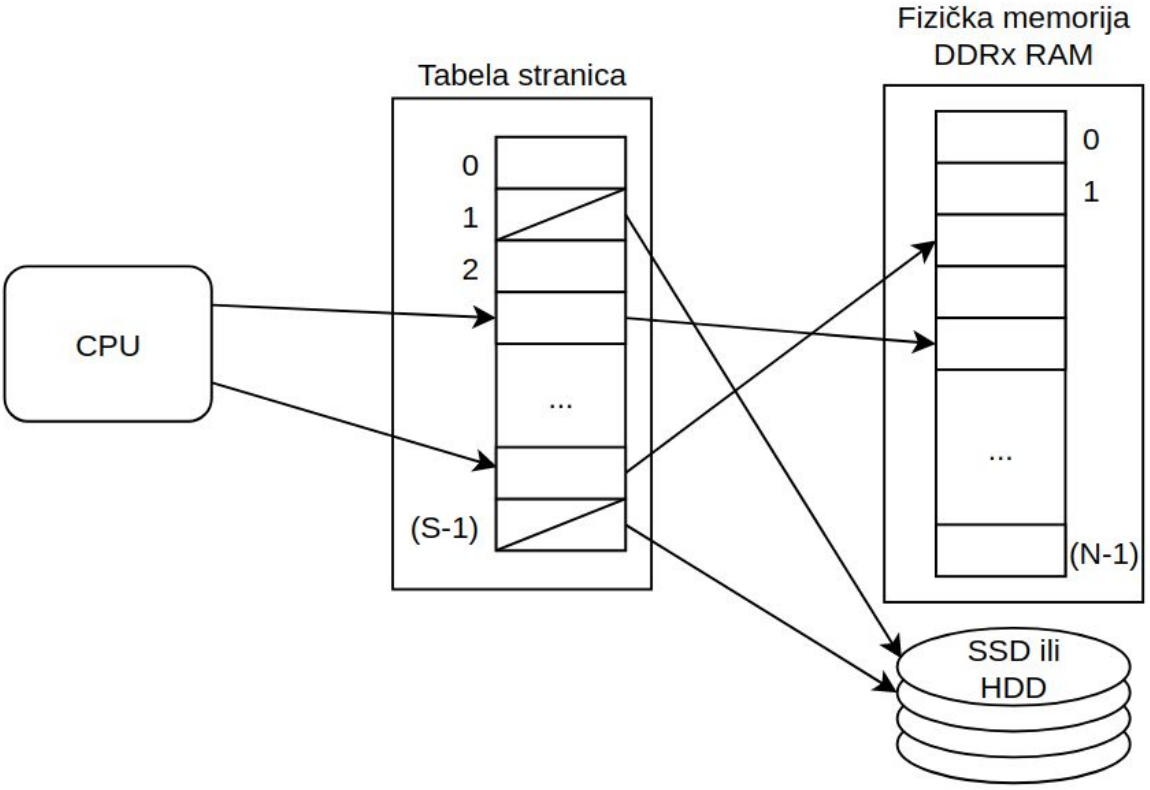
Vežba 4
Skrivene (keš) memorije

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 momorijskih 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, stranica, (*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 menagement 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 iskorištenje 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.

