SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 535

FPGA IZVEDBA PROCESORA S MINIMALNIM SKUPOM INSTRUKCIJA

Luka Dejanović

Zagreb, lipanj 2012.

(Umjesto ove stranice ide izvorni tekst zadatka)

FPGA izvedba procesora s minimalnim skupom instrukcija

Izraditi model procesora sa stogovnom arhitekturom i minimalnim skupom instrukcija. Procesor je potrebno implementirati korištenjem jezika za opis sklopovlja VHDL u tehnologiji logičkih polja, FPGA, porodice Xilinx Spartan-3E. Koristiti instrukcijsku riječ širine 5 bitova, dok je širina podataka 16 bitova. Ostvariti protočnu strukturu instrukcija, i pri tome paziti na vremenske odnose pojedinih faza. Provjeriti mogućnosti međusobnog povezivanja više ovako ostvarenih procesora u svrhu kooperacije u rješavanju određenih zadaća.

SADRŽAJ

1. Uvod

FPGA sklopovi u današnjoj hijerarhiji elektroničkih sklopova predstavljaju vrhunac tehnologije, ukoliko se u vidu ima fleksibilnost sustava. Pružaju vrlo visok stupanj slobode pri izradi složenih sustava, te jednostavnu ponovljivost i mogućnost brze izmjene dizajna. Dodatna prednost pred fiksno ožičenim procesorima jest mogućnost paralelizacije dizajna, pa je upravo zbog tih karakteristika FPGA sklop iz porodice Spartan-3E odabran kao platforma za razvoj ovog procesora. Iako procesori s minimalnim instrukcijskim skupom nisu česti, ponajviše zbog široke primjene registarskih računala (najčešće RISC računala), imaju određene prednosti pred njima, primjerice jednostavna izvedba sklopa za dekodiranje instrukcija. Ove prednosti, uz neke mane, su predstavljene u drugom poglavlju.

U trećem poglavlju predstavljen je ciljni sustav za implementaciju procesora. Prilikom implementacije na FPGA sklopu poželjno je koristiti što više namjenskih resursa sklopa. Ovo se prvenstveno odnosi na memorijske elemente, odnosno stogove, koji će biti implementirani korištenjem distribuiranih RAM resursa. Uz to, programsku i podatkovnu memoriju je potrebno izvesti korištenjem blok RAM-a.

Nadalje, u četvrtom poglavlju će biti opisan razvoj sustava s naglaskom na povećanje brzine izvođenja, zbog čega će se implementirati protočna struktura. Broj faza ovisi o koncepciji sustava, a kod jednostavnih računala se najčešće radi o tri faze: dohvat instrukcije, te njeno dekodiranje i izvođenje. Nakon modeliranja sustava, ispravnost njegova rada će biti provjerena funkcijskom simulacijom, a u konačnici će se sustav implementirati na stvarnom sklopovlju, te će biti predstavljene glavne značajke sustava, iskorištene logičke komponente te maksimalna frekvencija signala takta.

2. Računala s minimalnim skupom instrukcija

Računala s minimalnim skupom instrukcija (*MISC – minimal instruction set computer*) su procesori sa vrlo malim brojem osnovnih instrukcija i pripadajućim operacijskim kodovima. Ova računala najčešće posjeduju stogovnu arhitekturu, te se time uklanja potreba za specifikacijom operanada u instrukciji, pa je potreban manji broj bitova po instrukciji. Uz to, stogovna arhitektura je jednostavnija zbog toga što se sve operacije obavljaju nad najvišim (ili nekoliko najviših) elemenata stoga. Posljedica ovoga je i manja i jednostavnija jedinica za dekodiranje instrukcija, te brže izvođenje zasebnih instrukcija. S druge strane, negativna posljedica ovakve arhitekture je velika zavisnost slijeda instrukcija [1].

2.1. Stogovna računala

Osnovica ovih računala jest stog, ili takozvana LIFO struktura podataka (*Last In, First Out*). Ovi su stogovi poznati i kao „push down“ stogovi, te su konceptualno najjednostavniji način spremanja informacija u privremenim pohranama za česte računalne operacija kao što su matematički izrazi i rekurzivni pozivi podrutina.

Hardverski realizirani stogovi se koriste u računalima od kasnih 50-ih godina prošlog stoljeća. Inicijalno su se koristili za povećanje efikasnosti izvođenja viših programskih jezika. No, kroz veliki period vremena, bili su više ili manje zastupljeni kod dizajnera hardvera, te su na kraju u većini računala postali sekundarna struktura za rukovanje podacima. Usprkos velikom broju zagovornika stoga, računala koja koriste stog kao primarni mehanizam strukture podataka zapravo nikada nisu naišli na tako veliko odobravanje kao što je to slučaj sa računalima koja koriste registre. S pojavljivanjem VLSI procesora (*Very Large Scale Integration*), ponovno su se razmatrale konvencionalne metode dizajniranja računala. CISC računala (*Complex Instruction Set Computer*) su evoluirala u komplicirane procesore sa opsežnim setom instrukcija, a RISC računala (*Reduced Instruction Set Computer*) su dovela u pitanje ovaj pristup koristeći jednostavnije procesorske jezgre kako bi postigli veće procesne brzine za neke primjene. Tako je ponovno došlo vrijeme za razmatranje stogovnih računala kao alternativnog

dizajna računala. Novi stogovni strojevi temeljeni na VLSI tehnologiji pružaju dodatne pogodnosti koje na prijašnjim stogovnim računalima nisu bile moguće, te koriste sinergiju svojih značajki kako bi postigli impresivnu kombinaciju brzine, fleksibilnosti i jednostavnosti.

Stogovna računala pružaju nešto manju složenost procesora nego CISC računala, a puno manju složenost cjelokupnog sustava od RISC ili CISC računala. Dodatna prednost je postizanje ovih značajki bez potrebe za kompliciranim prevodiocima (*compiler*) ili pričuvnim memorijama (*cache*). Isto tako, postižu konkurentne razine sirovih performansi, te superiorne performanse za određene cjenovne razrede u većini programskih okruženja. Prve uspješne primjene ovih računala su bile u ugradbenim kontrolnim okruženjima, gdje su pokazala bolje performanse od ostalih pristupa dizajniranju sustava, i to s povećim razlikama. Također, stogovna računala pokazuju obećavajuće značajke kod izvođenja logičkih te funkcionalnih programskih jezika, te jezika za istraživanje umjetne inteligencije.

Najveća prednost nove generacije stogovnih računala nad njihovim prethodnicima jest mogućnost jeftinog izvođenja velikih i brzih stogova. Dok su u prošlosti stogovi bili dio programske memorije, današnja stogovna računala koriste odvojene memorijske module, ili čak i vanjske memorije za implementaciju stoga. Tako ova računala pružaju iznimno brze pozive potprograma i superiorne performanse u obrađivanju prekida i promjeni konteksta i zadataka. Kada se sve ove značajke spoje, dobivamo računalne sustave koji su brzi, kompaktni i fleksibilni.

2.1.2. Primjer softverske implementacije

Postoji nekoliko načina programiranja LIFO stogova u konvencionalnim računalima. Najizravniji način je alociranje polja u memoriji, te korištenje varijable sa indeksom najvišeg aktivnog elementa u polju. Oni programeri koji drže do učinkovitosti izvođenja će poboljšati ovu metodu alocirajući blok memorijskih lokacija, a u varijablu pohranjivati stvarnu adresu najvišeg elementa stoga. U oba slučaja, stavljanje podatka na stog (*push*) odgovara alokaciji nove riječi na stogu i stavljanju podatka u nju. Uzimanje podataka sa stoga (*pop*) odgovara uklanjanju

najvišeg elementa sa stoga i vraćanju uklonjenog podatka rutini koja je zatražila akciju *pop*.

Stogovi se najčešće postavljaju u najviše adresne regije računala, te najčešće rastu od najviše memorijske lokacije prema nižim, te time omogućavaju maksimalnu fleksibilnost korištenja memorije između kraja programske memorije i „vrha“ stoga. Element vrha stoga je uvijek element koji je posljednji stavljen na stog, te je prvi koji će se sa stoga ukloniti, pa stoga smjer „rasta“ stoga nije bitan. Element dna stoga je posljednji element na stogu, čijim uklanjanjem stog ostaje prazan. Važno svojstvo stoga jest to da oni u svom osnovnom obliku omogućuju pristup samo najvišem elementu podatkovne strukture. Kasnije će se pokazati kako ovo svojstvo ima duboke implikacije na kompaktnost programa, jednostavnost hardvera i brzinu izvođenja.

Stogovi su odlični mehanizmi za privremenu pohranu informacija kod procedura. Glavni razlog za ovo je omogućavanje rekurzivnih poziva procedure bez rizika uništavanja podataka od prijašnjih poziva rutine. Također, podržavaju *reentrant* funkcije. Dodatna prednost jest ta da se stogovi mog koristiti za prosljeđivanje parametara između procedura. Zaključno, stogovi štede memorijski prostor time što omogućuju da razne procedure iznova koriste isti memorijski prostor, umjesto rezerviranja prostora unutar procedura za privremene varijable.

Postoje drugi načini ostvarivanja stogova osim pristupa preko polja. Liste povezanih elemenata se mogu koristiti za alokaciju riječi stoga, bez potrebe da elementi stoga budu u redoslijedu u stvarnoj memoriji. Također, softverska gomila (*heap*) se može koristiti za alokaciju memorije za stog, iako je ovo pitanje raspolaganja memorijom, pošto je gomila zapravo nadskup stoga.

2.1.3. Primjeri hardverske implementacije

Hardverske implementacije stoga imaju očitu prednost u vidu bržeg upravljanja softverom. U računalima koja velikim dijelom instrukcija koriste stog, povećana učinkovitost je bitna za održavanje visokih performansi sustava. Dok se bilo koja softverska metoda upravljanja stogom može implementirati u hardveru, općenito korištena hardverska implementacija je rezerviranje susjednih lokacija memorije pomoću pokazivača stoga u toj memoriji. Najčešće se pokazivač stoga

izvodi pomoću namjenskog registra čiji se sadržaj povećava ili smanjuje tijekom *push* i *pop* instrukcija. Ponekad se pruža mogućnost dodavanja pomaka pokazivaču stoga, kako bi se pristupilo nižim elementima u stogu bez destrukcije ostalih elemenata, te bez određenog broja *pop* instrukcija. Često se stog izvodi u istoj memoriji kao i program, a ponekada se, zbog povećanja učinkovitosti, postavlja u zasebnu memoriju. Još jedan pristup implementaciji stoga u hardveru može biti korištenjem velikih posmačnih registara. Svaki posmačni registar se izvodi kao dugi lanac registara, gdje se jedan kraj lanca vidi kao bit na vrhu stoga. Kako bi dobili 32-bitni stog od N elemenata, potrebna su nam 32 posmačna registra postavljena jedan kraj drugoga, pri čemu svaki ima N bitova. Iako se ovaj pristup nije koristio u prošlosti, kod VLSI stogovnih računala ovakav način realizacije stoga može biti održiva alternativa klasičnoj implementaciji pomoću registra koji pokazuje na memorijsku lokaciju.

2.1.4. Važnost stogovnih računala

S teorijskog gledišta, stogovi su sami po sebi važni, pošto su osnovni i prirodni alat za korištenje pri obrađivanju kvalitetno strukturiranog programskog koda. Uređaji sa LIFO stogovima su potrebni za sastavljanje programskih jezika, te mogu biti potrebni pri prevođenju prirodnih jezika. Bilo koje računalo sa podrškom za stogovne strukture će vrlo vjerojatno učinkovitije izvoditi aplikacije koje zahtijevaju stog.

Programiranje stogovnih računala je po mnogima jednostavnije od konvencionalnih računala, te su programi za stogovna računala pouzdaniji od ostalih programa. Jednostavnije je pisati prevodioce za stogovna računala, zbog toga što postoji manje iznimnih slučajeva koji kompliciraju prevodilac. Budući da izvođenje prevodioca može zauzeti velik dio resursa sustava, iznimno je važno imati učinkovit prevodilac. Stogovna računala su učinkovitija pri izvođenju određenih programa, pogotovo programa koji su dobro modularizirani. Također, ona su jednostavnija od ostalih računala, te pružaju dobru procesnu moć koristeći malo hardvera. Područje s najvećim potencijalom za korištenje stogovnih računala su kontrolne aplikacije u ugradbenim sustavima u stvarnom vremenu, jer

zahtijevaju kombinaciju visoke procesne brzine, malih dimenzija i odličnog upravljanja prekidima [2].

2.2. Instrukcijski set

Zbog navedenih činjenica vezanih uz stogovna računala, instrukciji set ovih računala se razlikuje od instrukcijskog seta popularnijih i raširenijih RISC računala. Ovo se najviše odnosi na instrukcije za operacije nad stogovima, budući da su osnovni element pohrane podataka stogovi a ne registri. Veliku ulogu u razvoju stogovnih računala ima Charles Moore, koji je 1970-ih godina osmislio programski jezik Forth, čije su instrukcije implementirane u mnoštvu stogovnih računala. Mooreova ideja je bila razviti programski jezik koji ima veliku mogućnost portabilnosti, ponajviše zbog osobnih razloga, odnosno čestih promjena poslova i truda da se njegov rad objedini pod jednim programskim jezikom. Moore je uspio u svom naumu, što dokazuje činjenica kako je Forth 1978. godine bio prvi softver za Intelov 8086 mikroprocesor, a 1984. godine je bio prvi razvojni sustav za prvi Apple Macintosh. Forth se čvrsto oslanja na stogove, pa je zato izuzetno koristan kod MISC procesora. Posebice zbog toga što koristi tzv. *postfix* notaciju za matematičke operacije, koja odgovara stogovnim računalima, pošto se oba operanda prvo stavljaju na stog, te se zatim nad njima obavlja operacija[10]. Nakon toga se rezultat stavi na vrh stoga, te zapravo zamijeni operande.

Osnovni stog u ovakvim računalima je podatkovni stog, nad čijim se elementima obavljaju gotovo sve instrukcije. Ovisno o primjenama i složenosti sustava, procesor može imati dodatne stogove, a najčešći dodatni stog je tzv. povratni stog (*eng. return stack*), koji se koristi pri pozivu potprograma, te vraćanju iz istih. Pri pozivu potprograma, adresa sljedeće instrukcije se postavi na vrh povratnog stoga, te se pri izvođenju instrukcije za povratak iz potprograma, programsko brojilo napuni elementom sa vrha stoga. Ovakva funkcionalnost se mogla dobiti korištenjem povratnog registra umjesto stoga, čime štedimo na hardverskim resursima. No, prednost stoga u ovom slučaju je mogućnost ugniježđenih poziva instrukcija, kod kojih se pri svakom novom pozivu potprograma, njegova povratna adresa stavlja „na“ prošlu povratnu adresu. Tako se pri povratku iz potprograma uvijek vraćamo na mjesto posljednjeg poziva. Kod

ovakve primjene stoga, broj mogućih ugniježđenih poziva potprograma je ograničen dubinom stoga.

