljena struktura skalarnog procesora.....</i>	<i>12</i>
<i>Slika 3: Pojednostavljena struktura vektorskog procesora.....</i>	<i>13</i>
<i>Slika 4: Prikaz vektorskog procesora koji poseduje 1 liniju za egzekuciju vektorske instrukcije i procesora koji poseduje 4 [1, poglavlje 4].....</i>	<i>15</i>
<i>Slika 5: Reprezentacija logičke funkcije pomoću lukap tabele.....</i>	<i>18</i>
<i>Slika 6: Blok dijagram procesora. U uokvirenom delu su osnovne komponente koje čine RV32I skalarno jezgro, dok svi ostali blokovi pripadaju vektorskom jezgru.....</i>	<i>20</i>
<i>Slika 7: Blok šema skalarnog jezgra.....</i>	<i>21</i>
<i>Slika 8: Modifikacije skalarnog jezgra kako bi se omogućila njegova sprega sa vektorskim jezgrom.....</i>	<i>23</i>
<i>Slika 9: Metoda ulančavanja.....</i>	<i>31</i>
<i>Slika 10: Izvršavanje vektorske instrukcije na više različiti vektorskih linija.....</i>	<i>31</i>
<i>Slika 11: Mikroarhitektura vektorsko jezgra.....</i>	<i>32</i>
<i>Slika 12: Prikaz najbitnijih blokova unutar vektorskih linija.....</i>	<i>33</i>
<i>Slika 13: Realizacija množača pomoću više DSP ćelija.....</i>	<i>35</i>
<i>Slika 14: Vektorska registarska banka.....</i>	<i>36</i>
<i>Slika 15: Slika koja prikazuje talasne oblike prilikom čitanja podataka iz VRF modula.....</i>	<i>38</i>
<i>Slika 16: Primer operacije upisa u VRF modul.....</i>	<i>39</i>
<i>Slika 17: Operacija istovremenog čitanja i upisa u VRF modul.....</i>	<i>40</i>
<i>Slika 18: Primer operacije istovremenog upisa i čitanja.....</i>	<i>41</i>
<i>Slika 19: Interfejs Arbiter modula.....</i>	<i>43</i>
<i>Slika 20: Provera zavisnosti pristigle instrukcije i load instrukcija u lokalnoj memoriji.....</i>	<i>45</i>
<i>Slika 21: Sprega memorijskog podsistema i M_CU modula sa vektorskim linijama. .</i>	<i>48</i>

<i>Slika 22: Hijerarhija verifikacionog okruženja [11].....</i>	<i>54</i>
<i>Slika 23: Verifikaciono okruženje vektorskog jezgra.....</i>	<i>55</i>
<i>Slika 24: Talasni dijagrami izvršavanja jednostavnog programa.....</i>	<i>57</i>
<i>Slika 25: Integracija vektorskog procesora na zlybo razvojnoj ploči.....</i>	<i>60</i>
<i>Slika 26: Kritična putanja unutar vektorske linije.....</i>	<i>63</i>

Tabele

Tabela 1: Raspored bita u Vtype registru.....	26
Tabela 2: Zavisnost dinamičke širine elementa (SEW) of vsew[2:0].....	27
Tabela 3: Vrednost LMUL u zavisnosti od vlmul.....	27
Tabela 4: Tip instrukcije u zavisnosti od funct3 polja.....	28
Tabela 5: Karakteristike Zybo razvojne ploče.....	58
Tabela 6: Resursi koje koriste pojedinačni blokovi vektorskog procesora.....	60
Tabela 7: Iskorišćenost resursa kada je broj vektorskih linija 1.....	61
Tabela 8: Iskorišćenost resursa kada je broj vektorskih linija 4.....	61
Tabela 9: Iskorišćenost resursa kada je broj vektorskih linija 8.....	61
Tabela 10: Iskorišćenost resursa celog sistema kada je broj vektorskih linija 8.....	61

1 Uvod

2004. godine Denardovo skaliranje (*eng. Dennard scaling*) [1] prestaje da važi i frekvencije rada procesora se od tada sve sporije povećavaju. Ta promena je mikroprocesorsku industriju primorala da pronalazi nova rešenja kako bi povećala performanse procesora i ono što se pokazalo kao prekretnica jeste stavljanje akcenta na paralelizam prilikom obrade podataka (*eng. Data level parallelism*).

Trenutno, najuspešnija i najzastupljenija arhitektura za paralelnu obradu podataka jeste GPU (*eng. Graphics processing unit*). No, Vektorski procesori, koji datiraju još od 60-tih godina prošlog veka, su jedna od arhitektura kod kojih je ova vrsta paralelizma takođe izražena, ali su do skora smatrani jako „skupim“. Jedan razlog je broj tranzistora, ali drugi, možda i bitniji, potreba za DRAM (*eng. Dynamic Random Access Memory*) memorijama koje mogu dovoljno brzo da „nahrane“ vektorski procesor podacima [1, poglavlje 4]. Napretkom tehnologije, potrebom za što većom paralelizacijom obrade podataka, sa što većom energetsom efikasnošću, pojavom RISC-V instrukcijskog seta, ovi procesori su ponovo skrenuli pažnju na sebe. Iz tih razloga ovaj rad se bavi analizom i implementacijom vektorskog procesora baziranog na RISC-V arhitekturi.

1.1 Pregled rada po poglavljima

Poglavlje 2 je uvod u vektorsko procesiranje. Tu se opisuju različite vrste paralelizma (njihove prednosti i mane), pomoću jednostavnog primera objašnjen je način rada vektorskih procesora i opisana je RISC-V arhitektura. Takođe, objašnjeno je šta su FPGA platforme i zašto je vektorski procesor u ovom radu implementiran pomoću njih.

Poglavlje 3 posvećeno je implementaciji vektorskog procesora. Ono je podeljeno na tri velike celine: opis skalarnog jezgra, opis RISC-V vektorske ekstenzije i opis mikroarhitekture vektorskog jezgra.

U poglavlju 4 opisano je jednostavno verifikaciono okruženje zasnovano na UVM (*eng. Universal Verification Methodology*) metodologiji pomoću kojeg je potvrđen ispravan rad procesora.

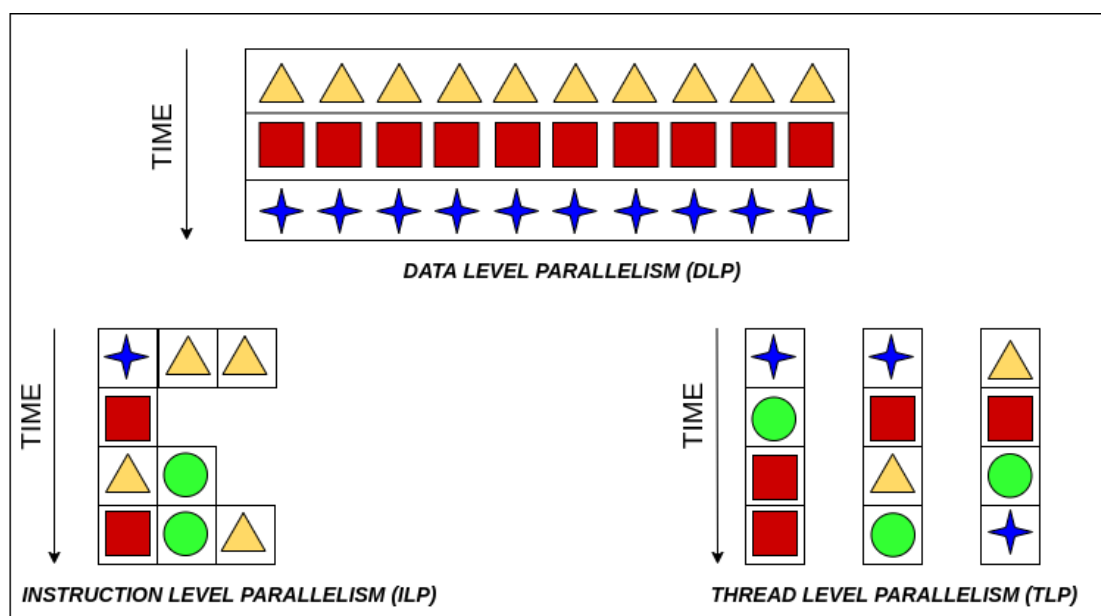
Poglavlje 5 je posvećeno analizi iskorišćenosti resursa i performansi prilikom implementacije procesora na *zybo* razvojnoj ploči.

Poglavlje 6 sumira rad i predlaže moguća poboljšanja u budućem radu.

2 Uvod u vektorske procesore.

Kako bi povećali performanse, preko granica koje dopušta trenutna tehnologija izrade čipova, sistemske arhitekture pribegavaju različitim vrstama paralelizacije. Na slici 1 su prikazane tri glavne kategorije:

- Paralelizam na nivou instrukcija, skraćeno ILP (eng. *Instruction Level Parallelism*), omogućava istovremeno izvršavanje više instrukcija iz jednog sekvencijalnog skupa. Primer su procesori sa protočnom obradom [1, poglavlje 3], kod kojih pre nego što se završi egzekucija prethodne instrukcije, kreće se sa narednom.
- Paralelizam na nivou niti, skraćeno TLP (eng. *Thread Level Parallelism*), omogućava istovremeno izvršavanje instrukcija iz više odvojenih skupova. Primer su multiprocesorski sistemi kod kojih svaki procesor može da izvršava njemu dodeljen skup instrukcija.
- Paralelizam na nivou podataka, skraćeno DLP (eng. *Data Level Parallelism*), omogućava izvršavanje jedne iste operacije istovremeno nad nizovima elemenata. Primer su vektorski procesori koji jednu vektorsku instrukciju primenjuju na više podataka istovremeno.



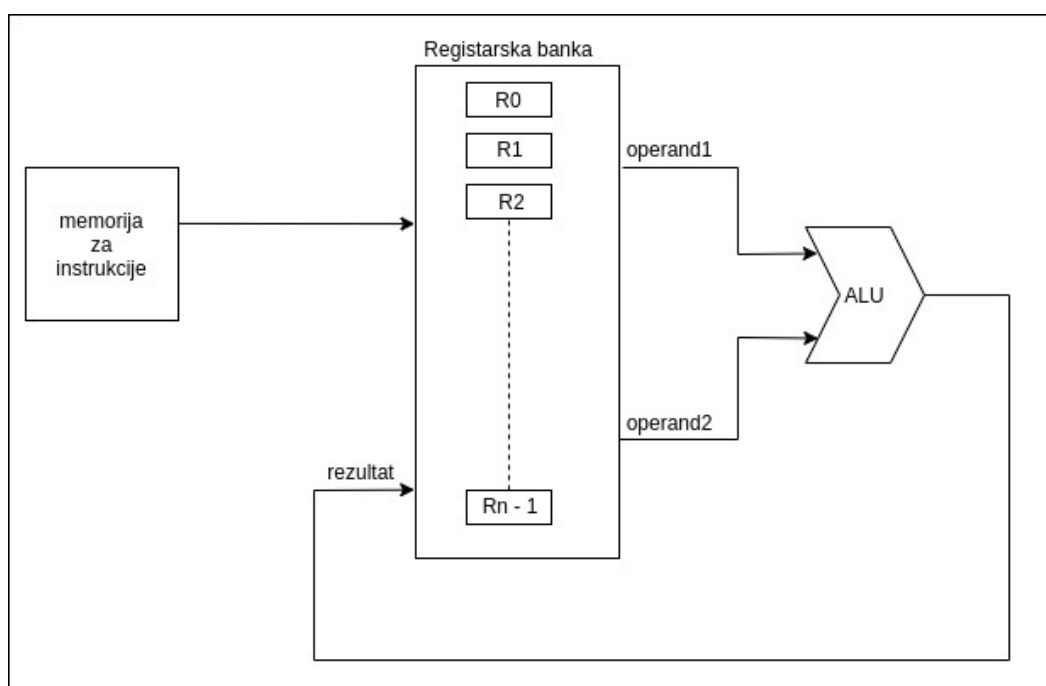
Slika 1: Tri vrste paralelizacije. Kvadrat, krug, trougao i plus, predstavljaju različite vrste instrukcija, dok svaki okvir oko njih predstavlja jednu instrukciju

Naravno, sistemske arhitekture nisu ograničene na eksploataisanje samo jedne vrste paralelizacije, tako na primer u multiprocesorskim sistemima pojedinačna jezgra su najčešće procesori sa protočnom obradom.

Od prethodne tri vrste paralelizma DLP se znatno bolje skalira, jer su podaci nad kojima vektorski procesor vrši određenu operaciju, na osnovu prihvaćene instrukcije, međusobno nezavisni, dok kod arhitektura koje iskorišćavaju TLP i ILP rešavanje zavisnosti između instrukcija zahteva dodatni hardver. Naravno, kada god se pravi sistem koji eksploatiše DLP mora da se postavi pitanje koliki set aplikacija zahteva visoki paralelizam prilikom obrade podataka. Na sreću, u trenutnom dobu postoji velika potreba za ovakvim sistemima, jer svi algoritmi vezani za mašinsko učenje, obradu slike, obradu zvuka, itd, mogu da se u velikoj meri paralelizuju.

2.1 Vektorsko procesiranje

Princip rada skalarnih procesora predstavlja dobar uvod u vektorske procesore, te će stoga početak ove sekcije biti posvećen tome. Skalarni procesor izvršava određene operacije nad unutrašnjim registrima, u zavisnosti od prihvaćene instrukcije, i međurezultat opet smešta u registre. Sledeća slika ilustruje pojednostavljenu strukturu skalarnog procesora:



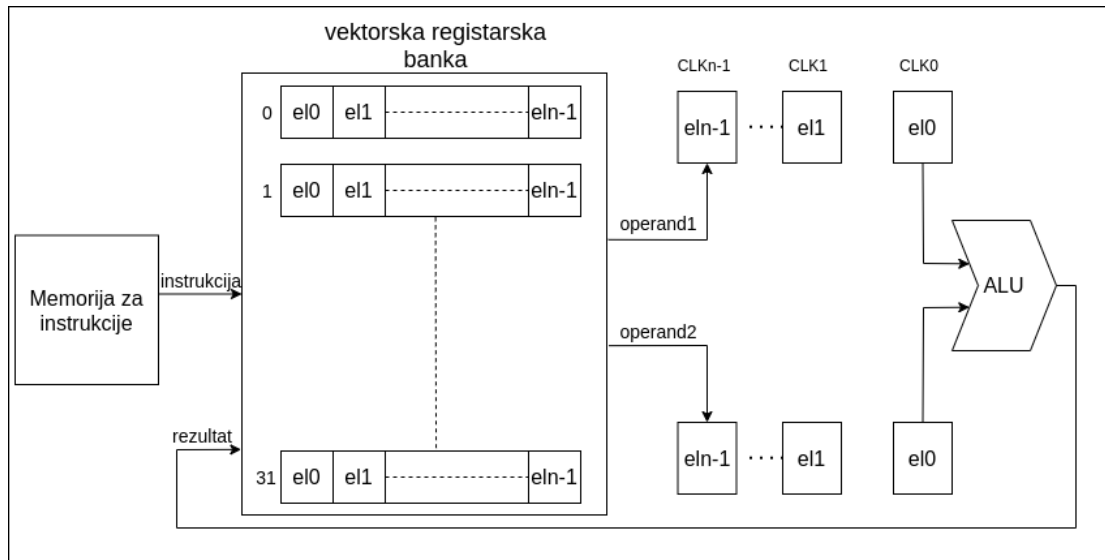
Slika 2: Pojednostavljena struktura skalarnog procesora

Na slici su prikazane 3 ključne komponente: memorija za instrukcije, registarska banka i ALU (aritmetičko logička jedinica). Na osnovu prihvaćene instrukcije unutrašnji registri će biti pročitani i sprovedeni do ALU komponente gde će se nad njima izvršiti određena operacija, nakon čega će oni ponovo biti smešteni u jedan od registara registarske banke. U nastavku je prikazana jedna instrukcija iz RISC-V arhitekture:

add rs10, rs11, rs12 (1)

Prethodna instrukcija sabira vrednosti u registrima rs11 i rs12 i rezultat smešta u registar rs10.

Struktura vektorskih procesora je jako slična prethodno opisanoj, kao što se može videti na sledećoj slici:



Slika 3: Pojednostavljena struktura vektorskog procesora

Ključne komponente su iste, osim što je registarska banka zamenjena vektorskom registarskom bankom. Razlika između te dve komponente jeste u tome što se registarska banka sastoji iz registara određene širine (broj bita je određen arhitekturom procesora), dok se vektorska registarska banka sastoji iz vektora, pri čemu svaki vektor unutar sebe sadrži određeni broj elemenata (širina pojedinačnog elementa je takođe određena arhitekturom). Prihvatom vektorske instrukcije, kao što se kod skalarnog procesora čitaju pojedinačni registri i izvršava određena operacija nad njima, tako se kod vektorskog procesora, iz vektorske registarske banke, čitaju vektori i jedna ista operacija se izvršava nad elementima unutar njih. Osnovna ideja rada vektorskog procesora je prikazana na prethodnoj slici. Unutar instrukcije se nalazi informacija kojim se vektorima pristupa i koju operaciju treba primeniti nad elementima unutar njih. Svakim taktom biće pročitani jedan par elemenata, počevši od elementa na indeksu „i“, izvršiće se određena operacija nad njima i rezultat će biti smešten u određeni vektor unutar vektorse registarske banke. Primer jedne vektorske instrukcije iz RISC-V arhitekture je prikazana u nastavku:

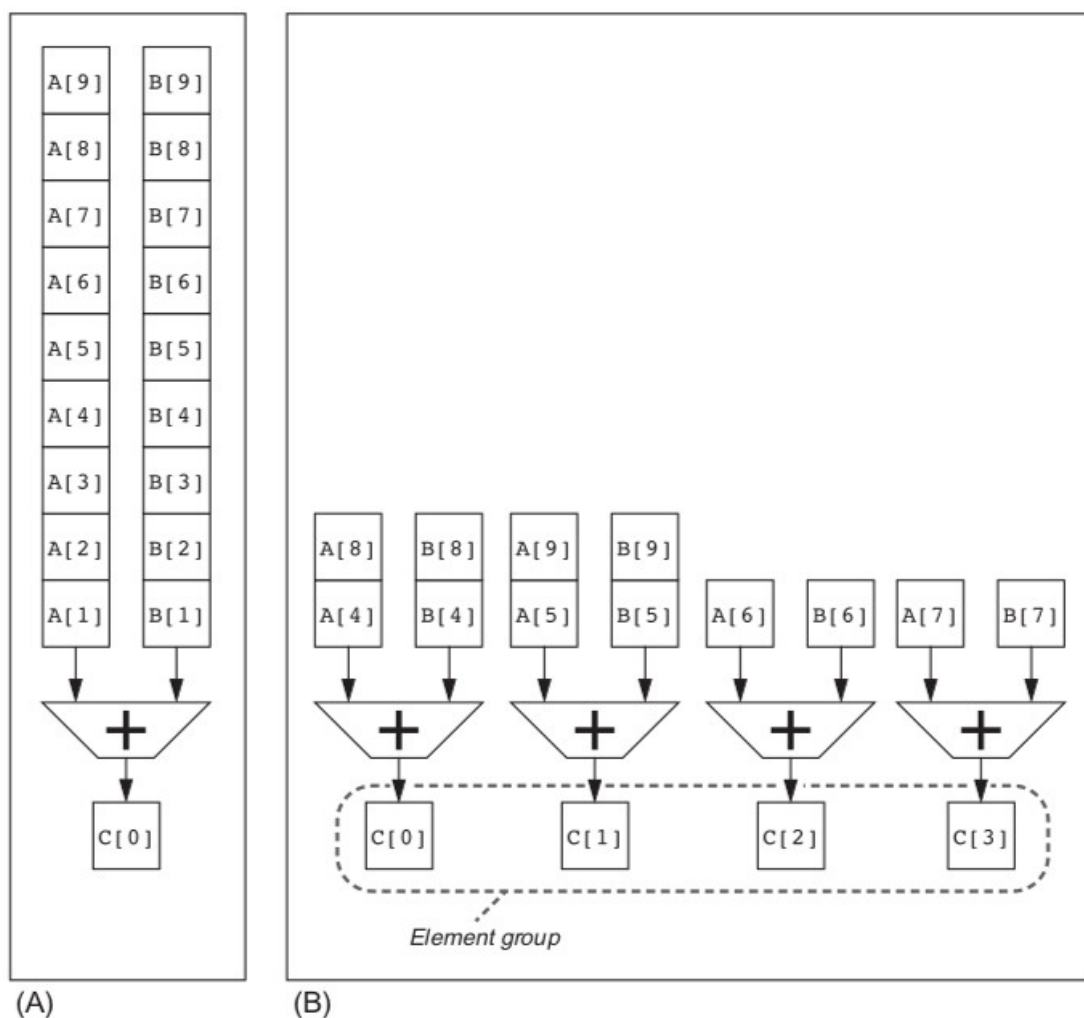
$$\text{vadd.vv } v10, v11, v12 \quad (2)$$

Prethodna instrukcija sabira pojedinačne elemente unutar vektora v11 i v12 i rezultat smešta u vektor v10. Ukoliko bi sličan rezultat trebao da se postigne na skalarnom procesoru (da se nad više elemenata izvrši operacija sabiranja) bilo bi neophodno izvršiti onoliko skalarnih instrukcija koliko ima elemenata u jednom vektoru.

