



UNIVERZITET U NOVOM SADU

FAKULTET TEHNIČKIH NAUKA

**KATEDRA ZA
MIKRORAČUNARSKU
ELEKTRONIKU**



IMPLEMENTACIJA PODSISTEMA SKRIVENE MEMORIJE ZA RISC-V PROCESOR

master teza

Kandidat
Đorđe Mišeljić

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 teza
Autor, AU:	Dorđe Mišeljić
Mentor, MN:	Dr Vuk Vranjković, docent
Naslov rada, NR:	Implementacija podsistema skrivene memorije za RISC-V procesor
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/gra fika/priloga)	8/45/0/3/29/0/0
Naučna oblast, NO:	Elektrotehnika i računarstvo
Naučna disciplina, ND:	Embedded sistemi i algoritmi
Predmetna odrednica/Ključne reči, PO:	RISC-V arhitektura, skrivena memorija, Zybo, FPGA
UDK	
Čuva se, ČU:	Biblioteka Fakulteta Tehničkih Nauka 21000 Novi Sad, Trg Dositeja Obradovića 6
Važna napomena, VN:	Nema
Izvod, IZ:	U ovom radu se modeluje podsistem skrivene memorije RISC-V procesora, za primenu na FPGA čipovima. Cilj je implementacija i testiranje predložene arhitekture na Zybo razvojnoj ploči.
Datum prihvatanja teme, DP:	DATUM KADA SE ODOBRI
Datum odbrane, DO:	DATUM KADA BRANIMO
Članovi komisije, KO:	Predsednik: Sretni dobitnik 1
	Član: Sretni dobitnik 2
	Član, Mentor: Prof. dr Vuk Vranjković
	Potpis mentora



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:		Đorđe Mišeljić
Mentor, MN:		PhD Vuk Vranjković, docent
Title, TI:		Implementation of caching subsystem for RISC-V processor, on Zybo board
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/appendi xes))		8/45/0/3/29/0/0
Scientific field, SF:		Electrical Engineering
Scientific discipline, SD:		Embedded systems and algorithms
Subject/Key words, S/KW:		RISC-V architecture, cache memory, Zybo, FPGA
UC		
Holding data, HD:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia
Važna napomena, VN:		
Abstract, AB:		This paper models caching subsystem for RISC-V processor, with FPGA application in mind. Goal is to implement one such architecture and test it on Zybo development board.
Accepted by the Scientific Board on, ASB:		DATUM KADA SE ODOBRI
Defended on, DE:		DATUM KADA BRANIMO
Defended Board, KO:	President:	Sretni dobitnik 1
	Member:	Sretni dobitnik 2
	Member, Mentor:	Ph.D Vuk Vranjković
		Mentor's signature

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, _____

Potpis studenta/kinje

	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 Milan Sečujski

Student:	Đorđe Mišeljić	Broj Indeksa	E1 5/2018
Oblast	Projektovanje digitalnih sistema		
Mentor	Dr Vuk Vranjković		
NA OSNOVU PODNETE PRIJAVA, PRILOŽENE DOKUMENTACIJE I ODREDBI STATUTA FAKULTETA IZDAJE SE ZADATAK ZA DIPLOMSI(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 podsistema skrivene memorije za RISC-V procesor

TEKST ZADATKA:

Modelovati podsistem skrivene memorije za RISC-V procesor u VHDL jeziku, te demonstrirati rad u simulaciji pomoću Vivado alata. Napraviti IP jezgro koje sačinjava procesor i podsistem skrivene memorije, te testirati rad na Zybo ploči.

Rukovodilac studijskog programa:

Mentor rada:

Primerak za: ☐ Studenta; ☐ Mentora;

Sadržaj

1 Uvod.....	8
2 Memorijski podsistem i keširanje.....	10
2.1 Virtuelna i fizička memorija.....	10
2.2 Tipovi memorija i hijerarhija memorijskog sistema.....	11
2.3 Direktno preslikan keš.....	14
2.4 Set asocijativan keš.....	15
2.4.1 Polisa evikcije.....	16
2.4.2 Polisa upisa.....	17
2.4.3 Polisa razdeljenosti.....	18
2.4.4 Polisa inkluzije keša.....	19
3 Modelovanje sistema za keširanje.....	20
3.1 RISC-V jezgro.....	20
3.2 Memorijski podsistem.....	23
3.2.1 Direktno mapiran sistem za keširanje.....	24
3.2.2 Simulacija direktno preslikanog keša.....	33
3.2.3 Set asocijativan drugi nivo sistema za keširanje.....	38
3.2.4 Simulacija sistema sa set asocijativnim drugim nivoom keša.....	42
4 Implementacija sistema na Zybo razvojnoj ploči.....	47
4.1 Pakovanje IP jezgra.....	47
4.2 Blok dizajn u Vivado alatu.....	49
4.3 Testiranje u Vitis alatu.....	50
5 Zaključak.....	54
6 Literatura.....	55

Slike

Slika 1: Mapiranje virtuelnih adresa.....	10
Slika 2: <i>Memorijski</i> podsistem.....	12
Slika 3: Podela adrese u polja.....	14
Slika 4: Direktno preslikan keš.....	15
Slika 5: Četvorosmerni set asocijativan keš.....	16
Slika 6: RV32I procesorsko jezgro.....	21
Slika 7: Razrešavanje hazarda zaustavljanjem protočne obrade.....	22
Slika 8: Hijerarhija VHDL modela RISCv procesora.....	23
Slika 9: Blok dijagram direktno preslikanog sistema za keširanje.....	27
Slika 10: Raspored bita u memorijama za čuvanje tagova.....	28
Slika 11: Keš kontroler prvog nivoa (direktno preslikani).....	29
Slika 12: Keš kontroler drugog nivoa.....	32
Slika 13: Prihvatanje bloka podataka.....	36
Slika 14: Konflikt u drugom nivou keša sa direktnim preslikavanjem.....	37
Slika 15: Raspored bita u memoriji za čuvanje tagova asocijativnog keša.....	38
Slika 16: Blok dijagram asocijativnog sistema za keširanje.....	39
Slika 17: Pomerački registar evicor_reg.....	40
Slika 18: Keš kontroler prvog nivoa (asocijativan drugi nivo).....	41
Slika 19: Stanje keševa u simulaciji za indeks 0.....	43
Slika 20: Nastavak stanja keševa u simulaciji za indeks=0.....	43
Slika 21: Evikcija prljavog bloka iz prvog nivoa keša.....	45
Slika 22: Evikcija prljavog bloka iz drugog nivoa keša.....	46
Slika 23: Procesor sa sistemom za keširanje, oklopljen AXI interfejsom.....	47
Slika 24: Blok dizajn sistema.....	49
Slika 25: Podešavanja AXI interkonekta.....	49
Slika 26: Rezultati implementacije.....	49
Slika 27: Iskorištenost resursa nakon implementacije.....	50
Slika 28: Potrošnja energije nakon implementacije.....	50
Slika 29: Blok dizajn sistema sa ILA modulom.....	50
Slika 30: Stanje terminala nakon izvršavanja C aplikacije.....	51
Slika 31: Prvo čitanje bloka iz memorije.....	52
Slika 32: Drugo čitanje bloka iz memorije.....	52
Slika 33: Treće čitanje bloka iz memorije.....	53
Slika 34: Upis bloka u memoriju.....	53

1 Uvod

Poslednjih par decenija, dve arhitekture skupa instrukcija procesora (*eng. Instruction Set Architecture*) su kontrolisale svetsko tržište: x86 i ARM. Kompanije Intel i AMD sa x86 arhitekturom su većinski proizvođači procesora visokih performansi za personalne računare i servere, dok ARM dominira u oblastima u kojima je potrošnja energije glavni faktor. Stoga se procesori sa ARM arhitekturom danas mogu naći u većini pametnih telefona, IoT (*eng. Internet of Things*) uređajima, *embedded* sistema kao i u poslednje vreme laptop računarima. Korišćenje ovih arhitektura zahteva dobijanje licence, što predstavlja veliku investiciju za nove kompanije, te dalje učvršćuje položaj industrijskih giganta i njihovu dominaciju nad tržištem.

Obećavajući faktor za promenu trenutnog stanja u hardverskoj industriji je nova, besplatna i otvorena arhitektura pod nazivom RISC-V. Nastala 2011. godine za naučno-istraživačke svrhe na Univerzitetu Berkley, Kalifornija, obezbeđuje novu eru inovacije u oblasti procesora kroz masovnu globalnu kolaboraciju. Začetnici RISC-V arhitekture, Asanović i Paterson, su bili motivisani da reše problem nedostatka fleksibilnosti sa trenutnim arhitekturama. Kako su im za istraživanja bile potrebne nove specijalizovane instrukcije, a nisu mogli da dobiju dozvolu od Intel-a ili ARM-a da modifikuju već postojeće arhitekture, odlučili su da naprave novu arhitekturu sa fokusom na rešavanje ovog problema.

2014. godine, organizovano je prvo zvanično lansiranje RISC-V arhitekture, te se od tada skupila dovoljna podrška da se oformi neprofitabilna organizacija pod nazivom "RISC-V Foundation". Danas, ona ima preko 200 članova od kojih su većina vodeće kompanije u oblastima dizajna i fabrikacije hardvera. Ova organizacija se bavi istraživanjem, unapređivanjem u održavanjem seta instrukcija, dok je dizajn mikroarhitekture kao i razvoj softvera prepušten nezavisnim organizacijama. Jedne od kompanija koje su najavile korišćenje RISC-V procesora u sistemima na čipu nove generacije su Nvidia, Samsung, Western Digital i Qualcomm [1].

RISC-V kroz modularizaciju omogućava korisnicima da naprave čip specifično konstruisan za željenu primenu. Krećući se od osnovnog seta instrukcija, moguće je dodati različite ekstenzije koje su specijalno dizajnirane za različite komercijalne i naučne svrhe. Sa ovim na umu, moguće je da sa različitim ekstenzijama, RISC-V nađe primenu kako u oblasti visokih performansi tako i u oblasti uređaja male potrošnje energije, te na taj način napravi otvoreni standard za hardver. Benefit ovoga je razvoj softvera koji će biti kompatibilan i raditi na svim sličnim RISC-V jezgrima. Zajednička arhitektura za različite primene znači da će se minimizovati napor za rekompajliranjem kodova, razvijanjem operativnih sistema i ostalog softvera [2].

Posebna oblast od interesa za RISC-V arhitekturu su FPGA (*eng. Field Programmable Gate Array*) uređaji. Ukoliko neko implementira RISC-V procesorsko jezgro na FPGA čipu, često obezbedi i RTL (*eng. Register Transfer Level*) izvorni kod. Budući da RISC-V ima i punu softversku podršku Linux organizacije, moguće je bez ikakvih izmena koda, besplatno i legalno podići operativni sistem baziran na Linux-u, na bilo kojoj FPGA razvojnoj ploči. Mnogi projekti ovog tipa već postoje, te se FPGA implementacije svakodnevno optimizuju i

nadograđuju zbog potpune vidljivosti kako RTL tako i Linux izvornog koda. Sa otvorenošću koda, dolazi i signurnost, što je Linux zajednica demonstrirala od svog začeca. Kada je RTL kod obezbeđen, ovo omogućava dublju inspekciju hardvera i nalaženje propusta u mikroarhitekturi koji mogu biti slaba tačka za softverski napad kao što je bio slučaj sa Spectre i Meltdown ranljivostima kod x86 arhitekture 2018 godine. Takođe se dobija i poverenje korisnika, zbog sve češće eksploatacije informacija i narušavanja privatnosti što će postajati sve bitnija prodajna tačka RISC-V arhitekture.

Sa svim prednostima koje RISC-V nudi, podrškom najuticajnih kompanija i potrebama tržišta, samo je pitanje vremena kada će ova arhitektura postati standard. U FPGA domenu, postoji velika zainteresovanost za dizajn *soft-core* RISC-V mikroprocesora, budući da će ove implementacije verovatno zameniti ulogu Xilinx-ovog MicroBlaze procesora koji je do sada bio jedino od retkih rešenja za podizanje operativnog sistema na FPGA čipu. MicroBlaze je takođe konfigurabilna besplatna i otvorena *RISC* arhitektura bazirana na predlozima iz popularne knjige profesora Patersona, te je čak bio prvi *soft-core* procesor uključen u glavno stablo izvornog koda Linux-ovog kernela. Ono što će izdvojiti RISC-V u ondosu na MicroBlaze je softverska podrška koja je uvek bila slaba tačka *soft-core* procesora, zbog malog broja korisnika i odžavaoca softvera.

Koliko god atraktivno zvučale nove RISC arhitekture sa optimizacijama instrukcija na nivou bita, ne sme se zaboraviti da je brzina pristupa memoriji i dalje usko grlo u performansama procesora. Svaki moderan čip od ARM Cortex-A5 do Intel Core i9 na sebi ima barem dva nivoa skrivene (keš, *eng. cache*) memorije koja zbog težnje da bude što veća, zauzima veliki deo ukupne površine čipa. Dakle, benefiti skrivenih memorija se ne smeju ignorisati, čak i u oblastima procesiranja sa malom potrošnjom energije.

Kao i kod ASIC implementacije, *soft-core* procesori na FPGA čipovima mogu iskoristiti različite tipove memorijskih resursa koji su na raspolaganju dizajneru, kao spregu između procesorske logike i DDR memorijskog čipa. Ovaj rad će predstaviti jedan način iskorišćenja konfigurabilnih memorijskih resursa koji su prisutni na svakom savremenom FPGA čipu, kako bi se implementiralo keširanje i zadovoljila memorijska zahtevnost RISV-V procesora.

U sledećem poglavlju će biti u kratkim crtama rezimirana struktura memorijskog podsistema računara. Biće objašnjena svrha *on-chip* keš memorije, te dve glavne strukture: direktno preslikan i set asocijativan keš. Takođe će biti predstavljene prednosti i mane tipičnih odluka koje dizajner keš podsistema mora da donese.

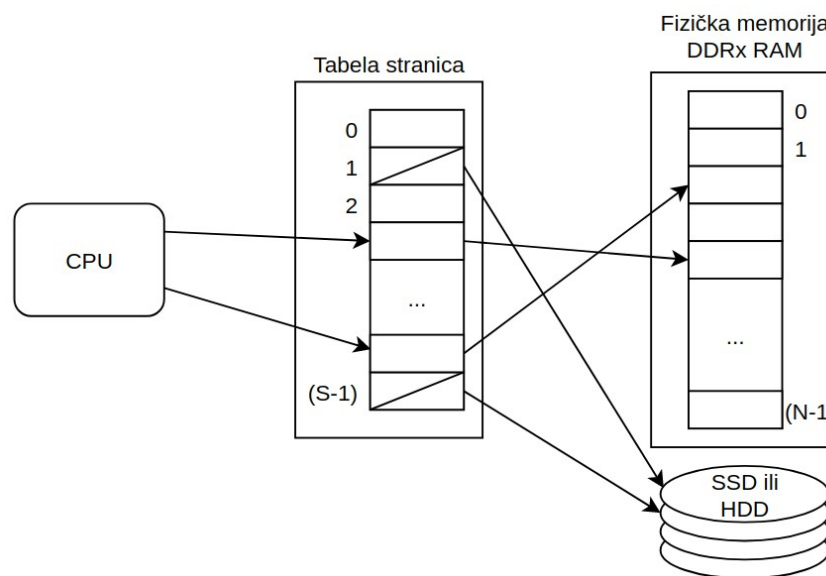
U trećem poglavlju će biti prvo predstavljen RISC-V procesor koji će se koristiti za testiranje keš podsistema. Nakon toga će u HDL jeziku biti modelovan direktno preslikan sistem za keširanje sa dva nivoa hijerarhije. Predstaviće se program pomoću kojega će se testirati kritični slučajevi u ovom sistemu, a zatim će biti predstavljeni rezultati simulacije. Drugi nivo keša će se zatim modifikovati da bude set asocijativan. U simulaciji će, na istom test programu, biti demonstrirani tipični prenosi podataka između memorijskih nivoa.

U četvrtom poglavlju će se RISC-V procesor sa keš podsistemom upakovati u IP jezgro a zatim u blok dijagramu povezati sa Zynq 7000 SoC-om. Koristeći Vivado i Vitis alate kompanije Xilinx, sistem će biti testiran na Zybo ploči pomoću samostalne C aplikacije.

2 Memorijski podsistem i keširanje

2.1 Virtuelna i fizička memorija

Svaki računar koji izvršava neku vrstu modernog operativnog sistema, ima operativnu memoriju koja je virtuelizovana, te korisnik (programer) vidi idealizovanu apstrakciju memorijskih resursa koji su prisutni u sistemu. Ovo stvara iluziju da je memorija u sistemu mnogo veća nego što jeste, te programer ne mora da zna koliki je stvarni fizički adresni prostor na računaru, niti ga mora kontrolisati. Na ovaj način, svaki proces koji se izvršava, može da referencira mnogo veći memorijski prostor nego što je prisutan na računaru.



Slika 1: Mapiranje virtuelnih adresa

Operativni sistem zajedno sa pomoćnim hardverom, mapira blokove virtuelnih adresa, stranice, (*eng. page*) na fizičke adrese, te čuva podatke o mapiranju u tabelama stranica (*eng. page tables*). Bez pomoći i bilo kakvih intervencija programera, specijalizovan hardver poznat kao "jedinica za upravljanje memorijom" (*eng. Memory management unit, MMU*) prevodi viruelne adrese na fizičke. Budući da se toku rada operativnog sistema može referencirati više memorije nego što je prisutno, sistem u fizičkoj memoriji održava samo one stranice koji se trenutno referenciraju, dok ostatak čuva na nekom tipu sekundarne memorije npr. poluprovodničkom (*SSD*) ili tvrom disku (*HDD*). Svaka virtuelna stranica mora biti mapirana na fizičku stranicu koja se nalazi u RAM memoriji ili na lokaciji na *SSD* ili *HDD* uređaju za masovno skladištenje podataka. Ukoliko se stranica nalazi na disku, operativni sistem će je preneti u operativnu memoriju, ali prvo mora neku drugu stranicu iz memorije da prebaci na disk kako bi se oslobodio prostor.

Virtuelna memorija ima mnoge prednosti za operativne sisteme kao što je bolje iskorišćenje memorije - rešavajući fizičku fragmentaciju, omogućava lakše upravljanje procesima i deljenje podataka između istih, kao i sigurnost - izolacijom između različitih procesa, korisnika i administratora, itd.

Međutim, u nekim slučajevima, *embedded* sistemima nije potreban kompletan operativni sistem, te bi u tim slučajevima izvršavanje koda kernela i virtuelizacija potencijalno bili kritični za održanje odziva u realnom vremenu. U tom slučaju se pribegava izvršavanju tkz. "*bare metal*" aplikacije, tj. već prevedenog jednostavnog mašinskog koda direktno na procesoru.

U ovom radu će biti ciljan čisto fizički memorijski sistem, dok bi bilo potrebno uvođenje dodatnog, prethodno pomenutog hardvera kako bi se obezbedila podrška za virtuelizaciju i rad savremenih operativnih sistema.

2.2 Tipovi memorija i hijerarhija memorijskog sistema

Sa tačke gledišta procesora, idealna memorija bi imala sledeće osobine:

- 1) Beskonačan kapacitet
- 2) Beskonačan propusni opseg
- 3) Bez kašnjenja
- 4) Besplatna

U tehnologijama koje trenutno posedujemo za izradu memorija, ovi zahtevi se protive jedan drugom. Veće memorije imaju manju maksimalnu brzinu rada. Sa linearnim povećanjem broja bajtova koji se skladište, usložnjava se logika za dekodovanje $\sim \log(n)$, te se smanjuje lokalnost i brzina propagacije signala usled većeg faktora grananja. Brža memorija je skuplja zbog kompleksnije tehnologije izrade. Veći propusni opseg zahteva više banaka, više portova, veću frekvenciju rada ili bržu tehnologiju - što ga čini skupljim.

Dve aktuelne tehnologije izrade brze, privremene (*eng. volatile*) memorije: SRAM i DRAM, demonstriraju ove konflikte u savremenim računarima. Čelija statičke (*eng. Static*) RAM se sastoji od dva invertora u pozitivnog sprezi gde se čuva bit informacije, te zajedno sa dva dodatna pristupna tranzistora za upis i čitanje sa linija, zahteva šest tranzistora po bitu u memoriji. Nasuprot tome, dinamička (*eng. Dynamic*) RAM-a čuva bit informacije u naelektrisanju kondenzatora, te je potreban samo jedan tranzistor po ćeliji kako bi se pročitala memorisana vrednost. Prisustvo kondezatora, te potreba za periodičnim osvežavanjem usled destruktivnog čitanja i curenja naelektrisanja kroz parazitivnu otpornost, čini ovu memoriju umnogome sporijom od statičke. Međutim, budući da se koristi samo jedan tranzistor-kondenzator par po bitu, gustina dinamičke memorije je skoro šest puta veća. Stoga je moguće dobiti veći kapacitet po jedinici površine silicijuma iz čega sledi i manja cena. Danas jedan GB statičke RAM košta u desetinama hiljada dolara dok ista veličina dinamičke RAM košta približno deset dolara [3].