Još jedan mogući dodatni stog koji se koristi kod ovakvih procesora je tzv. adresni stog. Najviši element ovog stoga služi kao pokazivač za memoriju, najčešće podatkovnu memoriju. Uz ovaj stog vezane su instrukcije za dohvat i pohranu u memoriju, uz opciju inkrementiranja pokazivača nakon dohvata/pohrane. Kao i kod povratnog stoga, istu smo funkcionalnost mogli dobiti korištenjem registra umjesto stoga, no i ovdje nam je u interesu sačuvati prethodnu adresu pokazivača. Ovo je, primjerice, korisno ako u memoriji imamo pohranjene koeficijente filtra, pa na adresni stog postavimo adresu za dohvat tih koeficijenata, i nakon proračuna novog uzorka, jednostavno uzmemo adresu koeficijenata sa stoga, i ostaje nam početna adresa, na kojoj smo obavljali neke druge operacije. Bitno je reći da se sve ove adrese na adresni stog postavljaju preko podatkovnog stoga. Time smo omogućili izračune adresa, ali i dodali jedan ciklus pri uzimanju adrese iz podatkovne memorije (jer njen put ide preko podatkovnog stoga, pa trebamo dvije instrukcije). No, kako smo ograničeni relativno malim brojem instrukcija, ovaj način nam daje najbolji omjer performansi i složenosti sustava. Postoje i drugi dodatni stogovi, od kojih je vjerojatno najčešći stog za varijable.

Uz instrukcije za pozive i povratke iz potprograma, ALU instrukcije i instrukcije za prebacivanje između stogova postoje i instrukcije za bezuvjetne i uvjetne skokove, te neke specifične instrukcije, kao što je primjerice instrukcija za množenje i akumulaciju. Detaljniji opis i pregled instrukcija predstavljen je u poglavlju *4. Izvedba procesora MISC16*.

3. FPGA sklopovi porodice Spartan-3

Procesor je potrebno dizajnirati pomoću jezika za opis sklopovlja, te nakon toga implementirati u tehnologiji logičkih polja, te se za to koristi FPGA (*Field Programmable Gate Array*) sklop porodice Xilinx Spartan-3E. Ova porodica FPGA sklopova je dizajnirana kako bi ispunila visoke zahtjeve primjena u potrošačkoj elektronici, s naglaskom na minimizaciji troškova. Porodica broji 5 članova te nudi gustoće između 100 000 i 1.6 milijuna logičkih vrata po sklopu.

Spartan-3E porodica polazi od uspjeha ranije porodice Spartan-3, te povećava količinu logike po ulazu odnosno izlazu, smanjujući time cijenu logičke ćelije. Nove značajke poboljšavaju performanse sustava i smanjuju troškove konfiguracije, a u kombinaciji s naprednom 90-nanometarskom tehnologijom omogućava veću funkcionalnost i postavlja nove standarde u industriji programabilne logike. Zbog svoje niske cijene ovi su sklopovi idealni za širok raspon primjena u potrošačkoj elektronici, uključujući širokopojasni pristup, kućne mreže, projekcije i prikaze te opremu za digitalne TV prijemnike. Ovakvi sklopovi imaju dodatne prednosti u odnosu na ASIC (Application Specific Integrated Circuits) sklopove, konkretno manje početne troškove, dugotrajan razvoj, te mogućnost jednostavnog nadograđivanja sustava, bez potrebe za dodatnim sklopovljem.

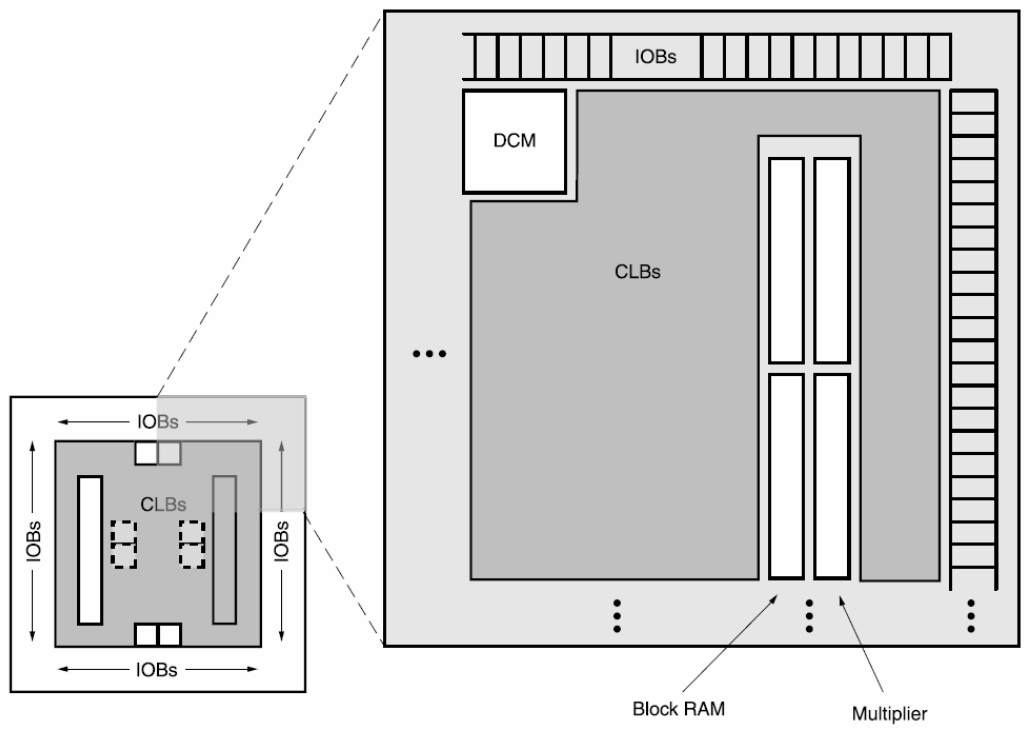
Glavne značajke FPGA sklopova ove porodice su:

- kombinacija niskih troškova i visokih performansi za zahtjevne primjene
- dokazana 90-nanometarska tehnologija
- sučelja s više naponskih razina i standarda
- fleksibilni logički resursi (posmačni registri, široki multipleksori, množila, podrška za distribuirani i blok RAM, itd.)
- do 8 sklopova za upravljanje taktom
- jeftina kućišta, izbor između QFP i BGA

Konkretno, koristi se inačica sklopa XC3S500E koja posjeduje 500 000 logičkih vrata (10 476 ekvivalentnih logičkih ćelija), te ukupno 1164 konfigurabilna logička bloka. Slijedi detaljniji pregled arhitekture sklopa.

3.1. Arhitektura Spartan-3E porodice FPGA sklopova

Sklopovi Spartan-3E porodice se sastoje od pet temeljnih programabilnih funkcijskih elemenata: konfigurabilnih logičkih blokova (CLB), ulazno/izlaznih blokova (IOB), blok RAM-a, blokova množila i blokova za digitalno upravljanje signalom takta (DCM). Ovi elementi su organizirani kao na slici 1. Prsten IOB-a okružuje pravilnu mrežu CLB-a, unutar koje su dva stupca sa blok RAM-om i množilima. Svaki stupac RAM-a se sastoji od nekoliko blokova RAM-a od 18Kbitova. DCM sklopovi se nalaze na dnu i na vrhu uređaja. Svi ovi elementi su povezani s bogatom mrežom linija koje služe za prijenos signala. Uz to, svaki od ovih elemenata ima dodatnu matricu sa sklopkama koje omogućavaju višestruko povezivanje na vezne elemente.

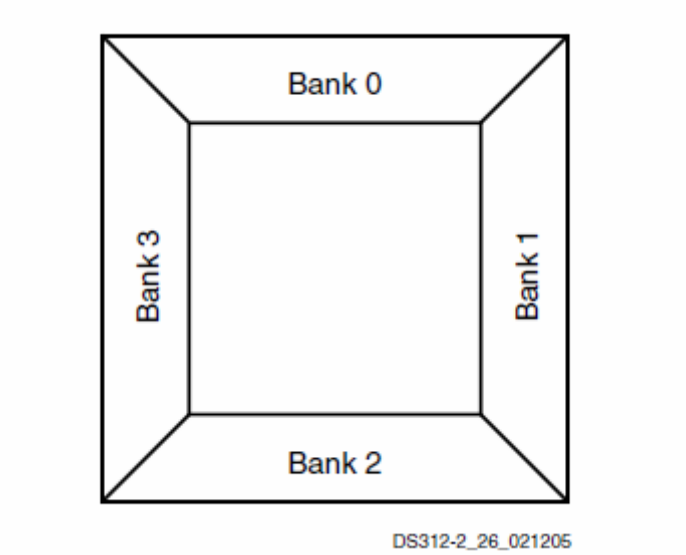


Slika 1. Arhitektura Spartan-3E porodice FPGA sklopova

3.1.1. Ulazno-izlazni blokovi

Ulazno-izlazni blokovi (*IOB – input/output blocks*) nude programabilno, jednosmjerno ili dvosmjerno sučelje između priključka kućišta i unutarnje logike FPGA sklopa. Svakom bloku se dodaje programabilno ulazno kašnjenje, te postoje tri glavna puta signala unutar IOB-a: izlazni put, ulazni put i upravljanje

visokom impedancijom izlaza. Svaki od ovih putova sadrži sklopovlje koje podržava prijenos dvostrukom brzinom (*double data rate – DDR*), a bistabili koji su zaslužni za DDR mogu se dijeliti između susjednih IOB-a. Isto tako, svi putovi signala koji ulaze u IOB imaju opciju invertiranja, te ukoliko u dizajnu stavimo invertor na neki od ovih putova, on je automatski apsorbiran u IOB. Ukoliko neki IOB ne koristimo, njegovo sklopovlje možemo koristiti u dizajnu. Dodatno, priključci ovih sklopova podržavaju širok raspon standarda za ulazno-izlazne signale, te se većina ulaza odnosno izlaza može koristiti za formiranje diferencijalnih parova, te se time podrška širi i na diferencijalne standarde. Definicija standarda u dizajnu se izvodi preko atributa IOSTANDARD koji se postavlja na određenu vrijednost. IO blokovi su grupirani u banke (4 banke za Spartan-3E porodicu), te se svi priključci neke banke konfiguriraju na isti način (slika 2.).

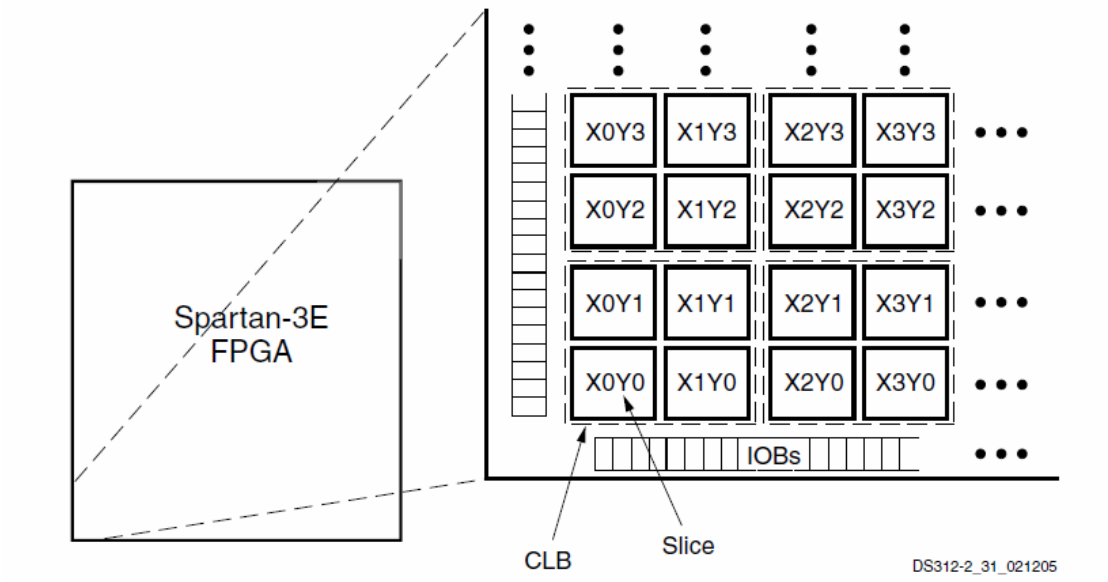


Slika 2. Banke ulazno-izlaznih blokova na Spartan-3E porodici

3.1.2. Konfigurabilni logički blokovi

Konfigurabilni logički blokovi (*CLB – configurable logic blocks*) sačinjavaju glavne logičke resurse za implementaciju kako sinkronih tako i kombinacijskih funkcija. Svaki CLB sadrži 4 odsječka (*eng. slices*), a svaki odsječak sadrži dvije prozivne tablice (*LUT – look up table*) koje ostvaruju logičke funkcije te dva namjenska elementa za pohranu koji se mogu koristiti kao bistabili (bilo SR

bistabili, bilo *latchevi*). Prozivne tablice se još mogu koristiti i kao distribuirani RAM kapaciteta 16 x 1 bit (RAM16) ili kao 16-bitni posmačni registri (SRL16), a dodatni multipleksori i linije za signal prijenosa (*carry bit*) pojednostavljuju široke logičke i aritmetičke funkcije. Najveći dio logike opće namjene u dizajnu se automatski mapira u resurse pojedinih odsječaka u CLB-ima. Svi CLB-i su identični, te su posloženi u pravilnu matricu, kao što prikazuje slika 3. Gustoća ovisi o inačici sklopa, a u korištenoj inačici su CLB-i posloženi u matricu sa 46 redaka i 34 stupca (ukupno 1164 konfigurabilna logička bloka).