2. Tipovi memorija i hijerarhija memorijskog sistema

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

1) Beskonačan kapacitet

2) Beskonačan propusni opseg

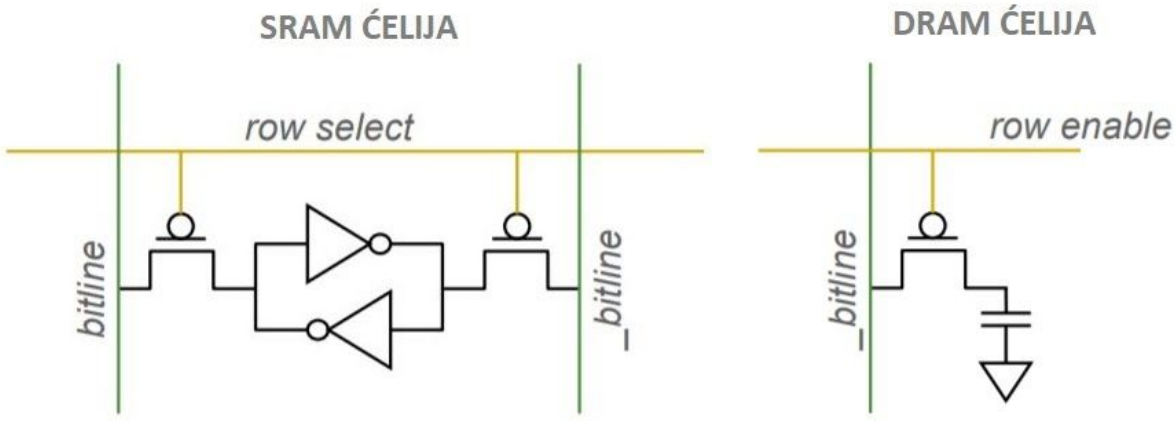
3) Bez kašnjenja

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 ~ 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*. *volotile*) memorije: SRAM i DRAM, demonstriraju ove konflikte u savremenim računarima. Ćelija 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 odziv ove memorije umnogome sporijim 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.



Slika 2: Tipične strukture ćelija za čuvanje bita informacije

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.

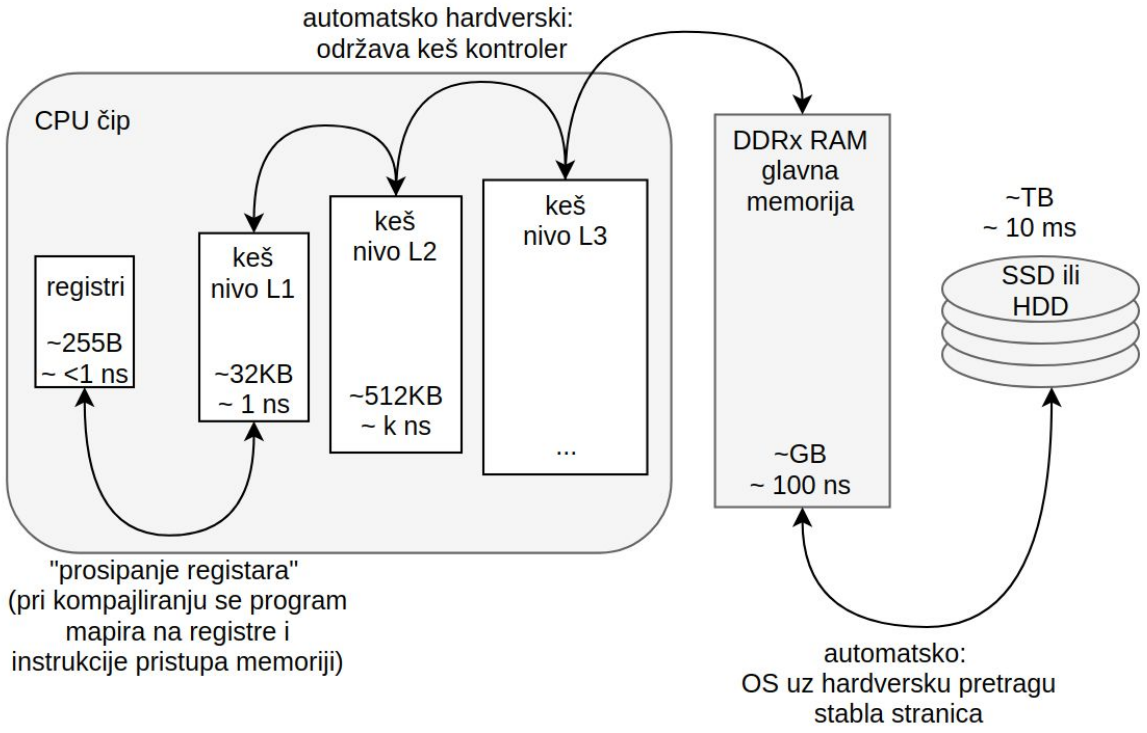
1. Vremenska lokalnost se izražava u tome, da ukoliko se pristupi nekom resursu, velika je verovatnoća da će u narednom bliskom vremenskom periodu ponovo pristupiti tom istom resursu.
2. Prostorna lokalnost se izražava u tome, da ukoliko se pristupi nekom resursu, postoji velika verovatnoća da će se 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.