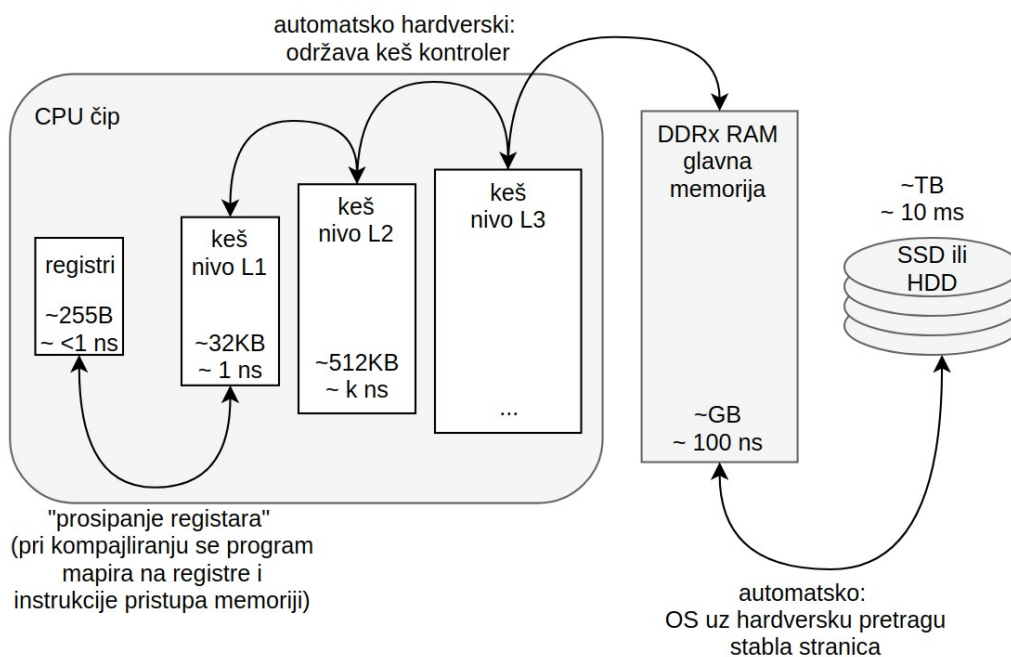
Ukoliko pogledamo "idealnu" memoriju, ona bi imala malo kašnjenje i veliki kapacitet. Ovo danas nije moguće ostvariti pomoću jedne vrste memorije. Kao i sa svim kompromisima u dizajnu hardvera pribegava se optimizaciji najčešćeg slučaja, te se iskorištavaju neke očekivane osobine programa kako bi se dizajnirao memorijski sistem.

Tipični programi će imati jako izražena svojstva prostorne i/ili vremenske zavisnosti pristupa memoriji - što znači da će procesor referencirati isti set memorijskih lokacija u kratkom vremenskom intervalu.

1. Vremenska lokalnost se izražava u tome, da ukoliko se pristupi nekom resursu, velika je verovatnoća da će u narednom kratkom vremenskom periodu ponovo pristupiti tom istom resursu.
2. Prostorna lokalnost se izražava u tome, da ukoliko se pristupi nekom resursu, postoji velika verovatnoća da će se u nekom kraćem vremenskom intervalu pristupiti resursu koji se nalazi na bliskoj memorijskoj lokaciji u odnosu na prvi.

Ove zavisnosti proizlaze iz kombinacije sledećih faktora. Načina izvršavanja programa - sekvencijalno izvršavanje instrukcija koje se skladište u memoriji jedna nakon druge. Strukture programa - gde se zavisni podaci smeštaju na bliskim memorijskim lokacijama. Tipičnih konstrukata programskih jezika kao što su petlje koje povećavaju vremensku lokalnost te linearnih struktura podataka (nizovi, strukture, unije) koje povećavaju prostornu lokalnost. Memoizacije - pamćenja rezultata zahtevnih proračuna za ponovnu upotrebu, itd.

Još 1965. godine je Vilks u radu "*Slave Memories and Dynamic Storage Allocation*" predložio da, ukoliko se iskoristi manja a brža pomoćna (eng. *slave*) memorija za čuvanje često referenciranih podataka, u praktčnim slučajevima će efektivna brzina pristupa memoriji biti bliža brzoj memoriji nego sporijoj [4]. Ovaj koncept je zatim implementiran u IBM 360/85 računar, gde je postojala pomoćna keš (skrivena, eng. *cache*) memorija od 16KB. Kako bi se dodatno, sem vremenske, iskoristila i prostorna lokalnost, podaci su se iz operativne u keš memoriju prebacivali u blokovima od 64B (bajtova) [5].



Slika 2: Memorijski podsistem

Od osnivanja ideje o keširanju podataka, svaki savremeni memorijski sistem se deli na više nivoa, gde se manje a brže memorije postavljaju što bliže procesoru, dok su veće a sporije memorije na periferiji. Danas, tipičan procesor ima nekoliko nivoa keša (*eng. cache levels*) koji se nalaze na istom čipu kao procesor, implementirani u SRAM tehnologiji. Niži nivoi su manjeg kapaciteta, optimizovanije fabrikacije tranzistora (veća brzina, potrošnja i površina) te su bliži procesoru za manje kašnjenje na linijama. Ukoliko se radi o višejezgarnom procesorskom čipu, jedan nivo keša (često nivo 3), je zajednički - deljen između jezgara. Slično kao i u prethodno pomenutom slučaju virtualizacije memorije, uz pomoć dodatnog hardvera (keš kontrolera) proces prebacivanja memorije između različitih nivoa keševa i operativne memorije se izvršava automatski, bez potrebe intervencije programera. Na osnovu svega prethodno pomenutog, tipičan memorijski sistem izgleda kao na slici 2.

Za datu memorijsku hijerarhiju, gde t_i predstavlja tehnološki zavisno kašnjenje pristupa memoriji na nivou hijerarhije i (krećući od strane procesora), realno vreme označeno sa T_i je veće jer zavisi i od ostalih nivoa hijerarhije memorije. Vreme T_i zavisi od udela "pogodaka" h_i (*eng. hit rate*) i "promašaja" m_i (*eng. miss rate*), za dati nivo memorije. Ovi brojevi govore koliko se često traženi podatak nalazi u memoriji na nivou i . Ukoliko se desi pogodak, procesor može da preuzme podatak iz memorije na nivou i , a u suprotnom se prelazi na nivo $i+1$. Na sličan način se redom ispituju nivoi $i+2 \dots n-1$ dok se ne pronađe željeni podatak. Iz ovoga proizlaze formule 1, 2 i 3.

$$h_i + m_i = 1 \quad (1)$$

$$T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1}) \quad (2)$$

$$T_i = t_i + m_i \cdot T_{i+1} \quad (3)$$

Formula broj 3 se dalje može rekursivno rastavljati zamenom člana T_{i+1} sa zavisnošću sa nivoom $i+2$. Za poslednji nivo hijerarhije memorije važi da je $h_{n-1} = 1$ dok je $m_{n-1} = 0$ jer se podatak mora nalaziti u njoj, te je $T_{n-1} = t_{n-1}$. Cilj pri dizajniranju sistema za keširanje je dobiti željeno vreme T_i unutar dozvoljenog budžeta.

Za implementaciju keširanja na Zybo razvojnoj ploči se mogu iskoristiti dva tipa memorijskih resursa koji su prisutni u programabilnoj logici:

- Blok RAM
 - postoji 60 blokova, gde se svaki može konfigurisati kao dvoprístupna memorija veličine 36Kb ili dve jednopristupne memorije veličine 18Kb [6]. Tip čitanja se može postaviti na *no_change*, *read_first* ili *write_first*. Čitanje memorijske lokacije se vrši na rastuću ivicu takta. Može se uključiti dodatan izlazni registar, čime se unosi dodatan takt kašnjenja pri čitanju, ali se dobija manje *clk-to-output* kašnjenje. Ukupno je na raspolaganju 2.1Mb obog tipa memorije.
- Distribuirani (LUT RAM)
 - Od prisutnih 17,600 *slice* modula, 6000 su tipa *slicem*, dok je ostatak tipa *slicel* [7]. Svaki *slicem* modul se može konfigurisati kao 64 bita brze RAM memorije [8]. Čitanje iz ove memorije je asinhrono, ali se može napraviti izlazni registar za

bolju *clk-to-output* karakteristiku. Ukupno je na raspolaganju 375Kb ovog tipa memorije.

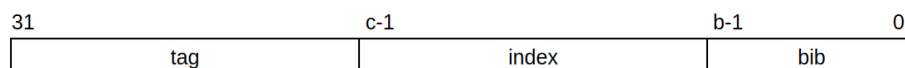
Brza a manja distribuirana (LUT) RAM se može iskoristiti za prvi nivo, dok sporija a veća blok RAM se može iskoristiti za drugi nivo keša.

Arhitektura (ISA) procesora fiksira dve veličine: adresni prostor (samim tim i širinu adrese) što je često 2^{32} ili 2^{64} , te granularnost memorije tj. maksimalnu širinu podatka što je 32 ili 64 bita. Ove dve veličine dele procesore na 32-bitne i 64-bitne. Ovaj rad će se baviti implementacijom 32-bitnog RISC-V procesora, te možemo usloviti da je veličina podataka $W=32$ bita, i da je širina adrese $m=32$ bita. Memorija u RISC-V arhitekturi je bajt-adresibilna, tj. minimalni podatak koji se može adresirati je jedan bajt. Kako su podaci veličine 4 bajta, to znači da će dva najniža bita u adresi ukazivati na specifičan bajt unutar podatka (reči).

2.3 Direktno preslikan keš

Kao što je pomenuto u poglavlju 2.2, podaci se između nivoa memorijske hijerarhije prenose u blokovima. Veličina bloka od B bajtova, znači da će nižih $b=\log_2 B$ bita u adresi referencirati bajt u bloku (*eng. byte in block, bib*). Sa veličinom keša od C bajtova, širina adrese keš memorije će biti $c=\log_2 C$ bita. Nižih c bita u adresi traženog podatka će indeksirati lokacije u kešu. Kod direktno preslikanog keša, kada se blok preuzima iz operativne memorije (*eng. fetch*), on se smešta na memorijske lokacije u kešu koje se poklapaju sa donjih c bita adrese bloka.

Može se zaključiti da postoji $2^{(31-c)}$ različitih blokova koji se mogu nalaziti na istoj lokaciji u kešu. Kako bi se znalo koji je blok trenutno na nekoj lokaciji u kešu, kada se on preuzima operativne memorije, gornjih $(31-c)$ bita pod nazivom "tag" se čuva u pomoćnoj memoriji za čuvanje tagova (*eng. tag store*). Ova memorija se često implementira u istoj tehnologiji izrade kao i keš (SRAM). Za svaki blok u keš memoriji kojih ima $2^{(c-b)}$, potrebna je jedna lokacija memoriji za čuvanje tagova. Širina te lokacije je širina tag polja $(31-c)$, te dodatan bit koji naznačava da li je taj tag validan.

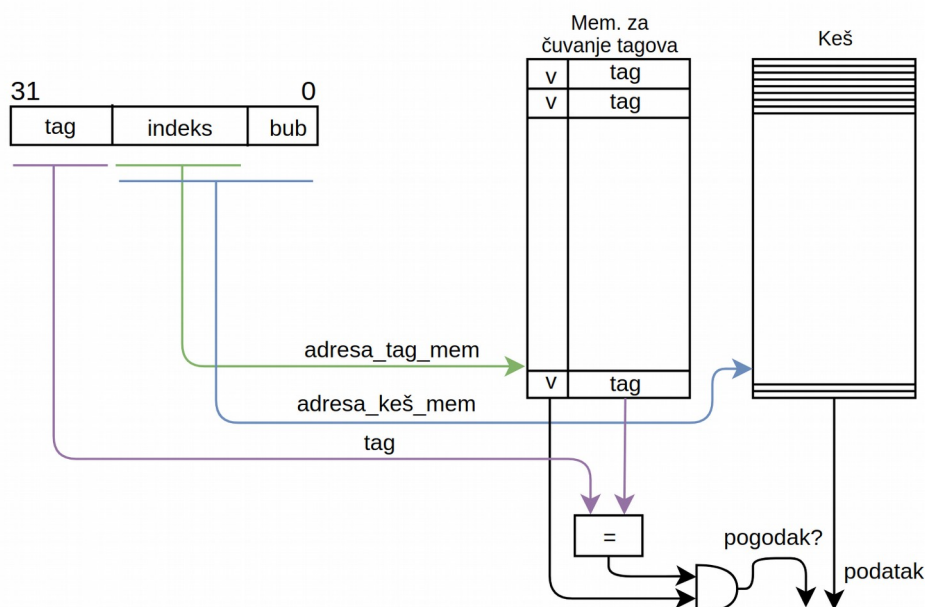


Slika 3: Podela adrese u polja

Ukoliko se c bita koristi za indeksiranje keš memorije, i b bita za indeksiranje bajta u bloku, veličina keš memorije je $8 \cdot 2^c$ bita a memorije za čuvanje tagova $(32-c) \cdot 2^{(c-b)}$ bita. Odnos bloka i veličine keš memorije se bira tako da se zadovolji vremenski zahtev T_i , a da udeo SRAM memorije utrošene na keš u odnosu na čuvanje taga, bude što veći.

Direktno mapirani keš funkcioniše na način koji je prikazan na slici 4. Kada se podatak referencira pomoću neke adrese, donjih c bita adresira keš memoriju, dok polje *index* adresira memoriju za čuvanje tagova. Iz memorije za čuvanje tagova se čita sačuvano tag polje i *valid* bit. Ukoliko se pročitani tag poklapa sa tagom u adresi, te je valid bit jednak jedinici, sledi da je pročitani podatak iz keš memorije validan - desio se pogodak (*eng. cache hit*). Procesor preuzima pročitani podatak i nastavlja izvršavanje. Ukoliko se pročitani tag ne

poklapa sa tagom u adresi, ili je valid bit jednak nuli - desio se promašaj (*eng. cache miss*). Procesor zaustavlja izvršavanje, dok se blok ne preuzme iz memorije i upiše u keš. Na adresi *index* u memoriji za čuvanje tagova se ažurira polje *tag*, te se *valid* bit postavlja na jedinicu. Čim se blok upiše, procesor preuzima referencirani podatak i nastavlja sa izvršavanjem. Kod memorijskih hijerarhija sa više nivoa keševa, tagovi se mogu paralelno čitati i porediti sa sačuvanim tagovima različitih nivoa keša, ili se može pristupiti čitanju taga iz sledećeg nivoa tek nakon što se nesio promašaj u prethodnom. Prvi način povećava performans ali uveliko i potrošnju energije.



Slika 4: Direktno preslikan keš

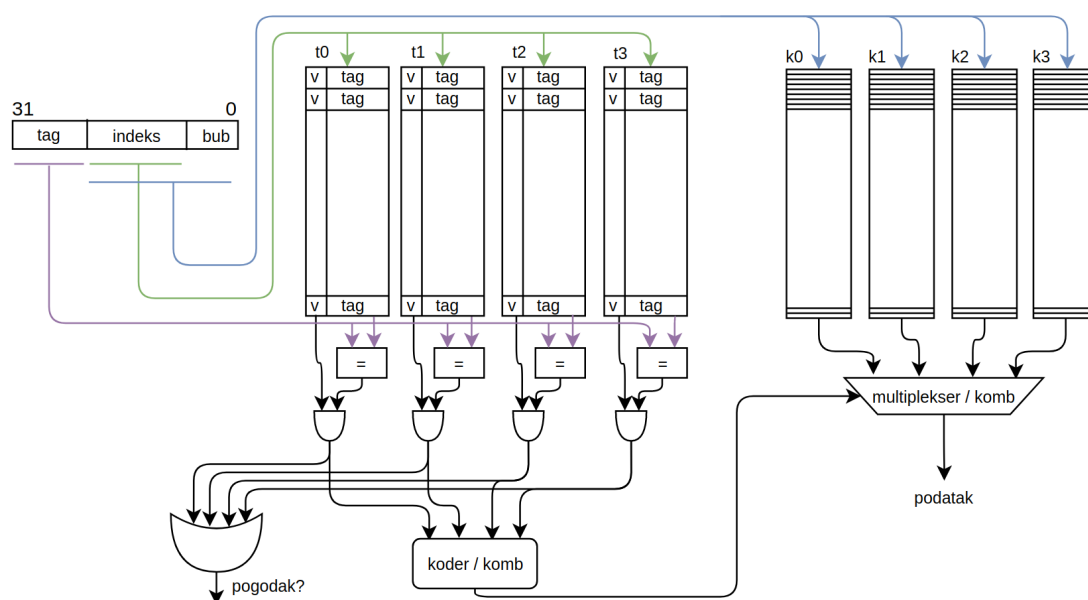
U ovom radu, prvi nivo keša će biti direktno mapiran. Jedina prednost direktnog mapiranja je minimalna dodatna logika, te najveća brzina rada. Ovo malo propagaciono kašnjenje logike za detekciju pogodaka će se iskoristiti zajedno sa brzinom odziva distribuiranog RAM-a na FPGA čipu kako bi se napravio prvi nivo keša koji u istom taktu proverava pogodak te obezbeđuje podatak bez zaustavljanja protočne obrade. Najveća mana keširanja sa direktnim preslikavanjem je da se lako može napraviti sekvenca pristupa memoriji koja proizvodi udeo pogodaka jednak nuli. To je situacija kada se naizmenično pristupa dvema memorijskim lokacijama sa istim indeksom. Ova situacija će biti prikazana u simulaciji u poglavlju 3.2.2.

2.4 Set asocijativan keš

Kako bi se izbegle sekvence pristupa koje lako dovode do nultog udela pogodaka, keš se često pravi da bude set asocijativan. Keš se podeli u više manjih jednakih delova, svaki sa svojom memorijom za čuvanje tagova. Ukoliko se podeli na N delova, u istom trenutku može da čuva N podataka sa istim indeksom. Za ovakav keš se kaže da je N -smerni set asocijativan keš. Sa većom asocijativnošću se udeo pogodaka povećava na račun dodatne logike i brzine rada. Potrebno je imati N komparatora za poređenje tagova, te je potrebno pomoću dodatne kombinacione logike proslediti podatak iz jednog od N smerova. Na slici broj 5 je prikazan

čtetvorosmerni set asocijativan keš. Očigledno je da se sa većom asocijativnošću povećava kritična putnja zbog dodatnog multipleksiranja signala. Ekstremni slučaj ove metode je potpuno asocijativan keš, gde bi svaki smer čuvao samo jedan blok. U praksi se ekstremni slučajevi asocijativnosti ne koriste, jer se takve sekvence pristupa memoriji retko ili nikada ne sreću, te nisu vredni dodatne logike i kašnjenja. U nastavku rada će biti implementiran parametrizovani N-smerni keš, sastavljen od blok RAM modula na FPGA čipu.

Sem dodatne kombinacione logike, usložnjava se i keš kontroler. Potrebno je implementirati logiku koja u situaciji kada se u svih N smerova nalaze validni blokovi, te se zatraži $N+1$ blok, na određeni način bira jedan od N blokova koji će biti izbačen (evikcija ili zamena). Ova dodatna logika često zahteva i uvođenje dodatnih bita u memoriji za čuvanje tagova.



Slika 5: Četvorosmerni set asocijativan keš

2.4.1 Polisa evikcije

Kada se keševi inicijalizuju, svi blokovi su nevalidni, te se N smerova popunjava redno. Može se desiti da se i u toku rada programa neki od blokova invalidira. Sve polise evikcije se pridržavaju pravila da ukoliko postoji nevalidan blok, on će biti zamenjen kada se zatraži novi blok. Ukoliko su svi blokovi u N smerova validni, onda se usvaja set pravila na osnovu kojih se odlučuje koji od blokova će biti zamenjen.

Neke od polisa su:

- 1) Najdavnije korišten (eng. *Least Recently Used*)
- 2) Nasumično (eng. *Random*)
- 3) Ne nedavno korišten (eng. *Not Most Recently Used, Pseudo LRU*)
- 4) Najređe korišten (eng. *Least Frequently Used*)
- 5) Hibridni

Najočiglednija polisa je izbaciti najdavnije korišten blok (*LRU*), međutim implementacija ove polise postaje da bude problem za keševe s većom asocijativnošću. Potrebno je pamtiti redosled pristupa blokovima, te čuvati redosled referenciranja. Sa rastom asocijativnosti raste i potreban broj bita za praćenje redosleda te je pri svakom referenciranju potrebno ažurirati bite svih smerova što takođe postaje energetski neefikasno. Ispostavlja se da nasumičan način evikcije pokazuje bolje rezultate od *LRU* kada je aktivni set podataka veći od asocijativnosti (*eng. set trashing*). U praksi se pokazuje da udeo pogodaka zavisi od programa na kojima se testira, te je prosečan udeo pogodaka sličan za *LRU* i nasumični. Iz ovog razloga se pribegava tehnikama koje predstavljaju kombinaciju *LRU* i nasumičnog algoritma koje se zovu "*Not MRU*" ili "*Pseudo LRU*" polise. Jedna od poznatijih polisa iz ove grupe je hijerarhijska, koja aproksimira *LRU* pomoću manje bita. U narednim poglavljima će se za *N*-smerni asocijativan keš implementirati pseudo *LRU* polisa pod imenom "Žrtva - Sledeća žrtva" (*eng. Victim - Next victim*).