2.2 Prednosti vektorskog seta instrukcija

Prethodni primer naveo je jednu prednost vektorskog procesora, ali postoji još razloga zašto je vektorski set instrukcija jako koristan:

- Vektorski set instrukcije pakuje više nezavisnih operacije u jednu kratku instrukciju, što dovodi do kompaktnog koda. Kod je kompaktan zato što jedna instrukcija može da opiše N operacija i da adresira $3N$ registarskih operanada. Ovo u mnogome smanjuje neophodnu propusnu moć prilikom prihvata instrukcija iz memorije i takođe eliminiše neophodne petlje koje se javljaju ukoliko skalarni procesor rešava isti zadatak, jer su one unutar vektorskog procesora implicitne.
- Vektorski set instrukcija umanjuje hardver neophodan prilikom dekodovanja vektorske instrukcije. Razlog za to je što se jedna vektorska instrukcija primenjuje N puta (na N elemenata). Takođe, N operacija koje se izvode su međusobno nezavisne, te iz tog razloga nije neophodan hardver za detekciju zavisnosti.
- Šablon koji vektorski procesor prati prilikom izvođenja operacija nad elementima vektorskog registra je regularan. Ukoliko se, na primer, izvršava operacija vektorskog množenja, element na indeksu i jednog vektorskog registra množi se sa elementom na indeksu i drugog vektorskog registra i smešta se na indeks i trećeg vektorskog registra. Ovaj regularni šablon omogućava visok stepen paralelizma, jer bi vektorski procesor mogao da se implementira pomoću više paralelnih linija, pri čemu bi svaka linija vršila operacije nad jednim delom vektorskih elemenata. Slika 4 ilustruje poređenje vektorskog procesora sa jednom linijom i sa četiri linije.
- Jedna od najbitnijih prednosti je što se vektorski set instrukcija može dodati kao ekstenzija na već postojeći skalarni set.



Slika 4: Prikaz vektorskog procesora koji poseduje 1 liniju za egzekuciju vektorske instrukcije i procesora koji poseduje 4 [1, poglavlje 4]

Iz prethodno opisanog vidi se da vektorsko procesiranje ima smisla, naravno, ovo važi jedino ukoliko je zadatak koji procesor treba da izvrši takav da paralelizam između podataka može da se eksploatiše u velikoj meri.

2.3 RISC-V ISA (eng. *Instruction Set Architecture*)

Arhitektura skupa instrukcija (eng. ISA, Instruction Set Architecture) opisuje na koji način određeni procesor funkcioniše i koje su njegove mogućnosti. Ona opisuje registre koje će procesor imati kao i sve mašinske instrukcije koje će podržavati. Takođe, ISA specificira do detalja šta koja instrukcija radi i na koji način će ona biti kodovana u memoriji. ISA predstavlja spregu između hardvera i softvera. Hardverski inženjeri dizajniraju i modeluju kola koja će izvršavati dati skup instrukcija. Softverski inženjeri pišu kod (operativne sisteme, kompajlere) bazirane na ovome skupu instrukcija [2].

Iz tog razloga prilikom projektovanja procesora neophodno je odabrati određenu arhitekturu, čiji set ili podset instrukcija će biti podržan. U ovom radu odabrana je RISC-V ISA. Ova arhitektura, koja je potekla sa Berkli (eng. *Berkeley*) univerziteta, je novi set instrukcija (ISA) koji je na početku bio zamišljen da podrži naučna istraživanja i edukaciju, ali za koji sada postoji nada da će postati arhitektura koja će biti besplatna i otvorena za sve industrijske implementacije. Cilj je bio da se napravi moderan skup instrukcija koji sadrži sve do danas najbolje ideje u dizajnu procesora. RISC-V je morao biti dosta jednostavniji od onih koji su danas u upotrebi, a istovremeno praktičan za moderne primene i namenjen za dizajn što brže hardverske implementacije. Još jedan cilj je bio da se napravi "čista" (pure) RISC arhitektura gde će se svaka instrukcija izvršiti za tačno jedan takt, pa instrukcije moraju biti jednostavne i ograničene [2].

Instrukcije RISC-V arhitekture su podeljene na skupove, pri čemu svaki skup ima svoju oznaku:

- **I** skup *integer* instrukcija.
- **M** Instrukcije množenje i deljenja celobrojnih podataka
- **A** Atomičke instrukcije
- **F** Instrukcije za operacije nad brojevima sa pokretnim zarezom, jednostruka preciznost (float)
- **D** Operacije brojevima sa pokretnim zarezom, dvostruka preciznost (double).
- **V** vektorski skup instrukcija

Takođe pored ovih podržane su još instrukcije sa oznakama: S, Q, C, E, L, P, B, T. No, detaljnije o njima može se pronaći u [3]. Svaki procesor zasnovan na RISC-V arhitekturi bi trebalo da implementira bazni „I“ skup instrukcije, dok su ostali opcioni.

U nastavku ovog rada biće korišćene sledeća skraćenica prilikom opisivanja arhitekture koja se koristi: RV32IV.

- **RV** je skraćenica za RISC-V.
- **32** govori da je procesor 32-bitni (podržani su 64-bitni i 128-bitni).
- **IV** govori koje setove instrukcija procesor podržava. U ovom primeru podržan je osnovni *integer* set instrukcija i vektorski set instrukcija. Ukoliko bi procesor bio proširen i operacijama množenja imao bi oznaku RV32IMV.

2.4 FPGA platforma i motivacija za njeno korišćenje

Vektorski procesor opisan u ovom radu je realizovan pomoću FPGA (*eng. Field Programmable Gate Arrays*) platforme. To su poluprovodnički uređaji zasnovani na matricama konfigurabilnih logičkih blokova povezanih pomoću programabilnih interkonekcija. Na ovim uređajima može biti implementiran bilo koji hardverski dizajn i što je najbitnije mogu biti reprogramirani kako bi se na njima implementirao neki drugi dizajn. Ova karakteristika izdvaja FPGA platforme od ASIC (*eng. Application Specific Integrated Circuit*) tehnologije izrade digitalnih integrisanih kola. No, mogućnost reprogramiranja dolazi uz cenu, a to je da hardverski moduli implementirani na FPGA platformi mogu da imaju znatno lošije performanse od onih izrađenih u ASIC tehnologiji. Ostale prednosti koje nudi FPGA platforma su:

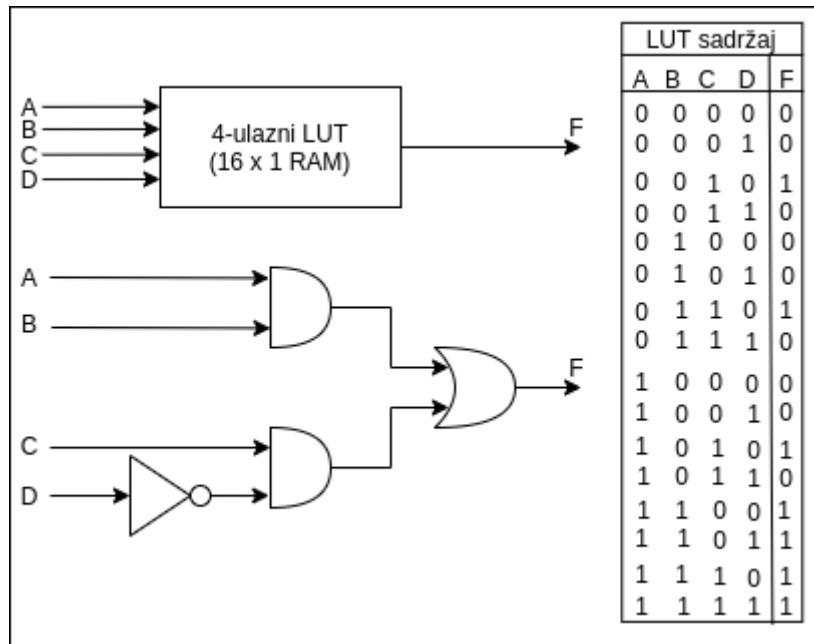
- Mala cena. Zbog mogućnosti reprogramiranja, implementacija sistema na FPGA platformi je mnogo jeftinija od sistema implementiranih pomoću ASIC tehnologije, jer za svaku modifikaciju neophodno je fabrikovati novi čip.
- Ubrzavanje izrade čipova u ASIC tehnologiji, jer pomoću FPGA platformi može da se testira ispravnost rada sistema pre nego što se fabrikuje čip.

Kao što je prethodno pomenuto, FPGA platforme se sastoje od konačnog broja predefinisanih logičkih blokova povezanih pomoću programabilnih interkonekcija i ulazno/izlaznih blokova koji omogućavaju komunikaciju sa „spoljnim svetom“. Osnovni logički blokovi koji postoje na FPGA platformama su:

- Flip flop – binarni registar koji se koristi kao sinhronizaciona logika za čuvanje stanja između dva otkucaja takta. Na svaku rastuću ivicu takta, stanje na ulazu flip flop ćelije se prosleđuje na njegov izlaz.
- LUT (*eng. Look Up Table*) – jako mali RAM realizovan kao lukap tabela (*eng. Look up table*), odnosno tabela sa predefinisanim vrednostima, pomoću kojeg se može realizovati bilo koja logička funkcija (*NAND, NOR, XOR*, itd). U zavisnosti od FPGA platforme, LUT ćelije mogu biti različite veličine. Na slici 5 je ilustrovana primer realizacije logičke funkcije pomoću lukap tabele.
- DSP (*eng. Digital signal processing*) – omogućava izvršavanje operacije pomnoži i akumuliraj (*eng. Multiply and accumulate*). Kod Xilinx [6] FPGA

platformi unutar DSP bloka postoji 18x25 množak sa dodatnom logikom za sabiranje.

- BRAM (*eng. Block RAM*) – je memorija koja se nalazi na FPGA platformi i uglavnom je podeljena na više blokova koji su 18Kb ili 36Kb veličine. Ona je korisna, jer jako često je potrebno vršiti operacije nad blokovima podataka i na ovaj način se omogućava brz pristup njima. Memorija se može implementirati i pomoću flip flop ćelija, ali to treba izbegavati, jer memorija veličine 36Kb bi zauzela većinu flip flop ćelija nekih FPGA platformi.



Slika 5: Reprezentacija logičke funkcije pomoću lukap tabele

Kako bi se prethodno opisani logički blokovi spojili u funkcionalnu celinu, prilikom dizajna sistema koriste se jezici za opis hardvera ili HDL (*eng. Hardware Description Language*). Dva najpoznatija su Verilog i VHDL. Nakon što je hardver opisan, određeni alati, kao što je Vivado alat kompanije Xilinx [6], to analiziraju i generišu izvršne fajlove pomoću kojih se FPGA platforme programiraju.

Sistemi koji se realizuju na FPGA platformama su najčešće akceleratori za aplikacije koje iskorišćavaju paralelizam između podataka. To su hardverski blokovi dizajnirani da obavljaju jednu vrstu zadatka sa ne toliko konfigurabilnih opcija. Oblasti u kojima je ovakav pristup zastupljen su: obrada slike i videa, mašinsko učenje (*eng. Machine learning*), emulacija hardvera, itd.

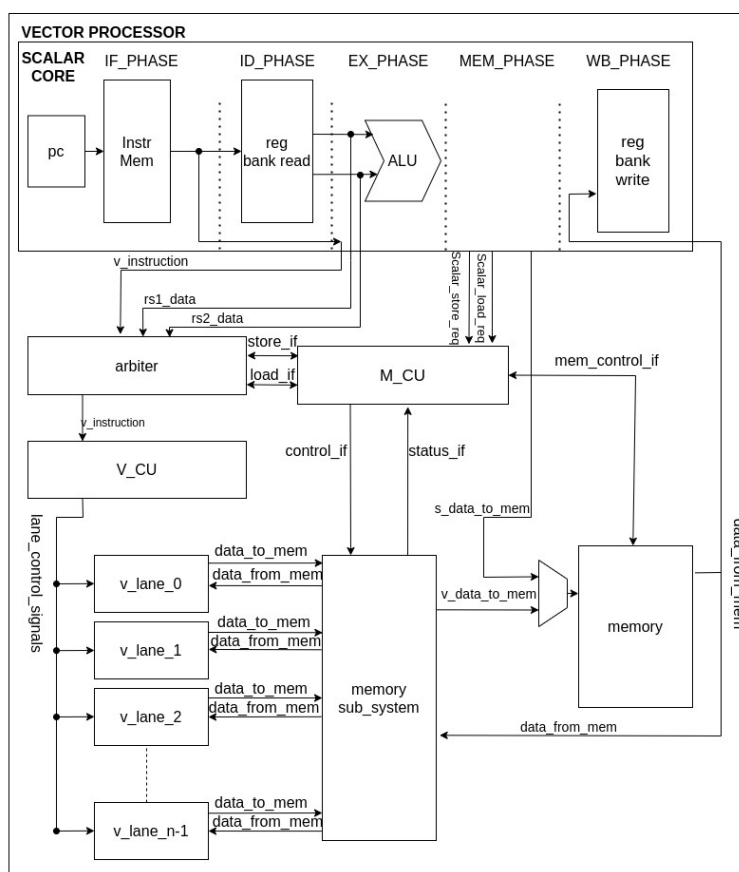
Ovaj rad istražuje alternativnu mogućnost korišćenja FPGA platforme za kreiranje vektorskog procesora kao akceleratora opšte namene. Procesor bi posedovao standardan set instrukcija, tako da bi svako, bez iskustva sa dizajnom hardvera, mogao da ga programira. Takođe, zbog mogućnosti reprogramiranja dizajna na FPGA

platformama, moćiće da se menjanju neke od karakteristika procesora kako bi se povećale performanse ili kako bi se optimizovala iskorišćenost resursa. Na primer, broj vektorskih linija (sekcija 2.2) bi mogao da se poveća, odnosno smanji, u zavisnosti od toga da li su od interesa performanse, ili zauzetost logičkih blokova na FPGA platformi.

3 Implementacija vektorskog procesora RV32IV

U ovoj sekciji biće opisana RISC-V vektorska ekstenzija kao i mikroarhitektura procesora baziranog na njoj. Osnovna ideja je da se pored skalarnog jezgra, koje podržava RISC-V *integer* set instrukcija, implementira i vektorsko jezgro koje bi se ponašalo kao ekstenzija na već postojeći set instrukcija. Opis implementacije podeljen je na sledeće sekcije.

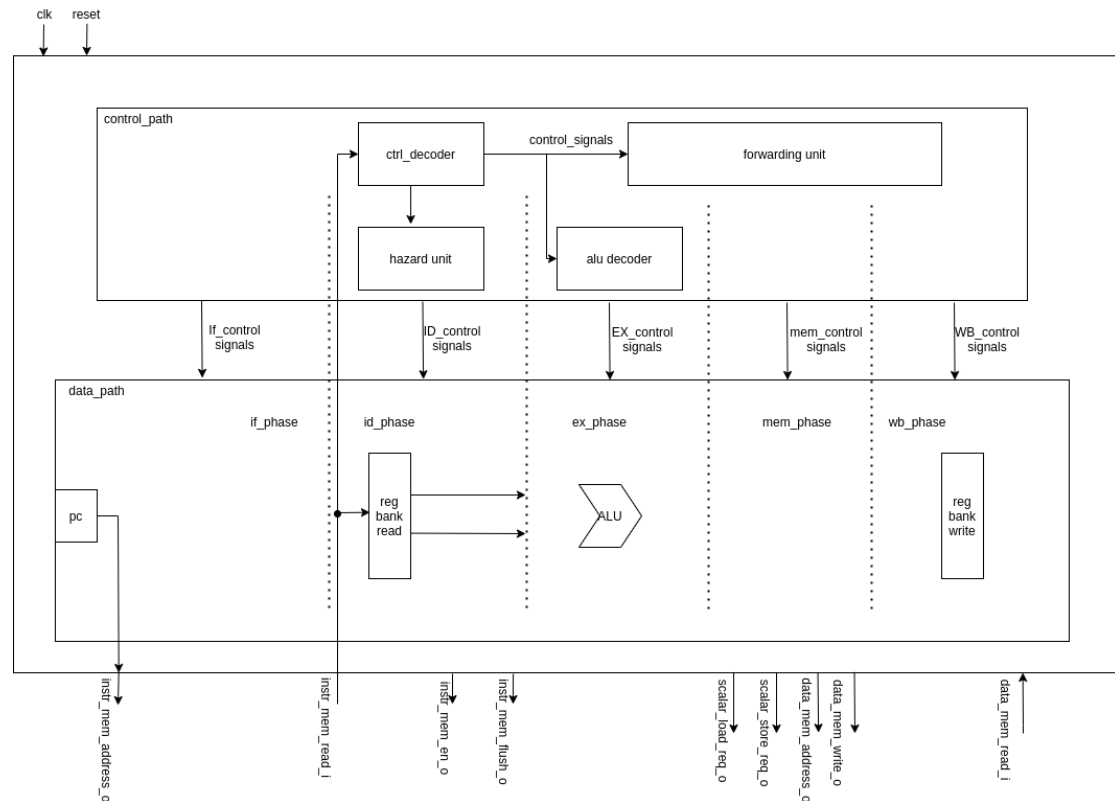
- **Opis skalarnog jezgra**, zaduženog za izvršavanje delova koda koji ne mogu da se vektorizuju, kao i za kontrolu grananja.
- **RISC-V vektorska ekstenzija i karakteristike vektorskog procesora.** U ovoj sekciji su ukratko opisani vektorski set instrukcija baziran na RISC-V arhitekturi, kao i koje njegove delove ova implementacija vektorskog procesora poseduje.
- **Mikroarhitektura vektorskog jezgra.** U ovoj sekciji su opisani osnovni gradivni blokovi vektorskog jezgra, kao i način komunikacije između njih.



Slika 6: Blok dijagram procesora. U uokvirenom delu su osnovne komponente koje čine RV32I skalarno jezgro, dok svi ostali blokovi pripadaju vektorskom jezgru

3.1 Opis skalarnog jezgra

Skalarno jezgro je 32-bitni procesor bez dinamičkog izvršavanja instrukcija (*eng. In-order processor*) koji implementira RISC-V *integer* set instrukcija. Pošto implementacija skalarnog procesora nije glavna tema ovog rada, u ovoj sekciji će biti objašnjene samo njegove glavne osobine i na koji način se postiže sprega sa vektorskim jezgrom. Naredna slika predstavlja izgled skalarnog procesora.



Slika 7: Blok šema skalarnog jezgra

Kao što se može videti sa prethodne slike skalarni procesor poseduje sledeći interfejs

- *clk* i *reset* - ulazni sinhronizacioni portovi.
- *instr_ready_i* – ulazni jednobitni signal koji naglašava da je pristigla instrukcija validna. Njegova vrednost će uvek biti postavljena na logičku 1. Razlog za to je što se instrukcije skladište unuta BRAM ćelija, kojima treba samo 1 takt kako bi postavili validan podatak na svoj izlaz.
- *data_ready_i* – Važi isto što i za *instr_ready_i* port.
- *instr_mem_address_o* – Izlazni-32 bitni signal koji predstavlja adresu na kojoj se nalazi naredna instrukcija koju procesor treba da izvrši.