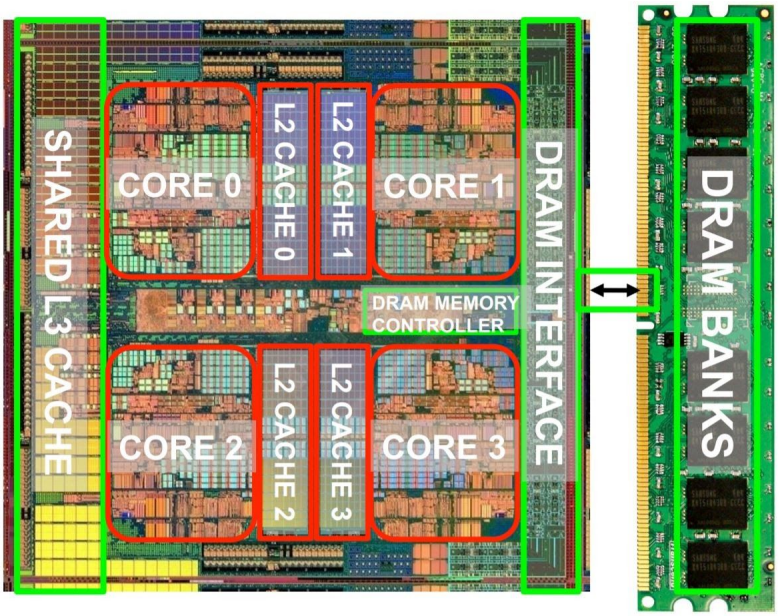
Još 1965. godine je Vilkis u radu "*Slave Memories and Dynamic Storage Allocation*" predložio da, ukoliko se iskoristi manja a brža pomoćna (*eng. slave*) memorija za čuvanje često referenciranih podataka, u praktčnim slučajevima će efektivna brzina pristupa memoriji biti bliža bržoj memoriji nego sporijoj [1]. Ovaj

koncept je zatim implementiran u IBM 360/85 računaru, 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) [2].

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 virtuelizacije 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 3.



Slika 3: Blok dijagram memorijskog podsistema



Slika 4: Memorijski podsistem u savremenom računaru

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 ... n-1* dok se ne pronađe željeni podatak. Iz ovoga proizlaze formule 1, 2 i 3.

$$h_i + m_i = 1 \tag{1}$$

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

$$T_i = t_i + m_i * T_{i+1} \tag{3}$$

Formula broj 3 se dalje može rekurzivno 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:

1. **Blok RAM**

   Postoji 60 blokova, gde se svaki može konfigurisati kao dvopristupna memorija veličine 36Kb ili dve jednopristupne memorije veličine 18Kb [3]. 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.

2. Distribuirani (LUT RAM)

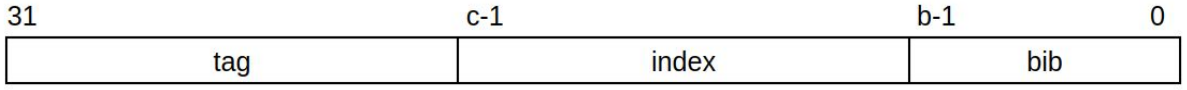
Od prisutnih 17,600 *slice* modula, 6000 su tipa *slicem*, dok je ostatak tipa *slicel* [4]. Svaki *slicem* modul se može konfigurisati kao 64 bita brze RAM memorije [5]. Čitanje iz ove memorije je asinhrono, ali se može napraviti izlazni registar za bolju *clk-to-output* karakteristiku. Ukupno je na raspolaganju 375Kb ovog tipa memorije.