Slika 3. Razmještaj CLB-a na Spartan-3E sklopu

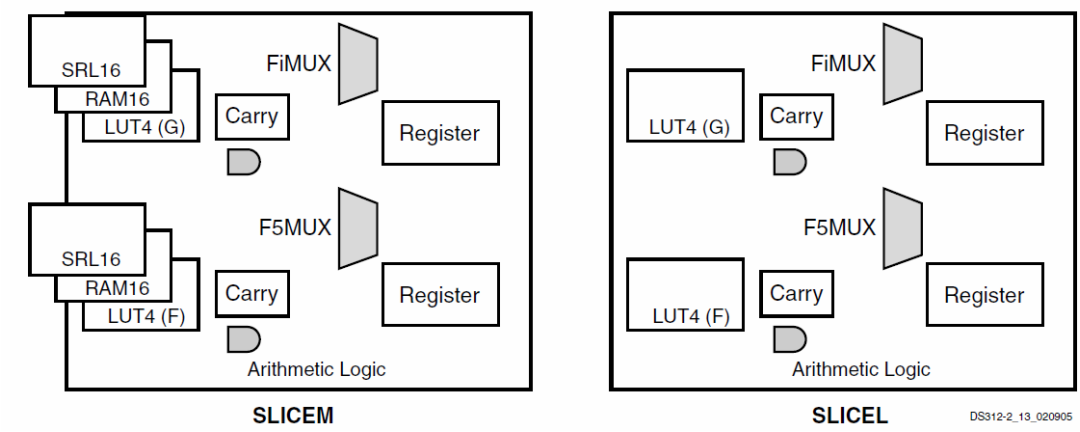
Svaki CLB se sastoji od 4 međusobno povezana odsječka, koji su grupirani u parove, te je svaki par organiziran kao stupac i ima zasebni lanac prijenosa. Lijevi par podržava logičke i memorijske funkcije, te se njegovi odsječci nazivaju SLICEM, dok desni par podržava samo logičke funkcije, pa se njegovi odsječci nazivaju SLICEL (slika 4.). Slijedi da polovica implementiranih prozivnih tablica podržava samo logičke funkcije, dok druga polovica podržava i memorijske funkcije (distribuirani RAM i posmačne registre), te su ta dva tipa naizmjenično postavljeni u stupce diljem matrice sa CLB-ima. SLICEL odsječci smanjuju površinu i cijenu CLB-a i cijelog uređaja, a pružaju i bolje performanse od SLICEM odsječaka.

Svaki odsječak sadrži dva LUT-a, koji služe kao funkcijski generatori, te elemente za pohranu i dodatnu logiku. Obje verzije odsječaka sadrže sljedeće elemente kako bi pružili logičke, aritmetičke i ROM funkcionalnosti:

- dva LUT-a sa četiri ulaza, F i G
- dva elementa za pohranu – bistabila
- dva multipleksora za široke funkcije, F5MUX i FiMUX
- logiku za signal prijenosa i aritmetiku

Dodatno, SLICEM odsječci podržavaju dvije dodatne funkcije:

- dva bloka distribuiranog RAM-a, 16 x 1 bit (RAM16)
- dva 16-bitna posmačna registra (SRL16)

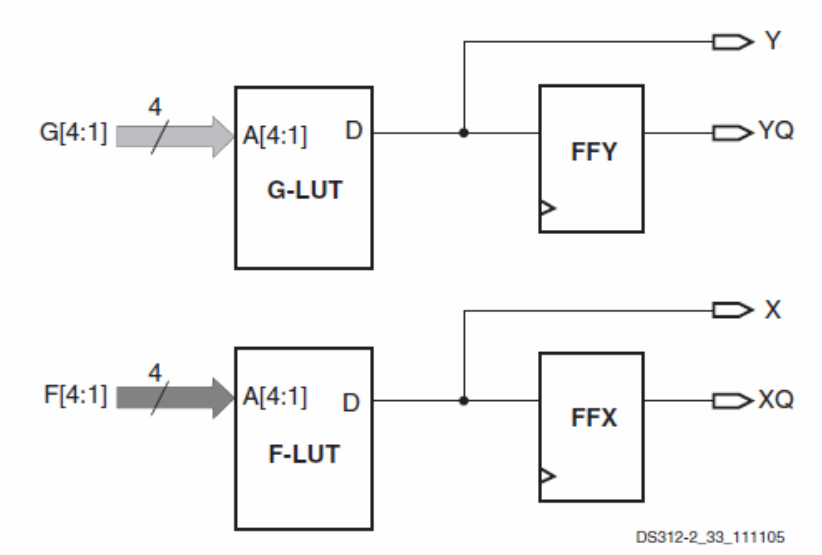


Slika 4. Izgled odsječaka CLB-a

Kombinacija LUT-a i bistabila formira tzv. logičku ćeliju. Dodatne značajke odsječka, kao što su primjerice široki multipleksori, logika za carry bit ili aritmetička vrata, pridonose kapacitetu odsječka, jer omogućavaju implementaciju logike koja bi inače zahtijevala dodatne LUT-ove.

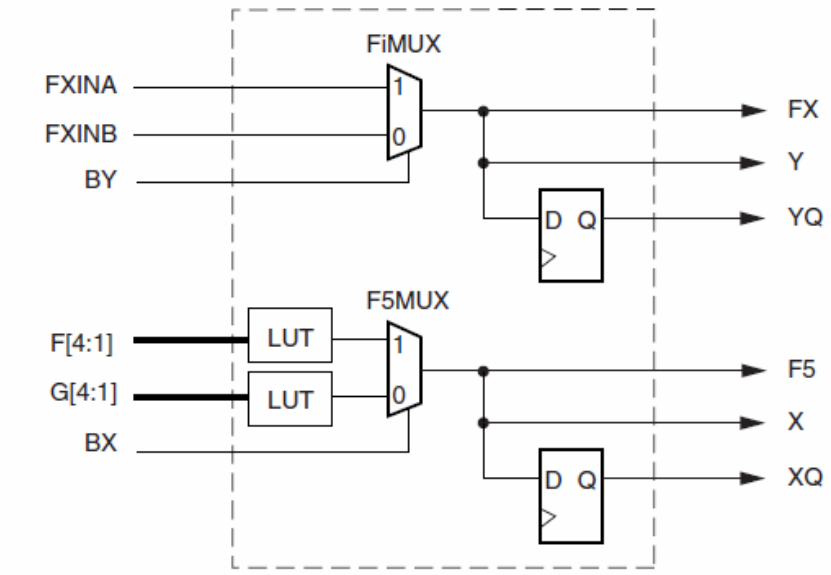
Prozivne tablice (LUT) su funkcijski generatori koji se temelje na RAM memoriji, i one su glavni resurs za implementaciju logičkih funkcija. Nadalje, LUT-ovi u SLICEM odsječcima se mogu konfigurirati kao 16-bitni distribuirani RAM ili kako posmačni registar. Svaki od dva LUT-a u odsječku (F i G) ima 4 logička ulaza (A1 do A4) i jedan izlazni signal (D). Tako možemo implementirati bilo koju Booleovu logičku funkciju koristeći samo jedan LUT. Funkcije s više ulaza se mogu implementirati kaskadiranjem LUT-ova ili korištenjem širokih multipleksora. Izlazni

signal LUT-a se može spojiti na multipleksor, na logiku za aritmetiku ili signal prijenosa, te izravno ili posredno (preko bistabila) na izlaz CLB-a (slika 5.).



Slika 5. Prozivne tablice u odsječku CLB-a

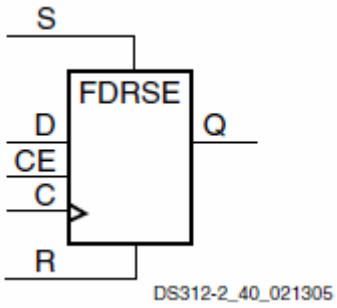
Namjenski multipleksori efikasno kombiniraju LUT-ove kako bi omogućili složenije logičke operacije. Svaki odsječak ima dva ovakva multipleksora, F5MUX u donjem dijelu odsječka, i FiMUX u gornjem, kao što je prikazano na slici 6. F5MUX kombinira dva LUT-a u odsječku, dok FiMUX kombinira dva ulaza u CLB koji su spojeni direktno na izlaze multipleksora istog odsječka ili drugih odsječaka. Ovisno načinu spajanja, FiMUX-om možemo ostvariti bilo koju funkciju od 6, 7 ili 8 ulaza, pa se tada FiMUX naziva F6MUX, F7MUX ili F8MUX.



Slika 6. Multipleksori u odsječcima CLB-a

Lanac za signal prijenosa, zajedno s logičkim sklopovima za aritmetiku, omogućava brzu i efikasnu implementaciju matematičkih funkcija. Carry signal se automatski koristi za većinu aritmetičkih funkcija u dizajnu. Ovi resursi se mogu koristiti i za stvaranje logičkih funkcija opće namjene, uključujući i jednostavne široke Booleove funkcije. Lanac prijenosa ulazi u odsječak kao CIN, te izlazi kao COUT, kontrolira ga nekoliko multipleksora, a spaja se izravno s jednog CLB-a na CLB iznad njega. Aritmetička logika uključuje logička vrata za operacije logičko I i isključivo ILI (postoje dva sklopa za svaku funkciju, XORF, XORG, te ANDF, ANDG), i ti sklopovi potpomažu LUT u implementaciji efikasnih aritmetičkih funkcija, primjerice brojila i množila (2 bita po odsječku).

Element za pohranu, tj. bistabil, može se konfigurirati kao SR-bistabil ili kao *latch*, a između ostaloga može služiti za sinkronizaciju podataka na signal takta. Postoje dva bistabila u odsječku, FFY i FFX. FFX može birati između kombinacijskog izlaza X ili premosnog signala BX, dok FFY dodatno ima fiksno množilo na izlazu D. Ovi elementi se nazivaju FDRSE (Flip-Flop, sinkroni reset, set, i signal za omogućavanje takta).

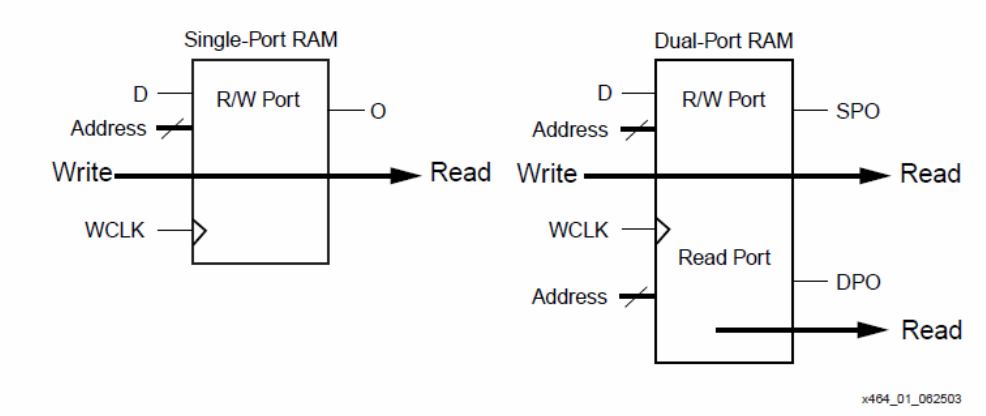


Slika 7. Bistabili u odsječcima CLB-a

3.1.3. Distribuirani RAM

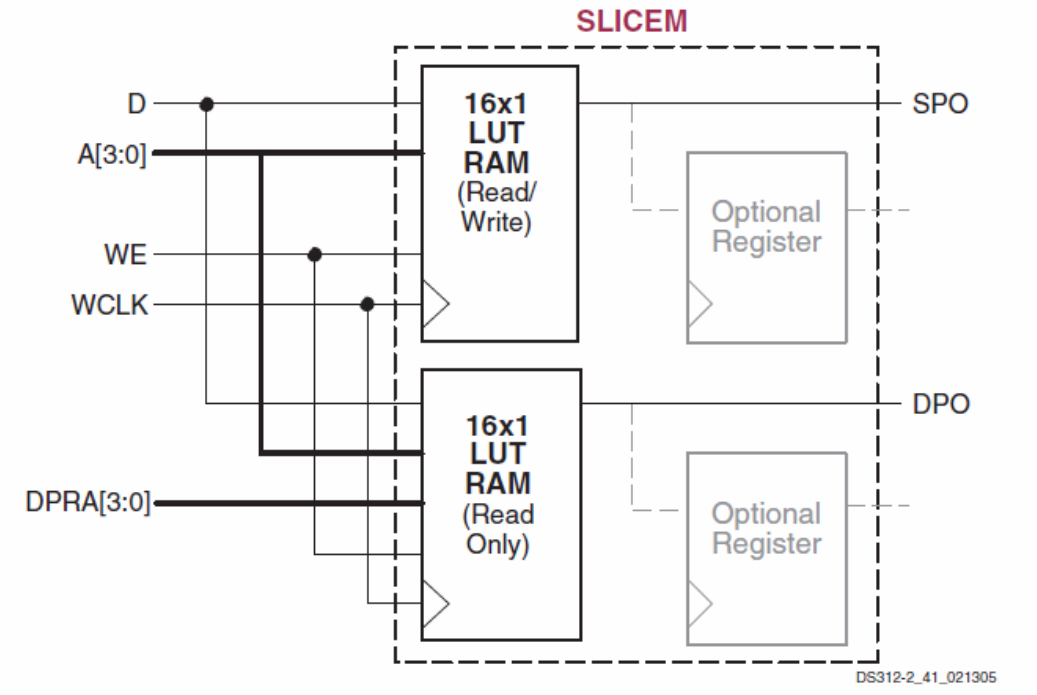
Dodatna opcija LUT-ova u SLICEM odsječcima je mogućnost konfiguracije kao distribuirani RAM. Ovakva memorija omogućava stvaranje spremnika podataka srednje veličine, bilo gdje na putu signala. Četiri ulaza u LUT postaju adresni signali, i time omogućavaju konfiguraciju 16 x 1 bit. Više LUT-ova iz

SLICEM odsječaka se može kombinirati na razne načine i time formirati memorije za pohranu više podataka, primjerice 16x4, 32x2 ili 64x1, unutar jednog CLB-a.



Slika 8. Jednopristupni i dvopristupni distribuirani RAM u CLB-u

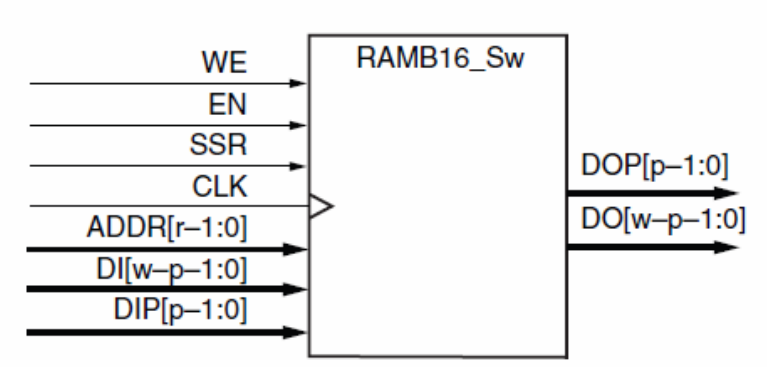
Distribuirani RAM se može konfigurirati i kao dvopristupni, te tada koristi dva LUT-a kako bi se omogućio pristup s dvije zasebne adresne linije. Kod ovakve konfiguracije, isti se podatak zapisuje u obje memorije, ali se razlikuju izlazni signali i adresne linije.