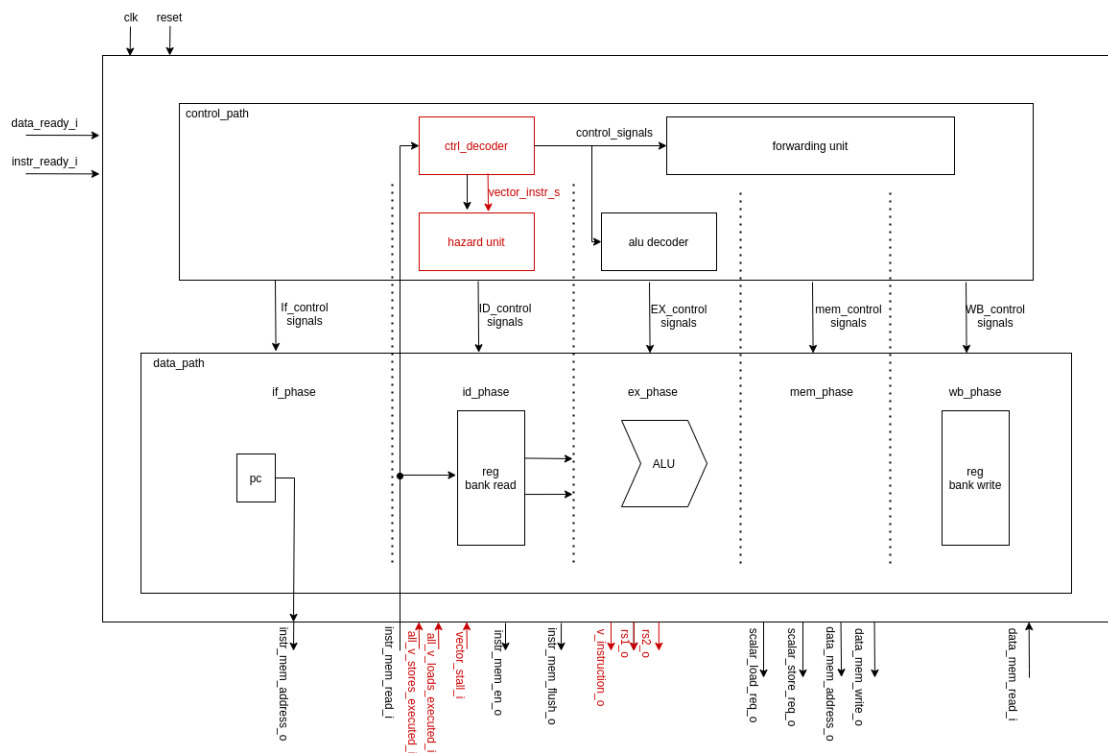
- *instr_mem_read_i* – Ulazni-32 bitni signal koji predstavlja instrukcija koju procesor treba da izvrši
- *instr_mem_flush_o* – Izlazni jednobitni signal koji ukoliko je postavljen na logičku 1₂ znači da naredna instrukcija koja treba da se učitava iz memorije treba da bude odbačena i da umesto nje treba da se postavi prazna (*eng. nop*) instrukcija. Ovo se dešava ukoliko procesor treba da izvrši instrukciju skoka (*eng. branch*).
- *instr_mem_en_o* – Izlazni jednobitni signal koji ukoliko je postavljen na logičku 0₂ zaustavlja dalje učitavanje instrukcija. Procesor to radi jer mora da razreši unutrašnje konflikte pre nego što nastavi sa daljim izvršavanjem instrukcija.
- *scalar_load_req_o* – Izlazni jednobitni signal koji ukoliko je postavljen na logičku 1₂ naglašava da skalarno jezgro ima zahtev za čitanjem iz memorije.
- *scalar_store_req_o* - Izlazni jednobitni signal koji ukoliko je postavljen na logičku 1₂ naglašava da skalarno jezgro ima zahtev za upis u memoriju
- *data_mem_address_o* – Izlazni 32-bitni signal koji predstavlja adresu memorije sa podacima sa koje skalarno jezgro želi da učitava ili na koju želi da upiše podatak.
- *data_mem_read_i* – Ulazni 32-bitni podatak koji se učitava iz memorije za podatke (rezultat izvršavanja *load* instrukcije).
- *data_mem_write_o* – Izlazni 32-bitni podatak koji se upisuje u memoriju za podatke (rezultat izvršavanja *store* instrukcije).

Kao što se može videti sa slike 7 jezgro poseduje 5 faza protočne obrade: *Instruction fetch*, *instruction decode*, *execute*, *memory* i *writeback*. Takođe, procesor je podeljen na dve velike celine: *data_path* i *control_path*. Unutar *data path* celine se nalaze sve komponente neophodne da bi se izvršila instrukcija, dok *Control path* celina, u zavisnosti od prihvaćene instrukcije, generiše odgovarajuće kontrolne signale koji upravljaju tim komponentama. Sa slike 7 se može videti da se unutar *control path* celine nalaze sledeće komponente:

- *ctrl_decoder* – je zadužen za dekodovanje pristigle instrukcije i za generisanje odgovarajućih kontrolnih signala na osnovu tog dekodovanja.
- *alu_decoder* – prima određene kontrolne signale od *ctrl_decoder* komponente, na osnovu kojih generiše kontrolne signale za aritmetičko logičku jedinicu
- *hazard_unit* – generiše signale za zaustavljanje (*eng. stall*) procesora ukoliko za tim postoji potreba.

- *forwarding_unit* – generiše kontrolne signale koji razrešavaju hazarde podataka (zavisnosti između instrukcija) koji nastaju uvođenjem protočne obrade.

Prethodno opisano je skalarni procesor koji ne komunicira sa vektorskim procesorom i da bi se ovo omogućilo neophodno je napraviti izmene prikazane na sledećoj slici:



Slika 8: Modifikacije skalarnog jezgra kako bi se omogućila njegova sprega sa vektorskim jezgrom

Sa slike se vidi da je skalarno jezgro prošireno sledećim portovima:

- *v_instruction_o* – Pošto je skalarno jezgro zaduženo i za prihvatanje vektorskih instrukcija, preko ovog porta se one prosleđuju vektorskom jezgrom. Kao što se sa slike može videti, instrukcija se prosleđuje iz EX faze jer su u tom trenutku rešeni svi hazardi podataka vezani za izlazne portove *rs1_o* i *rs2_o*.
- *rs1_o* i *rs2_o* – Uz instrukciju, vektorskom jezgrom se prosleđuju i podaci iz registarske banke, jer neke vektorske instrukcije rade sa skalarnim operandima. Jedan takav primer je vektorska *load* instrukcija sa korakom (eng. *Strided load*), njoj su neophodni vrednosti iz skalarnog registra, jer se u *rs1_o* nalazi bazna adresa sa koje kreće da preuzima podatke, a u *rs2_o* se nalazi udaljenost (eng. *offset*) narednog elementa. Ove dve vrednosti se takođe prosleđuju iz EX faze, jer su u tom trenutku rešeni svi hazardi podataka.

- *All_v_stores_executed_i* i *All_v_loads_executed_i* – Prilikom rada procesora, može da dođe do konflikta između skalarnih i vektorskih *load* i *store* instrukcija. Na primer, ukoliko pristigne skalarna *store* instrukcija, a vektorska *load* instrukcija se nije završila, neophodno je zaustaviti izvršavanje procesora, dok se ona ne izvrši. U suprotnom moglo bi da dođe do konflikta, jer bi skalarna *store* instrukcija mogla da prebriše vrednost u memoriji pre nego što je vektorski *load* učitao. Iz ovog razloga uvedena su gore pomenuta dva porta.
- *vector_stall_i* – Ukoliko je vektorski procesor zauzet izvršavanjem vektorske instrukcije, on će podići ovaj signal na logičku 1₂.

Sa slike se takođe može videti da je u skalarnom procesoru neophodno napraviti izmene unutar *control_path* celine, specifično, potrebno je nadograditi *ctrl_decoder* i *hazard_unit* blokove. *Ctrl_decoder* je potrebno izmeniti tako da je u mogućnosti da dekoduje vektorske instrukcije i da svaki put kada uoči takvu situaciju obavesti *hazard_unit* komponentu tako što će postaviti *vector_instruction_s* kontrolni signal na logičku 1₂. *Hazard_unit* komponentu treba proširiti tako da generiše signale za zaustavljanje procesora uzimajući u obzir nove ulazne portove i kontrolne signale. Sledeći scenariji treba da izazovu tako nešto:

1. *Vector_stall_i* = 1₂ i *vector_instruction_s* = 1₂. Ovo znači da se u ID fazi nalazi vektorska instrukcija, ali da je vektorsko jezgro još uvek zauzeto izvršavanjem prethodne instrukcije.
2. *all_v_stores_executed* = 0₂ i u ID fazi se nalazi skalarna *load* instrukcija.
3. *all_v_loads_executed* = 0₂ i u ID fazi se nalazi skalarna *load* instrukcija ili skalarna *store* instrukcija.

Sumacija ove sekcije je sledeća. Uloga skalarnog jezgra jeste da izvršava skalarni deo koda, kao i da vrši prihvatanje i prosleđivanje instrukcija vektorskom jezgru. Ako ne postoji zavisnost između instrukcija, oba jezgra mogu da funkcionišu paralelno.

3.2 RISC-V vektorska ekstenzija i karakteristike (eng. features) vektorskog procesora

Prilikom implementacije vektorskog jezgra praćena je verzija 0.8 RISC-V „V“ nacrt za vektorsku ekstenziju [4]. Ekstenzija je bazirana na vektorskim registarskim arhitekturama koje je uveo *Seymour Cray* 1970-ih godina, a ne na ranijem *packed SIMD* pristupu predstavljenom u *Lincoln Labs TX-2* digitalnom kompijuteru 1957 godine (iako je ovaj drugi pristup trenutno prihvaćen od strane komercijalnih setova instrukcija) [3]. Vektorski set instrukcija takođe poseduje mnoge karakteristike ranijih projekata, kao što su: *scale vector-thread* procesor sa MIT univerziteta, Berklijevi (eng. *Berkley*) *Maven* i *Hwacha* projekti, Berklijevi (eng. *Berkley*) *TO* i *VIRAM* vektorski procesori [3].

Ovaj nacrt opisuje sve karakteristike koje vektorska ekstenzija treba da poseduje da bi bila nazvana „bazna vektorska ekstenzija“ i da bi dobila ekstenziju „V“ u svom imenu. Termin bazna vektorska ekstenzija odnosi se na set instrukcija koji se koristi u standardnim serverskim platformama, no ukoliko neke *embedded* platforme ne zahtevaju sve instrukcije iz baznog seta, one mogu da implementiraju njegov podset, što je i učinjeno u ovom radu.

U nastavku će biti objašnjeni delovi 0.8 RISC-V „V“ nacrt a koje procesor u ovom radu implementira.

3.2.1 Programski model vektorske ekstenzije

Vektorska ekstenzija dodaje 32 vektorska registra (V0 – V31) na već postojeće registre skalarnog jezgra. Svaki vektorski registar biće predstavljen preko VLEN bita, pri čemu VLEN mora da bude stepen broja 2. To znači da ukoliko je potrebno da vektorski registar unutar sebe poseduje 32 elementa, pri čemu bi svaki bio 32-bitni, VLEN mora da bude 1024.

Pored vektorskih registara, ekstenzija dodaje i 2 neprivilegovana CSR (*eng. Control Status registers*) registra (*vtype* i *vl*) koji programeru mogu da daju veću kontrolu. U 0.8 RISC-V „V“ nacrtu [4] predviđeno je da postoji 7 neprivilegovanih CSR registara, ali zbog skupa instrukcija koje će ovaj procesor da poseduje zaključeno je da je dovoljno da poseduje samo *vtype* i *vl* registre.

Vtype (*eng. Vector type register*) je 32-bitni registar i stanje njegovih bita određuje na koji način se interpretiraju podaci unutar vektorskih registara. Sledeća tabela to ilustruje:

Tabela 1: Raspored bita u Vtype registru

Bit-i	Ime	Opis
XLEN-1	vill	<i>Ilegalna vrednost ukoliko je postavljen na 1₂</i>
XLEN-2:8		<i>rezervisan (upisati 0)</i>
7	vma	<i>Vector mask agnostic</i>
6	vta	<i>Vector tail agnostic</i>
5	vlmul[2]	<i>rezervisan</i>
4:2	vsew[2:0]	<i>standardna širina elementa, skraćeno SEW (eng. Standard element width)</i>
1:0	vlmul[1:0]	<i>Broj grupisanih vektora, skraćeno LMUL (Vector register group multiplier)</i>

Vsew postavlja dinamičku standardnu širinu elemenata (SEW) unutar vektorskog registra. Širina jednog elementa posmatra se kao VLEN/SEW i u zavisnosti od vsew, SEW može da ima sledeće vrednosti:

Tabela 2: Zavisnost dinamičke širine elementa (SEW) of vsew[2:0]

vsew[2:0]			SEW
0	0	0	8
0	0	1	16
0	1	0	32
0	1	1	64
1	0	0	128
1	0	1	256
1	1	0	512
1	1	1	1024

U ovom radu, prilikom implementacije vektorske ekstenzije vsew je fiksirano na vrednost „010“₂ i svi elementi unutar vektorskog registra se posmatraju kao 32-bitni i ne postoji mogućnost promene.

Promenom vrednosti **VLMUL** [1 : 0] bita moguće je grupisati više registara zajedno, tako da jedna vektorska instrukcija može da izvršava operacije nad više vektorskih registara. Odnosno, maksimalan broj elemenata na koje može da utiče jedna vektorska instrukcija je $VLEN/SEW * LMUL$, pri čemu je zavisnost *LMUL* (eng. *Vector length multiplier*) od *VLMUL* prikazana u sledećoj tabeli:

Tabela 3: Vrednost LMUL u zavisnosti od vlmul

vlmul		LMUL	VLMAX
0	0	1	$VLEN/SEW$
0	1	2	$2 * VLEN/SEW$
1	0	4	$4 * VLEN/SEW$
1	1	8	$8 * VLEN/SEW$

Vta (eng. *Vector Tail Agnostic*), **vma** (eng. *Vector Mask Agnostic*) i **vill** (eng. *Vector Type Illegal*) nisu podržani u ovoj implementaciji vektorske ekstenzije i vektorsko jezgro ih zanemaruje. Više o njima može se saznati u [4, poglavlje 3].

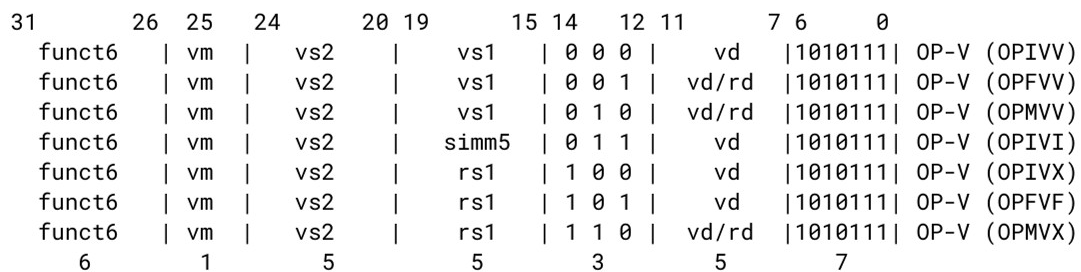
Unutar **vl** (eng. *Vector length*) **CSR** registra smeštena je vrednost koja određuje nad koliko elemenata vektorskog registra instrukcija treba da izvrši operaciju. Širina ovog registra određena je sa maksimalnim brojem elemenata koje jedna instrukcija sme da promeni i ta vrednost se dobija na sledeći način:

$$\log_2(\text{VLEN}/\text{SEW}_{\min} * \text{LMUL}_{\max}) \quad (3)$$

Na primer, ukoliko je $\text{SEW}_{\min} = 32$ i $\text{VLEN} = 1024$, širina *vl* registra bi bila: $\log_2(1024 / 32 * 8) = 9$.

3.2.2 Skup vektorskih instrukcija koje ekstenzija podržava

Nacrt vektorske ekstenzije [4] definiše 4 vrste instrukcija: vektorske *load* instrukcije, vektorske *store* instrukcije, aritmetičke instrukcije i AMO instrukcije. Format vektorskih aritmetičkih instrukcija je prikazan na sledećoj slici:



Slika 8. Formati vektorskih aritmetičkih instrukcija [4]

Sa slike se može videti da se one dele na 7 podskupova: OPIVV, OPFVV, OPMVV, OPIVI, OPIVX, OPFVF, OPMVX. Kom od prethodno navedenih podskupova aritmetička instrukcija pripada određuje grupa bita sa oznakom funct3 (biti 14:12 vektorske instrukcije). Pripadnost podskupu određuje dve stvari. Prvo, koji je tip instrukcije, odnosno da li je u pitanju integer, floating point ili konfiguraciona instrukcija. Drugo, *funct3* određuje da li se operacija izvodi između dva vektora, vektora i skalara, ili vektora i konstante (*eng. Immediate*). Sledeća tabela to ilustruje:

Tabela 4: Tip instrukcije u zavisnosti od funct3 polja

Funct3[2:0]			Tip instrukcije	Operandi
0	0	0	OPIVV	Vector-vector
0	0	1	OPFVV	Vector-vector
0	1	0	OPMVV	Vector-vector
0	1	1	OPIVI	Vector-immediate
1	0	0	OPIVX	vector-scalar
1	0	1	OPFVF	vector-scalar
1	1	0	OPMVX	vector-scalar
1	1	1	OPCFG	scalars-immediate

Integer instrukcije su podskupovi čija su prva 3 slova OPI ili OPM, instrukcije sa pokretnim zarezom (*eng. Floating point*) su podskupovi čija prva tri slova počinju sa OPF, dok konfiguracione instrukcije pripadaju OPCFG podskupu.

U nastavku su nabrojane sve aritmetičke instrukcije koje vektorska implementacija u ovom radu poseduje i njihova asemblerska predstava:

- Instrukcije koje pripadaju OPIVV podskupu:
 - Vektorsko sabiranje i oduzimanje (Vadd.vv, vsub.vv)
 - Vektorske logičke instrukcije (vxor.vv, vand.vv, vor.vv)
 - Vektorske instrukcije pomeranja (*eng. shift*) (Vsl.vv, vsrl.vv, vsra.vv)
 - Vektorske instrukcije poređenja (vmseq.vv, vmsne.vv, vmsltu.vv, vmslt.vv, vmsleu.vv, vmsle.vv)
 - Vektorske *min* instrukcije (vminu.vv, vmin.vv)
 - Vektorska *move* instrukcija (vmv.v.v)
 - Vektorska instrukcija spajanja (*eng. merge*) (Vmerge.vvm)
- Instrukcije koje pripadaju OPIVI podskupu:
 - Vektorsko sabiranje i oduzimanje (Vadd.vi, vsub.vi)
 - Vektorske logičke instrukcije (vxor.vi, vand.vi, vor.vi)
 - Vektorske instrukcije pomeranja (Vsl.vi, vsrl.vi, vsra.vi)
 - Vektorske instrukcije poređenja (vmseq.vi, vmsne.vi, vmsltu.vi, vmslt.vi, vmsleu.vi, vmsle.vi)
 - Vektorska *move* instrukcija (vmv.v.i vd)
 - Vektorska instrukcija spajanja (*eng. merge*) (Vmerge.vim)
- Instrukcije koje pripadaju OPIVX podskupu:
 - Vektorsko sabiranje i oduzimanje (Vadd.vx, vsub.vx)
 - Vektorske logičke instrukcije (vxor.vx, vand.vx, vor.vx)
 - Vektorske instrukcije pomeranja (Vsl.vx, vsrl.vx, vsra.vx)
 - Vektorske instrukcije poređenja (vmseq.vx, vmsne.vx, vmsltu.vx, vmslt.vx, vmsleu.vx, vmsle.vx)
 - Vektorske *min* instrukcije (vminu.vx, vmin.vx)
 - Vektorska *move* instrukcija (vmv.v.x vd)

- Vektorska instrukcija spajanja (*eng. merge*) (*Vmerge.vxm*)
- Instrukcije koje pripadaju OPMVV podskupu:
 - Instrukcije vektorskog množenja (*vmul.vv*, *vmulh.vv*, *vmulhu.vv*, *vmulhsu.vv*)
- Instrukcije koje pripadaju OPMVX podskupu:
 - Instrukcije vektorskog množenja (*vmul.vx*, *vmulh.vx*, *vmulhu.vx*, *vmulhsu.vx*).
- Instrukcije koje pripadaju OPCFG podskupu:
 - Instrukcije za modifikovanje CSR registara (*vsetvli*).