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).

3. Direktno preslikan keš

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.

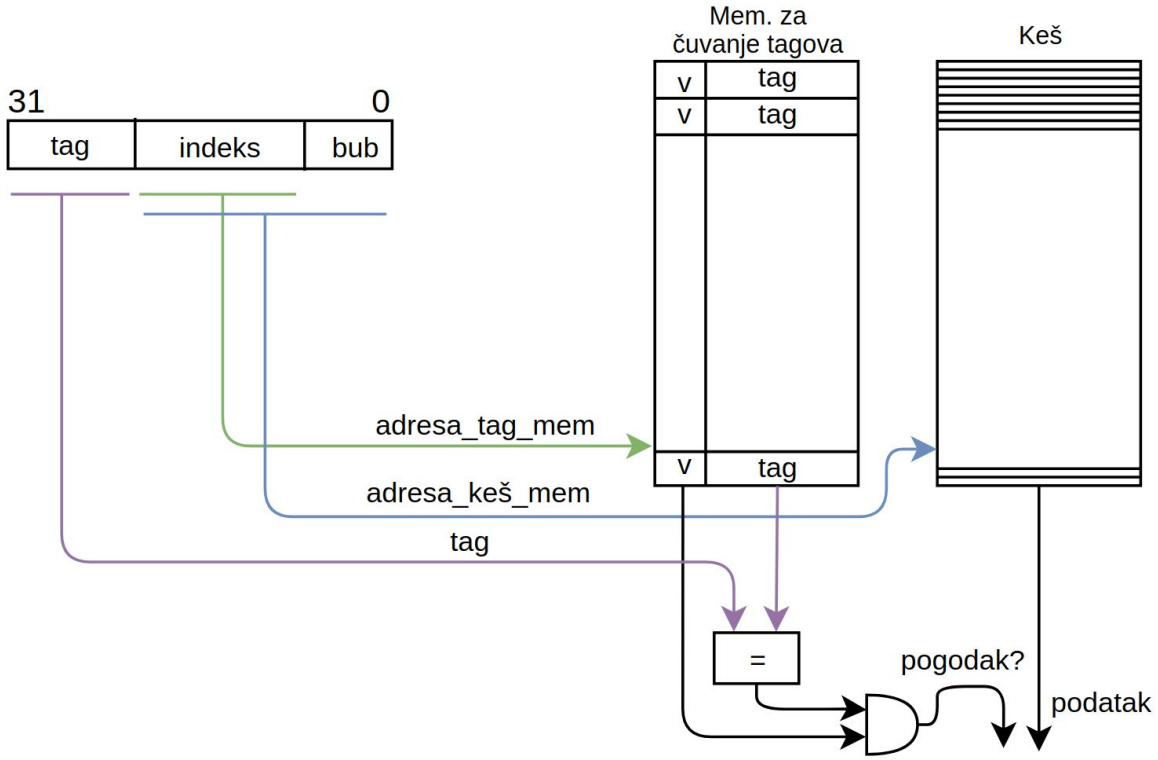


Slika 5: Podela adresa u polja

Ukoliko se *c* bita koristi za indeksiranje keš memorije, i *b* bita za indeksiranje bajta u bloku, veličina keš memorije je $8*2^c$ bita a memorije za čuvanje tagova $(32-c)*2^{(c-b)}$ 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. Premala veličina bloka ne iskorištava dovoljno efikasno prostornu lokalnost, te povećava količinu memorije potrebne za čuvanje tagova. Previše velika veličina bloka rezultuje u manjem broju blokova u kešu, te nedovoljno iskorištenom vremenskom lokalnosti. U tom slučaju se takođe i bespotrebno opterećuju magistrale pri promašajima u kešu zbog prebacivanja velike količine podataka.