Slika 9. Dvopristupni distribuirani tam unutar jednog odsječka

3.1.4. Blok RAM

Spartan-3E sklopovi sadrže između 4 i 36 namjenskih blok RAM-ova, koji su organizirani kao dvopristupni konfigurabilni blokovi od 18 Kbit-a. Blok RAM sinkrono pohranjuje velike količine podataka, dok je prethodno spomenuti distribuirani RAM prvenstveno namijenjen kao privremeni spremnik malih količina podataka.



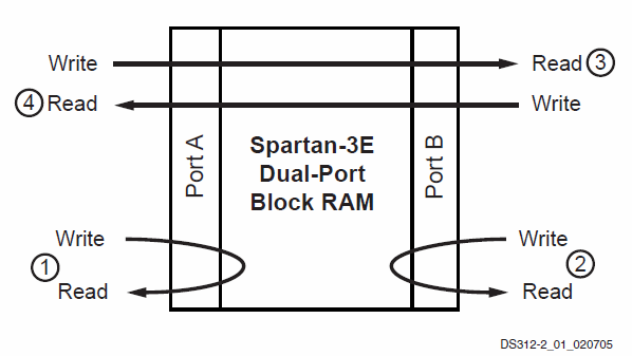
Slika 10. Jednopristupni blok RAM

Svaki blok RAM može se konfigurirati tako da se postavi inicijalna vrijednost sadržaja, zadane vrijednosti izlaznih signala, omjeri portova i načini upisa. Blok RAM se može koristiti kao jednopristupni i kao dvopristupni. Na sklopu su blok RAM-ovi formirani u jedan ili dva stupca, ovisno o veličini uređaja, te su okruženi konfigurabilnim logičkim blokovima. Pokraj blok RAM-a nalaze se hardverski izvedena množila širine 18 x 18 bitova, te dijele sabirnice za port A i port B sa blok RAM-om. Blok RAM ima dvopristupnu strukturu, dva porta A i B omogućavaju nezavisan pristup zajedničkom blok RAM-u, koji ima maksimalni kapacitet od 18432 bitova. Svaki od ovih portova ima zasebni skup podatkovnih, kontrolnih i signala takta za sinkrone operacije čitanja i pisanja. Postoje 4 osnovna putova podataka (slika 11.):

1. pisanje i čitanje s porta A
2. pisanje i čitanje s porta B
3. prijenos podataka s porta A na port B
4. prijenos podataka s porta B na port A

Svaki port se može zasebno konfigurirati za odabir različitih širina za ulaz podataka (*DI – data input*) ili izlaz podataka (*DO – data output*). Ukoliko se širine portova razlikuju, blok RAM automatski vrši funkciju poravnanja portova. Primjer

korištenja ovoga je upis 32-bitnih podataka na port A, te čitanje 16-bitnih podataka s porta B.



Slika 11. Osnovni putovi podataka za blok RAM

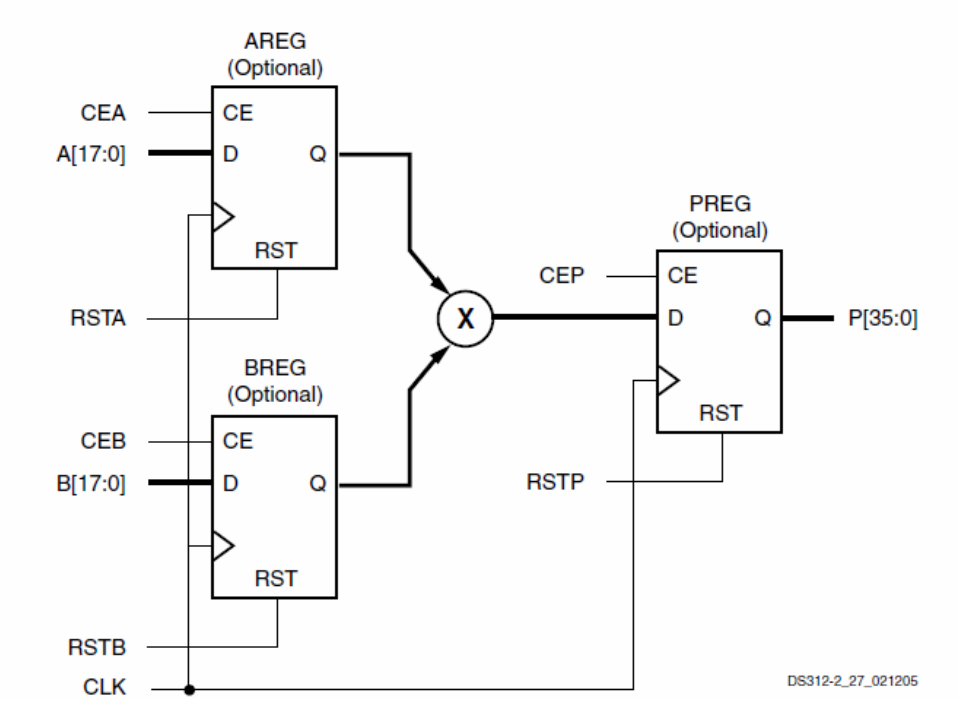
Blok RAM se može konfigurirati na razne načine s aspekta kapaciteta:

- 16K x 1 bit, 8K x 2 bita, 4K x 4 bita
- 2K x 9 bitova, 1K x 18 bitova, 512 x 36 bitova

Prve tri opcije ne podržavaju bitove pariteta, dok preostale tri sadrže opciju bitova pariteta (16Kbitova podataka, 2Kbit pariteta).

3.1.5. Namjenska množila

Porodica Spartan-3E nudi između 4 i 36 blokova namjenskih množila po uređaju. Ovi blokovi se nalaze pokraj blok RAM-a, te su također organizirani u jedan ili dva stupca. Osnovna namjena ovih množila je numeričko množenje brojeva u formatu dvojnog komplementa, ali mogu izvoditi i neke druge operacije, primjerice jednostavnu pohranu ili posmak podataka. Dodane opcije množenja pružaju LUT-ovi u odsječcima, kao što je već prije spomenuto. Svako množilo radi opciju množenja dva 18-bitna broja, s rezultatom u punoj preciznosti od 36 bitova, pri čemu su svi brojevi u formatu dvojnog komplementa.



Slika 12. Množilo s pripadajućim registrima i kontrolnim signalima

Dodatnu mogućnost predstavlja množenje bez predznaka, za što je potrebno postaviti najviši bit obje riječi u '0'. Svako množilo ima registre na ulazu i izlazu iz bloka (ulazi AREG i BREG, izlaz PREG), te time omogućava bolje performanse množila jer dopušta implementaciju protočne strukture (slika 12.). Također, svaki od ovih registara ima zasebni signal za omogućavanje takta i sinkroni reset, čime je olakšano spremanje nekih bitnih podataka, primjerice koeficijenata za filtre. Moguća je i implementacija množila manje širine, te je za to potrebna operacija proširivanja predznaka (*sign extension*) do pune širine od 18 bitova. Također je moguća i implementacija množila veće širine od prirodnih 18 bitova, te se to obavlja kombinacijom namjenskih množila i LUT-ova iz odsječaka CLB-a. Kako je već spomenuto, blok RAM i množila dijele neke resurse, što je moguće onemogućiti tako da blok RAM formiramo za kapacitet 512 x 36 bitova.

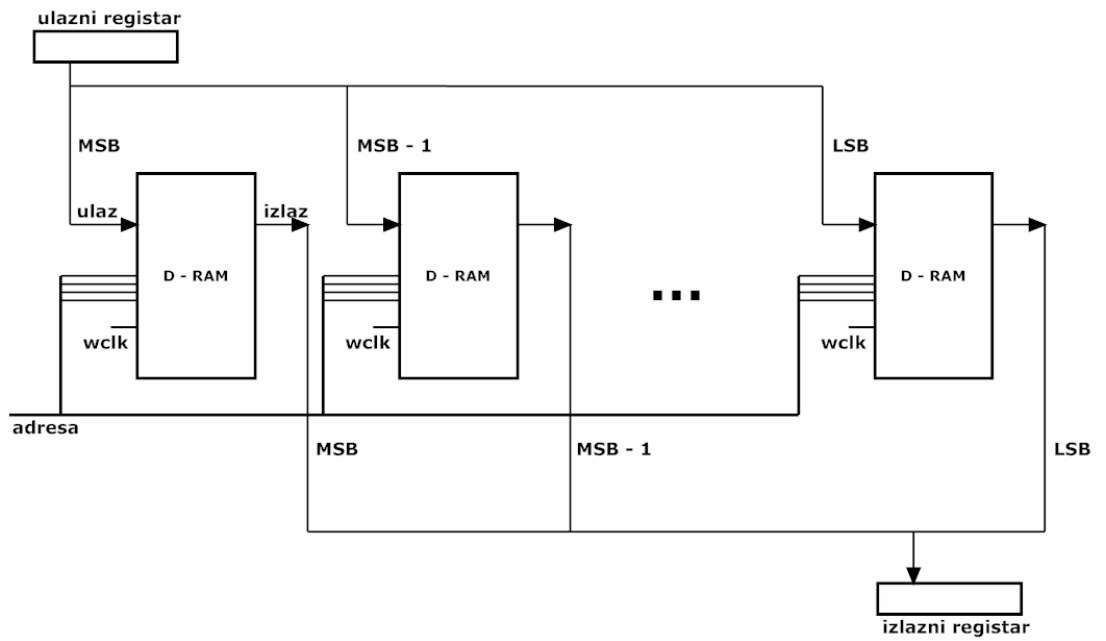
4. Izvedba procesora MISC16

Motivaciju i podlogu za razvoj procesora u VHDL-u i izvedbu na FPGA sklopu pružaju internetske stranice dizajnera stogovnih računala, najviše procesor P16 [8] te MSL16 [9]. Procesor P16 profesora Tinga daje uvid u ponašanje procesora, način izvedbe pojedinih jedinica i pregled osnovnih instrukcija, dok se MSL16 pokazuje korisnim u shvaćanju implementacijskog dijela, pošto je realiziran na Xilinxovom FPGA sklopu. U sljedećim potpoglavljima dan je razvoj i implementacija procesora MISC16.

4.1. Procesor MISC16

Za dizajn i izvedbu procesora u VHDL-u koristi se Xilinx-ovo razvojno okruženje, Xilinx ISE^{TM} 9.2i. Nakon pokretanja programa potrebno je stvoriti novi projekt (*File* => *New Project*) te je potrebno odabrati implementacijsku platformu (All, Spartan-3E, XC3S500E, FG320, -5). Za sintezu će se koristiti Xilinxov XST, a za simulacije ModelSim PE firme MentorGraphics. Nakon što je projekt kreiran, potrebno je dodati *source* datoteku u kojoj će se pisati VHDL kod (*Project* => *New Source* => *VHDL module*). Prije početka pisanja koda, još je potrebno kompajlirati biblioteke za simulaciju jezika za opis sklopovlja (u *Sources* prozoru odabrati platformu, te zatim u *Processes* prozoru proširiti opciju *Design Utilities* i pokrenuti proces *Compile HDL Simulation Libraries*).

Implementacija procesora MISC16 u VHDL-u započinje formiranjem najbitnijih stvari u MISC procesoru – stogova. O broju stogova je već bilo riječi, stoga je samo bitno reći da MISC16 ima 3 stoga: podatkovni, adresni i povratni. Svi stogovi su dubine 16 riječi, dok se njihove širine razlikuju ovisno o primjeni. Sva ova tri stoga su na FPGA sklopu implementirana korištenjem distribuiranog RAM-a, na način da jedan distribuirani RAM, odnosno jedan LUT odgovara jednom bitu u svim razinama stoga. Tako se stogovi formiraju na način da se „paralelno“ postavi određeni broj takvih istih distribuiranih RAM-ova, pri čemu se isti signal spaja na adresne ulaze, kako bismo istovremeno mogli pristupiti cijeloj riječi na stogu (slika 13). Na ovaj način nam je omogućen odabir širina pojedinog stoga, čime, pokazat će se, radimo uštedu resursa FPGA sklopa.



Slika 13. Formiranje stogova pomoću distribuiranih RAM-ova

4.1.1. Podatkovni stog

Podatkovni stog je najvažniji stog kod ovog procesora (i njemu sličnih) zbog toga što svi podaci prelaze preko njega, pa tako čak i adrese za memorijske dohvate prvo moraju biti učitane na podatkovni stog. Pošto je širina podataka kod MISC16 16 bitova, ovaj stog će biti širok 16 bitova, što znači da trebamo 16 distribuiranih RAM-ova u paraleli. Za formiranje stogova ćemo koristiti makro distribuiranog RAM-a iz biblioteke *Unisim*, te je stoga potrebno tu biblioteku uključiti na početku naše izvorne datoteke (*source file*).

```
library UNISIM;
use UNISIM.VComponents.all;
```

Željeni resurs se u dizajn može uključiti na dva načina: slijednim opisom i ravnopravnim jednadžbama ili strukturnim opisom. Kod slijednog opisa (procesa) i ravnopravnih jednadžbi VHDL kod je neovisan o implementacijskoj tehnologiji, te sintetizator na osnovi slijednog opisa prepoznaje makroe. Za razliku kod toga, kod strukturnog opisa, tj. instanci komponenata, VHDL kod je usko vezan uz korištenu platformu, te je potrebno poznavanje komponenata (*primitive*) koje podržava odabrana tehnologija i njeni alati. Ovaj pristup se koristi kada točno znamo koji resurs ili komponentu želimo iskoristiti jer instancirane komponente nisu podložne

optimizaciji. Za stogove znamo točno što želimo, pa stoga koristimo strukturni opis:

```
------------------  PODATKOVNI STOG  ---------------------
       DATA_STACK : for I in 15 downto 0 generate
       DSTACK : RAM16X1S
              port map (
                     O => d_out(I),     -- RAM output
                     A0 => ds_addr(0), -- RAM address[0] input
                     A1 => ds_addr(1), -- RAM address[1] input
                     A2 => ds_addr(2), -- RAM address[2] input
                     A3 => ds_addr(3), -- RAM address[3] input
                     D => d_in(I),          -- RAM data input
                     WCLK => clk,           -- Write clock input
                     WE => dw_en     -- Write enable input
                     );
       end generate DATA_STACK;
```

Vidimo kako je ulaz podatkovnog stoga predstavljen signalom `d_in`, a izlaz `d_out`. Adresa riječi u stogu je predstavljena signalom `ds_addr`. Način rada stoga bit će opisan u kasnijim poglavljima, uz instrukcije koje ga koriste.