Pored prethodno navedenih aritmetičkih vektorskih instrukcija implementirane su dve vektorske **load** instrukcije i dve vektorske **store** instrukcije:

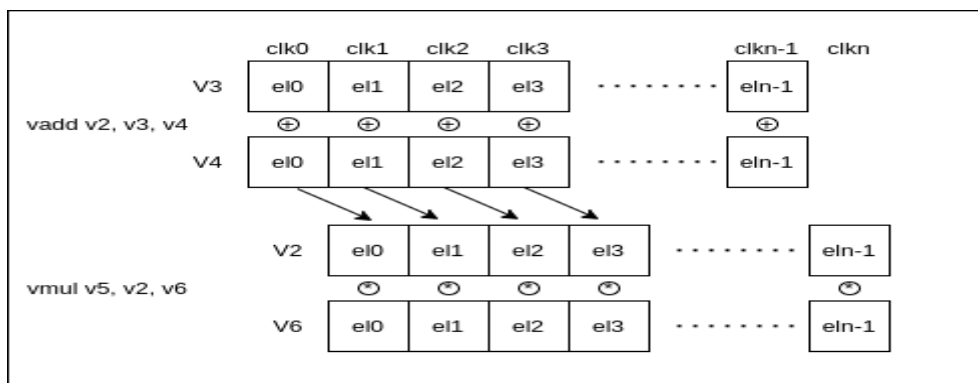
- Vektorska (*eng.unit strided*) **load**. U [4] je definisano više ovakvih instrukcija, ali samo jedna je implementirana i to *vlw.v*
- Vektorski (*eng.strided*) **load**. U [4] je definisano više ovakvih instrukcija, ali samo jedna je implementirana i to *vls.w.v*
- Vektorski (*eng.unit strided*) **store**. U [4] je definisano više ovakvih instrukcija, ali samo jedna je implementirana i to *vsw.v*
- Vektorski (*eng.strided*) **load**. U [4] je definisano više ovakvih instrukcija, ali samo jedna je implementirana i to *vss.w.v*.

3.2.3 Vektorsko maskiranje

Vektorsko maskiranje omogućuje da se određeni elementi vektorskog registra naznače kao neaktivni, što prouzrokuje da se ti elementi u destinacionim registrima neće modifikovati prilikom izvršavanja određene instrukcije. Da li će element vektorskog registra biti maskiran ili ne, određuju vrednosti unutar V0 vektorskog registra, specifično, LSB bit svakog elementa unutar V0. Ukoliko je LSB bit elementa u vektorskom registru V0 jednak 0₂, element na istom indeksu u destinacionom registru ne sme se modifikovati, u suprotnom sme. Vektorski procesor implementiran u ovom radu podržava vektorsko maskiranje.

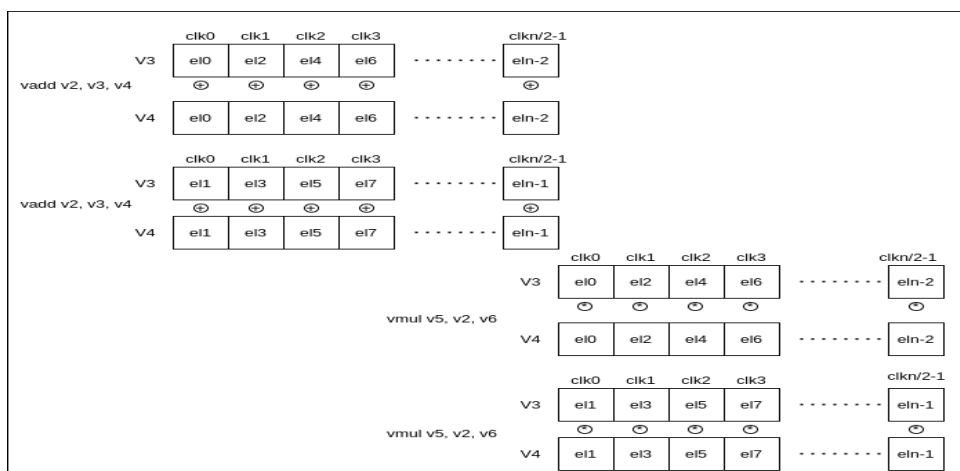
3.3 Mikroarhitektura vektorskog jezgra

Tradicionalno su vektorski procesori optimizovani za procesiranje dugačkih vektora i da bi povećali performanse oni se oslanjaju na protočnu obradu, gde više instrukcija može istovremeno da se izvršava na više različitih funkcionalnih jedinica (na primer na sabiraču i množaču). Prilikom paralelnog izvršavanja, instrukcije mogu biti međusobno zavisne i da bi se to rešilo koristi se metoda ulančavanja (*eng. Chaining*), prikazana na sledećoj slici:



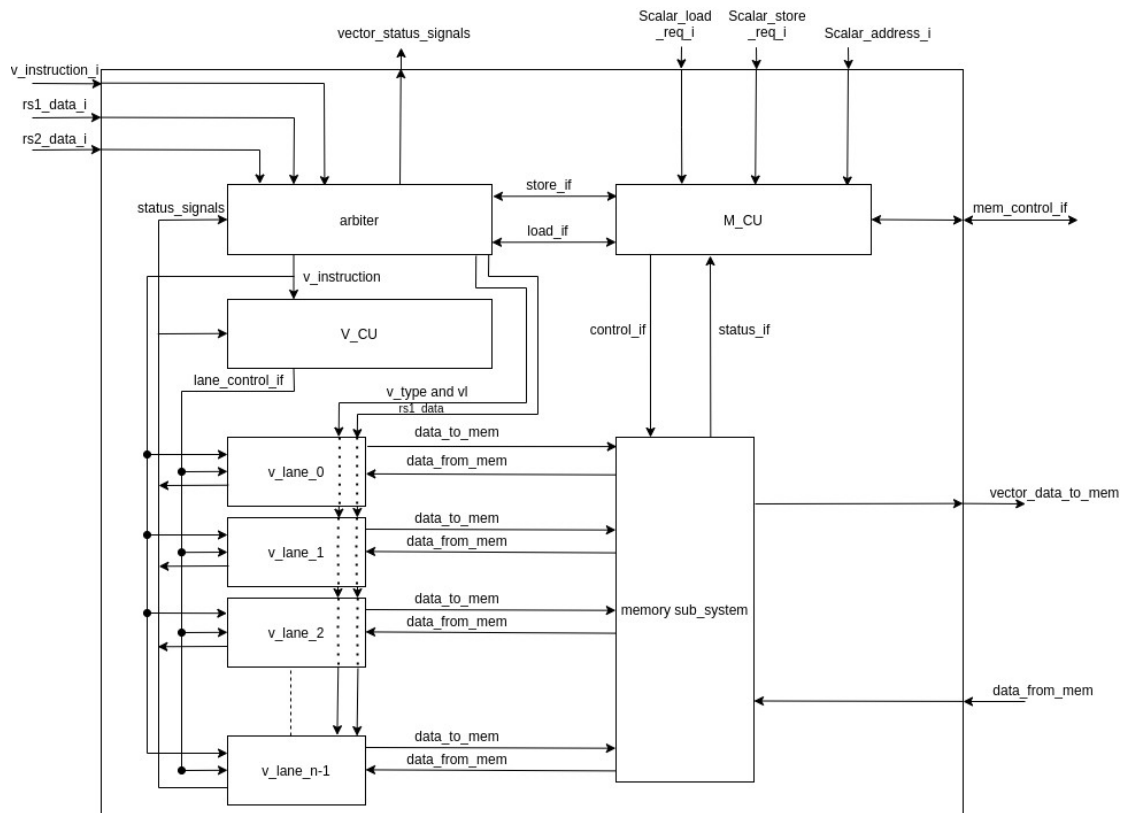
Slika 9: Metoda ulančavanja

U prethodnom primeru donja instrukcija (*vmul v5, v2, v6*) kreće sa svojim izvršanjem jedan takt kasnije. Iz tog razloga gornja instrukcija ima dovoljno vremena da završi proračun elemenata na nultom indeksu i da rezultat tog izvršavanja prosledi donjoj instrukciji. No, ulančavanje dovodi do povećanja kompleksosti vektorske registarke banke, jer je neophodno za svaku funkcionalnu jedinicu uvesti dodatni port za čitanje (sa kog zavisna instrukcija čita operand) i dodatni port za upis (preko kog zavisna instrukcija upisuje svoj rezultat). Ovakva vrsta vektorske registarske banke se jako teško može efikasno implementirati na FPGA platformama, te je iz tog razloga iskorišćen drugi način za povećanje performansi, prikazan na sledećoj slici:



Slika 10: Izvršavanje vektorske instrukcije na više različiti vektorskih linija

Osnovna ideja je da se izvršavanje jedne instrukcije podeli na više vektorskih linija. Na prethodnoj slici ilustrovan je primer gde se jedna vektorska instrukcija izvršava na dve vektorske linije, pri čemu jedna linija izvršava operacije samo nad elementima sa parnim indeksima, a druga linija izvršava operacije nad elementima sa neparnim indeksima. Na ovaj način ukupan broj ciklusa neophodan za izvršenje jedne instrukcije je onoliko puta manji koliko ima uposlenih vektorskih linija.



Slika 11: Mikroarhitektura vektorsko jezgra

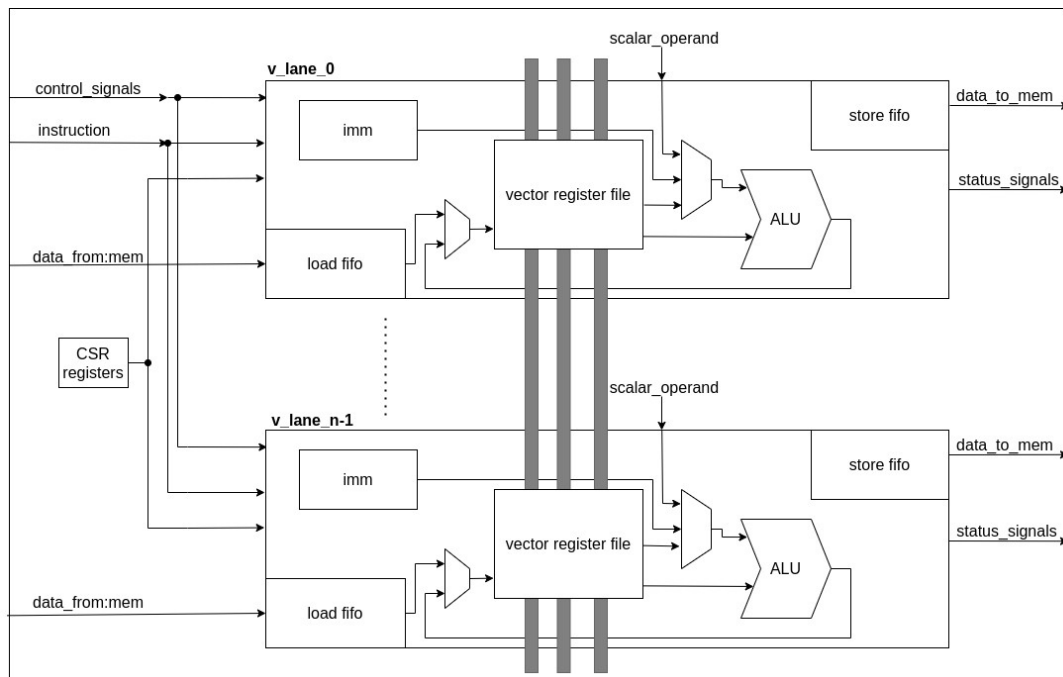
Na slici 11 je prikazana mikroarhitektura vektorskog jezgra koje implementira prethodno opisano i u narednim sekcijama su opisani svi blokovi koji ga čine. Njegov interfejs se sastoji iz sledećih portova:

- *v_instruction_i* – Instrukcija koju skalarno jezgro prosleđuje vektorskom
- *rs1_i* i *rs2_i* – Uz instrukciju, vektorskom jezgru se prosleđuju i podaci iz skalarne registarske banke, jer određene vektorske instrukcije rade sa skalarnim operandima.
- *Vector_status_signals* – U ovu grupu spadaju signali koji omogućavaju sinhronizaciju u radu između skalarnog i vektorskog jezgra, tu spadaju: *vector_stall_o* signal, *all_v_stores_executed* i *all_v_loads_executed*, koji su objašnjeni u sekciji 3.1.

- *Scalar_load_req_i*, *scalar_store_reg_i* i *scalar_address* su signali koje dolaze iz skalarnog jezgra i njihov smisao biće objašnjen u sekciji 3.3.5.
- *vector_data_to_mem* je izlazni port preko koga se podaci šalju u memoriju za podatke.
- *data_from_mem* je ulazni port preko koga vektorsko jezgro prima podatke iz memorije.
- *mem_control_if* je interfejs preko koga memorijska kontrolna jedinica (*M_CU* slika 10) kontroliše način komunikacije sa memorijom

3.3.1 Vektorske linije (eng. *Vector lanes*)

Na prethodnoj slici to su blokovi označeni sa *v_lane_0* do *v_lane_n-1*, što naznačava da je broj linija u vektorskom jezgru parametrizovan (u trenutnoj implementaciji broj linija mora biti stepen broja 2). Svaka vektorska linija sadrži kopiju funkcionalnih jedinica, deo vektorske registarske banke (polu ukoliko je broj linija 2, četvrtinu ukoliko je broj linija 4, itd), *load* i *store* fifo memorije i mrežu za rutiranje. Takođe, svaka vektorska linija ima u potpunosti isti interfejs i kontrolisana je od strane istih kontrolnih signala. U nastavku je prikazana detaljnija šema vektorskih linija:



Slika 12: Prikaz najbitnijih blokova unutar vektorskih linija

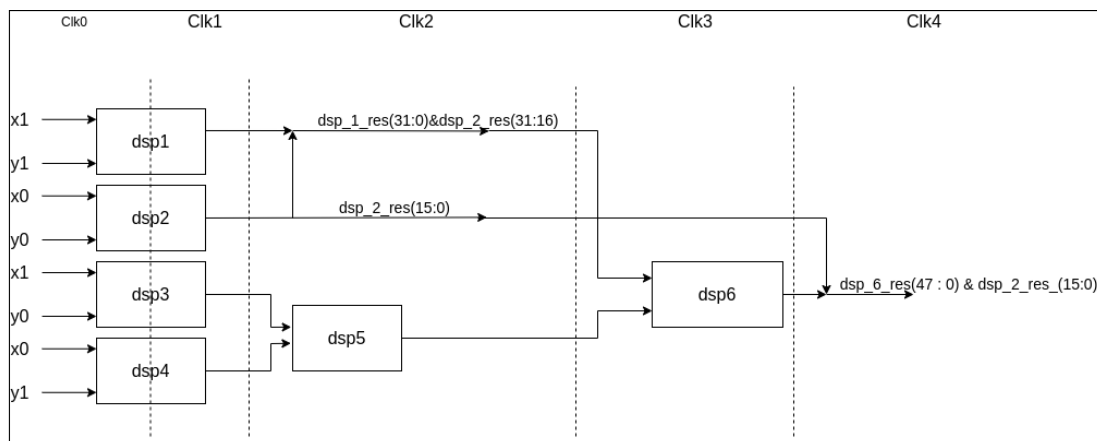
Vektorska registarska banka (VRF - vector register file) je distribuirana između vektorskih linija (označeno sivim linijama na slici 12) i na ovaj način izbegnuta je tradicionalna arhitektura jedne centralizovane banke [5] koja je zahtevala previše portova za upis i čitanje. Distribucija je realizovana tako da svaka

banka unutar vektorskih linija ima 32 vektorska registra sa $VECTOR_LENGTH/NUM_OF_LANES$ elemenata. $VECTOR_LENGTH$ je parametar koji određuje koliko elemenata ima u jednom vektoru, dok je NUM_OF_LANES parametar koji određuje koliko ima vektorskih linija. Na primer, ukoliko bi broj linija bio $NUM_OF_LANES = 2$ i $VECTOR_LENGTH = 32$, vektori unutar banki svake od vektorskih linija bi imali po 16 elemenata. Broj vektorskih linija ne sme biti veći od $VECTOR_LENGTH$, jer u tom slučaju bi postojale linije koje nemaju ni jedan element. Ovakav način particionisanja omogućuje da se prilikom realizacija VRF modula koristi blok RAM [7] koji je elementarna ćelija na FPGA platformama kompanije *Xilinx* [6]. No, to dolazi sa cenom. Da bi VRF mogao da obezbedi 2 porta za čitanje (istovremeno čitanje dva elementa u jednom taktu) i jedan port za upis neophodno je duplicirati blok RAM ćelije, jer jedan BRAM može da obezbedi samo jedan port za upis i jedan port za čitanje. Takođe, pošto je minimalna veličina jedne BRAM ćelije 2KB, ukoliko bi se VRF previše particionisao, došlo bi do situacije da se jedan deo BRAM ćelije uopšte ne koristi. Na primer ukoliko je $VECTOR_LENGTH = 32$ i $NUM_OF_LANES = 4$ broj elemenata unutar pojedinačnog VRF bloka je: $32 / 4 * 32 = 256$. Kako je svaki element širine 4 bajta, to znači da je za njihovo skladištenje potrebno 1024 bajta (1KB). No, čak i sa ovim nedostacima, particionisani VRF je „jeftiniji“ prilikom implementacije nego centralizovani.

Aritmetičko logička jedinica (ALU) omogućava izvršavanje svih operacija koje skup instrukcija naveden u sekciji 3.2.2 zahteva. Sve operacije se izvršavaju u istom taktu sem operacija množenja kojima treba 4 takta kako bi se izvršile (zbog 4 nivoa protočne obrade). Da bi se realizovao što efikasniji množač (rad na što većoj frekvenciji) i kako bi se što više iskoristili resursi na FPGA platformi, korišćene su DSP ćelije [8]. Kako ove ćelije unutar sebe implementiraju množač 18x25, jedna DSP ćelija nije dovoljna (zbog 32-bitne arhitekture neophodan je množač 32x32), ali ukoliko se iskoristi sledeća zavisnost, povezivanjem više DSP ćelija može da se realizuje množač 32x32:

$$X*Y = 2^{2k}*X1*Y1 + 2^k*(X1Y0 + X0Y1) + X0Y0 \quad (4)$$

X i Y su 32-bitne vrednosti, X1 predstavlja gornjih 16 bita promenljive X, a X0 donjih 16 bita. Isto važi i za Y, Y0 i Y1. Na osnovu prethodne jednačine se zaključuje da se 32-bitno množenje može rastaviti na više 16-bitnih množenja i ta činjenica je iskorišćena prilikom realizovanja množača pomoću DSP ćelija. U nastavku je prikazan blok dijagram jednog takvog sistema:



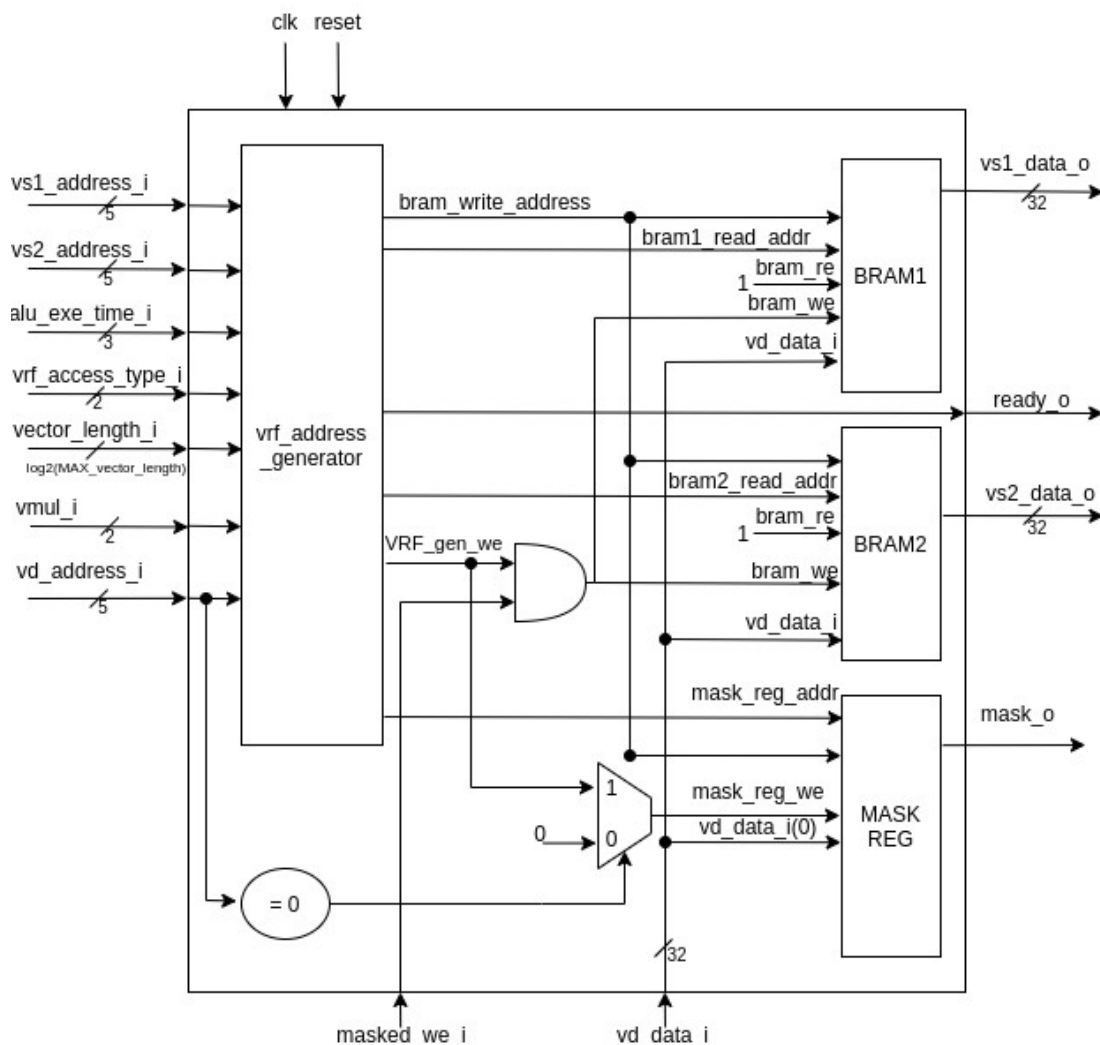
Slika 13: Realizacija množača pomoću više DSP ćelija

Sa slike se može videti da je neophodno 6 DSP ćelija. DSP1, DSP2, DSP3, DSP4 izvršavaju četiri množenja koja postoje u (4) i za to im je potrebno 2 takta, dok ostala dva izvršavaju operacije sabiranja za šta im treba 1 takt. Operacije pomeranja (*eng. shift*) su realizovane pravoremenim konkatanacijama rezultata određenih DSP ćelija.