Od *N* blokova u asocijativnom kešu samo se prate dva bloka: jedan označen kao žrtva (*eng. victim*) i jedan označen kao sledeća žrtva (*eng. next victim*), dok su svi ostali blokovi obični (*ordinary*). Polisa prati sledeća pravila:

- Kada se desi promašaj:
 - Blok označen sa "žrtva" se eviktuje, te se novi blok postavlja na njegovo mesto
 - Blok označen sa "sledeća žrtva" postaje "žrtva"
 - Nasumičan obični blok se označava sa "sledeća žrtva"
- Kada se desi pogodak sa blokom označenim kao "žrtva"
 - Blok označen sa "žrtva" postaje obični blok
 - Blok označen sa "sledeća žrtva" postaje "žrtva"
 - Nasumičan obični blok se označava sa "sledeća žrtva"
- Kada se desi pogodak sa blokom označenim kao "sledeća žrtva"
 - Blok označen sa "sledeća žrtva" postaje običan blok
 - Nasumičan obični blok se označava sa "sledeća žrtva"
- Kada se desi pogodak sa običnim blokom
 - Nije potrebno ažurirati blokove

2.4.2 Polisa upisa

Bitna odluka pri dizajniranju keševa je na koji se način rukuje upisima podataka u keš. Kada procesor izvrši "*store*" naredbu, te se desi pogodak, on promeni sadržaj nekog bloka u najnižem nivou keša. Pitanje je: kada ažurirati sledeći nivo keša sa novim podatkom? Ukoliko se keš dizajnira tako da se promeni sadržaj bloka samo najnižeg nivoa, dok blok zadržava prethodnu vrednost u sledećem nivou keša, sledeći nivo će biti ažuriran tek kada se taj blok eviktuje. Ovakav keš se naziva "upis-nazad" (*eng. write-back*) keš. Drugi način je da se pri upisu u blok nižeg nivoa keša, podatak takođe upiše i u sledeći nivo keša, tako da su podaci

sledećeg nivoa uvek saglasani sa prethodnim. Ovakav keš se naziva "upis-kroz" (*eng. write-through*) keš. Keširanje metodom "upis-nazad" podržava više upisa u isti blok pre nego što se blok eviktuje, te se podaci ažuriraju u sledećem nivou. To znači da se potencijalno štedi na protoku podataka između nivoa keševa a samim tim i količine utrošene energije. Međutim, za implementaciju ovakvog keša je potreban minimalno jedan dodatan bit po bloku, koji govori da li je on "prljav" tj. modifikovan, kako bi bilo moguće održati koherenciju između različitih nivoa keševa. Keširanje metodom "upis-kroz" je jednostavnije za implementirati zato što su keševi uvek ažurirani, te se pri evikciji blokova ne moraju proveravati keševi nižih nivoa. Sem veće potrošnje energije i opterećenja keševa viših nivoa, kod ove metode nema ni sjedinjavanja upisa - što znači da se više uzastopnih upisa na istu adresu, neće tretirati kao jedan upis u keš višeg nivoa.

Druga bitna odluka vezana za upise u keševe je da li se alociraju keš blokovi kada se desi promašaj pri instrukciji upisa. Pri metodi "alociraj pri promašaju upisa" (*allocate on write miss*), keš blok se preuzima iz keša višeg nivoa ili operativne memorije, te se nekon toga izvrši upis. Ova metoda omogućava da se rukuje promašajima upisa i čitanja na sličan način, te se pojednostavljuje logika keš kontrolera. Kod metode "ne alociraj pri promašaju upisa" se podatak direktno upisuje u sledeći nivo memorije, te se čuva prostor u kešu ukoliko je lokalnost upisa mala.

Iako je moguće napraviti keš sa četiri različita algoritma za upravljanje upisima, u praksi se koriste parovi:

- "upis-nazad" sa "alociraj pri promašaju upisa"
- "upis-kroz" sa "ne alociraj pri promašaju upisa"

Moguće je organizovati memorijski sistem da različiti nivoi keševa imaju različite polise za upis.

U ovom radu će postojati dva nivoa keša gde će oba biti metode "upis-nazad" i "alociraj pri promašaju upisa".

2.4.3 Polisa razdeljenosti

Većina savremenih procesora zahteva da se na svaku rastuću ivicu takta iz memorije preuzme instrukcija koju je potrebno dekodovati, a sem toga, potencijalno dodatan pristup memoriji ukoliko se radi o instrukciji upisa ili čitanja. Iz tog razloga, kako bi se izbegli prekidi rada procesora, mora se obezbediti interfejs ka memoriji koji omogućava dva čitanja ili čitanje i upis svaki takt. Stoga se keš može implementirati kao jedna dvoprístupna memorija ili kao dve fizički odvojene (za instrukcije i podatke). Prvi način implementacije se naziva ujedinjeni (*eng. unified*), a drugi se naziva razdeljeni (*eng. split*) keš. U savremenim procesorima, skoro uvek je memorijski sistem takav, da je najniži nivo keša razdeljen, dok su ostali ujedinjeni. Većina modernih procesora ima implementiranu protočnu obradu podataka, te se pri postavljanju komponenti na čip i optimizaciji rutiranja, desi da je logika faze za prihvatanje instrukcija na čipu udaljena od faze za upis/čitanje podataka. Razdeljeni keš omogućava da se keš za čuvanje instrukcija smesti na čipu blizu logike za prihvatanje i dekodovanje instrukcije. Na sličan način je moguće smestiti keš za čuvanje podataka blizu faze za pristup podacima u memoriji. U praksi je blizina logike na čipu ključna za ostvarivanje visokih performansi, do te

mere, da se danas i ne može naći procesor sa ujedinjenim prvim nivoom keša. Još jedan razlog za korišćenje razdeljenog keša u prvom nivou proizlazi iz toga da dodatna kombinaciona logika za implementaciju dvoprístupne memorije kod unificiranog keša usporava odziv memorije što danas nije prihvatljivo za prvi nivo keša. Razdeljeni keš takođe neutrališe mogućnost da česte reference na instrukcije preoptereće keš, te krenu da eviktuju blokove podataka, ili da se desi suprotan slučaj.

Mana razdeljenog keša je da ukupan prostor u kešu ne bude efikasno iskorišten. Može se desiti da se isti podatak nalazi u oba keša. U simulaciji se prikazuje da ujedinjeni keš ima bolje udele pogodaka od razdeljenog keša istog kapaciteta.

Sem toga, samomodifikujući (eng. *self-modifying*) kod može dovesti do problema sa koherencijom. Naime, ukoliko program ima mogućnost da modifikuje instrukcije u memoriji u toku izvršavanja programa, ovakav kod se naziva samomodifikujući. Razdeljeni prvi nivo keša, u kombinaciji sa metodom upisa "upis-nazad", uslovljava da promene u kešu za podatke neće biti vidljive u kešu za instrukcije. Dakle, pri izvršavanju samomodifikujućeg koda, promena vrednosti mašinske instrukcije će se desiti samo u kešu za podatke. Ukoliko arhitektura (ISA) zahteva da se ove situacije razreše u hardveru, potrebno je ubaciti dodatnu logiku što uslošnjava keš kontroler kao i memorije koje se koriste za implementaciju keša. RISC-V arhitektura ima opušteniji memorijski model, te ne zahteva implicitno razrešavanje problema koje stvara samomodifikujući kod. Jedini zahtev je implementacija instrukcije "*fence.I*" koja garantuje da će nakon njenog izvršavanja u kešu za instrukcije videti novije, modifikovane, vrednosti.

Na FPGA čipovima veliki problem predstavlja rutiranje, koje je mnogo ograničenije nego u ASIC implementaciji - zbog mnogo manjeg izbora za postavljanje komponenti te ograničenog broja kanala za rutiranje. Zbog ovoga se na rutiranju često gubi veći deo performansa implementirane logike. Iz ovih razloga je potencijalno dobar izbor da se prvi nivo keša implementira kao razdvojeni, dok će drugi nivo biti unificiran. Razdvojeni keš će imati podjednak kapacitet za instrukcije i podatke. Ova podjednaka raspodela prostora u praksi pokazuje najbolje prosečne udele pogodaka [9].

2.4.4 Polisa inkluzije keša

Kod inkluzivnog keša, prisustvo bloka u nižem nivou keša implicira postojanje kopije istog bloka u višem nivou keša. Kod ekskluzivnog keša, može se desiti da blok postoji u nižem nivou keša, a da isti blok nije prisutan u višem nivou keša. Inkluzivan keš žrtvuje ukupan kapacitet keša zbog dobiti na jednostavnosti implementacije keš kontrolera.

Kod procesora sa više jezgara, inkluzivan keš umnogome pojednostavljuje logiku za deljenje podataka. Ukoliko jedno jezgro na deljenoj magistrali napravi zahtev za određeni blok, druga jezgra mogu samo proveriti prisustvo tog bloka u najvišem nivou keša. Ukoliko blok nije u višem nivou, nije prisutan ni u nižim. Ovo čini inkluzivne keševe skalabilnijim za implementaciju multiprocesorskih čipova. Zbog dupliciranja blokova, udeo pogodaka u višim nivoima keša opada. U ovom radu će biti implementiran inkluzivan keš.

3 Modelovanje sistema za keširanje

3.1 RISC-V jezgro

Memorijski podsistem će se implementirati za jednostavno RISC-V procesorsko jezgro koje podržava set instrukcija RV32I. Procesor je 32-bitni, poseduje 32 registra, te je implementiran osnovni set instrukcija za rad sa celim brojevima (*eng. Integer*, stoga oznaka I). U procesoru je implementirana protočna obrada podataka, gde se instrukcije izvršavaju u pet faza. Detaljnija struktura procesora se može videti na slici 6.

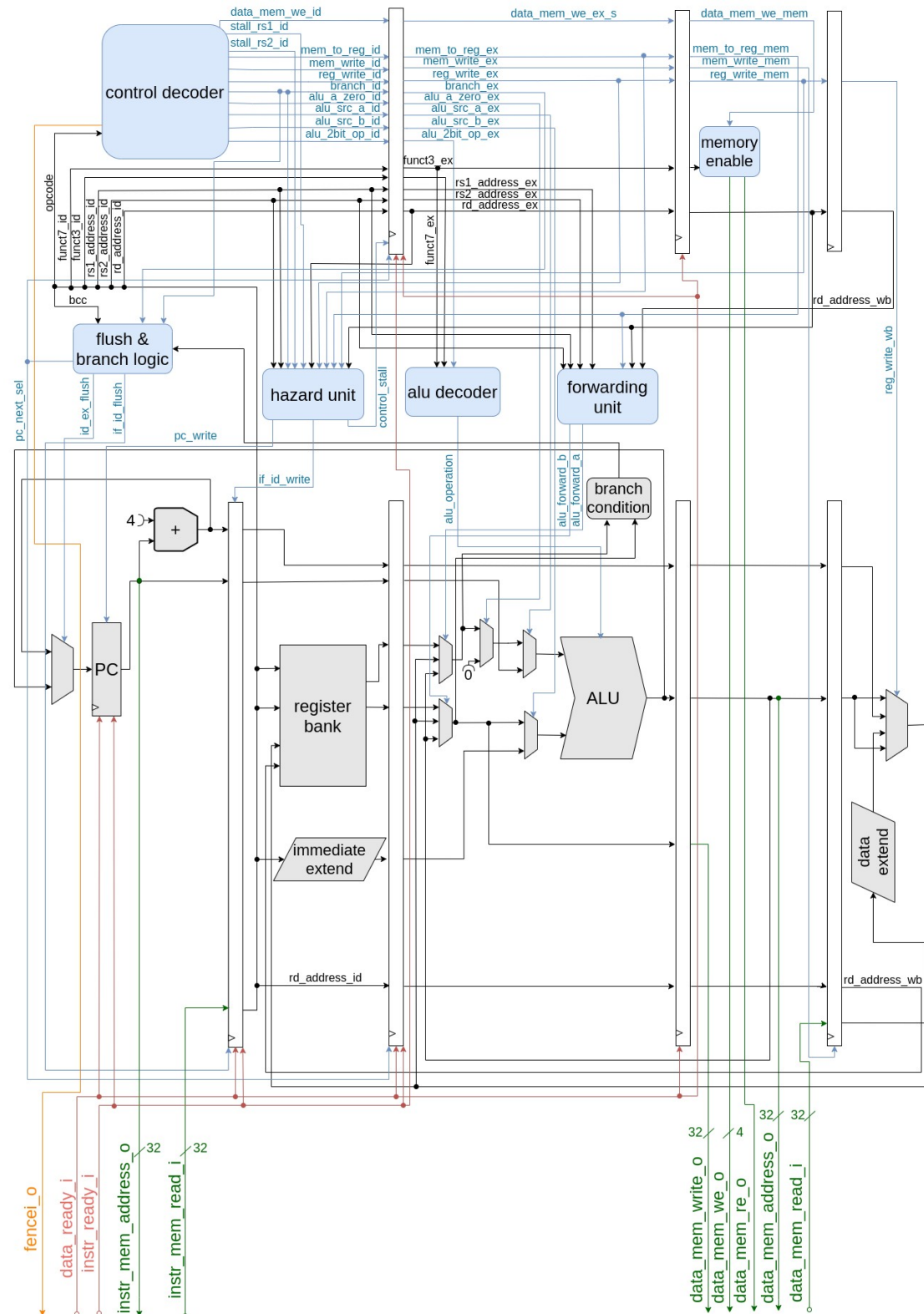
Faze protočne obrade i oznake koje će se koristiti za njih u nastavku teksta, su:

1. Faza prihvata instrukcije (*eng. Instruction fetch*, IF)
2. Faza dekodovanja instrukcije (*eng. Instruction decode*, ID)
3. Faza izvršavanja instrukcije (*eng. Execute*, EX)
4. Faza pristupa memoriji za podatke (*eng. Memory*, MEM)
5. Faza upisa u registarsku banku (*eng. Writeback*, WB)

Interfejs procesora sa memorijskim podsistemom se sastoji od sledećih portova:

- Globalnih signala: *clk* (*clock*, takt), *ce* (*clock enable*, *dozvola takta*) i *reset* (sinhroni reset, aktivan na nuli) koji su dovedeni do svih sekvencijalnih elemenata u procesoru.
- Interfejs sa memorijom za instrukcije:
 - *instr_mem_address_o* - za adresiranje memorije za instrukcije (izlazni, 32-bitni).
 - *instr_mem_read_i* - za preuzimanje instrukcija iz memorije (ulazni, 32-bitni).
- Interfejs sa memorijom za podatke:
 - *data_mem_address_o* - za adresiranje memorije za podatke (izlazni, 32-bitni).
 - *data_mem_read_i* - za preuzimanje instrukcija iz memorije (ulazni, 32-bitni).
 - *data_mem_write_o* - za upis podataka u memorije za podatke (izlazni, 32-bitni).
 - *data_mem_we_o* - signal dozvole upisa, (izlazni, 4-bitni, jedan bit po bajtu u podatku koji se upisuje u memoriju).
 - *data_mem_re_o* - signal dozvole čitanja (izlazni, 1-bitni,).
- Pomoćni signali:
 - *instr_ready_i* - signalizira da je memorija za instrukcije spremna za čitanje adresirane instrukcije, tj. desio se pogodak u kešu za instrukcije (ulazni, 1-bitni).
 - *data_ready_i* - signalizira da je memorija za podatke spremna za upis ili čitanje na adresiranu lokaciju, tj. desio se pogodak u kešu za podatke (ulazni, 1-bitni).

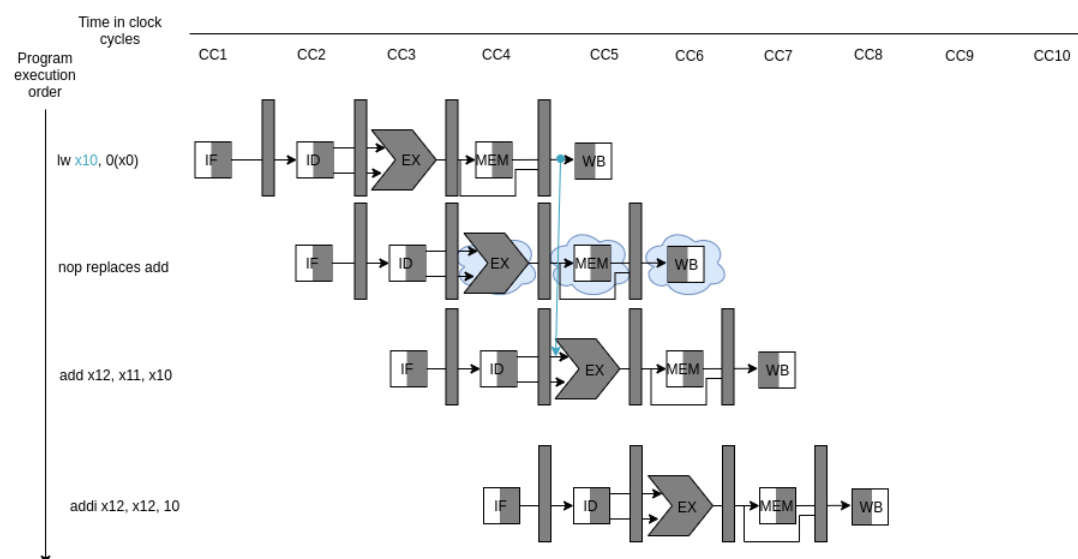
fencei_o - signalizira da se upravo dekodevala *fence.i* instrukcija, te da memorijski podsistem treba da preuzme mere za održanje koherencije između memorije za instrukcije i memorije za podatke (izlazni, 1-bitni).



Slika 6: RV32I procesorsko jezgro

Svi prethodno pomenuti portovi se mogu videti u donjem delu slike 6. Portovi za interfejs ka memoriji su označeni zelenom bojom, *ready* signali su označeni crvenom, dok je *fencei* pomoćni signal označen narandžastom bojom. Kao što se može zaključiti sa slike, prvi nivo memorije (keš za instrukcije i podatke) će biti implementirani pomoću memorija sa asinhronim čitanjem. To znači da će se pročitane instrukcije i podaci, po preuzimanju iz keša, dovesti direktno na ulaze registara koji odvajaju IF/ID i MEM/WB faze respektivno.

Uslovni i bezuslovni skokovi se izvršavaju kada instrukcija skoka dođe u EX fazu, gde se pomoću ALU jedinice računa ciljna adresa, a pomoću *branch* komparatora uslov. Ukoliko se desi skok, registri koji dele IF/ID i ID/EX faze se resetuju (*eng. flush*). Mnogi "čitanje nekon upisa" (*eng. read after write, RAW*) hazardi podataka se razrešavaju pomoću logike za prosleđivanje podataka (*eng. forwarding*). Međutim, na ovaj način nije moguće rešiti sve slučajeve. Na primer, ukoliko se desi da je instrukcija čitanja (*lw*), praćena bilo kojom instrukcijom koja kao operand koristi ciljni registar instrukcije čitanja, dešava se situacija gde se podatak ne može proslediti jer još uvek nije preuzet iz memorije, kada je potreban ALU jedinici. IF, ID i EX faze protočne obrade se moraju zadržati jedan takt, kako bi u sledećem taktu pročitani podatak bio prosleđen na ulaz ALU jedinice (videti sliku 7) [10]. Zadržavanje protočne obrade narušava performans procesora, za koji se teži da sve faze budu aktivne u svakom trenutku. Takođe, što ima više faza u protočnoj obradi, to su ove situacije češće, a trajanja zadržavanja procesora duža.



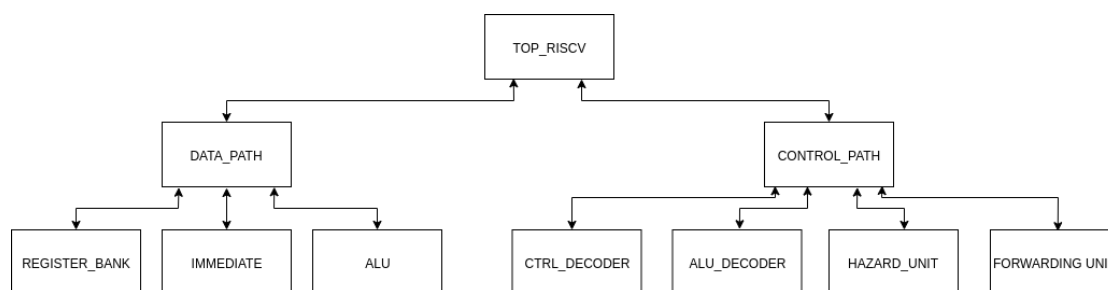
Slika 7: Razrešavanje hazarda zaustavljanjem protočne obrade

Zadržavanja procesora su neophodna i kada se desi promašaj u prvom nivou keša za instrukcije ili podatke. U tim slučajevima će zadržavanje procesora trajati sve dok se podatak ne preuzme iz sledećeg nivoa keša ili operativne memorije. Naravno, trajanje zadržavanja će biti mnogo manje kada se podatak preuzima iz drugog nivoa keša, što će biti demonstrirano u narednim poglavljima.

Ukoliko se desi promašaj u kešu za instrukcije ili podatke, nema potrebe zaustavljati sve faze izvršavanja protočne obrade, već samo one koje zavise od tog podatka. Na primer, ukoliko se

desi promašaj na kešu za instrukcije, a desi se pogodak u kešu za podatke, može se zaustaviti samo faza IF dok se faze ID, EX, MEM i WB mogu izvršiti bez smetnje. Takođe, ukoliko se desi promašaj u kešu za podatke, moraju se zaustaviti IF, ID, EX i MEM faze, dok se WB faza može izvršiti. Delimično zaustavljanje protočne obrade može povećati performans procesora, jer dok se čeka na keš za instrukcije, moguće je da će se nastavkom izvršavanja ostatka protočne obrade, razrešiti RAW hazardi koji bi zahtevali dodatno zaustavljanje procesora od jedan takt. Na ovaj način se čekanje na keš za instrukcije i čekanje na čitanje iz keša za podatke mogu preklopiti, te se uštediti jedna perioda takt signala. Ova metoda ima veće dobiti kod procesora sa više od pet faza protočne obrade. Iako će se ovde memorijski podsistem testirati na procesoru sa pet faza protočne obrade, procesoru će se slati dva posebna signala: jedan za informaciju da li se desio pogodak u kešu za instrukcije (*instr_ready*) i drugi za pogodak u kešu za podatke (*data_ready*).