4.1.2. Povratni stog

Povratni stog se koristi kod poziva i povratka iz potprograma, gdje se pri pozivu potprograma adresa sljedeće instrukcije stavlja na povratni stog (*return stack*), te se pri povratku taj podatak vraća u programsko brojilo, kako bi program nastavio gdje je stao u trenutku poziva. Programska memorija će na sklopu biti izvedena korištenjem blok RAM-a, te će biti formirana kao 1K x 16 bita, što znači da će programska memorija imati ukupno 1024 lokacije, te nam je za pristup svim lokacijama dovoljan signal od 10 bitova. Stoga možemo zaključiti kako nam povratni stog mora biti širok samo 10 bitova, te ga nema potrebe izvoditi identično kao i podatkovni (što bi značilo sa 16 bitova). Slijedi da nam za povratni stog treba 10 distribuiranih RAM-ova u paraleli:

```
------------------  POVRATNI STOG  ------------------------
       RETURN_STACK : for I in 11 downto 0 generate
       RSTACK : RAM16X1S
              port map (
                     O => r_out(I),     -- RAM output
                     A0 => rs_addr(0), -- RAM address[0] input
                     A1 => rs_addr(1), -- RAM address[1] input
                     A2 => rs_addr(2), -- RAM address[2] input
                     A3 => rs_addr(3), -- RAM address[3] input
                     D => r_in(I),          -- RAM data input
                     WCLK => clk,    -- Write clock input
                     WE => rw_en     -- Write enable input
                     );
       end generate RETURN_STACK;
```

Slično kao i kod podatkovnog stoga, podatkovni i kontrolni signali su: `r_in`, `r_out` i `rs_addr`. Način rada i formiranje podatka za pohranu na povratni stog bit će detaljnije objašnjeni u sljedećim poglavljima.

4.1.3. Adresni stog

Adresni stog se koristi kod pohranjivanja i dohvata podataka iz podatkovne memorije. Adresni stog bi se mogao izvesti kao adresni registar, no na ovaj način imamo mogućnost povremenog skoka na tablicu u podatkovnoj memoriji, bez gubitka informacije o prethodnoj adresi na koju smo primjerice upisivali dolazeće uzorke nekog signala. Podatkovna memorija će također biti izvedena kao blok RAM, konfiguriran kao 1K x 18 bitova, pa nam je i kod podatkovne memorije za pristup svim podacima dovoljna 10-bitna adresa. Tako će i adresni stog biti širine 10 bitova, te nam je za implementaciju ovog stoga dovoljno 10 paralelnih distribuiranih RAM-ova:

```
------------------  ADRESNI STOG  -------------------------
       ADDRESS_STACK : for I in 9 downto 0 generate
       ASTACK : RAM16X1S
              port map (
                     O => a_out(I),         -- RAM output
                     A0 => as_addr(0), -- RAM address[0] input
                     A1 => as_addr(1), -- RAM address[1] input
                     A2 => as_addr(2), -- RAM address[2] input
                     A3 => as_addr(3), -- RAM address[3] input
                     D => a_in(I),          -- RAM data input
                     WCLK => clk,    -- Write clock input
                     WE => aw_en     -- Write enable input
                     );
       end generate ADDRESS_STACK;
```

Kao i u prethodna dva slučaja, podatkovni i adresni signali su: `a_in`, `a_out`, `as_addr`.

4.1.4. Memorije

Programska memorija procesora MISC16 će, kako je već spomenuto, biti na sklopu realizirana kao blok RAM. No, kako od programske memorije očekujemo samo čitanje njenog sadržaja, taj blok RAM je potrebno konfigurirati kao ROM. Ovo se može učiniti na dva načina. Prvi način je instanciranjem potrebnog resursa, te postavljanjem inicijalnih vrijednosti svih bitova memorije, što bi značilo upisivanje 18 432 vrijednosti pri instanciranju komponente. Drugi, jednostavniji i kompaktniji način je korištenjem slijednog opisa i inicijalizacijom memorije preko datoteke [3, 7]. Kako bi se omogućila inicijalizacija putem tekstualne datoteke, potrebno je na početku izvorne datoteke uključiti potrebnu biblioteku (*TextIO*):

```
library std;
use std.textio.all;
```

Nakon toga je potrebno napraviti funkciju tipa *impure*, koja će čitati redak po redak iz datoteke i upisivati ih u memoriju. Postoji alternativa inicijalizaciji preko funkcije, a radi se o inicijalizaciji putem procesa, no u ovom slučaju je problem činjenica da se proces izvodi u početnom trenutku vremena, a pridjeljuje signalu u vremenu *delta*. Ovdje se javlja problem prekasne inicijalizacije memorije, pa stoga ovakav model nije moguće sintetizirati niti implementirati. Iz tog razloga inicijalizaciju memorije radimo putem funkcije, jer time dobivamo inicijalizaciju sadržaja memorije u *t = 0*. Funkcija koristi funkcije za čitanje retka bitova (tip *bit_vector*) iz biblioteke TEXTIO, te ga konvertira u *std_logic_vector* i pohranjuje u memoriju:

```
impure function read_file_func (file_name : string; AW : integer) return
ram_type is
      file     FID            : text open read_mode is file_name;
      variable file_row       : bit_vector(15 downto 0);
      variable trace_line     : line;
      variable memory_content : ram_type;
   begin
      for k in 0 to 2**AW-1 loop
         readline(FID, trace_line);
         read(trace_line, file_row);
         memory_content(k) := to_stdlogicvector(file_row);
      end loop;
      return memory_content;
   end function;
```

Sada nam preostaje stvoriti signal koji će nam predstavljati programsku memoriju, te ga inicijalizirati na sadržaj tekstualne datoteke:

```
signal block_ram  : ram_type := read_file_func(rom_content_file, AWidth);
```

No, budući da se u programu javljaju i konstante s kojima se računa, potrebno je, uz čitanje instrukcija, omogućiti i čitanje podataka iz programske memorije. Iz tog razloga, blok RAM resurs kojim je predstavljena programska memorija će biti dvopristupni, kako bi se s jednog izlaza čitale instrukcije a s drugog podaci. Ovo će detaljnije biti objašnjeno u sljedećem poglavlju kod opisa jedinice za dohvat instrukcija.

Podatkovna memorija se realizira na sličan način, s tom razlikom da za podatkovnu memoriju nije potrebna inicijalizacija, te je stoga samo potrebno stvoriti tip memorije i kreirati signal tog tipa:

```
type data_mem is array (1023 downto 0) of
std_logic_vector(15 downto 0);
signal DTM : data_mem;
```

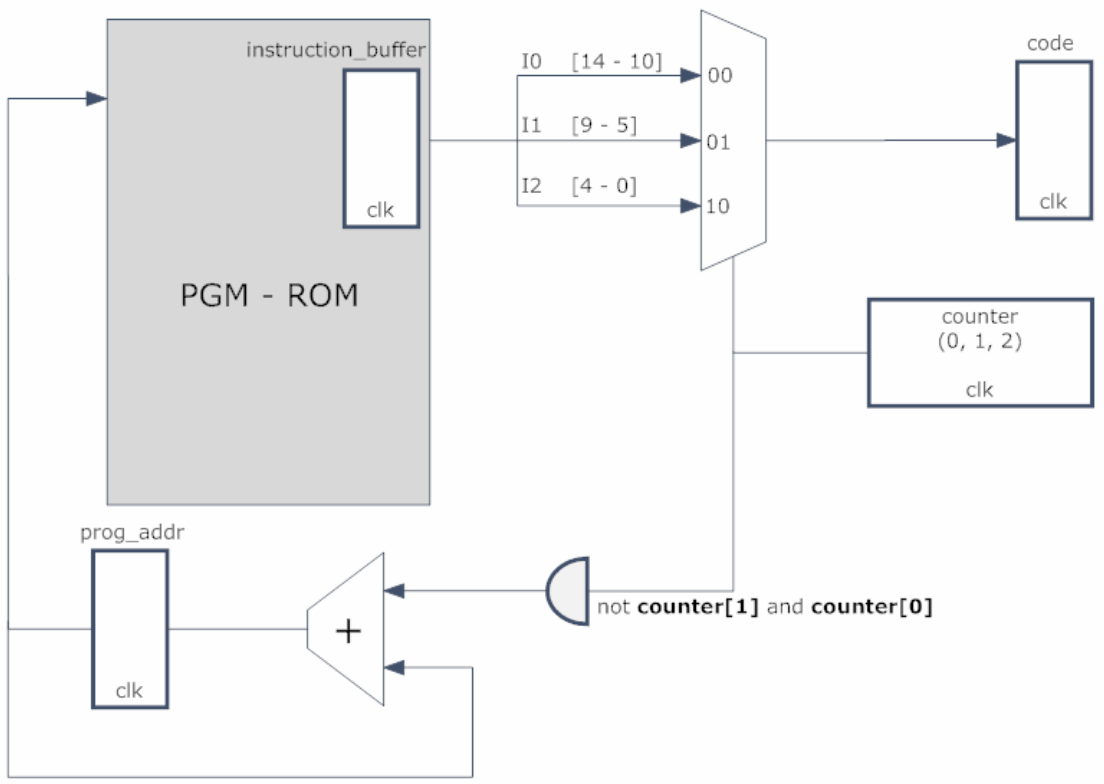
Nakon implementacije memorijskih elemenata, potrebno je implementirati jedinice koje zajedno čine procesorsku jezgru koja će obavljanjem zadanih instrukcija upisivati ili čitati iz postojećih stogova i memorija. Prva jedinica jest jedinica za dohvat instrukcija, koja je, kako ćemo pokazati, odgovorna za dvije faze protočne strukture, dohvat instrukcijskog spremnika iz programske memorije, te dohvat zasebnih instrukcija iz tog spremnika.

4.1.5. Jedinica za dohvat instrukcija

Kao je već spomenuto, jedinica za dohvat instrukcija je zaslužna za dvije faze protočne strukture. Prva faza je *dohvat retka iz memorije*, pri čemu se redak stavlja u izlazni registar blok RAM resursa. Ovaj registar je neizbježan, te je posljedica korištenja blok RAM-a, pošto čitanje sadržaja blok RAM-a mora biti sinkrono. Time je obavljena prva faza protočne strukture. Pošto su u jednom retku memorije pohranjene tri instrukcije, potrebno je implementirati mehanizam koji će ovisno o nekom brojilu odabrati jednu od te tri instrukcije i proslijediti je jedinici za dekodiranje. Ovaj mehanizam, s pripadnim izlaznim registrom čini drugu fazu protočne strukture, odnosno *dohvat instrukcije iz spremnika*. Iz ovoga slijedi kako su nam potrebna dva registra za te dvije faze protočne strukture:

```
signal instr_buffer : std_logic_vector(15 downto 0);
signal code : std_logic_vector(4 downto 0);
```

Kako bismo pojednostavili objašnjavanje jedinice za dohvat instrukcija, počet ćemo s osnovnom strukturom, čitanjem memorije redak po redak, te slijednim čitanjem instrukcija iz bloka. Na slici 13 vidimo blok shemu najjednostavnijeg oblika jedinice za dohvat instrukcija. Kao što vidimo, kada je brojač *counter* jednak 1, vrši se (zapravo na sljedeći rastući brid signala takta) inkrementiranje adresnog registra (kako bi pokazivao na sljedeći redak). Ovo se obavlja u situaciji kada je brojač jednak 1, kako bi se u sljedećem ciklusu signala takta nova adresa pojavila na izlazu adresnog registra. U tom trenutku, brojač ima vrijednost 2, te će se u sljedećem ciklusu signal takta u registru *code*, koji predstavlja instrukciju koja se dekodira, pojaviti treća instrukcija iz prethodno učitanog spremnika, a u isto vrijeme se učitava novi spremnik sa sljedeće memorijske lokacije. Ovime postižemo pravilnu protočnost *pipeline* strukture.



Slika 13. Blok shema osnovne jedinice za dohvat instrukcija

No, kako znamo da u programskoj memoriji uz retke instrukcija postoje i reci s konstantama, ovakva jedinica nije dovoljna, već ju je potrebno proširiti za dohvate podataka. Kao što je već prije rečeno, iz tog ćemo razloga programsku memoriju

implementirati pomoću dvopristupnog blok RAM-a. Ovime smo omogućili odvojeno čitanje instrukcija i podataka, koji će se na izlazu iz blok RAM-a pohranjivati u zasebne, te ćemo imati dva generatora adresa, jedan za instrukcije i jedan za podatke. Dodatnu složenost nam unosi činjenica da samo nekolicina instrukcija koristi konstante iz programske memorije. Iz tog razloga, broj podataka između dva bloka instrukcija varira u ovisnosti o instrukcijama. Slijedi da instrukcije možemo podijeliti na dvije grupe, u ovisnosti o tome koriste li neposredno podatak iz programske memorije. Srećom, *podatkovnih* instrukcija (koriste neposredno podatke iz programske memorije) je vrlo malo, samo četiri, a radi se o instrukciji koja konstantu stavlja na podatkovni stog (push), te o instrukcijama skokova (*jmp*, *jz*, i *call*). Detaljno objašnjenje ovih instrukcija slijedi u poglavlju *4.X.X Izvođenje instrukcija*. Stoga je korisno njihove operacijske kodove grupirati na način da su im tri najviša bita jednaka. Zbog jednostavnosti jedinice za dohvat instrukcija i podataka iz programske memorije, dodijelit ćemo im operacijske kodove u kojima su prva tri bita jednaka 1. Kao što smo već rekli, potrebno je nakon svakog dohvata instrukcijskog spremnika iz programske memorije zaključiti, na temelju programskih kodova instrukcija u tom spremniku, koliko podataka slijedu taj instrukcijski spremnik (slika 14), kako bi se generator programske adrese pravilno ažurirao i pokazivao na sljedeći instrukcijski blok.