Uloga **Load i Store FIFO memorija** (slika 12) je da budu posrednici između memorije sa podacima i VRF bloka. Prilikom izvršavanja *store* instrukcija nakon što se podaci smeste iz VRF bloka u *store FIFO*, odmah može da se krene sa izvršavanjem naredne instrukcije. *Load* instrukcije funkcionišu na sličan način, odnosno dok se podaci smeštaju u *load FIFO*, naredna instrukcija, ukoliko ne zavisi od prethodne *load* instrukcije, može da se izvršava. Nakon što se podaci iz memorije sa podacima smeste u *load FIFO*, mikro instrukcijom se oni odatle prebacuju u VRF. Dok se svi elementi ne prabace, vektorska linija je zaustavljena i ne mogu da se izvršavaju druge instrukcije. Takođe, nije moguće izvršavati *load* i *store* instrukcije paralelno, odnosno ukoliko jezgro smešta podatke iz memorije u *load FIFO*, ono nije u stanju da vrši smeštanje podataka iz *store FIFO* bloka u memoriju sa podacima, i obrnuto. Na ovaj način rešen je problem konzistentnosti memorije. Moduli koji su iskorišćeni kao *FIFO* memorije su već gotovi blokovi kompanije *Xilinx* [6] i njihova specifikacije može se pronaći u [7, poglavlje 2].

3.3.2 Vektorska registarska banka (VRF)

Kao što je prethodno pomenuto VRF je komponenta vektorske linije koja skladišti 32 vektorska registra, pri čemu svaki unutar sebe sadrži *VECTOR_LENGTH* broj elemenata, širine 32 bita. Na sledećoj slici je prikazan VRF blok i njegov interfejs:



Slika 14: Vektorska registarska banka

Interfejs VRF modula se sastoji iz sledećih portova:

- *clk* i *reset* su sinhronizacioni signali.
- *vs1_address_i* i *vs1_data_o* predstavljaju interfejs prvog porta za čitanje iz VRF modula. Vrednost na *Vs1_address_i* portu je adresa vektorskog registra iz kog instrukcija želi da čita, dok *vs1_data_o* predstavlja port na kome će se podaci tog vektorskog registra pojaviti.

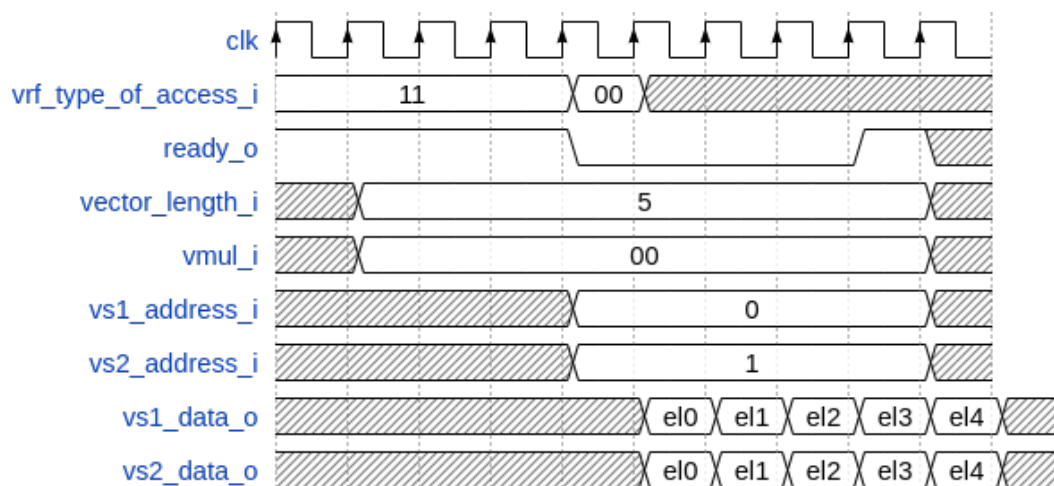
- *vs2_address_i* i *vs2_data_o* predstavljaju interfejs drugo porta za čitanje iz VRF modula. Za njih važi isto što i za prethodne.
- *vd_address_i* i *vd_data_i* predstavljaju interfejs za upis podataka u VRF modul. *Vd_address_i* je adresa vektorskog registra u koji se upisuje podatak, a *vd_data_i* je port preko koga se podaci prosleđuju VRF modulu.
- *vector_length_i* je ulazni port koji predstavlja stanje *vl* CSR registra. Na osnovu njega VRF modul zna koliko elemenata da pročita.
- *Vmul_i* je ulazni port koji predstavlja stanje *vtype* CSR registra, specifično, stanje njegova dva LSB bita, koji određuju da li se više registara spaja u jedan veliki.
- *Vrf_access_type_i* je ulazni port koji određuje način rada VRF modula. Postoje četiri vrste: samo čitanje, samo upis, čitanje i upis, nema zahteva za pristupom. U nastavku će biti objašnjen način rada sa VRF modulom kako bi se svaki od pristupa izvršio.
- *alu_exe_time* je ulazni port koji govori VRF modulu koliko taktova je neophodno ALU modulu da izvrši jednu operaciju. Ova informacija je bitna kada se vrši istovremeni upis i čitanje u VRF.
- *mask_o* je port na kome se pojavljuju biti koji omogućavaju maskiranje (opisano u sekciji 3.2.3). Na ovom portu se pojavljuju LSB biti svakog elementa vektora *V0*, svakog takta po jedan (počevši od indeksa 0), od trenutka kada se *vrf_access_type_i* postavi na neki od načina pristupa.
- *masked_we_i* je ulazni port dozvole upisa u VRF modul. Ukoliko je on 0₂ upis nije dozvoljen.
- *ready_o* – predstavlja statusni signal koji govori da li je VRF spreman za naredni pristup ili ne. Ukoliko je postavljen na logičku 1₂ znači da je spreman, u suprotnom nije. Sve dokle je on na logičkoj 0₂ ne sme se započinjati novi pristup jer može da dođe do neočekivanih rezultata.

Slika 14 pored interfejsa ilustruje unutrašnjost VRF modula. Kao što je prethodno pomenuto registarska banka se sastoji od dve BRAM ćelije koje unutar sebe imaju potpuno isti sadržaj, no, pored njih su neophodna još dva bloka: *vrf_address_generator* i *mask_reg*. Na prvi su direktno povezani neki od portova VRF modula i on na osnovu njih generiše adrese sa kojih treba da se pročitaju podaci iz BRAM blokova, odnosno adrese na koje treba da se upišu podaci. Takođe pored toga, on generiše signal *VRF_gen_we* koji u kombinaciji sa *masked_we* generiše dozvolu upisa u *BRAM1* i *BRAM2*. *Mask_reg* je registarska banka unutar koje se nalaze biti koji predstavljaju maske, odnosno unutar nje se nalazi kopija LSB bita svih elemenata

registra V0. Ti biti se koriste jedino ukoliko instrukcija zahteva rad sa maskama (*vm* polje instrukcije je 0₂).

U nastavku je opisana procedura koja mora da se ispoštuje prilikom svakog od pristupa VRF modulu. Za pristup „samo čitanje“ neophodno je uraditi sledeće:

1. Proveriti da li je *ready_o* = 1₂. Ako jeste, u narednom ciklusu uraditi sledeće korake.
2. Postaviti *vs1_address_i* i *vs2_address_i* na adrese vektorskih registara koji treba da se pročitaju.
3. Postaviti *vmul_i*.
 - *Vmul* = 0 – moguće čitanje samo jednog registra.
 - *Vmul* = 1 - moguće čitanje iz 2 vektorskih registra.
 - *Vmul* = 2 - moguće čitanje iz 4 vektorskih registra..
 - *Vmul* = 3 - moguće čitanje iz 8 vektorskih registra..
4. Postaviti *vector_length_i* broj elemenata koji treba da se pročita
5. Postaviti *vrf_type_of_access_i* na 10₂.
6. Kada je *vrf_type_of_access_i* postavljen, svakog takta pojaviće se novi podatak na *vs1_data_o* i *vs2_data_o* sve dok se ne pročita *vector_length_i* broj elemenata
7. Stanje na ulaznim portovima se ne sme menjati sve dok je *ready_o* = 0₂, a kada on skoči na jedinicu, u narednom taktu mogu da se izvrši naredni pristup. Primer čitanja je prikazan na sledećoj slici:

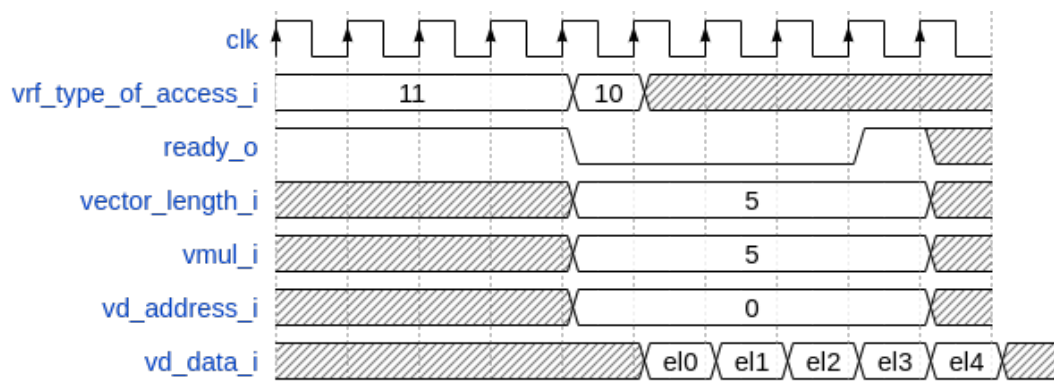


Slika 15: Slika koja prikazuje talasne oblike prilikom čitanja podataka iz VRF modula.

Prilikom izvršavanja operacije **čitanja** neophodno je uraditi sledeće:

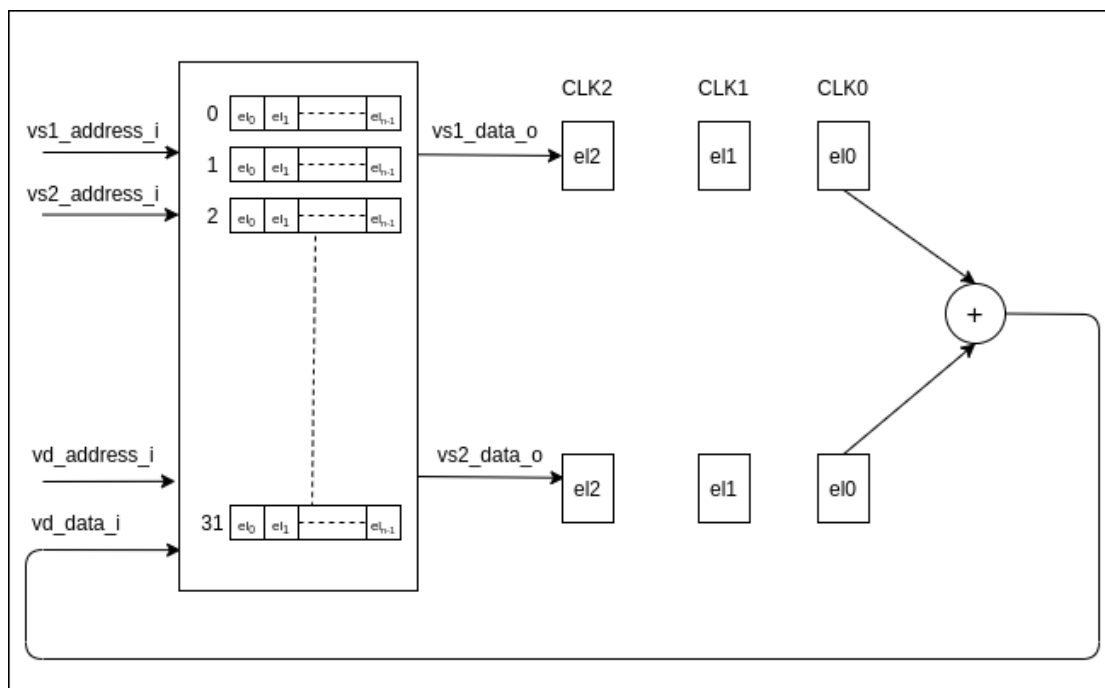
1. Proveriti da li je $ready_o = 1_2$. Ako jeste, u narednom ciklusu uraditi sledeće korake.
2. Postaviti **vd_address_i** na adresu vektorskog registra u koji treba da se upiše.
3. Postaviti **vmul_i**.
 1. Vmul = 0 – moguć upis u samo jedan vektorski registar.
 2. Vmul = 1 - moguće upis u 2 vektorska registra.
 3. Vmul = 2 - moguće upis u 4 vektorska registra.
 4. Vmul = 3 - moguće upis u 8 vektorska registra.
4. Postaviti **vector_length_i** na broj elemenata koliko treba da se upiše.
5. Postaviti **vrf_type_of_access_i** na 01_2 .
6. U narednom taktu postaviti podatak koji je potrebno upisati i svakog narednog takta postavljati podatke sve dok se ne upiše *vector_length* broj podataka.
7. Jedan takt pre nego što se poslednji element upiše, *ready_o* će biti postavljen na 1_2 .

Sledeća sliku ilustruje prethodno opisano:



Slika 16: Primer operacije upisa u VRF modul

Ukoliko je *masked_we_i* ulazni port na 0_2 upis nije dozvoljen.



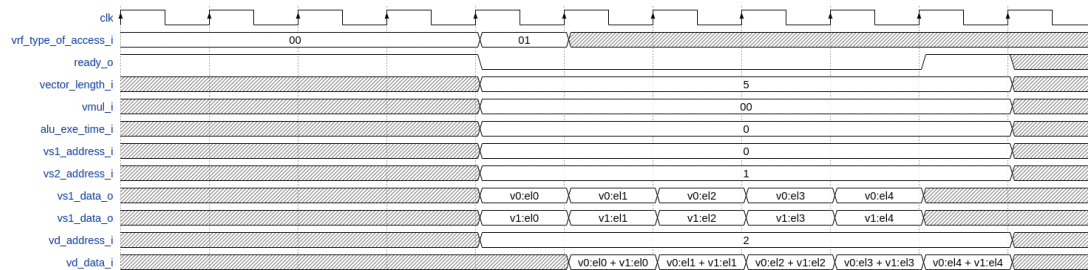
Slika 17: Operacija istovremenog čitanja i upisa u VRF modul

Poslednja vid pristupa je istovremen upis i čitanje. Ova opcija je bitna u situacijama kada se izvršavaju aritmetičke instrukcije, jer tada se jedan par elemenata iz vektorskih registara pročita, nad njima se izvrši određena operacija i oni se upišu nazad u VRF modul. Slika 17 to ilustruje. Prilikom izvršavanja ove operacije neophodno je uraditi sledeće:

1. Proveriti da li je $ready_o = 1_2$. Ako jeste, u narednom ciklusu uraditi sledeće korake.
2. Postaviti **vd_address_i** na adresu vektorskog registra u koji treba da se upiše.
3. Postaviti **vs1_address_i** i **vs2_address_i** na adrese vektorskih registara koji treba da se pročitaju.
4. Postaviti **vmul_i**.
 1. Vmul = 0 – moguć upis u samo jedan vektorski regisar.
 2. Vmul = 1 - moguće upis u 2 vektorska registra.
 3. Vmul = 2 - moguće upis u 4 vektorska registra.
 4. Vmul = 3 - moguće upis u 8 vektorska registra.
5. Postaviti **vector_length_i** na broj elemenata koliko treba da se upiše-pročita.
6. Postaviti **vrf_type_of_access_i** na 00_2 .
7. Postaviti **alu_exe_time_i** na broj koji predstavlja koliko taktova je potrebno aritmetičkoj jedinici da izbací validan rezultat, jer tada kreće proces upisa.

8. Nakon $alu_exe_time_i + 1$ taktova postaviti podatak koji je potrebno upisati. I svakog narednog takta postavljati podatke sve dok se ne upiše *vector_length* broj podataka.
9. Jedan takt pre nego što se poslednji element upiše, *ready_o* će biti postavljen na 1₂.

Sledeća slika ilustruje prethodno opisano:



Slika 18: Primer operacije istovremenog upisa i čitanja

Ukoliko je *vrf_type_of_access_i* postavljen na vrednost 11₂ to znači da nijedna operacija ne treba da izvrši i *ready_o* će automatski biti postavljen na 1₂.

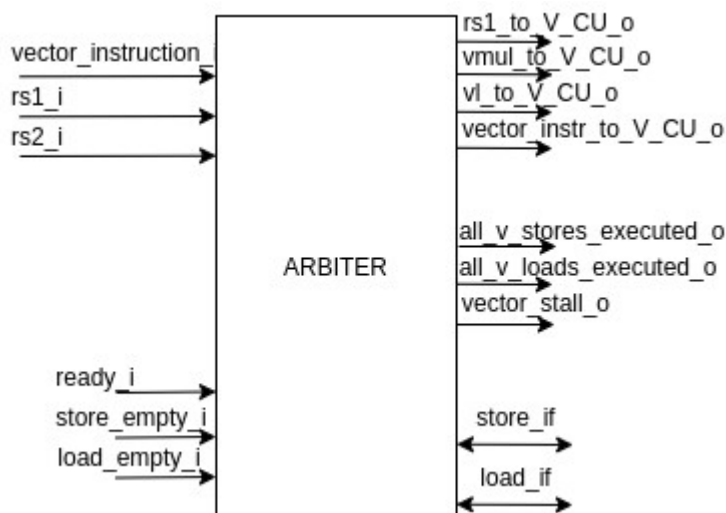
3.3.3 Vektorska kontrolna jedinica (V_CU)

Uloga ovog bloka je da na osnovu vektorske instrukcije koju dobija od *Arbiter* komponente, generiše kontrolne signale koji upravljaju vektorskim linijama. Sam blok je jednostavan dekodler koji posmatra polja prihvaćene instrukcije i na osnovu njih zaključuje koju operaciju vektorske linije treba da obave. Kontrolni signali koje generiše V_CU su sledeći:

- ***vrf_type_of_access_o*** – vrsta pristupa vektorskoj registarskoj banci (objašnjeno u 3.3.2).
- ***immediate_sign_o*** – ukoliko je 1_2 , *immediate* polje instrukcije se posmatra kao neoznačeno(*eng. unsigned*), u suprotnom kao označeno(*eng. signed*)
- ***alu_op_o*** – određuje koju operaciju aritmetičko logička jedinica treba da obavi
- ***mem_to_vrf_o*** – određuje šta se upisuje u VRF, rezultat ALU jedinice, podaci iz *load FIFO* memorije, vrednost sa „a“ ulaza alu jedinice ili stanje sa *vs2_data_o* porta VRF modula.
- ***store_fifo_we_o*** – signal dozvole upisa u *store FIFO* memoriju (generiše se jedino kada se izvršava *store* instrukcija).
- ***alu_src_a_o*** – određuje šta se propušta na „a“ ulat ALU jedinice, vrednost sa *vs1_data_o* porta VRF modula, vrednost na skalarnom portu ili konstanta .
- ***type_of_masking_o*** – U procesoru postoje dva načina maskiranja. Standardni, gde se ažuriraju samo oni elementi kod kojih je *mask* bit logička 1_2 , i nestadardni (kod merge instrukcija) gde *mask* bit ne treba da se posmatra iako je *vm* polje vektorske instrukcije logička 1_2 . Za prvi slučaj *type_of_masking_o* kontrolni signal je 0_2 .
- ***alu_exe_time_o*** – određuje koliko vremena je potrebno da bi se izvršila jedna ALU operacija (na primer, ukoliko je operaciji potreban 1 takt, njegova vrednost je 1).
- ***vs1_addr_src_o*** - Određuje šta će biti prosleđeno na *vs1_addr_i* port VRF modula. Ukoliko je 0_2 , prosleđuje se *vs1* (biti 19 do 15) polje vektorske instrukcije, u suprotnom *vd* polje (biti 11 do 7) vektorsko instrukcije.
- ***load_fifo_re_o*** – signal dozvole čitanja iz *load FIFO* memorije (generiše se jedino ukoliko *Arbiter* prosledi vektorsku *load* instrukciju).