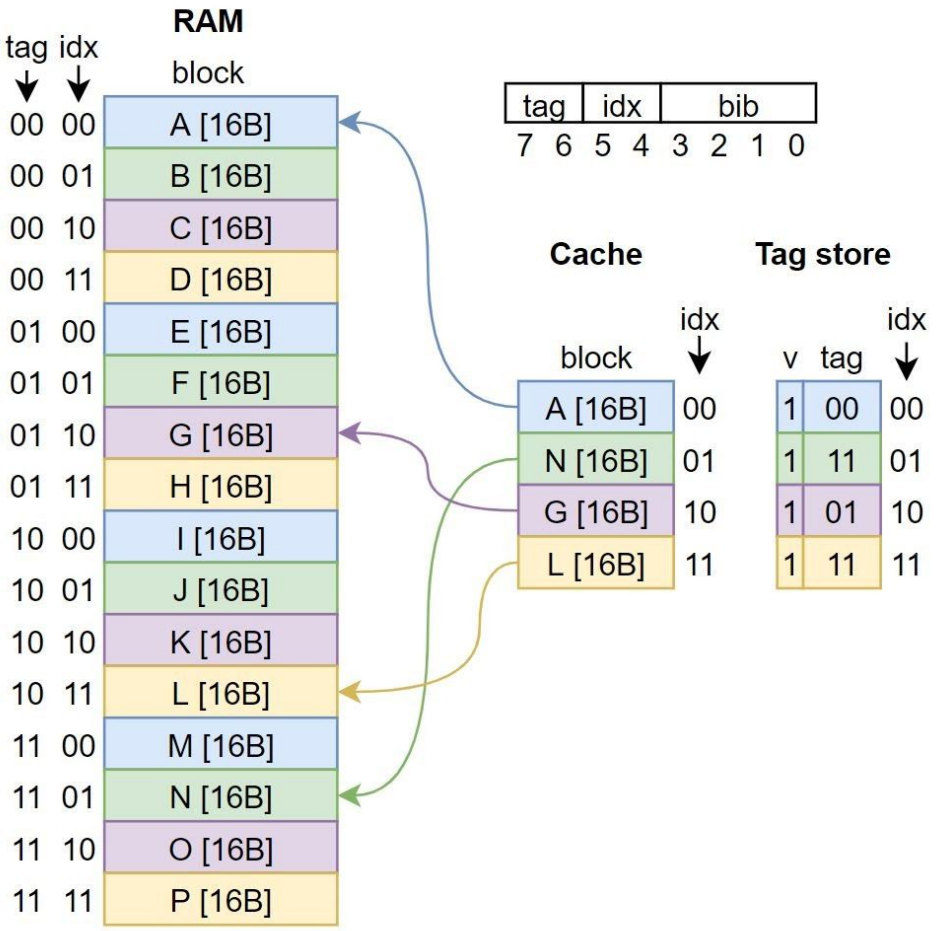
Direktno mapirani keš funkcioniše na način koji je prikazan na slici 6. Kada se podatak referencira pomoću neke adrese, donjih *c* bita adresira keš memoriju, dok polje *index* adresira memoriju za čuvanje tagova. Iz memorije za čuvanje tagova se čita sačuvano tag polje i *valid* bit. Ukoliko se pročitani 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 desio promašaj u prethodnom. Prvi način povećava performans ali uveliko i potrošnju energije.