Prethodno opisano procesorsko jezgro je modelovano u VHDL jeziku, sa hijerarhijom prikazanom na slici 8. Nakon preliminarne sinteze i implementacije se pokazuje da ovaj model, na Zybo ploči, može raditi na približno 100MHz, sa utroškom 1475 šestoulaznih lukap tabela i 1540 registara. Ovi rezultutati variraju u zavisnosti strategija sinteze i implementacije, te se može vršiti kompromis između količine zauzetih resursa i maksimalne frekvencije rada procesora.



Slika 8: Hijerarhija VHDL modela RISCv procesora

3.2 Memorijski podsistem

Ovaj rad će izvršiti implementaciju keširanja za prethodno opisano RV32I procesorsko jezgro na razvojnoj ploči Zybo, kompanije Digilent. Programabilna logika se nalazi na Zynq-7000 SoC-u, te je ekvivalentna po performansama Artix 7 familiji FPGA čipova kompanije Xilinx. Zybo ploča poseduje LPDDR3 (*eng. Low Power Dual Data Rate*) memorijski čip veličine 512MB, širine podataka 32 bita i maksimalnom brzinom prenosa od 1050Mbps. Ovaj memorijski čip će biti operativna memorija. Sa ovom memorijom je moguće komunicirati pomoću AXI Full interfejsa.

Za potrebe simulacije i prvobitnog testiranja sistema za keširanje, ova memorija će biti modelovana pomoću jednostavnog interfejsa koji se sastoji od sledećih portova:

- `addr_phy_o` - port za adresiranje memorije (izlazni, 32-bitni)
- `dread_phy_i` - port za čitanje podataka iz memorije (ulazni, 32-bitni)
- `dwrite_phy_o` - port za upis podataka u memoriju (izlazni, 32-bitni)

- `we_phy_o` - signal dozvole upisa podataka u memoriju (izlazni, jednoitni)

Ovaj interfejs će biti zamenjen AXI full interfejsom pri završnim fazama implementacije sistema (pakovanju modela u IP jezgro).

Prvo će biti modelovan i simuliran sistem sa direktno mapiranim drugim nivoom keša, a zatim će se model proširiti sa uvođenjem asocijativnosti.

3.2.1 Direktno mapiran sistem za keširanje

Kako bi se krajnjem korisniku omogućila fleksibilnost po pitanju zauzetih resursa i performansa memorijskog podsistema, tokom modelovanja će određene karakteristike keša biti parametrizovane. Za sada se uvode tri osnovna parametra.

- `BLOCK_SIZE` - veličina keš bloka u bajtovima
- `LVL1_CACHE_SIZE` - veličina keševa prvog nivoa (za instrukcije i podatke)
- `LVL2_CACHE_SIZE` - veličina keša drugog nivoa

Prvi nivo keša će biti razdeljen, što znači da će keš za instrukcije i keš za podatke biti dve jednodostupne memorije. Svaki od njih će takođe imati sebi pridruženu memoriju za čuvanje tagova. Prvi nivo vodi polisu "upis-nazad" sa "alociraj pri promašaju upisa". Procesor ima mogućnost upisa podataka u keš za podatke, te će se u njemu pridruženoj memoriji za čuvanje tagova, sem *valid* bita, čuvati i *dirty* bit. Budući da procesor može da upisuje podatke veličine bajta u keš za podatke, ova memorija mora da ima mogućnost dozvole upisa svakog pojedinačnog bajta (*eng. byte-write*). Procesor nema mogućnost promene sadržaja keša za instrukcije, tako da memorija za čuvanje tagova neće imati *dirty* bit.

Bitovi validnosti kod keša za instrukcije neće biti deo memorije za čuvanje tagova, već će biti čuvani u registrima. Ova odluka je donešena zbog implementacije podrške za *fence.I* instrukciju, koja treba da postavi sve *valid* bitove keša za instrukcije na nulu. Stoga se *valid* bitovi čuvaju u registru čime se omogućava invalidacija keša u jednom taktu (reset registra). Ovo bi bilo nemoguće ukoliko bi se pomoćni bitovi nalazili na različitim lokacijama u memoriji. Nakon invalidacije keša, pri preuzimanju sledećeg bloka iz drugog nivoa, keš kontroler će prvo ažurirati blok u drugom nivou sa novim vrednostima iz keša za podatke, a zatim će se preuzeti ažurirani blok u keš za instrukcije. Postoje alternativne metode za implementaciju *fence.I* instrukcije, ali bi oni zahtevali dodatan utrošak lukap tabela. Budući da će prvi nivo keša biti malih dimenzija, opravdano je da se koriste registri koji su u ovom slučaju manje kritičan resurs.

Na osnovu svega prethodno pomenutog, prvi nivo keširanja će se sastojati od sledećih memorijskih modula:

1. Keš za instrukcije

Jednodostupna memorija sa asinhronim čitanjem. Širine podatka je 32 bita a broj podataka je `LVL1_CACHE_SIZE/4`.

2. Memorija za čuvanje tagova keša instrukcija

Jednoprístupna memorija sa asinhronim čitanjem. Broj podataka je $LVL1_CACHE_SIZE/BLOCK_SIZE$ (broj blokova u kešu) a širina podataka ($32 - \log_2(LVL1_CACHE_SIZE)$), što je širina taga za prvi nivo keša.

3. Keš za podatke

Jednoprístupna memorija sa asinhronim čitanjem i bajt adresibilnom dozvolom upisa (*eng. byte-write*). Širine podatka je 32 bita a broj podataka je $LVL1_CACHE_SIZE/4$.

4. Memorija za čuvanje tagova keša podataka

Jednoprístupna memorija sa asinhronim čitanjem. Broj podataka je $LVL1_CACHE_SIZE/BLOCK_SIZE$ (broj blokova u kešu) a širina podataka ($32 - \log_2(LVL1_CACHE_SIZE) + 2$), što je širina taga za prvi nivo keša, plus "valid" i "dirty" bitovi.

Sve četiri memorije će biti implementirane kao distribuirani (LUT) RAM, o kome je bilo više priče u poglavlju 2.2. Ova odluka je donešena iz razloga što se distribuirani RAM mapira na *slicem* komponente koje su podjednako raspoređene po programabilnoj logici FPGA čipa. Cilj je da kao i u ASIC implementaciji procesora, keševi prvog nivoa budu što bliže fazama protočne obrade koje zahtevaju čitanje iz memorija (IF i MEM). Na ovaj način je moguće smanjiti kašnjenje zbog rutiranja, te dobiti bolje performanse kompletnog sistema. Za razliku od distribuiranog RAM-a, blok RAM moduli se često na FPGA čipovima nalaze u jednoj ili više kolona, kako bi se pojedinačni blokovi lakše konkatanirali u jedan veći [8]. Iz ovih razloga će blok RAM biti iskorišten za drugi nivo keša.

Prebacivanje podataka između keševa se vrši u blokovima, te drugi nivo keša ne mora imati mogućnost upisa pojedinačnih bajtova. Drugi nivo keša, kao i prvi, vodi polisu "upis-nazad" sa "alociraj pri promašaju upisa". Ovo znači da će u memoriji za čuvanje tagova, imati dodatne bitove "valid" i "dirty".

Sistem za keširanje će voditi polisu inkluzivnosti, te ukoliko blok nije prisutan u drugom nivou keša, ne sme biti prisutan ni u prvom. Iz ovog razloga uvodimo dva nova bita koja će takođe biti deo memorije za čuvanje tagova: "data" i "instruction". Ovi bitovi govore da li se blok nalazi u prvom nivou keša za podatke i instrukcije respektivno. Pri evikciji bloka iz drugog nivoa keša, sada je moguće invalidirati isti blok u kešu za podatke ili instrukcije ako je odgovarajući bit postavljen na jedinicu. Na ovaj način se održava inkluzivnost keša. Za direktno mapiran keš, se jednostavno pri evikciji mogu invalidirati oba keša prvog nivoa sa istim indeksom, ali ovo dovodi do bespotrebnih invalidacija a samim tim i većeg udela promašaja. Ova tehnika će imati još veći benefit kada bude implementiran set asocijativan keš u drugom nivou.

Blok RAM moduli su po specifikaciji dvoprístupni, te će ova osobina iskoristiti kako bi se uveo trud za dodatnom paralelizacijom i jednostavnijim rutiranjem. Jedan port keša drugog nivoa će biti namenjen samo razmeni blokova sa prvim nivoom keša, dok će drugi port biti namenjen komunikaciji sa operativnom memorijom. Sa ovakvom konfiguracijom se logika za kontrolu portova drugog nivoa keša, može rastaviti na dva jednostavnija konačna automata umesto jednog većeg.

Ova odluka je donešena iz dva razloga:

- 1) Potencijalno jednostavnije rutiranje i postavljanje (*eng. placement and routing*)
- 2) Podrške za višejezgarni sistem. Naime, ukoliko bi postojalo više jezgara povezanih na istu deljenu magistralu, svako jezgro bi moglo da postavi zahtev za određeni blok pre nego što se blok krene preuzimati iz operativne memorije. Ukoliko bi neko drugo jezgro imalo taj blok, moglo bi ga predati preko deljene magistrale, i tako izbeći pristup memoriji. U ovoj konfiguraciji bi se komunikacija sa ostalim sistemom mogla izvršavati preko porta namenjenog za to, dok bi se istovremeno, nesmetano, mogla izvršavati komunikacija sa prvim nivoom keša preko drugog porta.

Na osnovu svih prethodnih razmatranja, drugi nivo keša se sastoji od sledećih memorijskih modula:

1. Keš drugog nivoa

Dvopristupna memorija sa sinhronim čitanjem. Širine podatka su 32 bita a broj podataka je $LVL2_CACHE_SIZE/4$.

2. Memorija za čuvanje tagova drugog nivoa keša

Dvopristupna memorija sa asinhronim čitanjem. Broj podataka je $LVL2_CACHE_SIZE/BLOCK_SIZE$ (broj blokova u kešu) a širina podataka ($32 - \log_2(LVL2_CACHE_SIZE) + 4$), što je širina taga za prvi nivo keša, plus "valid", "dirty", "data" i "instruction" bitovi.

Na slici 9 se može videti blok dijagram prethodno opisanog sistema za keširanje. U gornjem delu slike se može primetiti interfejs ka procesorskom jezgrou, a u donjem delu interfejs ka operativnoj memoriji.

Prvi nivo keš podsistema se sastoji od prethodno opisanih memorijskih modula: keša za instrukcije (*instruction cache*), memorije za čuvanje tagova keša za instrukcije (*instruction tag store*), registra za čuvanje valid bita (*instruction valid reg*), keša za podatke (*data cache*), memorije za čuvanje tagova keša za podatke (*data tag store*). Komunikaciju između ovih memorijskih modula i keša drugog nivoa kontroliše automat "cache controller". Pomoću komparatora i logičkog "i" kola, se dobijaju signali *lv1d_hit_s* i *lv1i_hit_s*, te se prosleđuju automatu. Ovi signali su indikatori da li je trenutno adresirani podatak prisutan u odgovarajućem kešu prvog nivoa, te se dobija poređenjem taga zahtevane adrese, i taga pročitano iz memorije za čuvanje tagova. Poređenje se vrši pomoću komparatora, te se dobija jednobitni signal. Istovremeno se čita i *valid* bit iz registra (za keš instrukcija) ili iz memorije (za keš podataka). Ova dva signala se dovode na ulaz logičkog "i" kola. Još jedna komponenta koja je potrebna "cache controller" automatu, je brojački registar "*cc_cache_controller*". On je potreban za indeksiranje reči u bloku podataka, pri prenosu bloka između keševa prvog i drugog nivoa. Svi keševi su 32-bitni, te se blok mora prenositi u više taktova, stoga potreba za brojačem.

Memorijski moduli koji čine drugi nivo keš podsistema su: keš memorija (*lv2 cache*) i memorija za čuvanje tagova drugog nivoa (*lv2 tag store*). Prenos podataka između keša drugog nivoa i operativne memorije kontroliše automat "memory controller". Kao i u slučaju

Signali *flush_lvl1d_s*, *invalidate_lvl1d_s* i *invalidate_lvl1i_s* služe za održanje koherencije i inkluzije između dva nivoa keša. Njih šalje "memory controller" automat pri evikciji bloka, te na taj način govori "cache controller" automatu da je potrebno: proslediti određeni blok iz keša za podatke u drugi nivo keša, invalidirati blok u kešu za podatke ili invalidirati blok u kešu za instrukcije.

U nastavku poglavlja će se detaljnije opisati način rada ova dva automata. Pre toga je bitno uspostaviti raspored bita (*eng. bit ordering*) u memorijama za čuvanje tagova, kako bi se lakše razumeli ASM dijagrami. Na slici 10 se mogu videti ove tri memorije, te signali koji se koriste. Signali pod nazivom *dreada_*_ts_s* su signali koji se dovode direktno na portove za čitanje. Ovi signali se dele na dva signala sa nazivima: *lvl*_ts_bkk_s* i *lvl*_ts_tag_s*. Prvi predstavlja pomoćne bite (*eng. bookkeeping*) a drugi predstavlja tag polje.

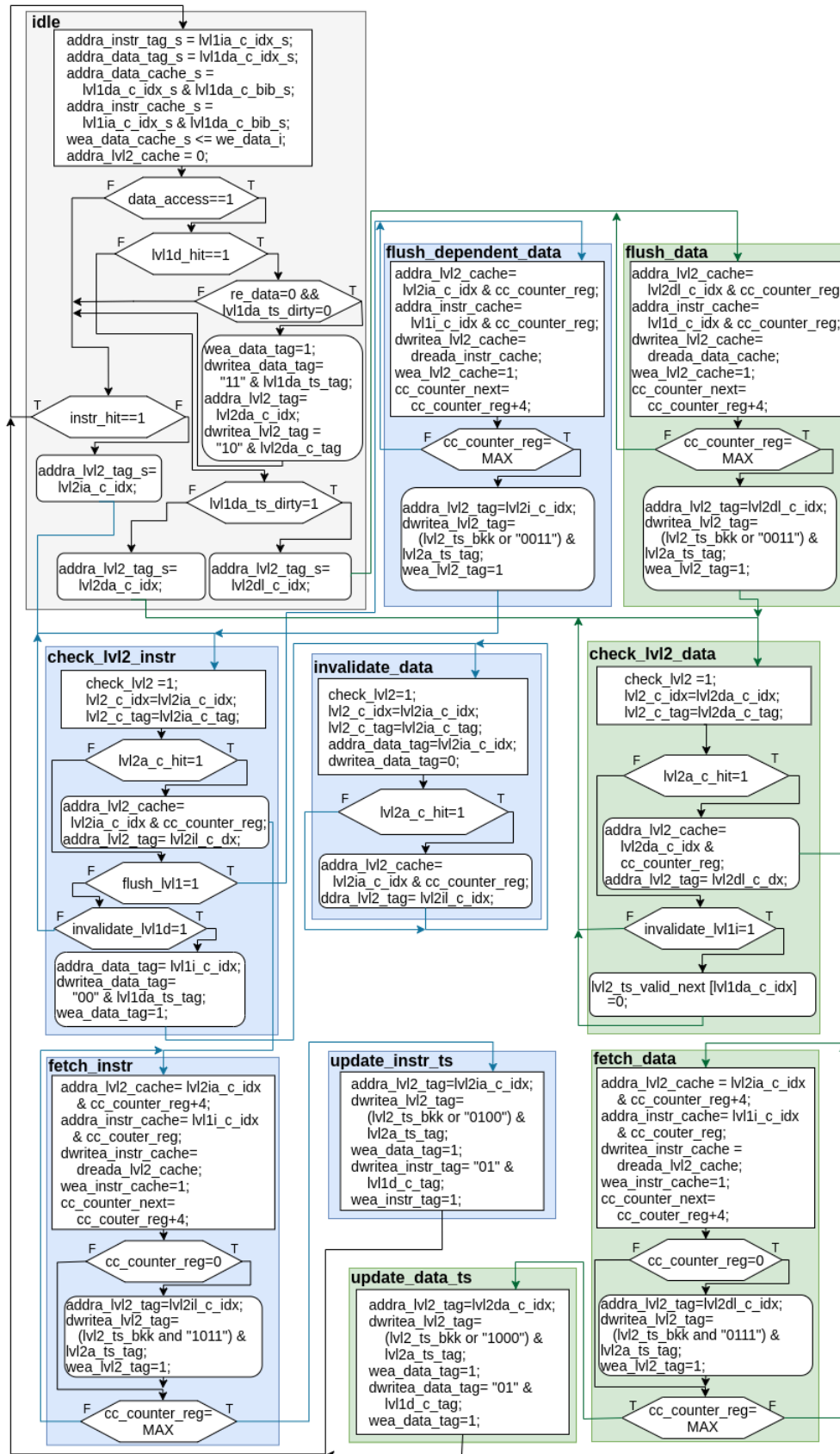
U memoriji za čuvanje tagova keša podataka, postoje dva pomoćna bita: *valid* (validan) i *dirty* (prljav) te njihove kombinacije označavaju sledeća stanja blokova:

- *valid*=0, *dirty*=0 - neinicijalizovan blok u kešu za podatke, blok nije validan
- *valid*=1, *dirty*=0 - blok je validan, podaci su ažurirani sa drugim nivoom keša
- *valid*=1, *dirty*=1 - blok je validan, podaci su noviji nego u drugom nivou keša
- *valid*=0, *dirty*=1 - nedozvoljena kombinacija bita

U memoriji za čuvanje tagova keša drugog nivoa, postoje četiri pomoćna bita: *dirty* (prljav), *valid* (validan), *instr* (blok je u kešu za instrukcije) i *data* (blok je u kešu za podatke). Kombinacije *valid* i *dirty* bitova označavaju sledeća stanja blokova u sistemu za keširanje.

- *valid*=0, *dirty*=0 - neinicijalizovan blok u kešu, blok nije prisutan u drugom nivou keša a samim tim ni u prvom nivou keša
- *valid*=1, *dirty*=0 - blok je validan, podaci su ažurirani sa operativnom memorijom
- *valid*=1, *dirty*=1 - blok je validan, podaci su noviji nego u operativnoj memoriji
- *valid*=0, *dirty*=1 - podaci su zastareli u odnosu na keš za podatke

Instruction tag store



Slika 11: Keš kontroler prvog nivoa (direktno preslikani)

Slika 11 predstavlja ASM dijagram "Cache controller" automata. Inicijalno stanje automata je **idle**. U ovom se stanju automat nalazi dok se god zahtevane instrukcije i podaci nalaze u kešu prvog nivoa, procesor se izvršava te su svi podaci dostupni bez kašnjenja. Signali koji dolaze od procesora se direktno prosleđuju na keš memorije, a na memorije za čuvanje tagova se prosleđuju indeks polja zahtevanih adresa. Memorije za čuvanje tagova su asinhronne, te se u istom taktu zna da li se desio pogodak ili promašaj u prvom nivou keša.

Na osnovu signala *data_mem_we* i *data_mem_re* se proverava da li se u trenutnom taktu uopšte zahteva pristup memoriji za podatke, jer ukoliko instrukcija nije tipa *load* ili *store*, nije ni bitno da li se desio pogodak sa prosleđenom, nevalidnom, adresom. Ukoliko je neki od bita ova dva signala jednak jedinici, sledi da se pristupa memoriji, te se proverava da li se desio pogodak (*lvl1d_hit*=1). Ako se desio pogodak, proverava se da li se radi o upisu u memoriju i da li je taj blok prethodno bio izmenjen. Ukoliko je slučaj da je *dirty* bit jednak nuli, a da se izvršava upis, potrebno je postaviti *dirty* bit u prvom nivou keša, i postaviti *dirty* bit a resetovati *valid* bit za taj blok u drugom nivou keša. U slučaju promašaja, novi blok se mora preuzeti iz memorije višeg nivoa. Ukoliko je trenutni blok prljav (*dirty* bit postavljen), znači da su podaci noviji od drugog nivoa keša, stoga se blok mora kopirati u drugi nivo (*eng. flush*) pre nego što se novi blok preuzme. U sličaju da je prljav, automat prelazi u stanje *flush_data*, a u suprotnom u stanje *check_lvl2_data*. Pri promašaju će automat svakako završiti u stanju *check_lvl2_data*, jer i nakon ažiranja drugog nivoa keša u stanju *flush_data*, automat prelazi u stanje *check_lvl2_data*. Ukoliko podaci u kešu za podatke prisutni i validni, u *idle* stanju se takođe proverava da li se desio pogodak u kešu za instrukcije. Ako je u pitanju promašaj, prelazi se u stanje *check_lvl2_instr*.

Stanja ***check_lvl2_data*** i ***check_lvl2_instr***, u koja se dolazi pri promašaju u nekom od keševa prvog nivoa, služe za proveru da li je taj blok prisutan u drugom nivou keša. Zahtevana adresa instrukcije ili podatka se poredi sa sadržajem memorije za čuvanje tagova drugog nivoa keša. U slučaju da blok nije prisutan ni u drugom nivou keša, automat čeka u ovom stanju, dok ga automat "memory controller" preuzima iz operativne memorije. Ako je blok prisutan, odmah se prelazi u stanje *fetch_instr* ili *fetch_data*. Stanja ***fetch_instr*** i ***fetch_data*** vrše preuzimanje bloka iz drugog nivoa keša u keš za instrukcije ili podatke. U ovim stanjima se ostaje dok se ne prebaci čitav blok, te u zavisnosti od veličine bloka zavisi i koliko će se dugo izvršavati ova stanja. Magistrale su 32-bitne (4 bajta), te je potrebno *BLOCK_SIZE/4* perioda takta za prebacivanje bloka podataka. Pri završetku prebacivanja bloka, prelazi se u stanje ***update_instr_ts*** ili ***update_data_ts***, gde se ažurira odgovarajuća memorija za čuvanje tagova.