INSTR	INSTR	INSTR
INSTR_P	INSTR	INSTR_P
	PODATAK_1	
	PODATAK_3	
INSTR_P	INSTR_P	INSTR_P
	PODATAK_1	
	PODATAK_2	
	PODATAK_3	
INSTR	INSTR_P	INSTR
	PODATAK_2	
INSTR	INSTR	INSTR_P
	PODATAK_3	

Slika 14. Sadržaj programske memorije

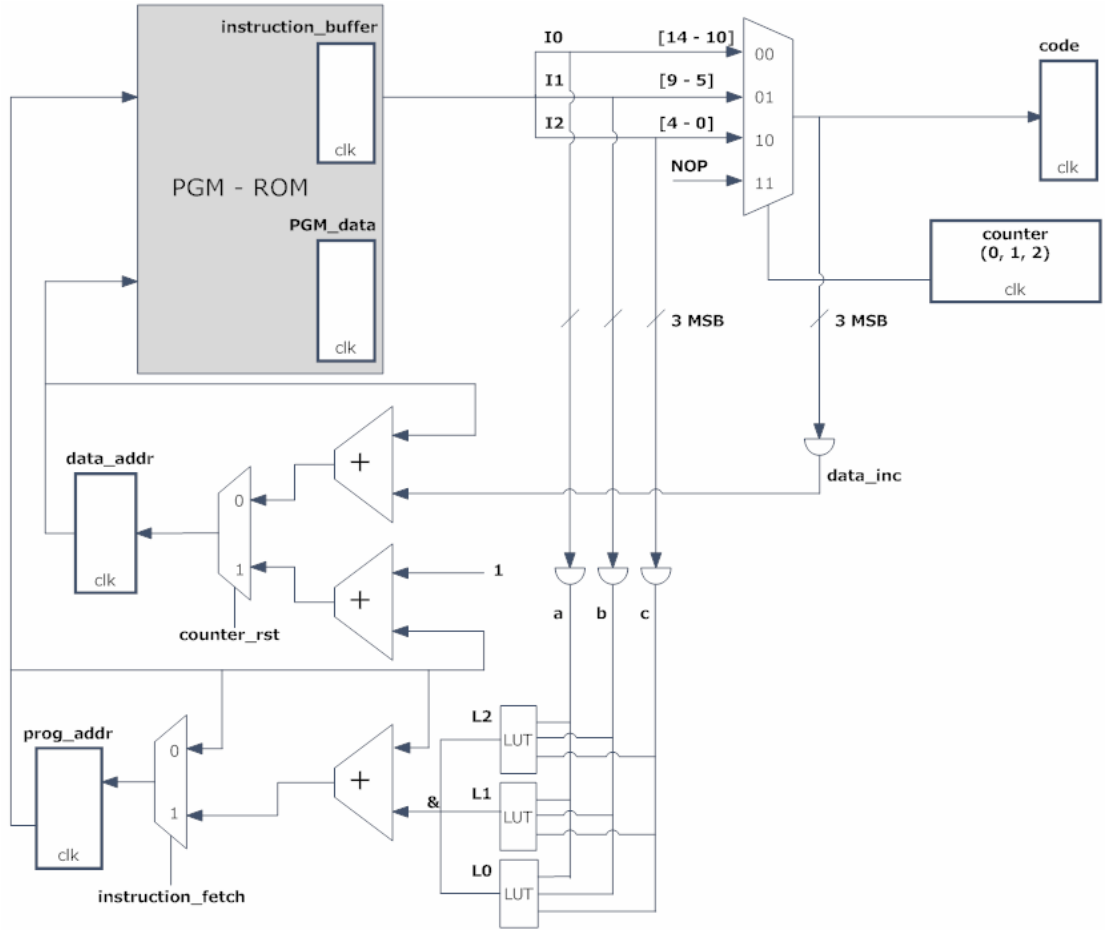
Iz ovog razloga ćemo implementirati kombinacijsku funkciju koja će u ovisnosti o 3 MSB-a (*MSB – most significant bit*) svake instrukcije u bloku generirati vrijednost za koju se povećava generator programske adrese. Budući da smo najviše bitove

ovih instrukcije zadali kao logičke jedinice, jednostavnim logičkim I sklopom s tri ulaza možemo zaključiti radi li se o podatkovnoj instrukciji. Ti signali (*a*, *b*, *c*) su ulazi u LUT-ove, čiji je rad opisan tablicom istinitosti na slici XY. Vrijednost inkrementa se dobije konkatenacijom izlaza LUT-ova (*L2 & L1 & L0*). Potrebno je naglasiti kako se vrijednosti inkrementa kreću od 1 do 4, gdje 1 označava instrukcijski spremnik bez *podatkovnih* instrukcija, dok se adresni generator inkrementira za 4 u „najgorem“ slučaju, kada imamo 3 *podatkovne* instrukcije u spremniku.

a	b	c	inkrement	L2	L1	L0
0	0	0	1	0	0	1
0	0	1	2	0	1	0
0	1	0	2	0	1	0
0	1	1	3	0	1	1
1	0	0	2	0	1	0
1	0	1	3	0	1	1
1	1	0	3	0	1	1
1	1	1	4	1	0	0

Slika 15. Tablica istinitosti i izlazi LUT-ova

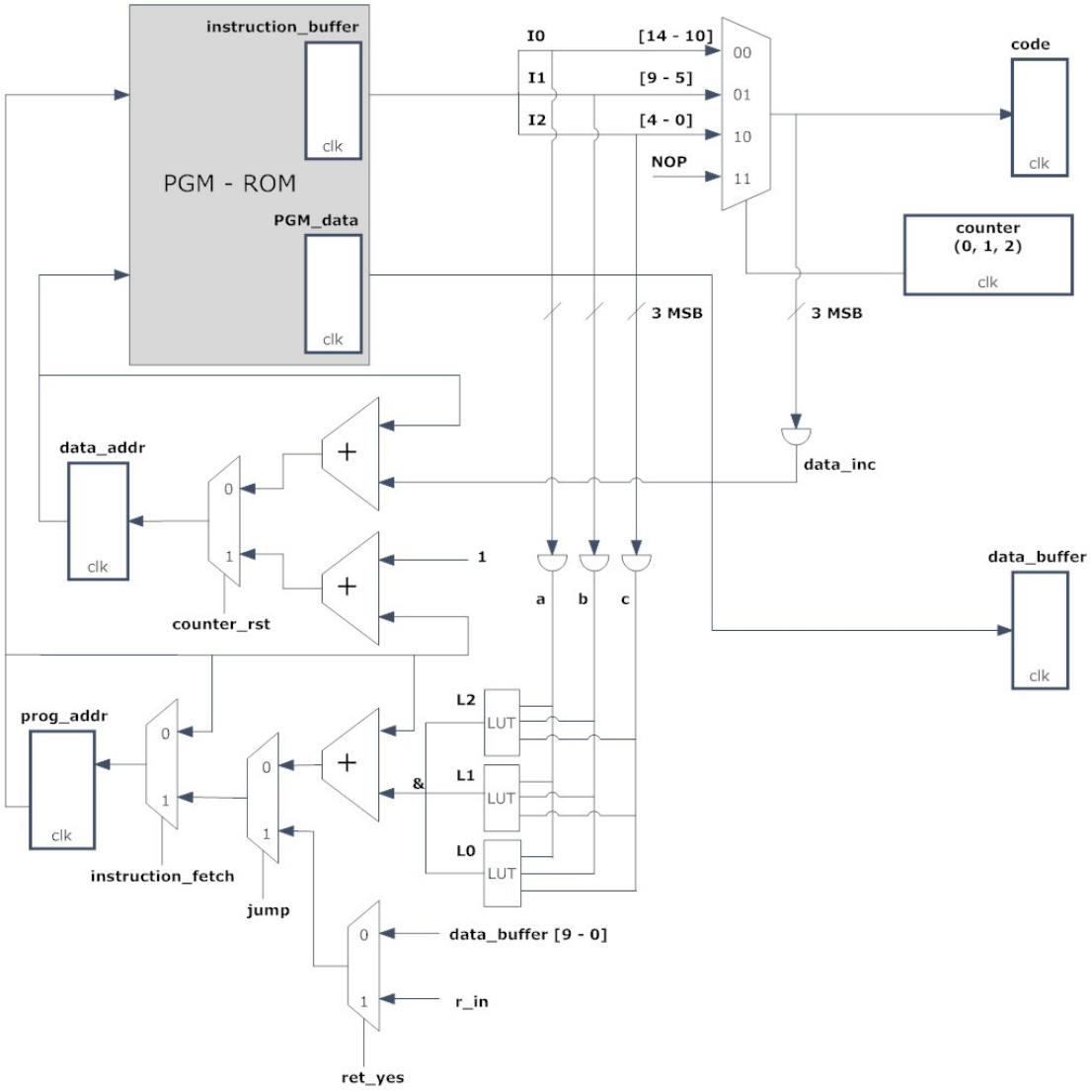
Na sličan način moramo inkrementirati i generator podatkovne adrese (za programsku memoriju). Podatkovna adresa će slijediti programsku adresu uvećanu za 1 u slučaju dohvata novog spremnika, odnosno kada pročitamo sve instrukcije iz prethodnog spremnika. Na taj način podatkovna adresa pokazuje na lokaciju iza instrukcijskog spremnika, odnosno na lokaciju na kojoj se nalazi podatak vezan uz prvu instrukciju u tom spremniku. Daljnje inkrementiranje generatora podatkovne adrese se vrši u ovisnosti o tri najviša bita trenutno izvođene (odnosno dekodirane) instrukcije. Pa tako, ukoliko trenutna instrukcija koristi podatak, potrebno je podatkovnu adresu povećati za 1, kako bi pokazivala na sljedeći podatak. Blok shema takve jedinice za dohvat prikazana je na slici XY.



Slika 16. Blok shema složenije jedinice za dohvat instrukcija

Ono što još nedostaje da bi jedinica za dohvat bila potpuna jest mogućnost promjene toka programa. Promjena toka programa vrlo je koristan resurs procesora, jer se time mogu implementirati mnoge složnije funkcionalnosti, kao što su primjerice petlje ili potprogrami. U tu svrhu, postoji nekoliko instrukcija koje provode skokove u programu. To su već spomenute *jmp* i *jz*, te instrukcije koje pozivaju i vraćaju se iz podrutine, *call* i *ret*. Ove četiri instrukcije možemo podijeliti u dvije grupe, u ovisnosti o izvoru podatka za skok. Kod prve tri navedene instrukcije, podatak dolazi iz programske memorije, odnosno iz podatkovnog spremnika programske memorije (primjećujemo kako su te instrukcije podatkovne, odnosno neposredno koriste podatke iz programske memorije). Posljednja instrukcija obavlja povratak iz podrutine, te podatak za skok koji ta instrukcija obavlja dolazi s povratnog stoga, koji smo već spomenuli pri implementaciji stogova korištenjem distribuiranog RAM-a. Kako bismo implementirali ovu funkcionalnost jedinice za dohvat instrukcija, potrebno je dodati još nekoliko multipleksora, te definirati kontrolne signale za njih. Kod generatora programske

adrese, treba umetnuti dodatni multipleksor, koji će, ukoliko je došlo do skoka, pročitati adresu sljedeće instrukcije s jednog od mogućih izvora. Ukoliko do skoka nije došlo, nastavlja se slijedno čitanje programa (kontrolni signal *jump*). Dodatni multipleksor je potreban kako bi se odredio spomenuti izvor skoka. Kao što smo pokazali, dva su moguća izvora podatka za skok, te se samo u slučaju instrukcije *ret* koristi registar s vrha povratnog stoga (kontrolni signal *ret_yes*). Uz ovo, postoji i potreba za izvanrednim resetiranjem brojača instrukcija u spremniku, zato što nakon svakog skoka treba početi izvoditi instrukcije od početka spremnika. Budući da postoji jedan ciklus „kašnjenja“ od postavljanja nove adrese, do dohvata novog spremnika (zbog prolaska tog signala kroz registar *data_addr*), slijedi da i signal za reset brojača mora stići jedan ciklus nakon aktivacije signala *jump* koji označava da je došlo do skoka. Ovo se vrlo jednostavno realizira korištenjem sinkronog registra koji na izlaz daje stanje signala *jump* iz prošlog ciklusa takta. Posljednja stvar koju je potrebno ostvariti jest kašnjenje podatka iz programske memorije. Naime, zbog postojanja faze *dekodiranja instrukcija* između dohvata i izvođenja, podatak vezan uz trenutnu instrukciju će biti prepisan ukoliko je i sljedeća instrukcija podatkovna. Kako bi se ovo izbjeglo, treba dodati dodatni registarski nivo na podatak koji se dohvati iz programske memorije, čime osiguravamo pravovremenost podatka za ispravno izvođenje *podatkovnih* instrukcija. Na slici XY je prikazana konačna verzija jedinice za dohvat koja implementira svu potrebnu funkcionalnost (zaseban dohvat instrukcijskog spremnika i podataka iz programske memorije, te mogućnost promjena toka programa), i time ostvaruje prve dvije faze protočne strukture: *dohvat instrukcijskog spremnika*, te *dohvat instrukcije iz spremnika*.