Slika 6: Blok dijagram direktno preslikanog keš

Na slici 7 je dat jednostavan primer bajt-adresibilne memorije sa 8-bitnom adresom. Ukupna veličina memorije je 256B. Blok je veličine 16 bajtova, te se najniža 4 bita ($\log_2 16$) adrese koriste za adresiranje bajta u bloku (eng. *byte in block*). Sa veličinom direktno mapiranog keša od 64B, mogu se čuvati četiri bloka iz operativne memorije, te je index (idx) polje širine dva bita. Različitim bojama u operativnoj memoriji su označeni blokovi sa istim indeksom. Kod direktno mapiranog keša, samo jedan od blokova sa određenim indeksom može istovremeno biti u keš memoriji. Na primer u kešu na adresi sa indeksom “0b00” može biti samo jedan od blokova: A, E, I ili M. Kako bi se znalo koji od ta četiri bloka se trenutno nalazi u kešu, njegovo “tag” polje se čuva u memoriji za čuvanje tagova (eng. *Tag store*).

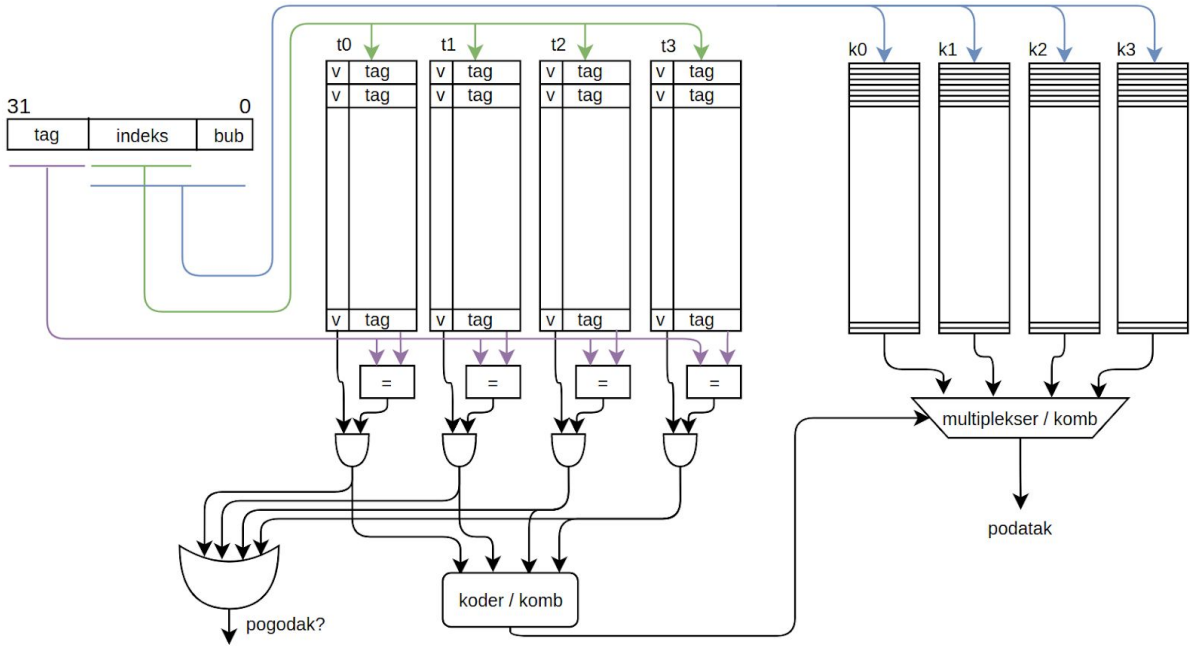


Slika 7: Primer jednostavnog direktno preslikanog keša

4. Set asocijativan keš

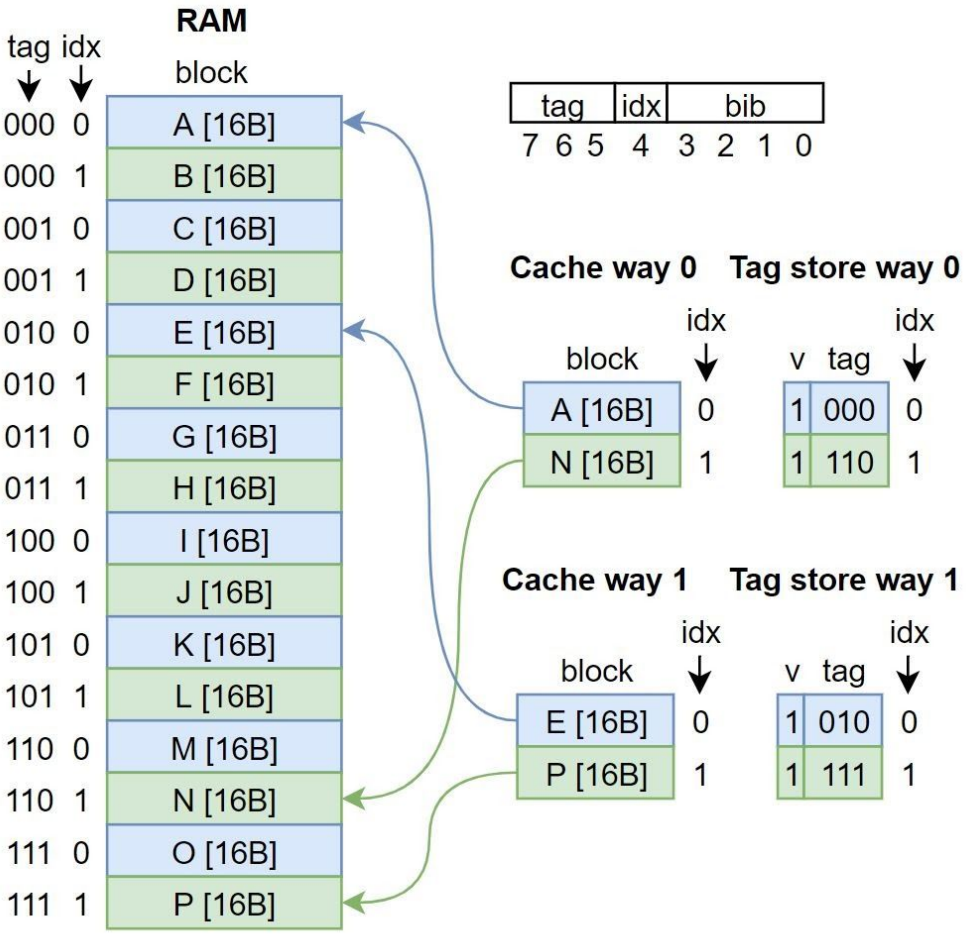
Kako bi se izbegle sekvence pristupa koje lako dovode do nultog udela pogodaka, keš se često pravi da bude set asocijativan. Keš se podeli u više manjih jednakih delova, svaki sa svojom memorijom za čuvanje tagova. Ukoliko se podeli na N delova, u istom trenutku može da čuva N podataka sa istim indeksom. 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 8 je prikazan četvorosmerni 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 8: Četvorosmerni set asocijatvan keš