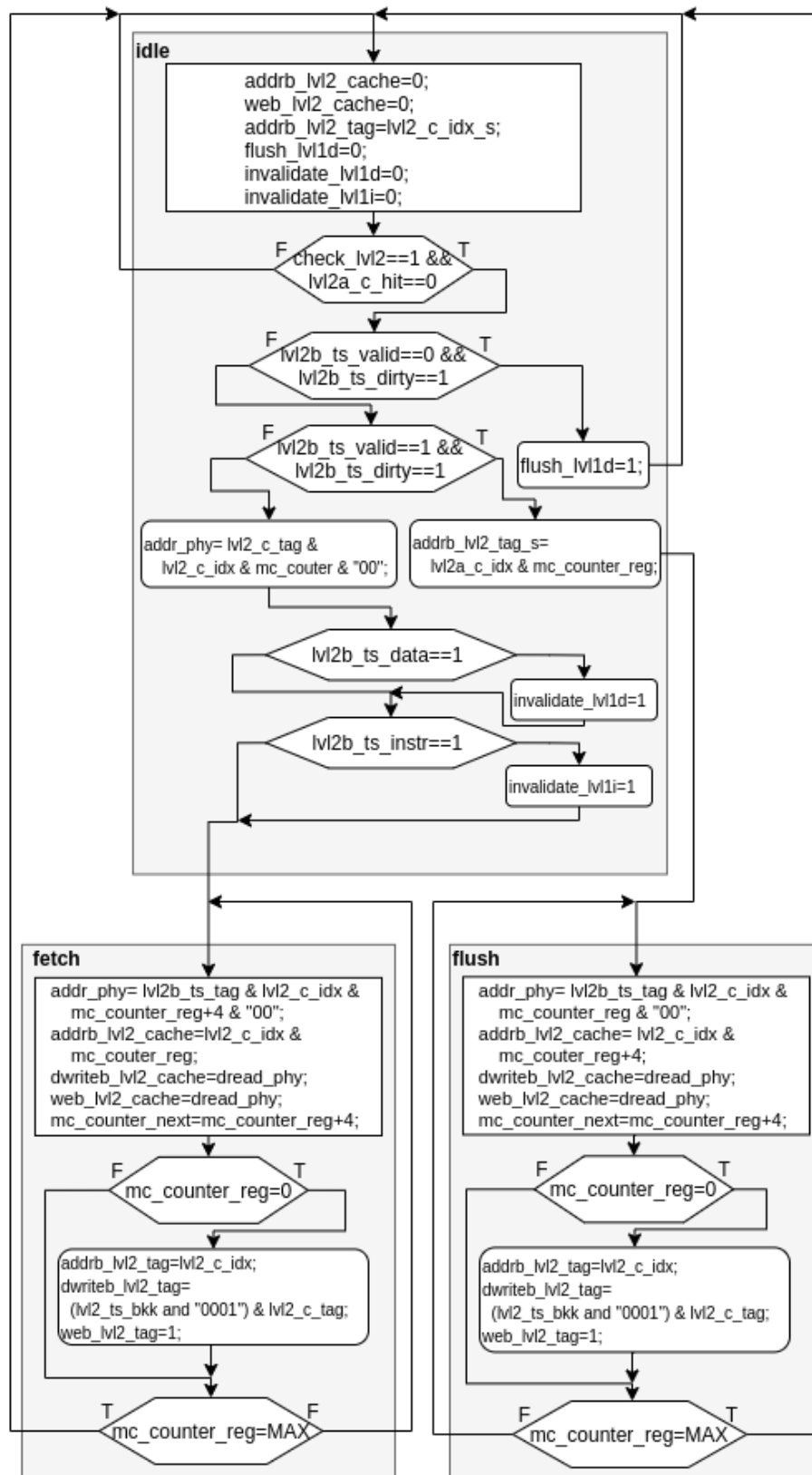
Dok "cache controller" automat čeka u stanju *check_lvl2_data*, te "memory controller" automat treba da eviktuje blok iz drugog nivoa keša koji je takođe prisutan u kešu za instrukcije, radi održanja inkluzije "memory controller" može da postavi signal *invalidate_lvl1i_s*. Ovo znači da je potrebno invalidirati taj blok u kešu za instrukcije. U kešu za instrukcije se valid biti čuvaju u registru, te se ovo može odraditi odmah na sledećoj rastućoj ivici takta.

Slično prethodnom slučaju, dok "Cache controller" automat čeka u stanju *check_lvl2_instr*, te "memory controller" automat treba da eviktuje blok iz drugog nivoa keša koji je takođe prisutan u kešu za podatke, radi održanja inkluzije, "memory controller" može da postavi

signal *invalidate_lvl1d_s*. Ovo znači da je potrebno invalidirati taj blok u kešu za podatke. *Valid* biti se u kešu za podatke se čuvaju u memoriji za čuvanje tagova, te se resetovanje valid bita vrši u posebnom stanju ***invalidate_data***. Ako "memory controller" automat treba da eviktuje blok sa kombinacijom pomoćnih bita *valid=0* i *dirty=1*, on postavlja signal *flush_lvl1d_s*. Ova situacija znači da je potrebno eviktovati blok, kod kojeg keš za podatke ima izmenjen (noviji) blok u odnosu na drugi nivo keša. Pre nego što se blok eviktuje, neophodno je ažurirati drugi nivo keša (kopiranjem bloka iz keša za podatke). Kopiranje bloka u drugi nivo keša se vrši u stanju ***flush_dependent_data***. Prethodno opisana dva stanja su neophodna za održanje inkluzije i koherencije u ovakvoj konfiguraciji keševa. Iako će se javljati u test programu sa direktno mapiranim drugim nivoom keša, kada se uvede asocijativnost, šanse za javljanje ovakvih situacija će biti mnogo manje, te će služiti samo za pokrivanje jako retkih, graničnih, slučajeva. Na slici 11 su plavom bojom označena stanja u koja automat "cache controller" može doći kada se desi promašaj u kešu za instrukcije, dok su zelenom bojom označena stanja u koja se može doći kada se desi promašaj u kešu za podatke.

ASM dijagram automata "memory controller" je prikazan na slici 12. Podrazumevano stanje je ***idle***, u kojem automat čeka da se desi promašaj u drugom nivou keša. Čim detektuje promašaj, iz memorije za čuvanje tagova se proveravaju pomoćni bitovi. Ukoliko je neophodno, šalju se prethodno opisani singali: *flush_lvl1d*, *invalidate_lvl1d* i *invalidate_lvl1i*.

Ukoliko je blok prljav u drugom nivou keša, potrebno ga je upisati u operativnu memoriju pre nego što se preuzme novi blok. Ovo se radi u stanju ***flush***. Nakon što je blok koji se eviktuje ažuriran sa operativnom memorijom, pristupa se preuzimanju zahtevanog bloka, što se izvršava u stanju ***fetch***.



Slika 12: Keš kontroler drugog nivoa

3.2.2 Simulacija direktno preslikanog keša

Prethodno opisani slučajevi će biti demonstrirani u bihevijalnoj simulaciji alata Vivado. Bitno je napomenuti da je u prikazanim simulacijama operativna memorija modelovana kao jednopristupna sinhrona memorija sa *read-first* tipom čitanja, te nisu modelovana dodatna kašnjenja pri komunikaciji sa memorijskim čipom, koja će se demonstrirati nakon implementacije projekta na Zybo ploči. Konfiguracija sistema za keširanje je sledeća:

- *BLOCK_SIZE* = 64B
- *LVL1_CACHE_SIZE* = 1024B = 1KB
- *LVL2_CACHE_SIZE* = 4*1024B = 4KB

U RISC-V asemblerskom jeziku je napisan program prikazan u kodnom listingu u nastavku teksta. Kod je asembliran, te je fajl sa binarnim kodom korišten za inicijalizaciju modela operativne memorije.

<pre>## Testiranje prihvata li a0, 0 #set x11 to zero outer1: li a1, 0 #set x11 to zero li a2, 64 # number of reads inner1: lw a7, 0(a0) addi a0, a0, 4 # increase adress (add offset) addi a1, a1, 4 # increase iterator (counts bytes to get to a block) beq a1, a2, einner1 jal inner1 eininner1: lui a4, 1 addi a4, a4, 64 beq a0, a4 eouter1 lui a0, 1 jal outer1 ## Testiranje upisa u keš za podatke eouter1: lui a0, 1 li a1, 0 li a2, 64 inner2: sw a0, 0(a0) # write address value to address (lvl1d), check if dirty is set addi a0, a0, 4 # increase adress (add offset) addi a1, a1, 4 # increase iterator (counts bytes to get to a block) beq a1, a2, einner2 jal inner2</pre>	<pre>## Testiranje pravilne evikcije blokova ## Preopterećivanje 4-smernog keša eininner2: li a5, 5 slli a5, a5, 12 li a3 1 outer3: addi a3, a3, 1 slli a6, a3, 12 addi a0, a6, 0 li a1, 0 li a2, 64 inner3: lw a7, 0(a0) # read address value, check if dirty data is flushed addi a0, a0, 4 # increase adress (add offset) addi a1, a1, 4 # increase iterator (counts bytes to get to a block) beq a1, a2, einner3 jal inner3 eininner3: beq a5, a6 eouter3 jal outer3 eouter3: nop li a3 1 jal outer3</pre>
---	--

Listing 1: Test program

Ovaj jednostavan asemblerski program će se koristiti za testiranje direktno preslikanog kao i asocijativnog keša. On izvršava sledeće memorijske operacije kako bi se testirali kritični slučajevi pri keširanju:

- 1) Čitanje svih podataka iz bloka na memorijskim adresama 0-64
- 2) Čitanje svih podataka iz bloka na memorijskim adresama 4096-4160

- 3) Upis podataka na sve lokacije bloka na memorijskim adresama 4096-4160
- 4) Čitanje svih podataka iz bloka na memorijskim adresama 8192-8256
- 5) Čitanje svih podataka iz bloka na memorijskim adresama 12288-12352
- 6) Čitanje svih podataka iz bloka na memorijskim adresama 16384-16448
- 7) Čitanje svih podataka iz bloka na memorijskim adresama 20480-20544
- 8) Ponovno izvršavanje koraka 4 - 8

Memorijske lokacije su izabrane kako bi za prethodnu odabrane veličine keša napravile konflikte u blokovima (adrese imaju ista indeks polja). Na slici 13 se može videti simulacija prvih 550ns izvršavanja prethodno opisanog programa.

Nakon što se završi sa resetom sistema, procesor kreće sa radom. Međutim, budući da je keš neinicijalizovan, instrukcije nisu dostupne, te je signal *instr_ready_s* jednak nuli. Signal *data_ready_s* je jednak jedinici iz razloga što se još uvek ne izvršava instrukcija pristupa memoriji. Promašaj u kešu za instrukcije rezultuje da "cache controller" automat prelazi u stanje *check_lvl2_instr*. Za zahtevanu adresu se dešava promašaj i u drugom nivou keša, te automat ostaje u istom stanju. "Memory controller" detektuje promašaj u drugom nivou, prelazi u stanje *fetch*, kreće da preuzima blok instrukcija iz operativne memorije i da ih upisuje u keš drugog nivoa (simulacioni trenutak 30ns, slika 13). Može se primetiti da signal *mc_counter_reg* kreće da broji indeksirajući reči unutar 64-bitnog bloka. Adresa operativne memorije *addr_phy_o* se menja zajedno sa brojačem, te se pročitani podaci upisuju u drugi nivo keš memorije, što se može primetiti na slici postavljanjem signala dozvole upisa *web* na logičku jedinicu. Čim "memory controller" upiše prvi podatak u keš drugog nivoa preko porta "b", on postavlja u memoriji za čuvanje tagova da je blok validan. "Cache controller" automat detektuje ovu promenu kroz signal *lvl2_hit_s*, prelazi u stanje *fetch_instruction*, i kreće da prebacuje blok u keš za instrukcije. Kao što se vidi sa slike 13, signal dozvole upisa u memoriju za instrukcije *wea* se postavlja na jedinicu u trenutku 60ns kada preuzimanje bloka iz drugog nivoa keša počinje. Izvršavanje stanja *fetch* i *fetch_instruction* je stoga preklapljeno kao što se vidi po stanjima brojača *cc_counter_reg* i *mc_counter_reg*. Za prebacivanje bloka od 64B iz jednog nivoa memorije u drugi je potrebno 16 perioda takt signala, što se vidi na slici po brojačima, čija je maksimalna vrednost 0xf. Kada se pruzme čitav blok u keš za instrukcije, "cache controller" ažurira memoriju za čuvanje tagova u stanju *update_instr_ts*. Sledeći takt (simulacioni trenutak 230ns) procesor kreće sa izvršavanjem instrukcija i ispunjavanjem protočne obrade.

Nedugo nakon toga, dešava se prvi zahtev za čitanjem iz memorije za podatke, što se vidi po signalu *data_mem_re* (simulacioni trenutak 290ns, slika 13). Signal *data_ready_s* se istovremeno postavlja na nulu zbog promašaja u kešu za podatke. "Cache controller" automat prelazi u stanje *check_lvl2_data* gde se dešava pogodak u drugom nivou keša što se vidi po signalu *lvl2_hit_s*. Naime, blok podataka koji se zahteva je 0-64, isti blok koji je prethodno bio zahtevan u kešu za instrukcije te prebačen u drugi nivo keša. Automat odmah prelazi u stanje *fetch_data_s* gde se blok preuzima u keš za podatke. Brojač *cc_counter_reg* redom indeksira reči koje se preuzimaju iz drugog nivoa keša preko porta "a". Signal dozvole upisa *wea* keša za podatke se postavlja na vrednost 0xf, što omogućuje upis četiri bajta svaku

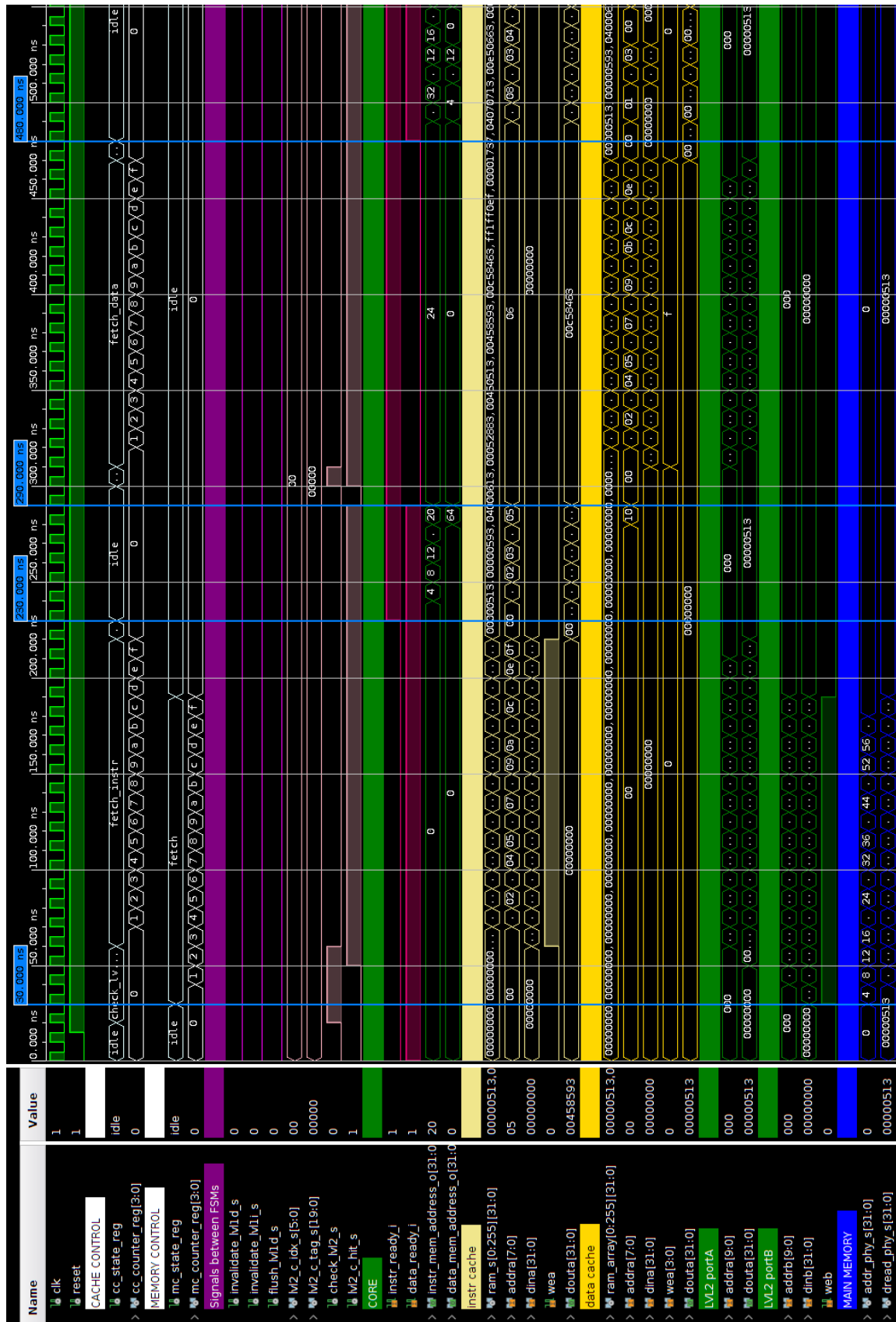
aktivnu ivicu takt signala. Nakon što je kompletan blok upisan, memorija za čuvanje tagova se ažurira u stanju *update_data_ts* (simulacioni trenutak 470 ns, slika 13). Sledeći takt se signal *data_ready* postavlja na logičku jedinicu, automat "cache controller" prelazi u stanje *idle* i procesor nastavlja sa radom.

Problem sa direktno mapiranim drugim nivoom keša nastaje kada se prvi put napravi zahtev za pristup bloku podatka 4096-4160. Naime, instrukcije programa se i dalje nalaze na adresama 0-64, dok procesor zahteva podatke sa lokacija 4096-4160. Ovo znači da keš za instrukcije i keš za podatke istovremeno zahtevaju blok koji ima isti indeks u drugom nivou keša. Donešena je odluka da keš bude inkluzivan, što znači da bi oba bloka istovremeno morala biti u drugom nivou keša, što je za direktno mapiran keš nemoguće. Ono što sledi je konflikt gde se dešava naizmenično preuzimanje ista dva bloka iz operativne memorije. Na slici 14 se u simulacionom trenutku 1680ns dešava promašaj u prvom nivou keša za podatke pri zahtevu bloka 4096-4160, te se može videti da signal *data_ready* pada na nulu. Kako bi "memory controller" automat ovaj blok smestio u drugi nivo keša, mora da invalidira onaj koji se trenutno nalazi na lokaciji sa istim indeksom. Trenutno se na lokaciji sa indeksom nula nalazi blok 0-64, koji je takođe i u oba keša prvog nivoa. "Memory controller" automat invalidira ovaj blok postavljanjem signala *invalidate_lvl1d_s* i *invalidate_lvl1i_s* (simulacioni trenutak 1690ns). Nastavlja se sa preuzimanjem bloka 4096-4160 iz operativne memorije i smeštanjem njega u drugi nivo keša kao i keš za podatke. Preuzimanje se završava u simulacionom trenutku 1900ns, kada *cc_state_reg* prelazi u *idle* stanje.

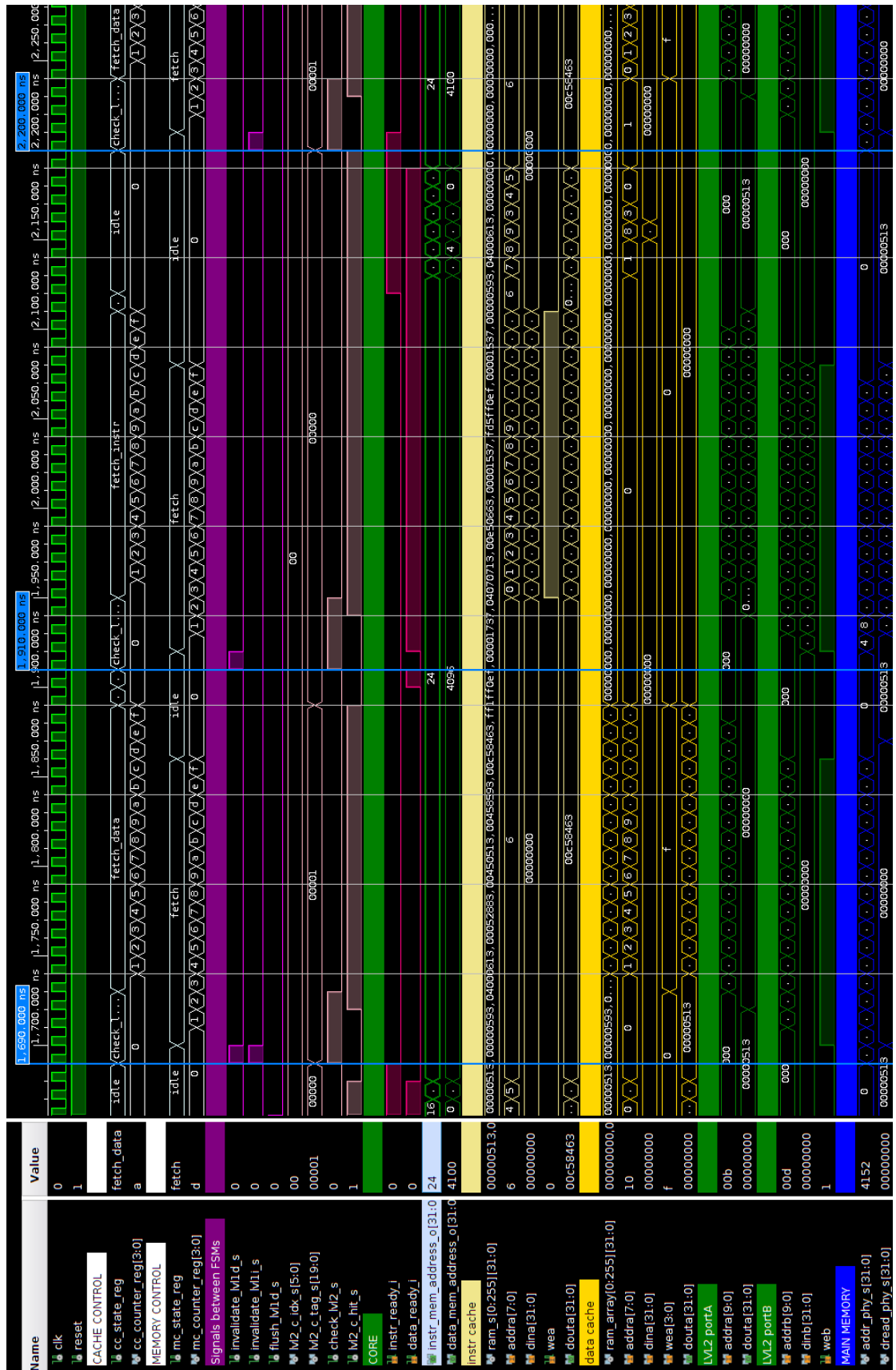
Iako je preuzet željeni blok u keš za podatke, problem nastaje zato što je eviktovan blok 0-64 iz keša za instrukcije, te protočna obrada ne može nastaviti sa radom zbog promašaja pri čitanju naredne instrukcije. Primetiti da je signal *instr_ready* na nuli. Dešava se promašaj i u kešu drugog nivoa, te se kreće sa preuzimanjem bloka 0-64 iz operativne memorije. Na sličan način, kako bi se blok 0-64 smestio u drugi nivo keša, mora se eviktovati blok koji je trenutno na toj lokaciji - blok 4096-4160. U simulacionom trenutku 1910ns "memory controller" automat postavlja signal *invalidate_lvl1d* čime se invalidira ovaj blok u kešu za podatke.