3.3.4 Arbiter

Pored vektorske registrarske banke i vektorskih linija, *arbiter* je treća najbitnija komponenta koja čini vektorsko jezgro. Njegova uloga je da prihvata instrukcije od skalarnog jezgra i da u zavisnosti od tipa prihvaćene instrukcije prosledi određene informacije *V_CU* modulu, *M_CU* modulu, vektorskim „linijama“, ili da ažirira stanja unutrašnjih registara. Sledeća slika prikazuje interfejs *Arbiter* modula:



Slika 19: Interfejs *Arbiter* modula

Pošto arbiter mora da komunicira sa nekoliko komponenti, njegov interfejs je podeljen na 6 skupova:

1. Preko 32-bitnih *instruction*, *rs1_i* i *rs2_i* ulaznih portova *Arbiter* modul dobija vektorsku instrukciju i podatke iz skalarnih registara.
2. *Ready_i*, *store_empty* i *load_empty* su jednobitni statusni ulazni portovi preko kojih *Arbiter* dobija informaciju da li su vektorske linije spremne za narednu instrukciju, da li su *store FIFO* memorije unutar vektorskih linija prazne i da li su *load FIFO* memorije unutar vektorskih linija prazne, respektivno.
3. *Rs1_to_V_CU_o*, *vmul_to_V_CU_o*, *vl_to_V_CU_o* i *vector_instr_to_V_CU_o* su 32-bitni izlazni portovi preko kojih *Arbiter* prosleđuje vektorskim linijama i *V_CU* modulu vrednost iz skalarnog registra, vrednost *vmul* iz *vtype* CSR registra, vrednosti iz *vl* CSR registra i vektorsku instrukciju respektivno.
4. *All_v_store_executed*, *All_v_loads_executed* i *vector_stall* su izlazni jednobitni statusni portovi preko kojih se skalarnom jezgru prosleđuje stanje vektorskog jezgra (sekcija 3.1).

5. *Load_if* je interfejs preko koga *Arbiter* šalje informacije neophodne *M_CU* modulu kako bi podatke iz memorije sa podacima smestio u *load FIFO* memoriju unutar vektorskih linija (detaljno objašnjenje u narednoj sekciji).
6. *Store_if* je interfejs preko koga *Arbiter* šalje informacije neophodne *M_CU* modulu kako bi podatke iz *store FIFO* memorije unutar vektorskih linija smestio u memoriju sa podacima (detaljno objašnjenje u narednoj sekciji).

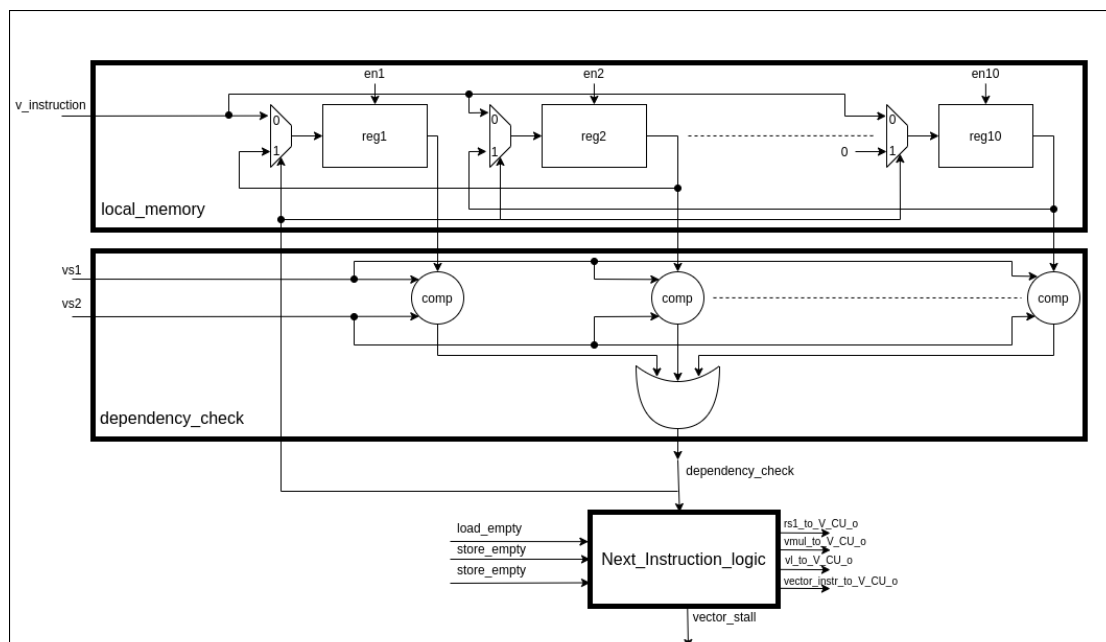
Ukoliko je *Arbiter* modul prihvatio **konfiguracionu** instrukciju, on je zadužen da modifikuje vrednosti unutar *vl_reg* i *vtype_reg* blokova, koji predstavljaju CSR registre objašnjene u sekciji 3.2.1. Ova dva registra su smeštena unutar *Arbiter* modula, jer su njihove vrednosti neophodne vektorskim „linijama“ i *M_CU* modulu, te ih je odatle najlakše prosleđivati.

Izvršavanje **Load** instrukcije je dosta kompleksnije nego izvršavanje bilo koje druge instrukcije. U sekciji 3.3.1 je pomenuto da unutar svake vektorske linije postoji *load FIFO* memorija i da je na taj način omogućeno da, dok se podaci koje zahteva *load* instrukcija preuzimaju iz memorije, izvršava neka druga instrukcija. Time su povećane performanse, ali uz cenu dodatne logike, jer vektorsko jezgro mora da bude u stanju da, ukoliko se pojavi instrukcija koja zavisi od rezultata izvršavanja *load* instrukcije, zaustavi dalji rad sve dok se zavisnost ne reši. Ta dodatna logika se nalazi u *Arbiter* modulu i prikazana je na slici 20:

- *Local_memory* blok – U njega se smeštaju pristigle *load* instrukcije uz vrednosti *vl_reg* i *vtype_reg* registara u trenutku prihvata. Vektorsku instrukciju je neophodno sačuvati jer se poređenjem njenog *vd* polja (biti 11 do 7) sa *vs2* i *vs1* poljima (biti 25 do 20 i 19 do 15 respektivno) novo pristiglih instrukcija, proverava zavisnost. Takođe, *load* instrukcija je neophodna jer ukoliko zavisnost postoji, ona mora da se prosledi *V_CU* komponenti zajedno sa *vl_reg* i *vtype_reg* vrednostima kako bi se podaci iz *load FIFO* memorije smestili u VRF. Razlog zašto se *vl_reg* i *vtype_reg* čuvaju je taj što se u međuvremenu mogla promeniti njihova vrednost, a *V_CU* modulu je neophodno proslediti one vrednosti koje su bile u tim registrima u trenutku kada je vektorska *load* instrukcija pristigla. Na slici 20 je prikazano da se lokalna memorija sastoji od 10 registara, pri čemu je svaki registar širok dovoljno da u njega stane konkatanirana vrednost vektorske instrukcije, *vl* registra i *vmul* vrednosti iz *vtype* registra. Ovo znači da unutar lokalne memorije može da stane 10 *load* instrukcija i ukoliko pristigne 11. vektorski procesor treba da zaustavi dalje prihvatanje instrukcija dok se prostor u lokalnoj memoriji ne oslobodi, odnosno dok se jedna *load* instrukcija ne izvrši. Podaci se dodaju u registre lokalne memorije redom, počevši od registra *reg1* pa do registra *r10*, dok se prilikom izbacivanja podatka (ukoliko

se otkrije zavisnost) sadržaj svih registara pomera u levo, odnosno reg1 dobija sadržaj od reg2, reg2 sadržaj od reg3, itd.

- *Dependency_check* blok – On vrši proveru zavisnosti tako što poređi polja *vs1* i *vs2* pristigle instrukcije sa *vd* poljima *load* instrukcija koje se nalaze unutar *local_memory* bloka. Sa slike 20 se može videti da *dependency_check* blok ima onoliko komparatora koliko ima registara unutar *local_memory* bloka, pri čemu je svaki registar (reg1 – reg10) povezan sa tačno jednim komparatorom. Ovo je urađeno kako bi prihvatom nove instrukcije mogla da se proveri njena zavisnost od svih *load* instrukcija koje se nalaze u *local_memory* bloku.
- *Next_instr_logic* blok - Njegova osnovna uloga je da, na osnovu statusnih signala koji potiču iz vektorskih linija, proverava da li su one spremne za izvršenje naredne instrukcije i ukoliko jesu da u narednom taktu prosledi instrukciju koju treba da izvrše. Takođe, ukoliko postoji zavisnost, ovaj blok zaustavlja dalje izvršavanje instrukcija (*vector_stall_o* = 1₂) i prosleđuje vektorskim linijama i V_CU modulu podatak iz registra reg1 *local_memory* modula, kako bi se podaci iz *load FIFO* memorija, unutar vektorskih linija, prebacili u VRF. Naravno, ovo može da se uradi jedino ukoliko *load FIFO* memorije nisu prazne (*load_fifo_empty* = 0₂) i ukoliko su se sve prethodne *store* instrukcije završile (*store_fifo_empty* = 1₂).



Slika 20: Provera zavisnosti pristigle instrukcije i load instrukcija u lokalnoj memoriji

Pored provere zavisnosti prilikom prihvata *load* instrukcije, *Arbiter* je takođe zadužen da prosledi *M_CU* modulu informacije neophodne za dobavljanje podataka iz memorije. Te informacije su:

- Bazna adresa – to je vrednost koju arbitar dobija iz skalarnog registra preko *rs1_i* ulaznog porta i ona predstavlja adresu od koje *M_CU* modul treba da počne da preuzima podatke.
- *Offset* – Ukoliko je prihvaćena *non-strided load* instrukcija offset ima vrednost 4 , a ukoliko je prihvaćena *strided load* instrukcija, offset je vrednost na *rs2_i* portu.
- Broj elemenata koji treba da se ekstrahuje iz memorije sa podacima (vrednost unutar *vl* registra).

Ove podatke *Arbiter* postavlja na *load_if* interfejs i poštujući odgovarajući dogovor (*eng.handshake*) *M_CU* modul će ih preuzeti, ukoliko nije zauzet izvršavanjem neke druge *load* ili *store* instrukcije. Ukoliko jeste, da bi se povećale performanse, arbitar skladišti neophodne informacije unutar dva *FIFO* modula i nastavlja sa radom. U prvi *FIFO* modul smeštaju se konkatanirane vrednosti sa portova *rs1_i* i *rs2_i*, dok se u drugi smešta vrednost *vl* registra. U obe *FIFO* memorije podaci se smeštaju u trenutku prihvata *load* instrukcije. Kada *M_CU* modul bude spreman, on će preuzeti ove podatke i izvršiti smeštanje podataka iz memorije u *load FIFO* memorije unutar vektorskih linija. *FIFO* memorije su neophodne jer više *load* instrukcija može da čeka da *M_CU* bude spreman da ih obradi i on će to raditi onim redom kojim su instrukcije pristizale.

Ukoliko arbitar prihvati *store* instrukciju, *Next_instruction_logic* će proslediti neophodne informacije na interfejs 3 Arbitra, ukoliko ne postoji zavisnost od neizvršenih *load* instrukcija i ukoliko su vektorske linije spremne za izvršenje naredne instrukcije. Takođe, kao i kod *load* instrukcija, arbitar je zadužen da prosledi *M_CU* modulu neophodne informacije kako bi prebacio podatke iz *store FIFO* memorija unutar vektorskih linija u memoriju za podatke. Te informacije su:

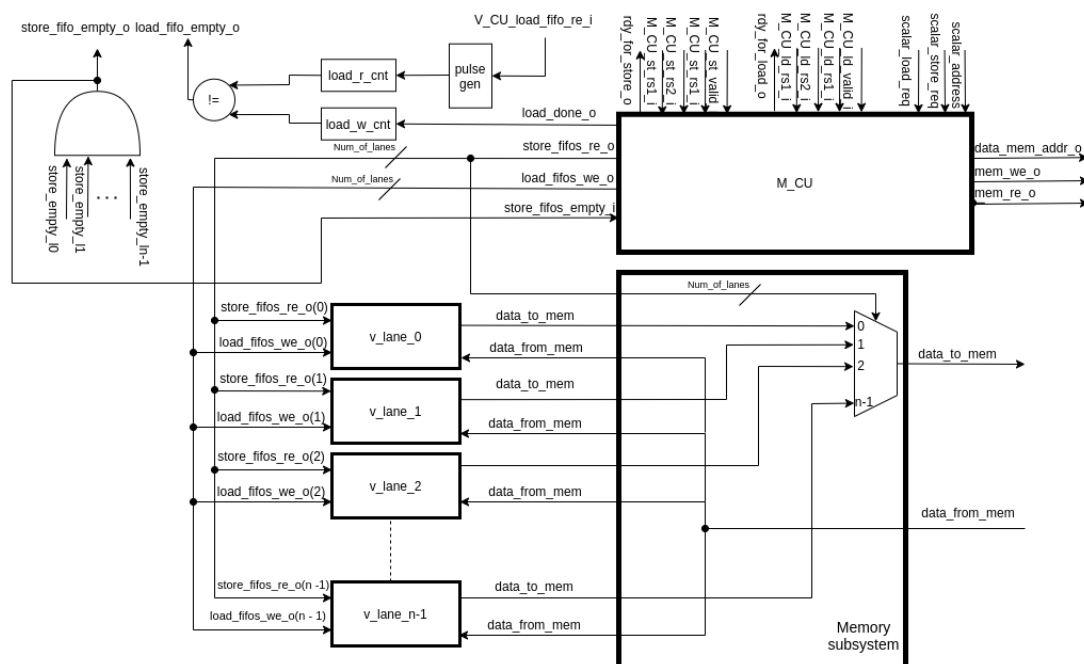
- Bazna adresa – to je vrednost koju arbitar dobija iz skalarnog registra preko *rs1_i* ulaznog porta i ona predstavlja adresu od koje *M_CU* modul treba da krene sa upisivanjem podataka.
- *Offset* – Ukoliko je prihvaćena *non-strided store* instrukcija offset ima vrednost 4 , a ukoliko je prihvaćena *strided store* instrukcija, offset je vrednost na *rs2_i* portu.
- Broj elemenata koji treba da se prebaci iz *store FIFO memorija* unutar vektorskih linija (vrednost unutar *vl* registra).

Ove podatke *Arbiter* postavlja na *store_if* interfejs i poštujući odgovarajući dogovor *M_CU* modul će ih preuzeti, ukoliko nije zauzet izvršavanjem neke druge *load* ili *store* instrukcije. Isto kao kod *load* instrukcija, ukoliko *M_CU* nije spreman za izvršavanje, ovi podaci se skladište u *FIFO* memorijama unutar arbitra sve dok *M_CU* modul ne bude spreman da ih preuzme.

Poslednji tip instrukcije koji *Arbiter* modul treba da prepozna jeste **vektorska aritmetička instrukcija**. Ukoliko se ona prihvati, *Next_instruction_logic* blok treba samo da prosledi tu instrukciju (zajedno sa trenutnim vrednostima *rs1_i* porta, *vl_reg* i *vtype_reg* registara) *V_CU* modulu i vektorskim linijama, ukoliko su one spremne za izvršavanje naredne instrukcije (*ready_i* = 1₂) i ukoliko ne postoji zavisnost od *load* instrukcija u *local* memory bloku. U suprotnom se zaustavlja rad vektorskog jezgra(*vector_stall_o* = 1₂).

3.3.5 M_CU modul i memorijski podsistem

Na sledećoj slici su prikazane dve komponente (M_CU modul i memorijski podsistem) koje su posrednici između memorije sa podacima i vektorskih linija:



Slika 21: Sprega memorijskog podsistema i M_CU modula sa vektorskim linijama

Sa slike 21 se može videti da je **memorijski podsistem** jako jednostavan i da se sastoji iz samo jednog multipleksera. Taj multiplekser je kontrolisan od strane M_CU modula i u zavisnosti od vrednosti *store_fifo_re_o* porta proslediće na *data_to_mem* port podatak iz samo jednog *store FIFO* modula unutar vektorskih linija prilikom izvršavanja vektorske *store* instrukcije.

M_CU modul je, pored upravljanja memorijskim podsistemom, zadužen da komunicira sa *Arbiter* komponentom i skalarnim jezgrom i da na osnovu informacija koje dobije od njih započne transakcije prenosa podataka iz memorije ka procesoru, ili obrnuto, od procesora ka memoriji. Komunikaciju sa prethodno navedenim komponentama M_CU modul obavlja preko sledećeg interfejsa:

- **Load_if** je skup portova preko kojih M_CU dobija informacije od *Arbiter* modula (objašnjeno u sekciji 3.3.4) neophodne kako bi se izvršio prenos podataka iz memorije sa podacima ka *load* FIFO memorijama unutar vektorskih linija. Portovi od kojih se sastoji ovaj interfejs su sledeći:
 - $M_CU_ld_rs1_i$ – Ulazni 32-bitni port preko koga M_CU dobija baznu adresu od koje treba da započne prihvatanje podataka.
 - $M_CU_ld_rs2_i$ – Ulazni 32-bitni port preko koga M_CU dobija offset

- *M_CU_ld_vl_i* – Ulazni port preko koga *M_CU* dobija informaciju koliko elemenata treba da ekstrahuje. Njegova širina je: $\log_2(\text{VECTOR_LENGTH} * 8)$. Ona je određena maksimalnim brojem elemenata unutar vektorskog registra.
- *M_CU_ld_valid_i* – Jednobitni ulazni port. Ukoliko je njegova vrednost 1, podaci na prethodnim portovima su validni.
- *rdy_for_store* – izlazni jednobitni port preko *M_CU* šalje informaciju da je spreman da preuzme podatke koji su postavljeni na *M_CU_ld_rs1_i*, *M_CU_ld_rs2_i*, *M_CU_ld_vl_i* portovima.
- ***Store_if*** je skup portova preko kojih *M_CU* dobija informacije od *Arbiter* modula (objašnjeno u sekciji 3.3.4) neophodne kako bi se izvršio prenos podataka od vektorskih linija ka memoriji sa podacima. Portovi od kojih se sastoji ovaj interfejs su sledeći:
 - *M_CU_st_rs1_i* – Ulazni 32-bitni port preko koga *M_CU* dobija baznu adresu od koje treba da započne smeštanje podataka.
 - *M_CU_st_rs2_i* – Ulazni 32-bitni port preko koga *M_CU* dobija ofset
 - *M_CU_st_vl_i* – Ulazni port preko koga *M_CU* dobija informaciju koliko elemenata treba da smesti u memoriju za podatke. Njegova širina je: $\log_2(\text{VECTOR_LENGTH} * 8)$. Ona je određena maksimalnim brojem elemenata unutar vektorskog registra.
 - *M_CU_st_valid_i* – Jednobitni ulazni port. Ukoliko je njegova vrednost 1, podaci na prethodnim portovima su validni.
 - *rdy_for_store* – izlazni jednobitni port preko *M_CU* šalje informaciju da je spreman da preuzme podatke koji su postavljeni na *M_CU_st_rs1_i*, *M_CU_st_rs2_i*, *M_CU_st_vl_i* portovima.
- ***Scalar_ld_st_if*** je skup portova preko kojih *M_CU* modul dobija zahteve od skalarnog jezgra za pristup memoriji sa podacima. Portovi koji ga čine su:
 - *Scalar_load_reg_i* – Preko njega skalarno jezgro šalje zahtev za čitanje iz memorije.
 - *Scalar_store_reg_i* - Preko njega skalarno jezgro šalje zahtev za upis u memoriju.
 - *Scalar_address_i* – Preko njega *M_CU* modul dobija adresu sa koje skalarni procesor želi da pročita podataka, odnosno na koju želi da upiše podatak.