Na slici 9 je dat jednostavan primer bajt-adresibilne memorije sa 8-bitnom adresom. Ukupna veličina memorije je 256B. Blok je veličine 16 bajtova, te se najniža 4 bita ($\log_2 16$) adrese koriste za adresiranje bajta u bloku (eng. *byte in block*). Sa veličinom dvosmernog asocijativnog keša od 64B, mogu se čuvati četiri bloka iz operativne memorije, pri čemu svaki smer čuva dva. Budući da se unutar smera čuvaju dva bloka, širina index (idx) polja je jedan bit. Različitim bojama u operativnoj memoriji su označeni blokovi sa istim indeksom. Kod dvosmernog asocijativnog keša, dva bloka sa određenim indeksom mogu istovremeno biti u keš memoriji, jedan blok u svakom smeru. Na primer u kešu u smerovima sa indeksom “0b00” mogu biti dva bloka iz skupa: A, C, E, G, I, K, M, O. Na primeru sa slike se A blok nalazi u smeru 0, a E blok u smeru 1 keša. Kako bi se znalo koji od blokova se trenutno nalazi u kojem smeru, njihova “tag” polja se čuvaju u memorijiama za čuvanje tagova odgovarajućeg smera (eng. *Tag store*). Obratiti pažnju da za razliku od direktno preslikanog keša iste veličine sa slike 7, sada blokovi A i E mogu istovremeno biti u kešu.