Zatim se kreće sa preuzimanjem bloka 0-64 iz operativne memorije što se može primetiti na slici prelaskom "cache controller" automata u *fetch_instr* stanje i "memory controller" automata u *fetch* stanje. Blok se upiše u drugi nivo keša i keš za instrukcije. Na slici se može primetiti da oba automata prelaze u *idle* stanje, te da zatim procesor izvršava nekoliko instrukcija (simulacioni trenutak 2120ns).

Ubrzo se dolazi do sledeće instrukcije pristupa memoriji (iz bloka na adresi 4096-4160). Dešava se promašaj u kešu za podatke kao i u drugom nivou keša. Eviktuje se blok 0-64 postavljanjem signala *invalidate_lvl1i_s* (simulacioni trenutak 2200ns). Kreće se sa preuzimanjem bloka 4096-4160 iz operativne memorije, čime se ciklus ponavlja. Ovaj ciklus naizmeničnog invalidiranja blokova se izvršava za svaki pristup bloku 4096-4160, što neće biti prikazano u simulaciji. Većina vremena se provodi u prebacivanju podataka između memorija za koje vreme procesor ne izvršava program već čeka na podatke ili instrukcije. Ova situacija je retka, međutim može dovesti do kritičnog pada performansa procesora (približno potpunom zaustavljanju) koji nije prihvatljiv ni sa kakvom verovatnoćom. Iz ovog primera se može zaključiti da je neophodno da drugi nivo keša bude barem dvosmerno asocijativan.



Slika 13: Prihvat bloka podataka



Slika 14: Konflikt u drugom nivou keša sa direktnim preslikavanjem

3.2.3 Set asocijativan drugi nivo sistema za keširanje

U ovom poglavlju će se modifikovati model iz prethodnog, tako da drugi nivo keša postane N-smerno asocijativan. Kako bi se krajnjem korisniku omogućila fleksibilnost pri konfiguraciji sistema za keširanje, uvode se dva parametra:

- *LVL2C_ASSOCIATIVITY* - asocijativnost drugog nivoa keša, podrazumevana vrednost je 4. Ova vrednost govori koliko će puta biti repliciran keš i memorija za čuvanje tagova drugog nivoa.
- *TS_BRAM_TYPE* - tip blok RAM memorija u kojima se čuvaju tagovi drugog nivoa keša. Ovaj parametar je tipa string, i podržava dve vrednosti:
 - *"HIGH_PERFORMANCE"* - uključen izlazni registar, čitanje podataka zahteva dve periode takta, ali se dobija manje kašnjenje od rastuće ivice do uspostavljanja stabilnog signala.
 - *"LOW_LATENCY"* - isključen izlazni registar, podaci se čitaju na prvu rastuću ivicu takta, lošije vreme uspostavljanja signala.

Polisa evikcije koja će biti implementirana je pseudo LRU polisa pod imenom "Žrtva - Sledeća žrtva" (*eng. Victim - Next victim*). Ova polisa je bila detaljnije opisana u poglavlju 2.4.1, te se ovde neće zalaziti u sam algoritam. Nezavisno od asocijativnosti keša, ova polisa zahteva dva dodatna pomoćna bita u memoriji za čuvanje tagova: "victim" i "next-victim". Raspored bita u memoriji za čuvanje tagova će biti kao na slici 15. Svi signali koji su korišteni za interfejs sa drugim nivoom direktno preslikanog keša, zadržavaju imena, ali postaju nizovi bit-vektora. Nizovi imaju N članova, gde se svaki koristi za pristup jednom od memorijskih modula drugog nivoa keša.

Level 2 tag store						
<i>dreadda_lv2_ts_s [i]</i>						
<i>lv2a_ts_nbkk_s [i]</i>		<i>lv2a_ts_bkk_s [i]</i>				<i>lv2a_ts_tag_s [i]</i>
NEXTV	VICTIM	DATA	INSTR	DIRTY	VALID	TAG

Slika 15: Raspored bita u memoriji za čuvanje tagova asocijativnog keša

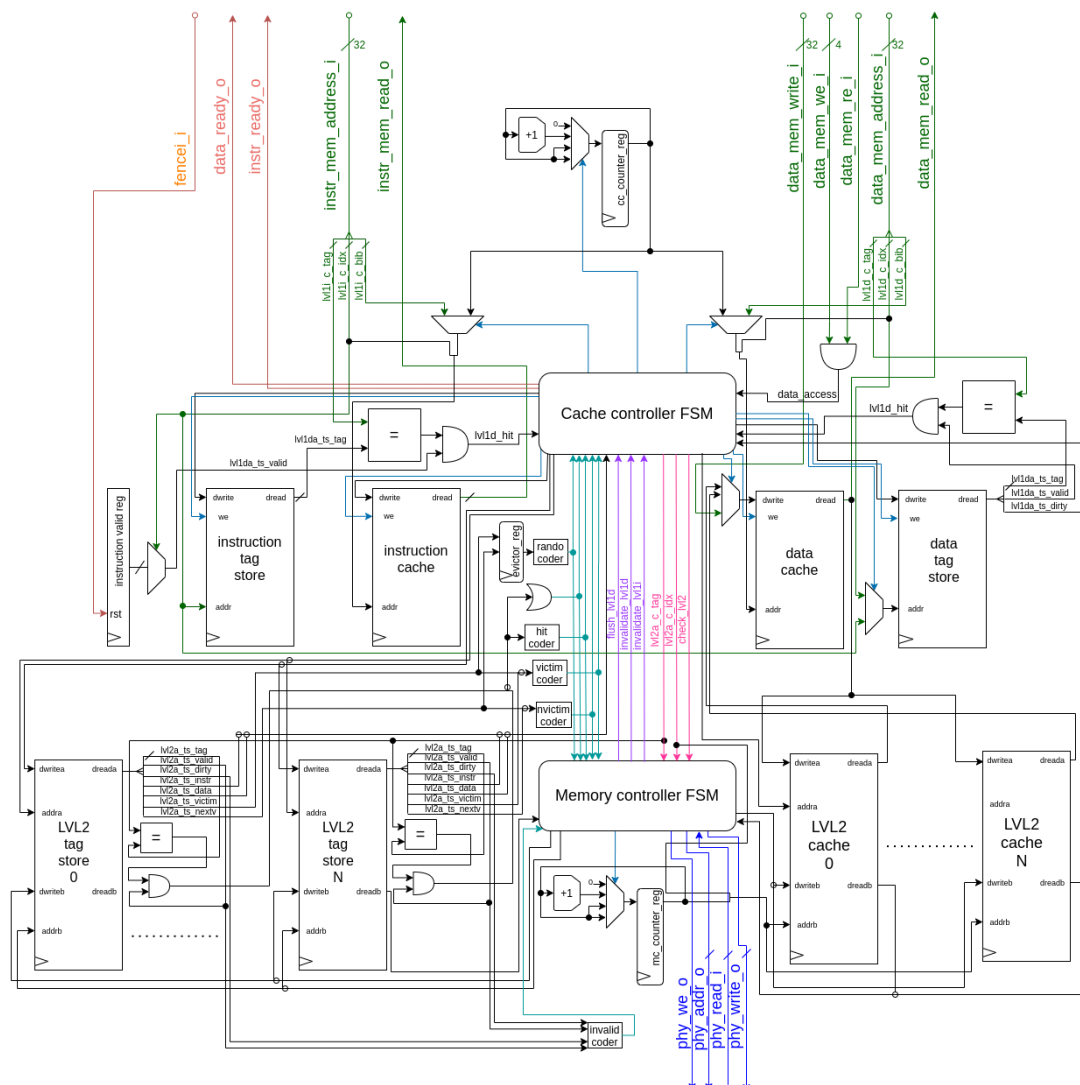
Na slici 15 je prikazan blok dijagram modifikovanog sistema za keširanje, sa N-smernim asocijativnim drugim nivoom. Na blok dijagramu su zbog preglednosti, kratki spojevi označeni punim krugom na grananju linija, dok je konkatanacija (spajanje signala) označena kružnicama. Kao što se može primetiti sa dijagrama, prvi nivo keša je strukturalno ostao nepromenjen, kao i komunikacija između automata. U drugom nivou su replicirani memorijski moduli i uvedena dodatna kombinaciona kola.

Kada se god komunicira sa drugim nivoom keša, potrebno je znati sa kojim od N memorijskih modula se treba komunicirati. Svaki memorijski modul ima poseban komparator koji poredi tagove kao i logičko "i" kolo za proveru validnosti. Za dobijanje signala *lv2_hit_s* je dovoljno izlaze ovih kola dovesti na N-bitno logičko "ili" kolo, kao što je prikazano na dijagramu. Međutim, nije dovoljno znati samo da li se desio pogodak, već i u kojem od N

memorijskih modula se nalazi željeni podatak. Stoga je potrebno uvesti dodatne signale koji će se koristiti za indeksiranje smerova asocijativnog keša.

- *lvl2_hit_index* - indeks smera u kojem se desio pogodak
- *lvl2_victim_index* - indeks smera koji je označen kao žrtva
- *lvl2_nextv_index* - indeks smera koji je označen kao sledeća žrtva
- *lvl2_invalid_index* - indeks smera koji nije inicijalizovan
- *lvl2_rando_index* - indeks nasumičnog običnog bloka
- *lvl2_dflush_index*, *lvl2_iflush_index* - pomoćni signali za kopiranje u drugi nivo

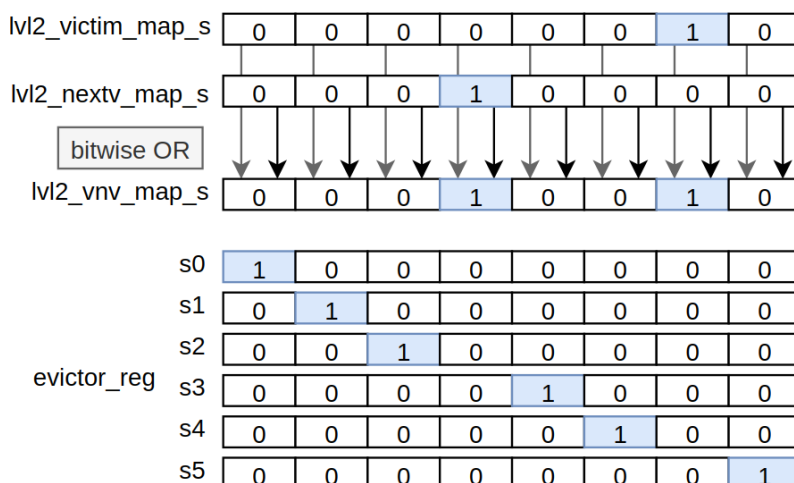
Signal *lvl2_hit_index* daje indeks smjera u kojem se desio pogodak, te se dobija tako što se izlazi kombinacione logike za detekciju pogotka svih smerova spoje u N-bitni signal i dovedu na ulaz N-bitnog prioritetskog kodera. Signali *lvl2_victim_index* i *lvl2_nextv_index* se dobijaju na sličan način: pomoćni bitovi *victim* i *nextv* svih smerova se konkataniraju te dovedu na prioritetske kodere.



Slika 16: Blok dijagram asocijativnog sistema za keširanje

Za dobijanje signala *lvl2_invalid_index* se pomoćni bitovi *valid* i *dirty* svakog smera zasebno dovedu na "ni" kolo a zatim se izlazi tih kola konakataniraju i dovedu na prioritetni koder. Ovaj signal služi za dobijanje indeksa neinicijalizovanog smera u asocijativnom kešu. Kada se preuzima blok iz operativne memorije, polisa je da ukoliko postoji neinicijalizovan smer, da se smesti na to mesto. Nema potrebe vršiti evikciju validnih blokova koji se koriste dok god postoji neiskorišten smer, stoga potreba za ovim signalom. U ovu svrhu se uvodi i jednobitni pomoćni signal *lvl2_invalid_found_s*, koji indikuje da li postoji neiskorišteni smer za dati indeks.

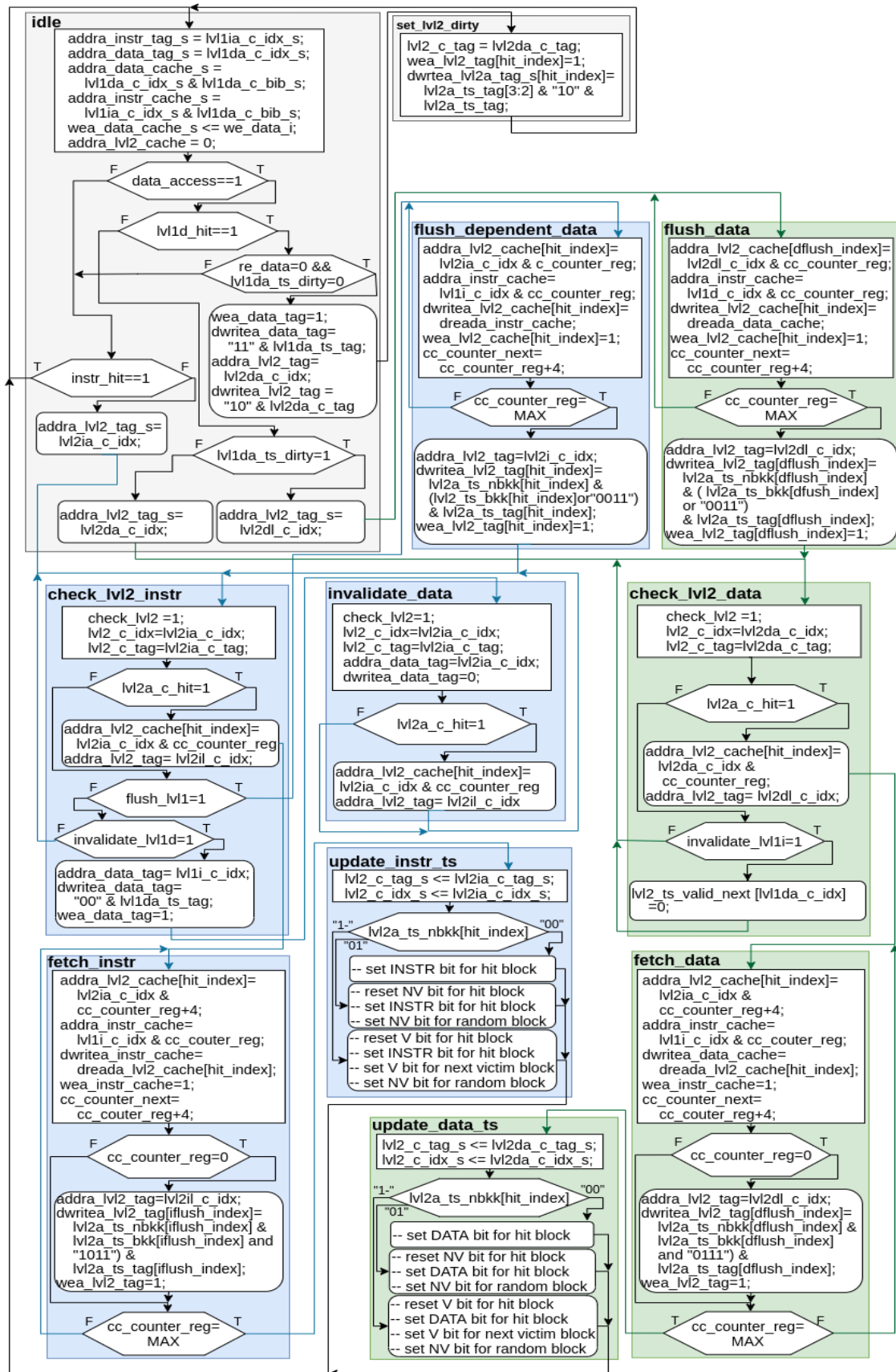
Korištena polisa za evikciju nalaže da je u algoritmu često potrebno izabrati nasumičan "običan" blok kako bi se on postao žrtva ili sledeća žrtva. Običan blok je onaj kod kojeg su pomćni bitovi *victim* i *nextv* postavljeni na nule. Od N smerova u asocijativnom kešu, postojaće N-2 obična bloka. Pseudo LRU polise evikcije često poseduju elemente nasumičnosti. Prava nasumičnost se u elektronici retko sreće te se pribegava drugim načinima za dobijanje ovakvih brojeva. U ovoj implementaciji će se napraviti pomerački registar koji će svaki takt menjati stanje te na takav način napraviti pseudo-nasumičnost. Prvo će se spojiti *victim* i *nextv* bitovi svih smerova, na isti način kao i za dobijanje prethodno pomenutih *lvl2_victim_index* i *lvl2_nextv_index* signala. Zatim će se odraditi logička operacija "ili" nad bitovima ova dva signala. Rezultujući signal *lvl2_vnv_map_s* je jednostavno signal gde nule predstavljaju pozicije "običnih" blokova. Svaki takt će pomerački registar *evictor_reg* pomeriti svoj sadržaj do sledećeg slobodnog običnog bloka. Signal *evictor_reg* se zatim, kao i u prethodnim slučajevima, dovodi na prioritetni koder čime se dobija signal *lvl2_rando_index*. Za osmosmerni asocijativan keš, stanja pomeračkog registra bi bila kao na slici 17.



Slika 17: Pomerački registar *evictor_reg*

Svi prethodno pomenuti signali su prikazani tirkiznom bojom na slici 16.

Na slici 18 se može videti ASM dijagram "Cache controller" automata kada je drugi nivo keša N-asocijativan. Većina stanja je ista kao i u direktno mapiranom kešu, s tim da su u ovom slučaju signali za interfejs sa kešom drugog nivoa nizovi, te se moraju koristiti prethodno pomenuti pomoćni **_index* signali.



Slika 18: Keš kontroler prvog nivoa (asocijativan drugi nivo)

Sa dijagrama se vidi da je uvedeno jedno novo stanje: *set_lvl2_dirty*. Kada se prvi put postavlja *dirty* pomoćni bit u nekom bloku, potrebno je i ažurirati isti bit u drugom nivou keša. Kako bi se znalo u kojem od smerova se nalazi traženi blok, potrebno je pročitati sadržaj memorija za čuvanje podataka. Stoga, prvi upis u neki blok unosi kašnjenje od jednog takta.

Jedina dva stanja koja su drastično promenjena u odnosu na direktno mapiranu implementaciju su: *update_instr_ts* i *update_data_ts*. U ovim stanjima se implementira algoritam polise za evikciju. U zavisnosti od signala *lvl2a_ts_nbkk* (pomoćnih bita *victim* i *nextv*) je potrebno ažurirati iste bite po pravilima algoritma.

Sva dodatna kombinaciona logika za indeksiranje smerova postaje kritična putanja u sistemu sa asocijativnim keširanjem, pogotovo za veće brojeve smerova. Iz tog razloga je uveden parametar *TS_BRAM_TYPE* koji ukoliko je postavljen na *HIGH_PERFORMANCE*, smanjuje propagaciono kašnjenje na izlazima blok RAM modula memorija za čuvanje tagova. Ovo se dobija na račun dodatnog takta kašnjenja. Stoga je potrebno dva takta dok se ne dobije informacija da li se desio pogodak u drugom nivou keša i sa kojim od N smerova se komunicira. Uzimajući ovo u obzir, bitno je napomenuti da je u tom slučaju potrebno čekati dodatan takt u stanjima: *set_lvl2_dirty*, *check_lvl2_instr*, i *check_lvl2_data*.