- **Data_memory_interface** je skup portova preko kojih M_CU šalje zahteve za upisom, odnosno čitanje memoriji sa podacima. Ti portovi su:
 - *mem_we_o* – izlazni jednobitni port dozvole upisa u memoriju.
 - *mem_re_o* – izlazni port dozvole čitanja iz memorije
 - *data_mem_addr_o* – izlazni port preko koga memorija dobija informaciju na koju adresu da izvrši upis, odnosno čitanje.
- **Lane_control_signals** je interfejs preko koga M_CU šalje kontrolne signale za upravljenje vektorskim linijama, njega čine sledeći portovi:
 - *store_fifos_re* – izlazni port preko koga M_CU generiše signale dozvole čitanja iz *store FIFO* modula unutar vektorskih linija. Širina ovog porta je vrednost parametra *NUM_OF_LANE*, odnosno jednak je broju vektorskih linija.
 - *load_fifos_we* – izlazni port preko koga M_CU generiše signale dozvole upisa u *load FIFO* memorije unutar vektorskih linija. Širina ovog porta je vrednost parametra *NUM_OF_LANE*, odnosno jednak je broju vektorskih linija.
 - *load_done_o* – izlazni jednobitni port na kome M_CU generiše puls kada je završio sam jednom vektorskom *load* instrukcijom.
- **Lane_status_signals** interfejs čini samo jedan port, *store_fifos_empty_i* preko koga M_CU dobija informaciju da li je *store FIFO* memorija unutar vektorskih linija prazna.

Proces prenosa podataka između memorije i procesora započinje ukoliko arbiter ili skalarno jezgro pošalju zahteve za prenosom podataka između memorije u procesora. U slučaju zahteva za izvršenje vektorske **load** instrukcije (*Arbiter* modul je postavio M_CU_ld_valid na logičku 1₂) M_CU prolazi kroz sledeće korake:

1. Postavlja logičku 1₂ na *rdy_for_load_i* port u trajanju od jednog takta kako bi Arbitru naglasio da je preuzeo informacije sa *load_if* interfejsa.
2. Postavlja na *data_mem_addr_o* port adresu sa koje treba da se pročita podatak.
3. Postavlja logičku 1₂ na *mem_re_i* port usled čega će memorija sa podacima u narednom taktu da postavi podatak na *data_from_mem_i* portu.
4. Svakog narednog takta M_CU inkrementira adresu na *data_mem_addr_o* portu za ofset vrednost, kako bi se učitao podatak sa naredni adrese.
5. Kako podaci pristižu M_CU modul ih raspoređuje po vektorskim linijama, tako što u prevovremenim trenucima generiše signale dozvole upisa u *load*

FIFO memorije unutar vektorskih linija preko *load_fifos_we_o* porta. Podaci se raspoređuju u cikličnom maniru.

6. Kada učitava *vl* (eng. *vector_length*) broj elemenata, *load* instrukcija je gotova i *M_CU* generiše puls na *load_done_o* portu.

U slučaju izvršavanja vektorske **store** instrukcije *M_CU* prolazi kroz slične korake kao i prilikom izvršavanja *load* instrukcije:

1. *M_CU* proverava da li postoji zahtev za izvršenje vektorske *store* instrukcije (*M_CU_st_valid_i* = 1₂) i da li je *store_fifo_empty* = 0₂ (u svim *store FIFO* memorijama unutar vektorskih linija postoji bar jedan podatak), ako jeste, prelazi na sledeće korake.
2. Postavlja logičku 1₂ na *rdy_for_store_i* port u trajanju od jednog takta kako bi Arbitru naglasio da je preuzeo informacije sa *store_if* interfejsa.
3. Generiše signale dozvole čitanja za *store FIFO* memorije unutar vektorskih linija preko *store_fifos_re_o* porta, tako da se svakog takta čita jedan podatak iz jedne vektorske linije u cikličnom maniru.
4. Postavlja logičku 1₂ na *mem_we_i* port usled čega će memorija sa podacima da upiše podatak pročitani iz vektorskih linija.
5. Postavlja na *data_mem_addr_o* adresu na koju treba da se upiše podatak. Svakog takta *M_CU* inkrementira adresu na ovom *portu* za ofset vrednost.
6. Kada se upiše *vl* broj elemenata *store* instrukcija je gotova.

Izvršavanje **skalarnih load i store** instrukcija je mnogo jednostavnije. Ukoliko *M_CU* detektuje zahtev od skalarnog procesora za čitanjem iz memorije, on će na *data_mem_addr_o* port postaviti vrednost sa *scalar_address_i* ulaznog porta, a *mem_re_o* port će postaviti na logičku 1₂ i u narednom taktu će skalarno jezgro dobiti podatak. Zahtev za upisom od strane skalarnog jezgra se obrađuje na sličan način, stim što se signal *mem_we_o* postavlja na logičku 1₂.

Pored *M_CU* modula i memorijskog podsistema na slici 20 su prikazana još dva bloka, **load_w_cnt i load_r_cnt** i oni su jednostavni brojači na gore. Prvi se inkrementuje za 1 svaki put kada *M_CU* izvrši jednu *load* instrukciju, a drugi se inkrementuje za 1 svaki put kada *V_CU* postavi kontrolni signal *V_CU_load_fifo_re_i* na 1₂ (prebaci podatke iz *load FIFO* memorije u VRF). Ukoliko su ta dva brojača jednaka, *load_fifo_empty_o* izlazni port je 0₂, a u suprotnom 1₂. Ovo je statusni port se prosleđuje Arbitru i na osnovu njega on zna da se unutar *load FIFO* memorija nalaze podaci i da ako dođe do zavisnosti između instrukcija može da izvrši *load* instrukciju iz njegove lokalne memorije.

4 Funkcionalna verifikacija vektorskog jezgra

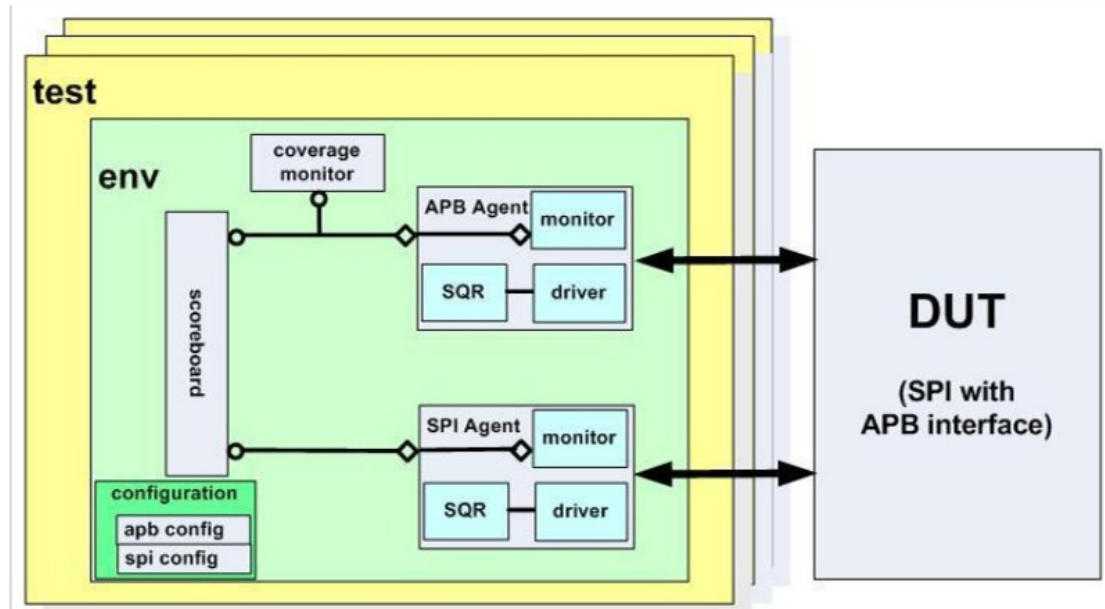
Kako bi se testirala ispravnost implementiranog vektorskog jezgra napravljeno je jednostavno verifikaciono okruženje zasnovano na UVM metodologiji (*eng. Universal Verification Methodology*). UVM omogućava efikasnu i iscrpnu verifikaciju i njen glavni princip jeste razvoj univerzalnih verifikacionih komponenti sa mogućnošću ponovnog korišćenja (*UVC – Universal Verification Components*). Karakteristike UVM metodologije su:

- Zrelost – UVM kodna bazna je zasnovana na OVM (*eng. Open Verification Methodology*) bibliotekama, sa određenim modifikacijama.
- Otvorena kodna baza (*eng. Open source*).
- Kompatibilnost i portabilnost – Svi poznati komercijalni simulatori podržavaju UVM.

UVM metodologija koristi principe objektno orijentisanog programiranja kako bi realizovala klasnu hijerarhiju. Na slici 22 je prikazano standardno UVM verifikaciono okruženje i osnovne komponente koje čine njegovu hijerarhiju su:

- *UVM test* – Je komponenta na vrhu hijerarhije. Njena glavna uloga je instancioniranje *environment* komponente koja se nalazi jedan stepenik ispod nje, njeno konfigurisanje i pokretanje sekvenci zaduženih za generisanje stimulusa koji će se proslediti na interfejs dizajna koji se verifikuje.
- *UVM environment* – Je komponenta na drugom nivou hijerarhije. Njena uloga je da instancionira komponente verifikacionog okruženja koje međusobno komuniciraju. Tipične komponente koje ona instancionira su *UVM Agent* i *UVM scoreboard*.
- *UVM scoreboard* – Je komponenta čija je glavna funkcija provera ispravnog funkcionisanja dizajna koji se testira. *Scoreboard* uglavnom dobija informacije od *UVM* agenta (stanje na ulaznim/izlaznim portovima dizajna), koje propušta kroz neki referentni model kako bi dobio očekivani rezultat i uporedio ga sa stvarnim.
- *UVM agent* – Je komponenta koja unutar sebe takođe grupiše verifikacione komponente zadužene za rad sa interfejsom dizajna koji se testira. Tipičan *UVM Agent* unutar sebe instancionira:
 - *UVM sequencer* komponentu - zaduženu za sinhronizaciju prenosa podataka između *UVM* sekvence i *UVM driver* komponente.
 - *UVM driver* - zadužen da podatke prihvaćene od sekvence postavi na interfejs dizajna koji se testira, poštujući protokol tog interfejsa.

- UVM *monitor* – sakuplja podatke sa interfejsa dizajna koji se testira i prosleđuje ih ostatku verifikacionog okruženja na dalju analizu.
- UVM sekvenca (*eng. Sequence*) – je objekat zadužen za generisanje stimulusa koji UVM *driver* treba da prosledi na interfejs dizajna koji se testira. Ona nije komponenta UVM hijerarhije.

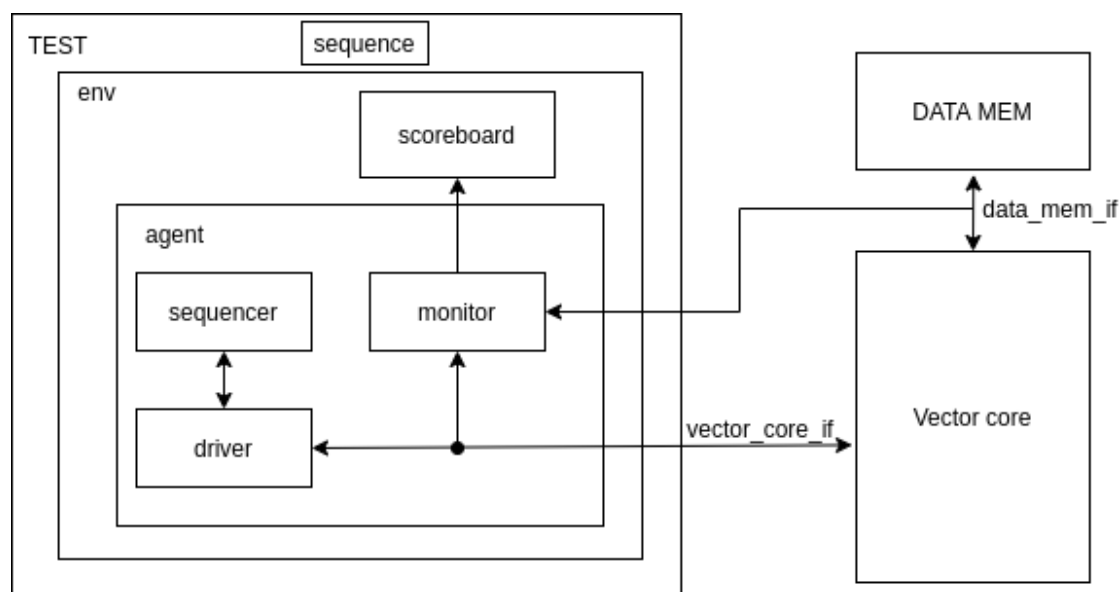


Slika 22: Hijerarhija verifikacionog okruženja [11]

Svi prethodno navedeni objekti i komponente se nalaze u UVM klasnoj biblioteci koja omogućava brzo kreiranje dobro konstruisanih i ponovno upotrebljivih komponenti. Bolji uvid u UVM metodologiju može se dobiti iz [12].

4.1 Opis verifikacionog okruženja

Na sledećoj slici je prikazano verifikaciono okruženje, zasnovano na UVM metodologiji, pomoću kojeg je proveren ispravan rad vektorskog procesora.



Slika 23: Verifikaciono okruženje vektorskog jezgra

Osnovna ideja je da verifikaciono okruženje prosleđuje vektorskom jezgru predefinisani set instrukcija (da imitira skalarno jezgro) i da na osnovu podataka koje ono upisuje u *data_mem* modul proverava da li su instrukcije izvršene ispravno. Ovo podrazumeva da je unutar verifikacionog okruženja realizovan referentni model koji poseduje svoj VRF modul i prilikom svake izvršene *store* instrukcije on proverava da li se vrednosti u njegovom VRF modulu poklapaju sa podacima koje vektorsko jezgro upisuje u *data_mem* blok. Razlog zašto instrukcije nisu randomizovane je zato što bi morao da se ispita preveliki prostor stanja, a izvršavanjem predefinisanih programa proveren je rad vektorskog jezgra u određenim „kritičnim“ scenarijima.

Data_mem modul, kao što se vidi se prethodne slike, nalazi se van verifikacionog okruženja i on je instanca BRAM bloka koji predstavlja memoriju sa podacima. Podaci unutar ove memorije su inicijalizovani nasumičnim vrednostima.

Interfejs preko koga verifikaciono okruženje komunicira sa vektorskim jezgrom čine sledeći portovi: *v_instruction_i*, *rs1_i*, *rs2_i*, *vecto_stall_o*, *all_v_stores_executed*, *all_v_loads_executed*, *Scalar_load_req_i*, *scalar_store_reg_i*, *scalar_address_i*. No, kako bi verifikaciono okruženje moglo da proverava da li vektorsko jezgro prosleđuje ispravne podatke *data_mem* bloku, ono mora da ima pristup i njegovom interfejsu, a njega čine sledeći portovi: *data_to_mem*, *data_from_mem*, *mem_we_o*, *mem_re_o*.

Za generisanje predefinisano programa korišćen je parser [9] realizovan za potrebe ovog projekta, koji asemblerski kod prevodi u binarni. On je u mogućnosti da parsira asemblerski kod RISC-V *integer* instrukcija, kao i vektorskih instrukcija koje implementira vektorsko jezgro u ovom projektu.

Kako bi se utvrdilo da li vektorsko jezgro funkcioniše na ispravan način, osmišljeno je nekoliko programama napisanih u asemblerskom kodu i pomoću njih se pogađaju sledeći scenariji:

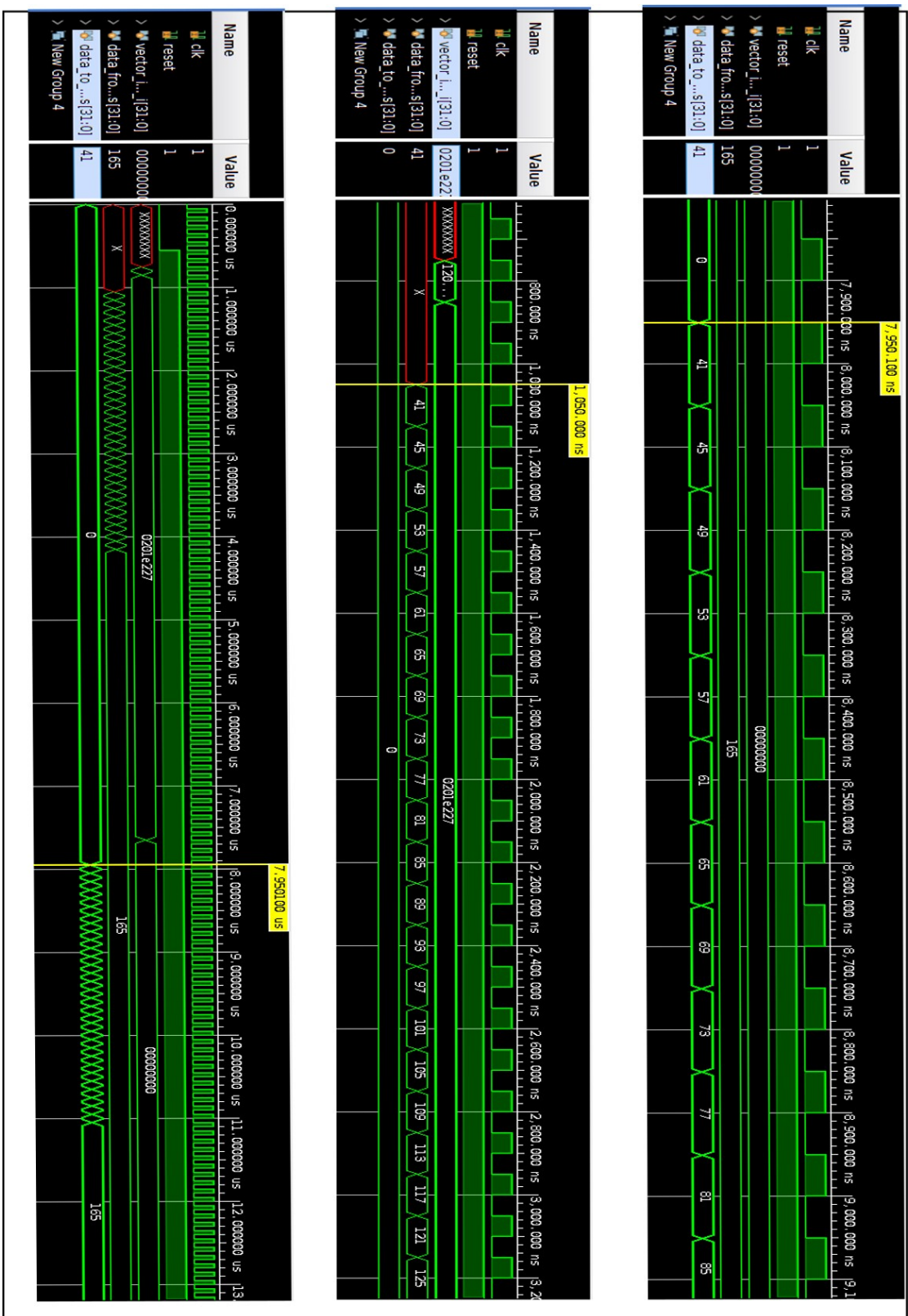
- Izvršavanje vektorskih *load* i *store* instrukcija.
- Izvršavanje aritmetičkih vektorskih instrukcija.
- Provera ispravnosti rada ukoliko nakon *load* instrukcije pristigne instrukcija koja je nezavisna od nje
- Provera ispravnosti rada ukoliko nakon *load* instrukcije pristigne instrukcija koja je zavisna od nje. Ova provera je neophodna jer procesor može dinamički da izvršava instrukcije nezavisne od *load* instrukcija, ali ukoliko se pojavi zavisna instrukcija procesor bi trebalo da zaustavi svoj radi dok se *load* ne izvrši.
- Provera rada ukoliko postoji više uzastopnih *load* instrukcija.
- Provera rada ukoliko postoji više uzastopnih *store* instrukcija.
- Provera rada ukoliko se pomoću konfiguracionih instrukcija menjanju *vl* i *vtype* CSR registri.
- Ispitivanje prethodnih scenarija sa različitim vrednostima parametra *NUM_OF_LANES*, odnosno promenom broja vektorskih linija koje vektorsko jezgro poseduje (1, 2, 4, 8, 16, 32).