Slika 9: Primer jednostavnog dvosmernog set asocijativnog keša

4.1. Polisa evikcije keša

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 (*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.

4.2. Polisa upisa keša

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.

4.3. Polisa razdeljenosti keša

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 dvopristupna 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 prihvat 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 prihvat 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 korištenje razdeljenog keša u prvom nivou proizlazi iz toga da dodatna kombinaciona logika za implementaciju dvopristupne 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 preopterete 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 ujedninjeni 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 usložnjava 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 prestavlja 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 [6].

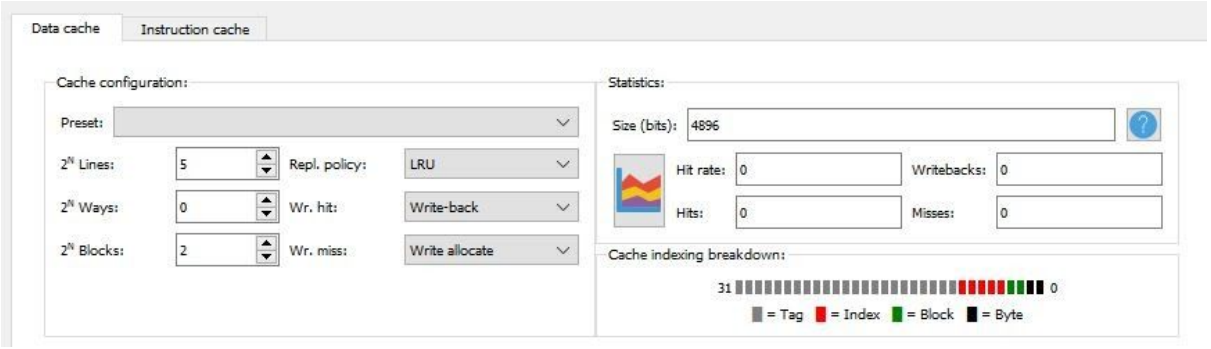
4.4. Polisa inkluzije keša

Kod inkluzivnog keša, prisusvo 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 umnognome pojednostavlje 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.

Zadatak 1:

U Ripes simulatoru je moguće podesiti konfiguraciju razdeljenog keš podsistema te simulirati njegov rad. Napisati jednostavan program za testioranje memorijskog podsitema. Ispitati ponašanje memorijskog podsistema nad različitim veličinama keša, bloka i broja smerova u asocijativnom kešu. Simulirati konfliktne promašaje u kešu za indeks = 0. Osmotriti ponašanje keša za različite kombinacije polise upisa: *write-back* ili *write-trough* kao i *write allocate* ili *non write allocate*. Ispitati *Least Recently Used* i *Random* metode evikcije blokova iz keša. Pri ispitivanju obratiti pažnju na količinu bitova koji se utroše na memorije za čuvanje tagova (eng. *Tag store*).



Zadatak 2:

Modelovati u VHDL-u operativnu memoriju koja se ponaša kao tipičan DRAM čip. Postoji kašnjenje odziva pri čitanju iz memorije od 20 taktova, nakon čega je moguće isčitati čitav blok podataka, reč po reč, na svaku rastuću ivicu takt signala. Zatim modelovati jedan nivo što jednostavnijeg keš podstistema, koji će napraviti spregu između *single-cycle* RISC-V procesora i operativne memorije.

Literatura:

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

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

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

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

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

Preuzeto sa:

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

[5] 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

[6] James Bell, David Casasent i C. Cordon Bell An Investigation of Alternative Cache Organizations, IEEE Trans. On Electronic Computers1974.