ASM dijagram "memory controller" automata neće biti ponovo prikazan, zato što stanja i funkcionalnost ostaju isti kao na slici 12 za slučaj direktno preslikanog keša. Kako bi se napravilo mesto za novi blok, uvek se eviktuje blok označen kao žrtva, te po pravilu algoritma:

- blok označen kao sledeća žrtva postaje žrtva
- ukoliko postoji neinicijalizovan blok, on se označava kao sledeća žrtva, a u suprotnom slučaju se uzima nasumičan obični blok

3.2.4 Simulacija sistema sa set asocijativnim drugim nivoom keša

Simulacija će na sličan način kao i za direktno mapiran sistem za keširanje biti odrađena u Vivado alatu. Koristiće se isti program predstavljan u kodnom listingu 1.

Konfiguracija sistema za keširanje je sledeća:

- *BLOCK_SIZE* = 64B
- *LVL1_CACHE_SIZE* = 1024B = 1KB
- *LVL2_CACHE_SIZE* = 4*1024B = 4KB
- *LVL2C_ASSOCIATIVITY* = 4
- *TS_BRAM_TYPE* = "HIGH PERFORMANCE"

Ovo znači da će keš za instrukcije kao i keš za podatke biti 1KB, dok će drugi nivo keša biti asocijativan sa četiri smer, gde je svaki smer veličine 4KB. Program je dizajniran kako bi se testirala evikcija modifikovanih blokova. Cilj je da se detektuje evikcija modifikovanog bloka 4096-4160 upisom novih podataka nazad u operativnu memoriju.

Pri resetu sistema, svi blokovi oba nivoa keša su neinicijalizovani. U drugom nivou keša je smer 0 označen kao žrtva a smer 1 kao sledeća žrtva (slika 19, trenutak t0).

Na potpuno isti način kao i na slici 13 za direktno mapirani keš, blok 0-64 se prvo smešta smer 0 drugog nivoa keša, te istovremeno i u keš za instrukcije. Program zatim zahteva čitanje tog bloka, koji se prihvata u keš za podatke. Smer 1 postaje žrtva, a neinicijalizovani smer 3 postaje sledeće žrtva (slika 19, trenutak t2).

U koraku 2 programa, se čita blok 4096-4160. Prihvat ovog bloka je u direktno mapiranom kešu pravio konflikt prikazan na slici 19. Međutim u asocijativnom kešu je moguće čuvati više različitih blokova sa istim indeksom, te se jednostavno ovaj blok smešta u smer 1. Smer 3 postaje žrtva, a neinicijalizovani smer 2 postaje sledeća žrtva (slika 19, trenutak t2).

U koraku 3 programa, upisuju se vrednosti 4096-4160 na lokacije 4096-4160. Prvi nivo keša poštuje polisu tipa "upis-nazad", te se ništa ne menja u drugom nivou keša, jedino se ovaj blok u prvom nivou označava kao prljav (slika 19, trenutak t3)

Budući da se u svim koracima programa radi sa blokovima čiji je indeks jednak nuli, na sledećim slikama će zbog preglednosti biti prikazana stanja keševa samo za tu vrednost indeksa.

		t0	t1	t2	t3
level 1	instr.	-	0-64	0-64	0-64
	data	-	0-64	4096-4160	4096-4160
level 2	way0	- v	0-64 o	0-64 o	0-64 o
	way1	- nv	- v	4096-4160 o	4096-4160 o
	way2	- -	- -	- nv	- nv
	way3	- -	- nv	- v	- v

Slika 19: Stanje keševa u simulaciji za indeks 0

U koraku 4 test programa, zahteva se čitanje podataka sa lokacija 8192-8256. U ovom trenutku, blok u kešu za podatke je prljav, te je potrebno prvo kopirati nove vrednosti u drugi nivo, pre nego što se preuzme željeni blok. Ovo je prva situacija gde više nema neinicijalizovanih običnih blokova da se izaberu kao sledeća žrtva. Na nasumičan način je između prva dva smer, izabran smer 0 kao sledeća žrtva (slika 20, trenutak t4).

		t4	t5	t6	t7	t8
level 1	instr.	0-64	0-64	-	-	-
	data	8192-8256	12288-12352	16384-16448	20480-20544	8192-8256
level 2	way0	0-64 nv	0-64 v	16384-16448 o	16384-16448 o	16384-16448 nv
	way1	4096-4160 o	4096-4160 o	4096-4160 nv	4096-4160 v	8192-8256 o
	way2	- v	12288-12352 o	12288-12352 o	12288-12352 nv	12288-12352 v
	way3	8192-8256 o	8192-8256 nv	8192-8256 v	20480-20544 o	20480-20544 o

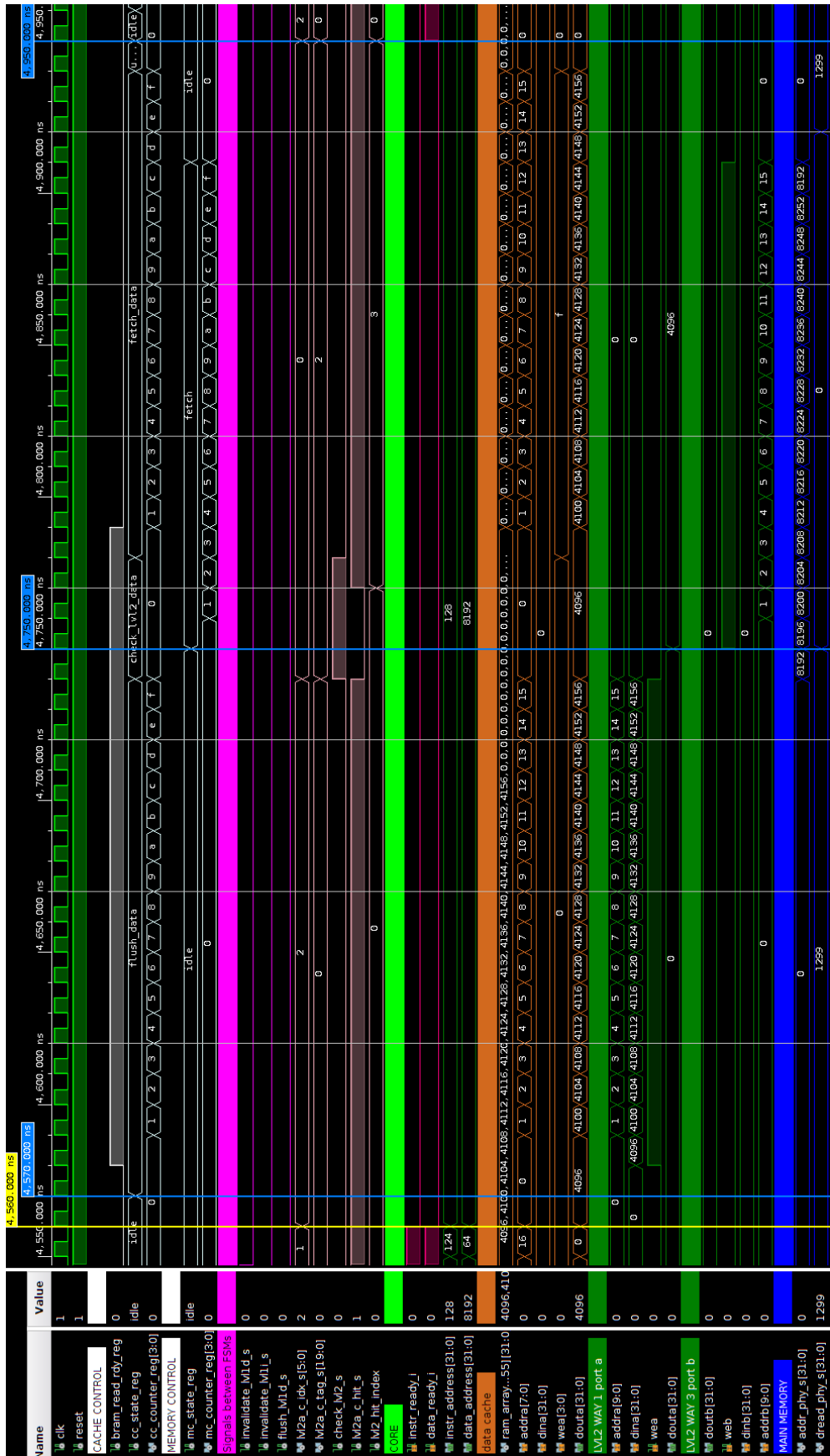
Slika 20: Nastavak stanja keševa u simulaciji za indeks=0

Na slici 21, u trenutku 4.560ns se zahteva čitanje podatka iz bloka 8192-8256, te zbog promašaja u kešu za podatke, signal *data_ready* pada na logičku nulu. Zato što je blok u kešu za podatke prljav, automat "cache controller" prelazi u stanje *flush_data*, kako bi prvo ažurirao drugi nivo keša. Novi podaci se preko porta "a" prenose u smer 1 drugog nivoa keša. Može se primetiti da se signal dozvole upisa *wea* smera 1 postavlja na jedinicu, te da se podaci 4096-4156 upisuju na prvih 16 mesta. Tek nakon što se drugi nivo keša ažurira, može se pristupiti prihvatu bloka 8192-8256 iz operativne memorije. U simulacionom trenutku 4750ns, automat "memory controller" prelazi u stanje *fetch*. Dok je u ovom stanju, čita blok iz memorije, te pomoću porta "b" upisuje u smer 3 koji je označen kao žrtva. Istovremeno, "cache controller" automat u stanju *fetch_data*, pomoću porta "a" smera 3, upisuje ovaj blok u memoriju za podatke. Nakon što se poslednja reč bloka upiše u keš za podatke, "cache controller" prelazi u stanje *update_data_ts* gde se ažurira memorija za čuvanje tagova. Na sledeću rastuću ivicu, se *data_ready* signal postavlja na jedinicu (simulacioni trenutak 4950ns).

U koraku br. 5 test programa, zahteva se čitanje podataka sa lokacija 12288-12352. Blok se upisuje u smer 2, dok je na nasumičan način smer 3 odabran kao sledeća žrtva (slika 20, trenutak t5). Korak br. 6 na sličan način prihvata blok 16384-16448. On se smešta u smer 0, u kojem je do sada bilo prvih 16 instrukcija test programa. Stoga se evikcijom ovog bloka mora invalidirati blok u kešu za instrukcije (slika 20, trenutak t6). Korak br. 7 test programa prihvata blok 20480-20544 u smer3, te u ovom trenutku smer 1 postaje žrtva (slika 20, trenutak t7).

Na slici 22, u trenutku 10710ns se zahteva ponovno čitanje podataka iz bloka 8192-8256, koji se ne nalazi u drugom nivou keša (signal *lvl2_hit_s* je jednak nuli, *data_ready* pada na nulu). Žrtva je blok u smeru 1, koji sadrži prljav blok 4096-4160. Kako bi se eviktovao ovaj blok, moraju se podaci kopirati u operativnu memoriju pre prihvata novog bloka. "Memory controller" automat stoga u trenutku 10740ns prelazi u stanje *flush*. U ovom stanju se pomoću porta "b" smera 1 čita reč po reč i upisuje u operativnu memoriju na adrese 4096-4156. Neposredno nakon toga, u trenutku 10910ns, "memory controller" prelazi u stanje *fetch*. U ovom stanju se čita blok podataka iz operativne memorije sa lokacija 8192-8252 te preko porta "b" upisuje u smer 1 drugog nivoa keša. Na isti način kao i u prethodnim primerima, istovremeno se blok prebacuje u keš za podatke, što se vidi na slici kada je "cache controller" u stanju *fetch_data*. Upis bloka u keš za podatke se završava, ažurira se memorija za čuvanje tagova te se signal *data_ready* postavlja na jedinicu (simulacioni trenutak 11110ns).

Bitno je obratiti pažnju da je program napisan tako da u beskonačnoj petlji čita četiri različita bloka sa istim indeksom. Ovo u određenom trenutku mora dovesti do evikcije prljavog bloka u operativnu memoriju. Nakon implementacije sistema na Zybo ploči, neće se imati uvid u stanja keš memorija, te će se testirati da li se izvršava prethodno opisana evikcija sa slike 22. Ukoliko se stanje operativne memorije na lokacijama 4096-4160 promeni sa inicijalnih nula na vrednosti 4096-4160, prihvati i evikcija između različitih nivoa keševa i memorije rade pravilno, kao i u simulaciji.



Slika 21: Evikcija prljavog bloka iz prvog nivoa keša

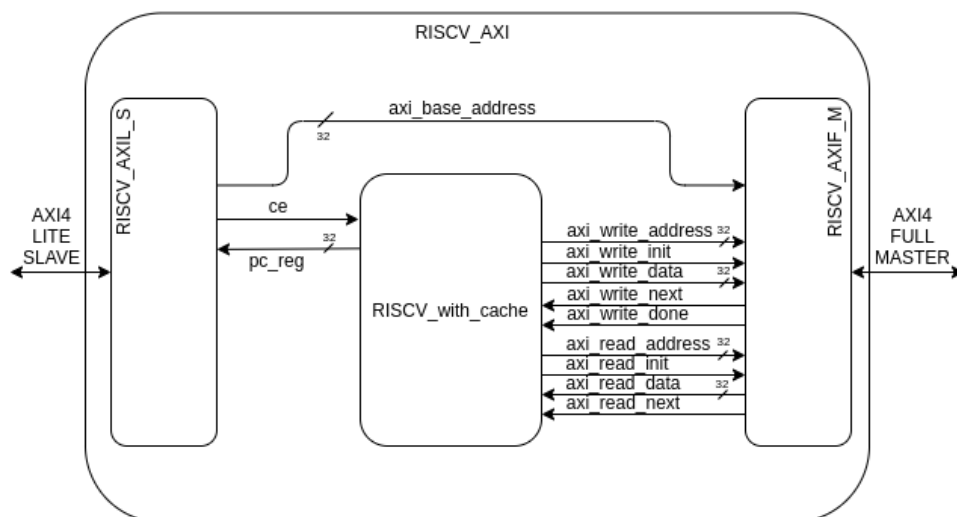
4 Implementacija sistema na Zybo razvojnoj ploči

Implementacija sistema će biti izvršena u Vivado alatu kompanije Xilinx. Ciljana razvojna ploča je Zybo, kompanije Digilent, koja na sebi ima Zynq-7000 sistem na čipu. Na ovome SoC-u se nalazi dvojezgarni ARM Cortex-A9 procesor koji radi na frekvenciji 650 MHz. Ploča poseduje dva DDR3 memorijska čipa, koji prave 32-bitni interfejs ka memoriji kapaciteta 512MB i propusnim opsegom 1050Mbps. Na Zynq SoC-u već ugrađen memorijski kontroler sa 8 DMA kanala za direktan pristup memoriji.

Za potrebe testiranja, sistem za keširanje sa RISC-V procesorom će biti implementiran u programabilnoj logici, te će jedan od memorijskih kanala biti iskorišten za komunikaciju sa DDR memorijom. Interfejs sa memorijom je tipa *AXI3-Full* širine 64 bita.

4.1 Pakovanje IP jezgra

Kako bi u bilo moguće uvesti kontrolne i statusne signale za procesor kao i omogućiti da keš sistem pravilno komunicira sa DDR memorijskim čipom, moraju se modelovati *AXI-Lite Slave* i *AXI-Full Master* interfejsi. Zapakovano IP jezgro će izgledati kao na slici 23, te se sastoji od dva dodatna modula:



Slika 23: Procesor sa sistemom za keširanje, oklopljen AXI interfejsom

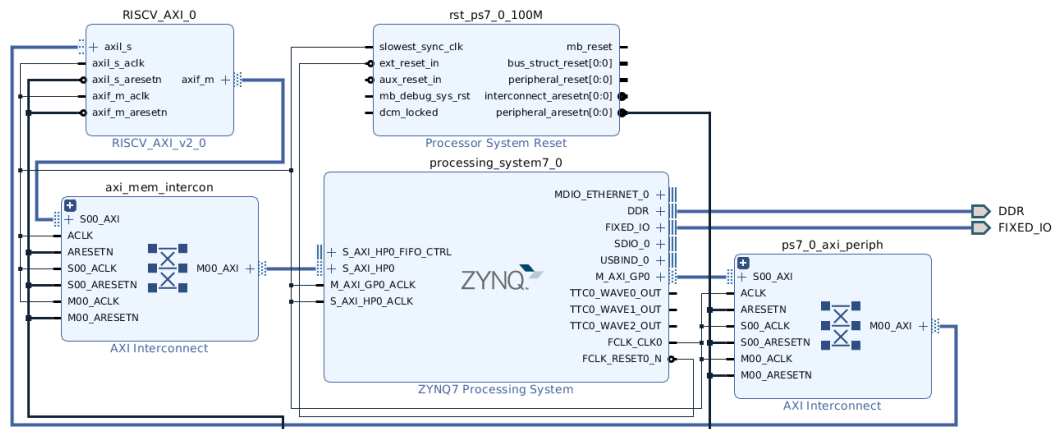
- **RISCV_AXIL_S** - autogenerisani modul koji implementira *AXI Lite Slave* interfejs. U ovom interfejsu će biti korištena samo 3 registra:
 - *axi_base_address* - bazna adresa svih transakcija koje se izvršavaju ka operativnoj memoriji preko AXI Full interfejsa. Na ovaj način, procesor će krenuti da izvršava program počevši od adrese *axi_base_address*.
 - *ce* - signal dozvole rada procesora. Pri resetu sistema je na nuli, te procesor čeka. Kada se upiše jedinica na bit najmanje važnosti (*LSB*) u ovom registru, procesor i sistem za keširanje kreću sa radom.

- *pc_reg* - statusni registar koji indikuje trenutno stanje programskog brojača. Na ovaj način je moguće pročitati u kojoj fazi izvršavanja programa se nalazi procesor.
- ***RISCV_AXIF_M*** - autogenerisani modul koji implementira *AXI Full Master* interfejs. Kako bi se inicirala transakcija na ovom interfejsu potrebno je komunicirati sa ovim modulom pomoću sledećih portova:
 - *axi_base_address* - 32-bitna bazna adresa (offset) koja se sabira sa traženim adresama za transakcije upisa i čitanja.
 - *axi_write_address* - 32-bitna početna adresa upisa blok podataka u memoriju.
 - *axi_write_init* - port za iniciranje transakcije upisa. Kada se detektuje rastuća ivica, modul započinje transakciju upisa bloka podataka.
 - *axi_write_data* - 32-bitni port za slanje podataka u memoriju.
 - *axi_write_next* - port koji indicira da je memorija spremna za upis sledećeg 32-bitnog podataka u bloku.
 - *axi_write_done* - port koji indicira da je blok podataka uspešno upisan u memoriju.
 - *axi_read_address* - 32-bitna početna adresa za čitanje bloka podataka iz memorije.
 - *axi_read_init* - port za iniciranje transakcije čitanja. Kada se detektuje rastuća ivica, modul započinje transakciju čitanja bloka podataka.
 - *axi_read_data* - 32-bitni port preko kojeg se čitaju podaci.
 - *axi_read_next* - port koji indicira da je memorija spremna za čitanje sledećeg 32-bitnog podataka u bloku.

RISCV_with_cache modul koji je simuliran u prethodnom poglavlju, je potrebno modifikovati kako slanje i čitanje bloka podataka iz memorije radio preko komunikacije sa interfejsom modula *RISCV_AXIF_M* kao na slici 23. Jedine modifikacije su u "memory controller" automatu koji i dalje zadržava ista tri stanja: *idle*, *fetch* i *flush*. U stanju *fetch* se podaci preuzimaju pomoću prethodno opisanih *axi_read_** signala, dok se u stanju *flush* podaci upisuju u memoriju pomoću *axi_write_** signala. Čitanje i upis više nisu sa jednim taktom kašnjenja. Automat postavi signal *axi_read_init* na jedinicu, te čeka u stanju *fetch* dok se signal *axi_read_next* ne postavi na logičku jedinicu. Kada se memorija bude spremna, blok podataka se preko *axi_read_data* porta može čitati u kontinuitetu (4B po periodu takta). Na sličan način, automat postavi signal *axi_write_init* na jedinicu, te čeka u stanju *flush* dok se signal *axi_write_next* ne postavi na jedinicu. Čim se transakcija čitanja pravilno inicira, moguće je poslati blok podataka preko *axi_write_data* porta u kontinuitetu (4B po periodu takta).

4.2 Blok dizajn u Vivado alatu

Prethodno opisano IP jezgro je povezano sa Zynq procesorskim sistemom kao na slici 24. 32-bitni AXI4 Master IP jezgra *RISCV_AXI* je povezan na 64-bitni AXI3 Slave interfejs sa Memorijkog kontrolera preko *axi_mem_intercon* modula. Interkonekt modul pravi spregu između AXI4 i AXI3 protokola, te vrši pakovanje podataka u 64-bitni interfejs.



Slika 24: Blok dizajn sistema

Dodatno je omogućena funkcija interkonekta za unutrašnji FIFO bafer, kako bi potencijalna memorijska kašnjenja unutar transakcije bila neprimetna iz tačke gledišta *RISCV_AXI* IP jezgra (slika 25). Na ovaj način, kada se pošalje ili primi prvi podatak, ostali podaci iste transakcije će biti poslati ili primljeni u kontinuitetu, tj. takt za taktom, bez pauza.