Slika 17. Konačna verzija jedinice za dohvat

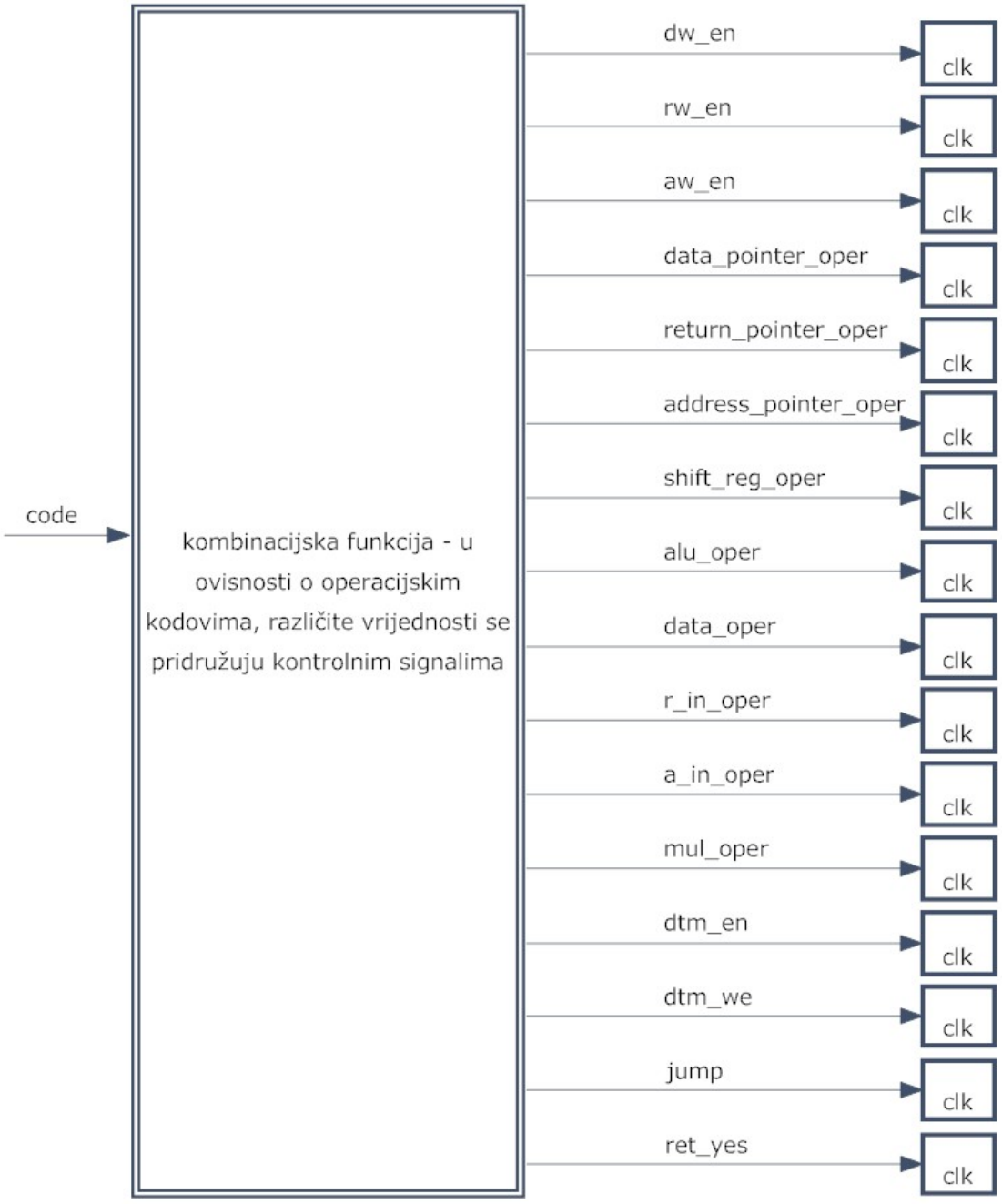
U sljedećem odsječku koda prikazani su kontrolni signali, te kombinacijska funkcija za generiranje inkrementa programske adrese (*L2*, *L1* i *L0*). Dodavanjem funkcionalnosti bilo je potrebno složenije formiranje kontrolnih signal, pošto neke kombinacije imaju nepoželjne posljedice. Tako, primjerice, kod signala za dohvat novog instrukcijskog bloka (*instruction_fetch*), moramo onemogućiti aktiviranje tog signala jedan ciklus nakon dohvata novog spremnika. Kada to ne bismo učinili, postojala bi mogućnost da se „nova“ instrukcija dohvati odmah nakon skoka (zbog vrijednosti brojača). Cjelokupni opis jedinice za dohvat instrukcija nalazi se u poglavlju *9. Dodatak A: VHDL kod procesora*.

```
-- counter reset zbog skoka
process(clk)
begin
	if rising_edge(clk) then
		if reset = '1' then
			cnt_rst_jump <= '0';
		else
			cnt_rst_jump <= jump;
		end if;
	end if;
end process;
-- reset brojaca
counter_rst <=  (counter(1) and not counter(0)) or cnt_rst_jump;
-- dohvat sljedece instrukcije, bilo zbog slijeda instrukcija ili skoka
instr_fetch <= ((not counter(1) and counter(0)) or jump) and not
		cnt_rst_jump;

a <= instr_buffer(14) and instr_buffer(13) and instr_buffer(12);
b <= instr_buffer(9) and instr_buffer(8) and instr_buffer(7);
c <= instr_buffer(4) and instr_buffer(3) and instr_buffer(2);

L2 <= a and b and c;
L1 <= (not a and not b and c) or
	(not a and b and not c) or
	(not a and b and c) or
	(a and not b and not c) or
	(a and not b and c) or
	(a and b and not c);
L0 <= (not a and not b and not c) or
	(not a and b and c) or
	(a and not b and c) or
	(a and b and not c);
-- konkatenacija izlaza LUT-ova
pa_adder <= L2 & L1 & L0;
```

Ovime je implementirana jedinica za *dohvat instrukcijskog spremnika*, te *dohvat instrukcije iz spremnika*. Sljedeća faza protočne strukture jest *dekodiranje instrukcija*, te je to tema sljedećeg poglavlja.

4.1.6. Dekodiranje instrukcija

Dekodiranje instrukcija se obavlja sinkrono sa signalom takta. U tu svrhu, napisan je proces po uzoru na procesor P16 [8], gdje se u ovisnosti o sadržaju registra *code*, odnosno registra koji sadrži jednu od tri instrukcije iz spremnika, različite vrijednosti pridružuju raznim kontrolnim signalima, kako je prikazano na slici XY.



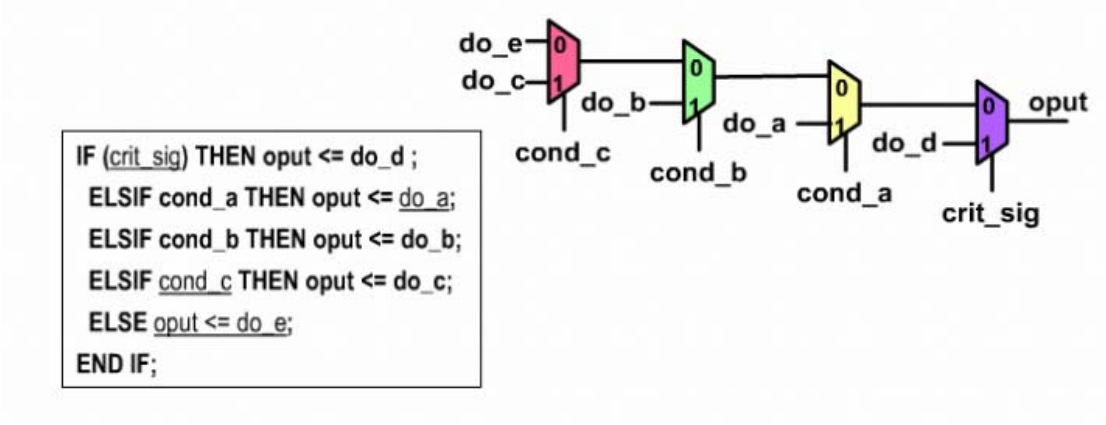
Slika 18. Blok shema dekodiranja instrukcija

Dekodiranje signala izvodi kombinacijska instrukcija s 5 ulaznih varijabli, što znači da ova jedinica koristi dva LUT-a i pripadajući multipleksor MUXF5, na način koji je objašnjen u poglavlju *3.1.2. Konfigurabilni logički blokovi*. Slijedi opis svih implementiranih instrukcija s pripadajućim operacijskim kodovima. Prisjetimo se, ograničeni smo brojem bitova pojedine instrukcije, te u ovom slučaju imamo maksimalno 32 instrukcije (5 bitova po instrukciji):

```
constant nop :      std_logic_vector(4 downto 0) := "00000";
constant push :     std_logic_vector(4 downto 0) := "11100";
constant call :     std_logic_vector(4 downto 0) := "11101";
constant jmp :      std_logic_vector(4 downto 0) := "11110";
constant jz :       std_logic_vector(4 downto 0) := "11111";
constant pop :      std_logic_vector(4 downto 0) := "00010";
constant dup :      std_logic_vector(4 downto 0) := "01001";
constant swap :     std_logic_vector(4 downto 0) := "01010";
constant ret :      std_logic_vector(4 downto 0) := "00011";
constant neg :      std_logic_vector(4 downto 0) := "01011";
constant comp :     std_logic_vector(4 downto 0) := "01100";
constant add :      std_logic_vector(4 downto 0) := "01101";
constant sub :      std_logic_vector(4 downto 0) := "01110";
constant i_and :    std_logic_vector(4 downto 0) := "01111";
constant i_or :     std_logic_vector(4 downto 0) := "01000";
constant i_xor :    std_logic_vector(4 downto 0) := "10100";
constant shla :     std_logic_vector(4 downto 0) := "10010";
constant shll :     std_logic_vector(4 downto 0) := "10011";
constant shra :     std_logic_vector(4 downto 0) := "10000";
constant shrl :     std_logic_vector(4 downto 0) := "10001";
constant mul :      std_logic_vector(4 downto 0) := "10101";
constant mac :      std_logic_vector(4 downto 0) := "10111";
constant sta :      std_logic_vector(4 downto 0) := "11000";
constant lda :      std_logic_vector(4 downto 0) := "11001";
constant store :    std_logic_vector(4 downto 0) := "11010";
constant sti :      std_logic_vector(4 downto 0) := "11011";
constant load :     std_logic_vector(4 downto 0) := "00110";
constant ldi :      std_logic_vector(4 downto 0) := "00111";
constant str :      std_logic_vector(4 downto 0) := "00100";
constant ldr :      std_logic_vector(4 downto 0) := "00101";
```

Ovime je implementirano ukupno 30 instrukcija, što znači da je u budućim proširenjima moguće dodati još dvije.

Kreiranjem konstanti koje predstavljaju instrukcije smo olakšali dekodiranje, jer sada možemo uspoređivati registar *code* s pojedinim konstantama, i u ovisnosti o konstanti aktivirati potrebne signale. Ova usporedba se u VHDL-u može napraviti na dva načina: *if-then-else* naredbama ili *case* naredbama.



Slika 19. Sinteza *if-then-else* naredbe [11]

U ovom slučaju prednost ima *case* naredba, pošto je XST realizira pomoću paralelnih multipleksora, čime imamo manje logičkih nivoa, te možemo postići

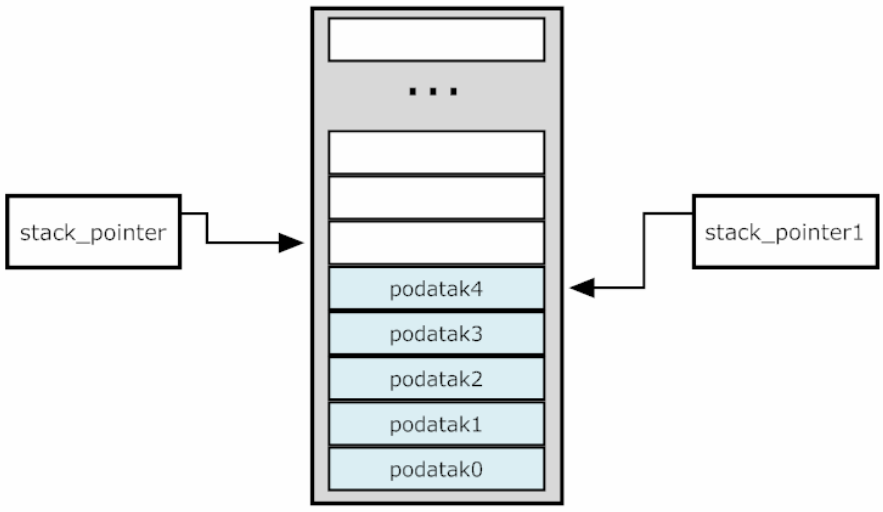
veću brzinu u dizajnu. Za razliku od toga, kod *if-then-else* naredbi, XST nekim signalima daje prioritet, dok neki moraju proći kroz sve LUT-ove i sve logičke nivoe, čime povećavamo kašnjenje tih signala i gubimo na brzini (slika 19) [11]. Ponekad ne možemo izbjeći *if-then-else* naredbe, te je tada potrebno signale koji su nam vremenski kritični staviti u prvi stupanj *if-then-else* naredbe, jer time taj signal ima najmanji put do izlaza, kao što je prikazano na slici 14. Iz sličnih razloga preporučuje se izbjegavanje ugniježđenih *if-then-else* naredbi.

Dodatni problem kod *if-then-else* ili *case* naredbi nam predstavlja nenamjerno uključivanje *latcheva*. Ova pojava je problematična kod sinkronog dizajna jer latch kvari sinkroni put, te otežava integriranim alatima (*Xilinx Timing Engine*) pravilno analiziranje puta. Razlog tome je činjenica da *latch* daje izlazni signal na promjenu ulaznog signala, dok god je *enable* ulaz aktivan, te drže zadnji izlaz. Suprotno tome, sinkroni registri daju izlazni signal na temelju ulaza na brid signala takta. Budući da su latchevi nepoželjni, XST nam nakon procesa sinteze daje upozorenje ukoliko naš dizajn uključuje *latch* (*WARNING:Xst:737 - Found n-bit latch for signal <signal name>*), što svakako moramo provjeriti prilikom svake sinteze. Razlog pojavljivanja *latcha* leži u nepotpunoj definiciji *if-then-else* ili *case* naredbi [12]. Naime, u svakoj grani tih naredbi moramo dodijeliti vrijednosti svim signalima, iako se možda u nekim realizacijama uvjeta neki signali ne koriste, kako XST ne bi na mjesto tog signala stavio *latch*. Postoji nekoliko rješenja ovog problema, ovisno o uzroku. Ako koristimo *case* naredbu, trebamo dodati sve moguće slučajeve. Ukoliko je slučajeva koji se ne koriste mnogo, možemo koristiti naredbu *„when others =>“*, no ona može stvoriti nepotreban višak logike, što može rezultirati većim i sporijim dizajnom, pošto bilo koje nepoznato stanje treba logiku koja bi ga dovela u poznato. Slično je i kod *if-then-else* naredbi, koje trebaju imati zaključnu *else* granu. Najčešće rješenje, kao što je već spomenuto, je dodjeljivanje vrijednosti istim signalima u svim granama[13]. U slučajevima kada je takvih signala mnogo, bolji način je dodjeljivanje vrijednosti svim signalima izvan *case* naredbe, te unutar istog procesa, kako bi se pri svakom izvršavanju procesa svim signalima dodijelila neka zadana vrijednost. Ovaj način ćemo koristiti kao bi osigurali izostanak *latcheva* u procesu dekodiranja instrukcija.