Primer jednostavnog programa kojim se pogađa jedan od prethodnih scenarija prikazan je u nastavku:

```
vlw.v $4, ($2), 1
```

```
vsw.v $4, ($3), 1
```

U ovom primeru proverava se ispravnost rada *load* instrukcije, tako što se prvo upišu podaci u vektorski registar V4, a nakon toga se pomoću *store* instrukcije oni pročitaju iz V4 i smeste u memoriju sa podacima (*data_mem* blok na slici 23). Kao što je pomenuto na početku ove sekcije, verifikaciono okruženje proverava ispravnost rada samo prilikom izvršavanja *store* instrukcija, odnosno da li su podaci koji se smeštaju u *data_mem* blok ispravni. Na sledećoj slici prikazan je talasni dijagram izvršavanja prethodnog programa:



Slika 24: Talasni dijagrami izvršavanja jednostavnog programa

Na svim talasnim dijagramima posmatraju se vrednosti na 3 porta: *vector_instructio_i*, *data_from_mem_i* i *data_to_mem_o*. Preko prvog procesor prihvata vektorsku instrukciju, preko drugog prima podatke iz *data_mem* bloka i preko trećeg šalje podatke u *data_mem* blok. Na levom talasnom dijagramu je prikazano izvršavanje obe instrukcije. Kao što se sa slike može videti, prvo se izvršava *load* instrukcija koja podatke smešta iz memorije u vektorski procesor. *Store* instrukcija mora da sačeka dok se podaci ne smeste u VRF modul vektorskog jezgra, jer ona zavisi od *load* instrukcije i tek nakon toga može da krene da se izvršava (trenutak 7,950.100 ns na talasnom dijagramu). U ovom primeru *load* instrukciji je potrebno 64 takta da smesti podatke u VRF, jer je NUM_OF_LANES = 1, odnosno 32 takta je potrebno da se podaci pročitaju iz memorije i smeste u vektorske linije i još 32 takta da vektorska linija prebaci podatke u VRF. Desni talasni dijagram pokazuje da vektorska *store* instrukcija smešta iste podatke u memoriju koje je prethodno pročitala *load* instrukcija.

Nakon što se simulacija završi, verifikacioni okruženje će ispisati broj koji demonstrira da li se rezultat izvršavanja *store* instrukcije poklapa sa očekivanim, i to je prikazano na sledećoj slici za prethodni primer:

```
uvm_test_top.env.scoreboard [vector_core_scoreboard] vector lane scoreboard num of matches: 32
uvm_test_top.env.scoreboard [vector_core_scoreboard] vector lane scoreboard num of miss matches: 0
```

Slika 22. Rezultat simulacije

Num of matches je broj koji govori koliko podataka, upisanih u *data_mem* blok, se poklapa sa očekivanim rezultatima verifikacionog okruženja, dok je *num of miss matches* predstavlja broj nepoklapanja. Sa slike se vidi da je broj poklapanja 32, odnosno *store* je trebalo da pročita 32 elementa iz procesora i smesti ih u VRF, što se i desilo.

Prethodno opisana procedura je izvršena za sve scenarije za koje se smatralo da su kritični.

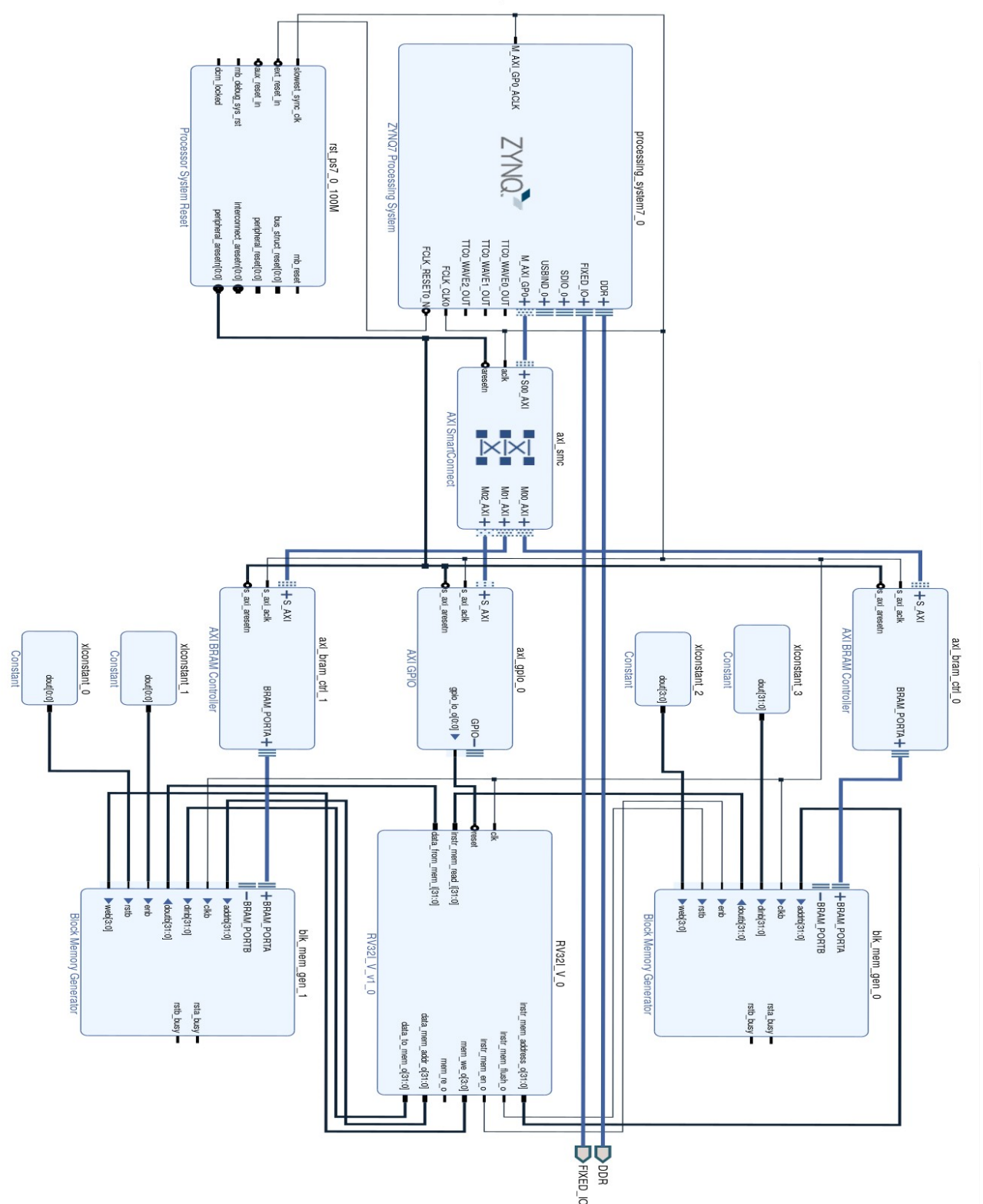
5 Performanse i iskorišćenost resursa

Ova sekcije je posvećena analizi utrošenih resurse prilikom implementacije vektorskog procesora opisanog u sekciji 3 kao i analizi performansi nakon sinteze i implementacije. Za to je iskorišćen Vivado alat kompanije Xilinx [6], verzija 2020.1. Pored Vivado alata, za testiranje sistema iskorišćena je zybo razvojna ploča koja pripada Zynq®-7000 familiji sistema na čipu (*eng. System on chip*) i njene najbitnije karakteristike su opisane u sledećoj tabeli:

Tabela 5: Karakteristike Zybo razvojne ploče

Naziv	Količina
6-ulazne LUT ćelije	26400
Flip-flop ćelije	35200
BRAM	240KB
DSP	80
DDR	512 MB DDR3

Na sledećoj slici je prikazana integracija vektorskog procesora sa komponentama zybo razvojne ploče. Tu se može videti da vektorsko jezgro radi u sprezi sa procesorskim sistemom (*processing_system7_0*) zybo razvojne ploče, koje je zaduženo da upiše u *blk_mem_gen_0* program koji vektorsko jezgro izvršava, a u *blk_mem_gen_1* podatke koje vektorsko jezgro treba da obradi. *Blk_mem_gen_0* i *blk_mem_gen_1* su gotovi moduli kompanije Xilinx koji se koriste kao memorije, pri čemu za skladištenje podataka koriste BRAM ćelije. Veličina memorije za podatke (*blk_mem_gen_1*) 64KB, a veličina memorije za instrukcije (*blk_mem_gen_0*) je 2KB.



Slika 25: Integracija vektorskog procesora na zybo razvojnoj ploči

5.1 Iskorišćenost resursa

U sekciji 3 je objašnjeno da je implementacija vektorskog procesora parametrizovana i da se na taj način mogu povećavati performanse na uštrb resursa, i obrnuto, smanjivati iskorišćenost resursa na uštrb performansi. U nastavku je prikazan proračun iskorišćenosti BRAM i DSP ćelija na osnovu parametara NUM_OF_LANES i VECTOR_LENGTH. Naredne dve tabele prikazuje koliko resursa koriste određeni blokovi vektorskog procesora:

Tabela 6: Resursi koje koriste pojedinačni blokovi vektorskog procesora

Komponenta	BRAM	DSP
<i>Arbiter</i>	12 KB	0
<i>Vector lane VRF</i>	VECTOR_LENGTH/NUM OF LANES $\leq 16 \Rightarrow$ 4KB VECTOR_LENGTH/NUM OF LANES $> 16 \Rightarrow$ $((\text{VECTOR_LENGTH} * 32 * 32 / 8 * 2) / 1024)$ KB	0
<i>Vector lane store fifo</i>	2KB	0
<i>Vector lane load fifo</i>	2KB	0
<i>Vector lane ALU</i>	0	6
<i>Scalar_Core</i>	0	0

Prethodna tabela prikazuje koliko resursa troši jedna vektorska linija. Da bi se dobila ukupna potrošnja neophodno je prvo pomnožiti broj resursa koje zahtevaju komponente unutar vektorske linije sa NUM_OF_LANES parametrom i sumirati sa resursima koje zahtevaju ostale komponente. Na primer, ukoliko bi NUM_OF_LANES bio 4 i VECTOR_LENGTH = 1024 iskorišćenost resursa bi bila: 24 DSP ćelije i 44 KB BRAM.

U prethodnom proračunu nije prikazano koliko je LUT i flip-flop ćelija iskorišćeno, jer je taj proračun mnogo komplikovaniji. U nastavku su prikazani izveštaji Vivado alata o utrošenosti resursa nakon sinteze i implementacije sistema sa slike 25, kada je VECTOR_LENGTH = 1024 i NUM_OF_LANES = 1, 4, 8. U ovom izveštaju biće uvrštena iskorišćenost LUT, flip-flop, LUTRAM, BRAM i DSP ćelija.

Tabela 7: Iskorišćenost resursa kada je broj vektorskih linija 1

Resurs	Zauzeto	Dostupno	Zauzeto %
LUT	3324	17600	18.38
LUTRAM	8	6000	0.13
Flip-Flop	2179	35200	6.19
BRAM	24	240	10
DSP	6	80	7.5

Tabela 8: Iskorišćenost resursa kada je broj vektorskih linija 4

Resurs	Zauzeto	Dostupno	Zauzeto %
LUT	6351	17600	36.09
LUTRAM	8	6000	0.13
Flip-Flop	2391	35200	6.79
BRAM	44	240	18.33
DSP	24	80	30

Tabela 9: Iskorišćenost resursa kada je broj vektorskih linija 8

Resurs	Zauzeto	Dostupno	Zauzeto %
LUT	10217	17600	58.05
LUTRAM	16	6000	0.27
Flip-Flop	2626	35200	7.46
BRAM	76	240	31.67
DSP	48	80	60

Bitno je obratiti pažnju da prethodni izveštaji o iskorišćenosti resursa uzimaju u obzir samo vektorski procesor, a ne ceo sistem sa slike 25. Radi poređenja u sledećoj tabeli je prikazan izveštaj utrošenosti resursa celog sistema, kada je broj vektorskih linija 8:

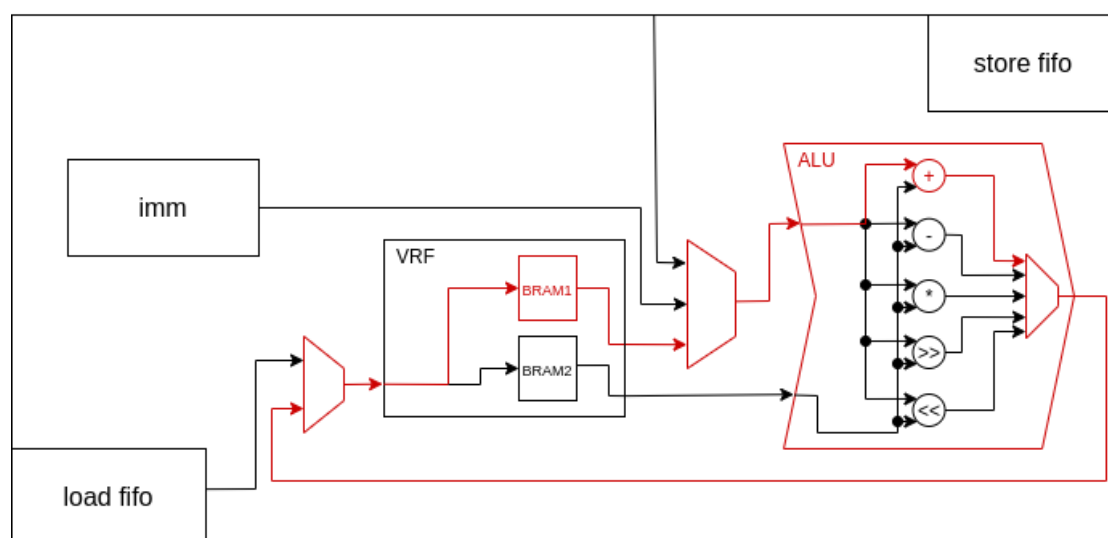
Tabela 10: Iskorišćenost resursa celog sistema kada je broj vektorskih linija 8

Resurs	Zauzeto	Dostupno	Zauzeto %
LUT	15156	17600	86.11
LUTRAM	883	6000	14.72
Flip-Flop	7897	35200	22.43
BRAM	36.5	60	60.83
DSP	48	80	60

5.2 Performanse

Maksimalna frekvencija rada sistema takođe zavisi od parametara VECTOR_LENGTH i NUM_OF_LANES. Nakon implementacije sistema sa slike 25 (sa parametrima VECTOR_LENGTH = 1024 i NUM_OF_LANES = 1, 2, 4, 8) frekvencija rada procesora je oko 95 MHz. Ukoliko je broj vektorskih linija 1, ova frekvencija može da bude 96 MHz, a razlog za to je manji broj iskorištenih logičkih ćelija na FPGA platformi i samim tim veća efikasnost prilikom povezivanja (*eng. routing*). Ono što takođe dovodi do manje frekvencije jeste količina iskorišćenosti BRAM ćelija, jer to otežava posao alatu koji vrši implementaciju prilikom povezivanja. Ukoliko se memorija za podatke spusti na 2KB, maksimalna frekvencija sistema se podiže na 100 MHz.

Osim problema prilikom rutiranja, nakon vremenske (*eng. Timing*) analize, uočena je jedna kritična putanja u vektorskim linijama. Ta putanja, naznačena crvenom bojom na slici 26, uzima u obzir propagaciono vreme čitanja i upisa iz, odnosno u BRAM memoriju unutar VRF modula, mreža za rutiranje i sabirača unutar ALU jedinice. Poboljšanje za ovu kritičnu putanju bi bilo uvođenje protočne obrade prilikom izvršavanja operacije sabiranja. Takođe, to bi bilo dobro uraditi za sve ostale operacije jer alat može, nakon uvođenja protočne obrade za operaciju sabiranja, neku od njih da uvrsti u kritičnu putanju.



Slika 26: Kritična putanja unutar vektorske linije

Na prethodnoj slici, unutar ALU jedinice, su prikazane samo najbitnije operacije, odnosno izostavljene su operacije: jednakosti, nejednakosti, minimum, maksimum, logičke operacije, veće od, manje od.

Množenje, iako složenija operacije od sabiranja, nije deo kritične putanje jer je prilikom izvršavanja ove operacije već uvedena protočna obrada.

6 Zaključak

U ovom radu izvršena je vektorska ekstenzija 32-bitnog procesora, koji implementira RISC-V integer set instrukcija. To je urađeno tako što je pored skalarnog jezgra implementirano vektorsko jezgro, prateći verziju 0.8 RISC-V - „V“ nacrtu za vektorsku ekstenziju [4].

Vektorska ekstenzija dodaje 32 vektorska registra (V0 – V31) na već postojeće registre skalarnog jezgra. Svaki vektor unutar sebe sadrži VECTOR_LENGTH broj elemenata, pri čemu je svaki element širine 32 bita.

Za razliku od standardne implementacije procesora pomoću ASIC tehnologije, ovaj rad istražuje alternativnu mogućnost kreiranja vektorskog procesora koristeći FPGA platformu. Specifičnost ovih platformi je mogućnost reprogramiranja, čime bi mogle da se menjanju neke od karakteristika procesora kako bi se povećale performanse ili kako bi se optimizovala iskorišćenost resursa. U ovom radu je ta karakteristika iskorišćena uvođenjem promenljivog broja vektorskih linija, čime je procesor parametrizovan. Povećanjem broja vektorskih linija povećavaju se performanse, jer jedna vektorska instrukcija može da se izvrši n puta brže (n je broj vektorskih linija), no, ovo dolazi uz cenu, jer se povećava količina neophodnih resursa.

Kako bi se povećale performanse, pored promenljivog broja vektorskih linija, u samoj mikroarhitekturi vektorskog procesora omogućeno je delimično dinamičko izvršavanje instrukcija. Pošto vektorske *load* instrukcije zahtevaju više vremena za njihovu egzekuciju, omogućeno je da se, dok *load* instrukcija preuzima podatke iz memorije, izvršava naredna instrukcija, ukoliko je nezavisna.

Naravno, implementacija procesora u ovom radu ima mnoštvo prostora za poboljšanje. Memorija u kojoj se skladište podaci je sada u FPGA delu sistema na čipu i za njenu implementaciju se koriste BRAM ćelije. Mana ovog pristupa je mala količina raspoloživih BRAM ćelija (na zybo razvojnoj ploči svega 240KB). Bolje rešenje bi bilo da se koristi DDR memorija sistema, no to bi zahtevalo modifikacije vektorskog procesora (specifično *M_CU* modula) i implementaciju memorijskog sistema koji bi sakrio kašnjenje (*eng. latency*) prilikom pristupa DDR memoriji (na primer keš memorija). Implementirani skup instrukcija bi mogao da se proširi dodatnim instrukcijama, na primer instrukcijama redukcije, pomoću kojih bi se poboljšalo izvršavanje operacija množenja matrica, konvolucionih operacija i sl.

7 Bibliografija

- [1] J. L. Hennessy and D. A. Patterson. Computer Architecture - A Quantitative Approach, Sixth Edition. Morgan Kaufmann, 2017
- [2] Đ. Mišeljić i N. Kovačević – Napredni mikroprocesorski sistemi, Upoznavanje sa RISC-V procesorom, 2019.
- [3] A. Waterman, K. Asanović. The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2, CS Division, EECS Department, University of California, Berkeley, 2017.
- [4] “Working draft of the proposed RISC-V “V” vector extension,” 2020, accessed on February 15, 2020. [Online]. Available: <https://github.com/riscv/riscv-v-spec>
- [5] C. Kozyrakis and David Patterson - Overcoming the Limitations of Conventional Vector Processors, Proceedings of the International Symposium on Computer Architecture, 2003
- [6] URL: <https://www.xilinx.com/>
- [7] 7 Series FPGAs Memory Resources user guide UG473 (v1.14), 2019
- [8] 7 Series DSP48E1 Slice user Guide UG479 (v1.10), 2018
- [9] URL: <https://github.com/Nikola2444/RISCV-RV32I-Assembler>
- [10] *Zybo reference manual.*
- [11] *Mentor graphics Verification Academy, UVM Cookbook.*
- [12] *Acclera System initiative, Universal Verification Methodology (UVM) 1.2 User's Guide*