Slika 25: Podešavanja AXI interkonekta

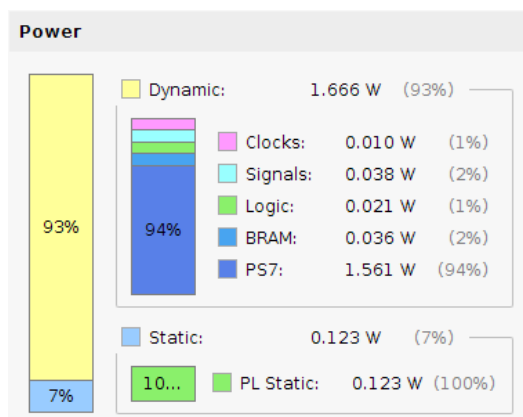
Sinteza i implementacija su izvršene nad sistemom sa slike 24. Frekvencija rada sistema je podešena na 100Mhz. Vrednosti parametara sistema za keširanje su postavljene kao i u simulaciji. Na slici 26 se može videti da je implementacija uspešno završena za ovu konfiguraciju sistema. Iskorištenost resursa nakon implementacije se može videti na slici 27, a potrošnja energije na slici 28.

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS
✓ synth_final (active)	constrs_1	synth_design Complete!					
✓ impl_final	constrs_1	phys_opt_design (Post-Route) Complete!	0.027	0.000	0.022	0.000	0.000

Slika 26: Rezultati implementacije

Resource	Utilization	Available	Utilization %
LUT	5060	17600	28.75
LUTRAM	512	6000	8.53
FF	3308	35200	9.40
BRAM	8	60	13.33
BUFG	1	32	3.13

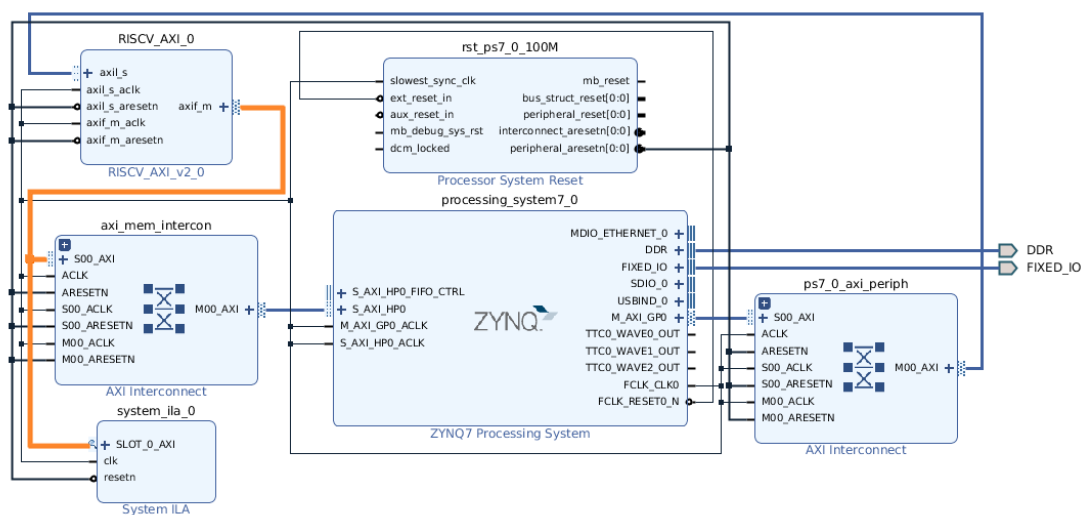
Slika 27: Iskorištenost resursa nakon implementacije



Slika 28: Potrošnja energije nakon implementacije

4.3 Testiranje u Vitis alatu

Za svrhe testiranja će se u blok dizajn prikazan na slici 24, dodati modul *System ILA* (*Integrated Logic Analyzer*). Ovaj modul će biti povezan na AXI Full interfejs *RISCV_AXI* modula, te će koristiti blok RAM module da zabeleži transakcije pri testiranju na Zybo ploči (slika 29). Na ovaj način možemo proveriti da li se transakcije koje inicira sistem za keširanje poklapaju sa zahtevima programa.



Slika 29: Blok dizajn sistema sa ILA modulom

Test program je urađen kao samostalna C aplikacija. Prvobitno se napravi statički niz "main_memory" čiji će opseg služiti kao memorija za RISC-V procesor. Niz se sastoji od 32-bitnih podataka, što znači da ukoliko se želi dobiti stvarna ofset lokacija u memoriji, potrebno je indeks člana niza pomnožiti 4 (memorija je bajt adresibilna, a niz nije). Pri deklaraciji se niz inicijalizuje na sve nule. Za proveru da li je stek pravilno inicijalizovan, u terminalu se ispisuju prvi i poslednji blok u ovom nizu.

Mašinski kod test programa je u smešten u niz "assembly" u pomoćnom fajlu "assembly.h". U jednostavnoj "for" petlji se mašinski kod smešta na najniže pozicije u nizu. Ponovo se ispisuje prvi blok u nizu koji sada sadrži prvih 16 instrukcija test programa. Takođe se ispisuje u terminalu blok na adresama 4096-4160. Ove lokacije su jednake nuli zbog prvobitne inicijalizacije niza.

Pomoću AXI Lite interfejsa se adresa početnog člana niza "main_memory" pošalje u registar za ofset. Ova adresa se takođe ispisuje u terminalu. Procesor je sada konfigurisan da čita i izvršava instrukcije počevši od adrese prvog člana niza (gde se nalazi mašinski kod test programa). Neposredno nakon toga, se preko istog interfejsa registar za signal dozvole takta postavi na logičku jedinicu, te procesor kreće sa izvršavanjem programa.

Nakon kraćeg kašnjenja, resetuje se registar za signal dozvole takta, čime se izvršavanje programa zaustavlja. Ponovo se u terminalu ispisuje sadržaj blokova 0-64 i 4096-4160. Može se приметiti da je vrednost niza na memorijskim lokacijama 4096-4160 promenjena. Ovo znači da se test program uspešno izvršio i da je prljavi blok uspešno eviktovan nazad u memoriju, kao što je bio slučaj u simulaciji. Rezultati izvršavanja programa se mogu videti na slici 30.

```
PC value: 0
CE value: 0

**** Checking memory array to see if it's initialized to all zeros ****
*** First and last block of memory, each location is 32b = 4B ***
*** Pairs [address:data] will be displayed ***

* First block *
0:0x00000000; 4:0x00000000; 8:0x00000000; 12:0x00000000; 16:0x00000000; 20:0x00000000; 24:0x00000000; 28:0x00000000;
32:0x00000000; 36:0x00000000; 40:0x00000000; 44:0x00000000; 48:0x00000000; 52:0x00000000; 56:0x00000000; 60:0x00000000;

* Last block *
8128:0x00000000; 8132:0x00000000; 8136:0x00000000; 8140:0x00000000; 8144:0x00000000; 8148:0x00000000; 8152:0x00000000; 8156:0x00000000;
8160:0x00000000; 8164:0x00000000; 8168:0x00000000; 8172:0x00000000; 8176:0x00000000; 8180:0x00000000; 8184:0x00000000; 8188:0x00000000;

**** Checking memory array after initialization with machine code ****
*** First block and target block will be displayed ***

* First block [0-64]: *
0:0x00000513; 4:0x00000593; 8:0x04000613; 12:0x00052883; 16:0x00450513; 20:0x00458593; 24:0x00c58463; 28:0xff1ff0ef;
32:0x00001737; 36:0x04070713; 40:0x00e50663; 44:0x00001537; 48:0xfd5ff0ef; 52:0x00001537; 56:0x00000593; 60:0x04000613;

* Target block [4096-4160]: *
4096:0; 4100:0; 4104:0; 4108:0; 4112:0; 4116:0; 4120:0; 4124:0;
4128:0; 4132:0; 4136:0; 4140:0; 4144:0; 4148:0; 4152:0; 4156:0;

This is first memory address 0x112300

**** Checking memory array after program execution ****
*** First block and target block will be displayed ***

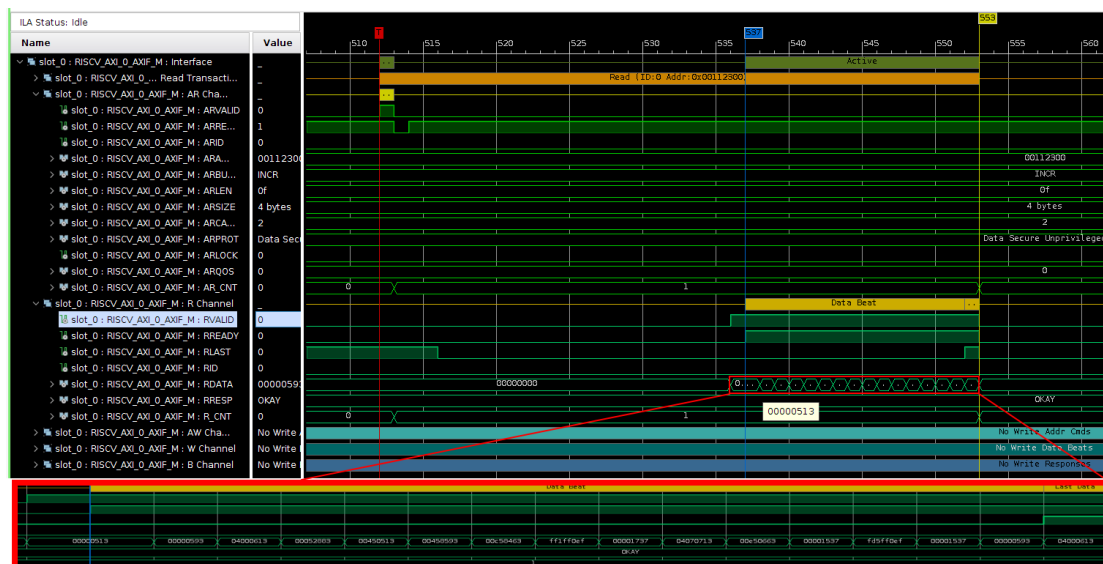
* First block [0-64]: *
0:0x00000513; 4:0x00000593; 8:0x04000613; 12:0x00052883; 16:0x00450513; 20:0x00458593; 24:0x00c58463; 28:0xff1ff0ef;
32:0x00001737; 36:0x04070713; 40:0x00e50663; 44:0x00001537; 48:0xfd5ff0ef; 52:0x00001537; 56:0x00000593; 60:0x04000613;

* Target block [4096-4160]: *
4096:4096; 4100:4100; 4104:4104; 4108:4108; 4112:4112; 4116:4116; 4120:4120; 4124:4124;
4128:4128; 4132:4132; 4136:4136; 4140:4140; 4144:4144; 4148:4148; 4152:4152; 4156:4156;

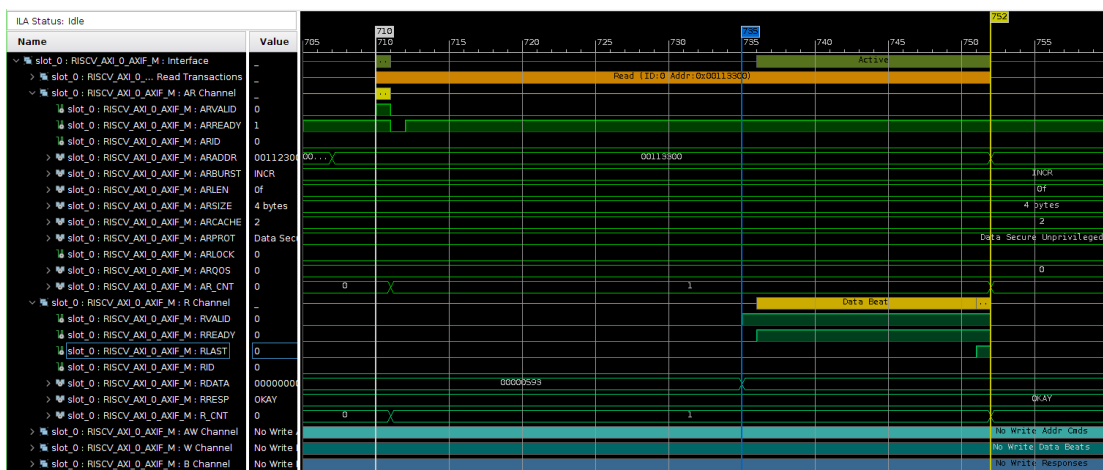
* If target blocks have values 4096-4156, the program executed as expected *
```

Slika 30: Stanje terminala nakon izvršavanja C aplikacije

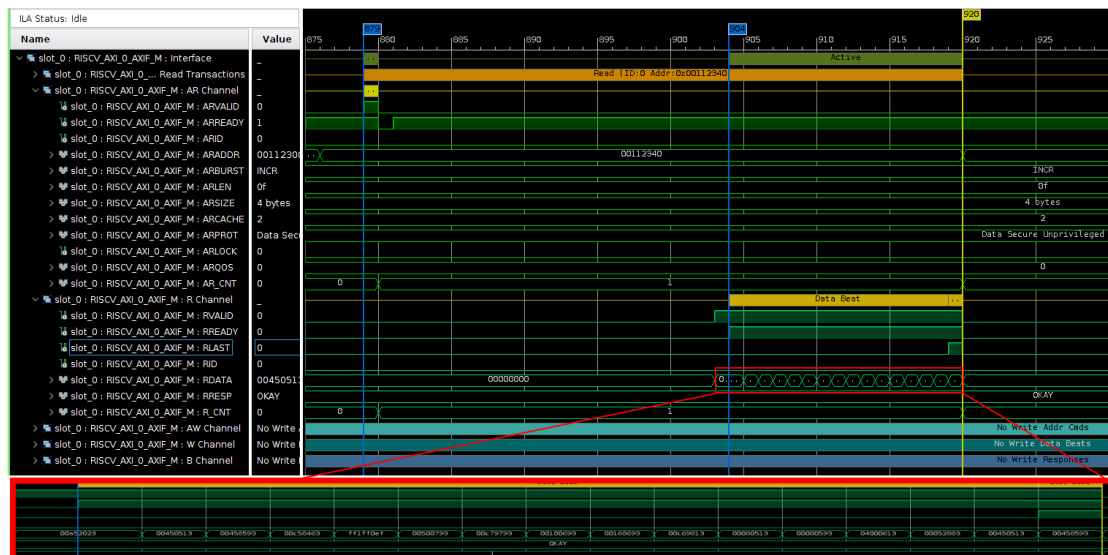
Za dodatnu proveru se mogu proveriti stanja ILA modula. Kada se konfigurira da detektuje transakcije čitanja iz memorije, nakon pokretanja aplikacije se dobiju tri transakcije. Prva transakcija čita sa adrese 0x0012300 što je početna adresa niza "main_memory" iz aplikacije. Pročitani podaci su mašinski kod instrukcija test programa (slika 31). Drugo čitanje je sa lokacija 0x0013300, što je ofset od 4096 u odnosu na početnu adresu. Pročitane vrednosti su sve nule, što se poklapa sa simulacijom (slika 32). Poslednje čitanje koje je *ILA* modul sačuvao, je sa lokacije 0x112340, što je ofset od 64 u odnosu na početnu adresu. Dakle, sledeće čitanje je drugi blok instrukcija mašinskog koda, što se vidi iz pročitanih podataka (slika 33).



Slika 31: Prvo čitanje bloka iz memorije

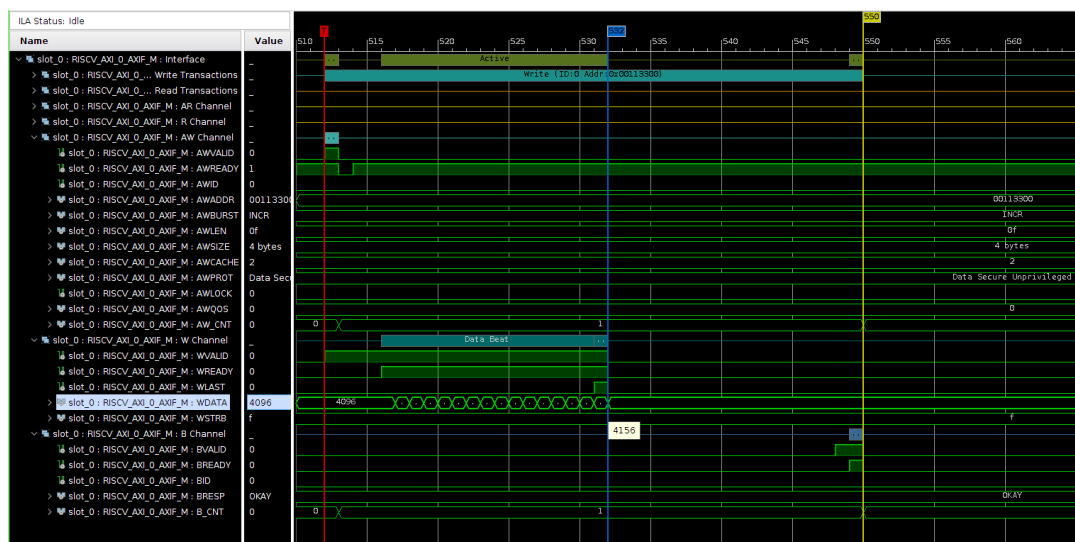


Slika 32: Drugo čitanje bloka iz memorije



Slika 33: Treće čitanje bloka iz memorije

Iz rezultata *ILA* modula se može izvući još jedna bitna informacija. Od zahteva bloka do čitanja poslednjeg podataka iz memorijskog čipa utroši se oko 42 takta. Sam prenos podataka zahteva 16 taktova, dok se ostalih 26 izgubi u komunikaciji sa memorijskim čipom. Ukoliko se ovo uporedi sa prenosom bloka iz drugog nivoa keša što zahteva 16 taktova za prenos podataka plus 3 takta za proveru tagova i ažuriranje pomoćnih bita (2 takta u *low_latency* konfiguraciji), očigledno je zašto je memorijski sistem sa više nivoa keša neophodan za dobre performanse procesora. Može se takođe zaključiti da za manje blokove, odnos korisnih taktova (tokom kojih se prenose podaci) u odnosu na utrošene u komunikaciji sa DDR čipom, naglo opada.



Slika 34: Upis bloka u memoriju

Ukoliko se *ILA* modul podesi za detekciju upisa u memoriju, dobije se jedna transakcija kao rezultat. To je transakcija u kojoj se eviktuje prljavi blok 4096-4160, čija je detekcija bila cilj test programa (slika 34). Od inicijalizacije transakcije upisa u memoriju, do odgovora memorije da je upis izvršen uspešno, utroši se 38 taktova.

5 Zaključak

Modularna RISC-V arhitektura izgleda kao obećavajući standard za širok spektar primena u računarstvu. Posebno primamljiva primena predstavlja svet programabilnog hardvera i FPGA čipova, gde je RISC-V idealna zamena za dosadašnje *soft-core* arhitekture. Veliko zainteresovanje i podrška koju je dobila RISC-V organizacija, uzrokovala je stvaranju mnoštva alata i softverske podrške što je uvek bila slaba tačka ostalih *soft-core* arhitektura. Kada se dodatno uračuna i otvorenost različitih RTL implementacija procesora, očigledno je da će RISC-V postati standard za primenu na FPGA pločama.

Ovaj rad demonstrira da su kašnjenja pristupa memoriji neprihvatljiva čak i za najjednostavnija procesorska jezgra koja rade na relativno niskim frekvencijama. Kako bi se premostila razlika između brzine rada procesora i tipičnog DDR3L čipa, u *soft-core* primeni RISC-V arhitekture, mogu se iskoristiti već postojeći memorijski resursi na FPGA ploči kako bi se implementirao jednostavan a efikasan sistem za keširanje. Distribuirani (LUT) RAM može da obezbedi prvi nivo keša malog kapaciteta ali dovoljno brzog odziva da zadovolji zahteve pristupa memoriji bez zaustavljanja protočne obrade. Nasuprot tome, blok RAM može da obezbedi dovoljno velik kapacitet keša kako bi se aktivan set podataka aplikacije čuvao na FPGA čipu.

HDL model koji je predstavljen u radu je samo jedan način realizacije sistema za keširanje. Potrebno istražiti alternativne implementacije kao i optimizacije datog sistema. Podela portova drugog nivoa keša za specifičnu namenu zajedno sa implementiranom inkluzivnošću i pamćenjem prisustva blokova prvog nivoa u pomoćnim bitima, izgleda kao dobro rešenje za sistem sa više RISC-V jezgara. Prisustvo ovih informacija u drugom nivou keša u kombinaciji sa slobodnim portom u memoriji za čitanje tagova, je odlična osnova za laku implementaciju MOESI (*Modified, Owned, Exclusive, Shared, Invalid*) ili nekog sličnog algoritma za održanje koherencije između jezgara.

6 Literatura

[1] RISC-V Foundation, *In The News*.

Preuzeto sa: <https://riscv.org/news/in-the-news/>

[2] RISC-V Foundation, *Why RISC-V?*

Preuzeto sa: <https://riscv.org/why-risc-v/>

[3] James, C.H. *Memory Technology and Organization*. [predavanje] Department of ECE, Carnegie Mellon University, 2020.

Preuzeto sa: <http://users.ece.cmu.edu/~jho/courses/ece447/S20handouts/L14.pdf>

[4] Wilkes, *Slave Memories and Dynamic Storage Allocation*, IEEE Trans. On Electronic Computers, 1965.

[5] Liptay, *Structural aspects of the System/360 Model 85 II: the cache*, IBM Systems Journal, 1968.

[6] Xilinx, *7 Series FPGAs CLB User Guide (UG474)*, 2016

Preuzeto sa:

https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

[7] Xilinx, *Zynq-7000 SoC Data Sheet: Overview (DS190)*, 2018

Preuzeto sa:

https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[8] Xilinx, *7 Series FPGAs Memory Resources User Guide (UG473)*, 2018

Preuzeto sa:

https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

[9] James Bell, David Casasent i C. Cordon Bell *An Investigation of Alternative Cache Organizations*, IEEE Trans. On Electronic Computers 1974.

[10] Đ. Mišeljić i N. Kovačević – *Napredni mikroprocesorski sistemi, Upoznavanje sa RISC-V procesorom*, 2019.