Budući da se radi o stogovnoj arhitekturi, nakon svake instrukcije ažurirat će se sadržaj jednog ili više stogova. Isto tako, zbog mogućnosti pohranjivanja nekih podataka u podatkovnu memoriju (predstavljenu na sklopu blok RAM resursom),

neke će instrukcije upisivati ili čitati podatke iz ove memorije. Kako bismo omogućili upise i čitanja sa stogova, svaki stog posjeduje ulazni registar, koji je postavljen „iznad“ tog stoga i predstavlja vrh stoga. Na ovaj način moguće je postići izvršavanje operacija nad najviša dva elementa stoga, pri čemu je najviši predstavljen registrom, a drugi najviši izlazom iz distribuiranog RAM-a. Dodatno, kod podatkovnog stoga postoji dodatni registar, pošto jedna instrukcija (*multiply and accumulate – mac*) koristi tri podatka sa stoga. Ovime smo opisali ukupno 5 mogućih odredišta za rezultate instrukcija. Blok shema i prikaz odredišnih resursa su prikazani slikom XY, te objašnjeni u poglavlju *4.1.7. Izvođenje instrukcija*. Tih 5 odredišta je opisano pomoću 16 kontrolnih signala, od kojih je 5 signala za omogućavanje (*enable* signali), 2 signala za skokove te 9 signala koji kontroliraju razne multipleksore. Dva signala koja se koriste kod skokova su *jump* i *ret_yes*. *Jump* signal označava da je dekodirana instrukcija za skok, te taj signal kontrolira multipleksor koji određuje izvor sljedeće instrukcije (slika 17). Signal *ret_yes* se aktivira ako je dekodirana instrukcija za povratak iz potprograma, odnosno instrukcija *ret*. Ovaj signal kontrolira multipleksor za odabir izvora skoka, pošto se kod ove instrukcije adresa nove instrukcije dolazi s povratnog stoga (detaljnije objašnjeno u prethodnom poglavlju). Kod pohranjivanja ili čitanja podataka iz podatkovne memorije, potrebno je aktivirati signale koji su potrebni blok RAM resursu, a to su signal za omogućavanje memorije (*dtm_en*), te signal za omogućavanje upisa u memoriju (*dtm_we*). Nadalje, kod pohranjivanja podataka na stogove, potrebni su nam signali za omogućavanje upisa u distribuirani RAM, te su to signali *dw_en*, *rw_en* i *aw_en*, pri čemu oni omogućavaju upise na podatkovni, povratni i adresni stog. Isto tako, budući da su stogovi izvedeni pomoću distribuiranog RAM-a, potrebno je imati dva adresna signala za svaki stog, pri čemu jedan pokazuje na prvu praznu lokaciju na stogu, dok drugi pokazuje na vrh stoga (slika 20). Ova činjenica proizlazi iz načina rada komponenti od kojih su izgrađeni stogovi, odnosno *RAM16X1S*. Kod ove se komponente podatak s podatkovnog ulaza (*d_in*) stavlja na podatkovni izlaz (*d_out*) uz stabilan signal za adresu, na sljedeći rastući brid signala takta, ako je aktivan signal za omogućavanje upisa (*we*). Kada se adresni signal promijeni, na podatkovnom izlazu se pojavi „stari“ podatak, te se na sljedeći rastući brid signala takta postavi podatak s podatkovnog ulaza [15]. Iz ovoga slijedi da su nam potrebna dva adresna signala kako bismo mogli u jednom ciklusu ili pohraniti podatak na stog ili

uzeti podatak s njega. Pri tome se na adresni ulaz distribuiranih RAM-ova koji sačinjavaju svaki stog se postavlja jedan od ova dva adresna signala u ovisnosti o signalu za omogućavanje upisa. Pa tako, ukoliko upisujemo podatak na stog, „aktivan“ je signal koji pokazuje na prvu praznu lokaciju, te se na tu lokaciju stavlja novi podatak, uz inkrementiranje oba pokazivača. Na isti način, ako čitamo podatak sa stoga, „aktivan“ je pokazivač na najviši način, te se nakon uzimanja podatka sa stoga oba pokazivača umanjuju.



Slika 20. Stog i pripadni pokazivači

Ovime smo opisali još jedan kontrolni signal (zapravo tri signala, po jedan za svaki stog), a to je signal koji određuje koja se vrijednost pridružuje pokazivačima stoga, odnosno obavlja li se njihovo inkrementiranje ili dekrementiranje. Dodatno, pošto smo rekli da jedna instrukcija koristi prva tri elementa s podatkovnog stoga, nakon izvođenja te instrukcije potrebno je smanjiti vrijednosti pokazivača stoga za 2. Preostali kontrolni signali su vezani uz registre koji predstavljaju vrhove stogova, te oni kontroliraju multipleksore koji određuju ulaze u svaki od tih registara (*shift_reg_oper*, *alu_oper*, *mul_oper*, *data_oper*, *r_in_oper* i *a_in_oper*).

U nastavku slijedi popis svih implementiranih instrukcija, s pripadnim resursima koje koriste, odnosno koji se aktiviraju prilikom dekodiranja tih instrukcija.

Instrukcije *push* i *pop*

Instrukcija push učitava podatak iz programske memorije i stavlja ga na vrh podatkovnog stoga. Mehanizam pravilnog učitavanja podataka iz programske

memorije je pokazan u poglavlju *4.1.5. Jedinica za dohvat instrukcija*. Tamo je podatak vezan uz određenu instrukciju pročitan iz programske memorije u izlazni registar blok RAM-a, te je zatim napunjen u podatkovni spremnik (*data_buffer*) kako bi nakon faze dekodiranja bio na raspolaganju instrukciji koja ga zahtjeva, te nam sada samo preostaje učitati taj podatak u ulazni registar stoga u povećati pokazivače na vrh stoga. Kako je već objašnjeno, kod podatkovnog stoga postoje dva registra koja predstavljaju vrh stoga, kako bi se mogle implementirati instrukcije koje vrše operacije nad dva ili tri najviša elementa stoga.

Instrukcija pop obavlja operaciju suprotnu onoj instrukcije push, odnosno uzima podatak s podatkovnog stoga, uz smanjenje pokazivača na stog. Ovaj podatak se dalje nigdje ne pohranjuje, te da je, ukoliko ga želimo sačuvati, potrebno prethodno pohraniti u memoriju ili na neki od preostalih stogova. Budući da instrukcije *push* i *pop* koriste samo podatkovni stog, potrebno je prave vrijednosti pridružiti signalima koji kontroliraju izvor podataka za pokazivače na podatkovni stog, te dva ulazna registra za podatkovni stog. Dodatno, kod instrukcije *push* moramo omogućiti upis u distribuirane RAM-ove, aktivacijom pripadnog signala, koji ujedno kontrolira koji signal vodimo na adresni ulaz distribuiranog RAM-a.

```
when push =>
      dw_en <= '1'; data_pointer_operation <= "01"; data_oper <= "1000";
      d_in_oper <= "01";
when pop =>
      data_pointer_operation <= "10"; data_oper <= "0011";
      d_in_oper <= "10";
```

Instrukcije *call* i *ret*

Ove se instrukcije koriste za pozive potprograma, odnosno povratke iz njih. Stoga ove instrukcije moraju ili pohraniti podatak na povratni stog (*call* stavlja adresu na koju se potrebno vratiti), ili ga uzimaju (*ret* uzima taj podatak i prosljeđuje ga jedinici za dohvat). Slijedi da ove instrukcije koriste samo resurse vezane uz povratni stog, odnosno, pokazivače na stog i ulazni registar. Dakako, kao i kod podatkovnog stoga, potrebno je omogućiti upis na stog tijekom *call* instrukcije, što se obavlja aktiviranjem potrebnog signala. Dodatno, obje instrukcije moraju aktivirati signal za skok, dok *ret* instrukcija aktivira i signal *ret_yes* kojim upravlja multipleksorom, te kao izvor adrese za skok postavlja vrh povratnog stoga.

```
when call =>
       rw_en <= '1'; return_pointer_operation <= "01"; r_in_oper <= "10";
       jump <= '1';
when ret =>
       return_pointer_operation <= "10"; r_in_oper <= "11";
       ret_yes <= '1'; jump <= '1';
```

Instrukcije *dup* i *swap*

Postoje još dvije instrukcije koje obavljaju stogovne operacije nad podatkovnim stogom. Radi se o instrukcijama *dup* i *swap* koje svoj korijen vuku iz programskog jezika Forth. Budući da je stog sam po sebi slijedan, ubacujemo podatke jednog za drugim, korisno je imati instrukciju koja će zamijeniti prva dva podatka na vrhu stoga, što čini instrukcija *swap*. S druge strane, instrukcija *dup* duplira podatak na vrhu stoga. Ovo je korisna operacija kada trebamo, primjerice, izračunati kvadrat nekog broja, tada ga je potrebno staviti na stog, te duplirati. Ove su instrukcije vrlo slične instrukciji *push*, jer koriste samo podatkovni stog. Instrukcija *dup* kopira vrh stoga, te ga stavlja u drugi registar na vrhu podatkovnog stoga, uz inkrementiranje pokazivača. Instrukcija *swap* je još jednostavnija, te ona samo zamijeni dva najviša elementa stoga, tj. zamijeni vrijednosti registrima koji su postavljeni „iznad“ podatkovnog stoga.

```
when dup =>
       dw_en <= '1'; data_pointer_operation <= "01"; d_in_oper <= "01";
when swap =>
       data_oper <= "0011"; d_in_oper <= "01";
```

ALU instrukcije

Slijedi nekolicina instrukcije koje ostvaruju aritmetičke i logičke instrukcije, pa stoga koriste iste resurse: ALU multipleksor i podatkovni stog. No, neke od ovih instrukcija vrše operacije samo nad prvim podatkom na vrhu stoga, te zbog toga nije potrebno mijenjati pokazivače na stog. To su instrukcije *comp*, koja komplementira prvi element na stogu, te instrukcija *neg*, koja vrši aritmetičku negaciju prvog podatka na stogu (dvojni komplement). Ostale instrukcije vrše operaciju nad dva najviša elementa stoga, te je kod tih instrukcija potrebno smanjiti pokazivače na stog. Radi se o instrukcijama za zbrajanje i oduzimanje (*add* i *sub*), te o instrukcijama koje implementiraju operacije logičko I, ILI i isključivo ILI (*i_and*, *i_or*, *i_xor*).

```
when neg =>
       data_oper <= "0101"; alu_oper <= "001";
when comp =>
       data_oper <= "0101"; alu_oper <= "010";
when add =>
       data_oper <= "0101"; alu_oper <= "011";
       data_pointer_operation <= "10"; d_in_oper <= "10";
when sub =>
       data_oper <= "0101"; alu_oper <= "100";
       data_pointer_operation <= "10"; d_in_oper <= "10";
when i_and =>
       data_oper <= "0101"; alu_oper <= "101";
       data_pointer_operation <= "10"; d_in_oper <= "10";
when i_or =>
       data_oper <= "0101"; alu_oper <= "110";
       data_pointer_operation <= "10"; d_in_oper <= "10";
when i_xor =>
       data_oper <= "0101"; alu_oper <= "111";
       data_pointer_operation <= "10"; d_in_oper <= "10";
```

Instrukcije za posmak

Postoje ukupno četiri instrukcije za posmak, te se njima mogu implementirati aritmetički posmak ulijevo i udesno (*shla*, *shra*) te logički posmaci ulijevo i udesno (*shll*, *shrl*). Aritmetički posmak ulijevo (*shla*) ekvivalentan je množenju podatka s 2, jer zadržava predznak na način da posmiče nižih 15 bitova dok najviši bit, koji u formatu dvojnog komplementa označava predznak, ostaje na svom mjestu. Slično je i kod posmaka udesno (*shra* – dijeljenje s 2), samo što se kod te operacije radi tzv. proširivanje predznaka (*eng. sign extension*). To znači da se udesno posmiče nižih 15 bitova, dok se najviši bit kopira na upražnjeno mjesto. Za razliku od aritmetičkog, logički posmak (*shll* i *shrl*), ne mari za bit predznaka, te se tom operacijom svih 16 bitova posmiče u bilo kojem smjeru, te se na prazna mjesta dodaje nula.

Pošto se ove instrukcije izvode samo nad najvišim elementom stoga, nije potrebno ažurirati pokazivače na stog, već se samo upravlja multipleksorima kao bi potreban podatak došao do vrha podatkovnog stoga.

```
when shla =>
       data_oper <= "0110"; shift_reg_oper <= "00";
when shll =>
       data_oper <= "0110"; shift_reg_oper <= "01";
when shra =>
       data_oper <= "0110"; shift_reg_oper <= "10";
when shrl =>
       data_oper <= "0110"; shift_reg_oper <= "11";
```

Instrukcije vezane uz množilo

Ove instrukcije tiču se namjenskog bloka množila na sklopu Spartan-3E. Radi se o instrukciji *mul*, koja obavlja množenje dvaju podataka na vrhu stoga, te o instrukciji *mac*, koja uz množenje ta dva podatka obavlja i pribrajanje trećeg (gledano s vrha stoga). Množilo je u dizajn uključeno slijednim opisom, te se koristi kombinacijsko množilo [3]. Kako je već spomenuto u poglavlju *3.1.5. Namjenska množila*, množila uključena u Spartan-3E FPGA sklop imaju mogućnost ostvarivanja protočne strukture prilikom množenja,, budući da imaju ulazne i izlazni registar. No, ovdje se ti registri ne koriste pošto se sama operacija množenja odvija u posljednjoj fazi protočne strukture, pa bi uključivanje registara zahtijevalo dodatne faze u *pipeline-u*. Ovime se ne gubi gotovo ništa na brzini dizajna, zbog sličnih vremenskih odnosa na ostalim dijelovima procesora. Budući da se koriste podaci koji se nalaze dublje na stogu, potrebno je smanjiti pokazivače na stog. Instrukcija *mac* je jedina instrukcija koja koristi tri podatka sa vrha podatkovnog stoga, pa zbog toga treba nakon njenog izvođenja umanjiti pokazivače na stog za 2. Obje ove instrukcije koriste podatkovni stog, i pripadni multipleksor, te dodatni multipleksor koji određuje radi li se o operaciji množenja ili o množenju s akumulacijom.

```
when mul =>
       data_oper <= "0111"; mul_oper <= '1';
       data_pointer_operation <= "10"; d_in_oper <= "10";
when mac =>
       data_oper <= "0111"; data_pointer_operation <= "11";
       d_in_oper <= "10";
```

Instrukcije za skokove

Postoje dvije instrukcije za skokove u programu (bez pohranjivanja na povratni stog). Bezuvjetni skok obavlja se instrukcijom *jmp*, a postoji i instrukcija za uvjetni skok (*jz*), koja se ne izvršava ukoliko podatak na vrhu stoga nije jednak nuli (u tu svrhu postoji jednobitni signal *zero*). Stoga ove instrukcije moraju samo aktivirati signal *jump*.

```
when jmp =>
       jump <= '1';
when jz =>
       jump <= zero;
```

Instrukcije *sta*, *lda*, *str* i *ldr*

Kao što smo već napomenuli, svi podaci idu preko podatkovnog stoga, pa iz tog razloga postoje instrukcije koje prosljeđuju podatke s jednog stoga na drugi, odnosno s podatkovnog na ostale, i nazad na podatkovni. Podsjetimo, vrh adresnog stoga se koristi kao pokazivač za podatkovnu memoriju, te je stoga korisno imati mogućnost računanja adresa, pošto u memoriji možemo na raznim mjestima imati pohranjene neke bitne podatke. Na sličan način, vrh manipulacijama nad podatkom na vrhu povratnog stoga moguće je mijenjati povratnu adresu, primjerice u ovisnosti o nekom uvjetu. Tako instrukcije za prebacivanje podataka s podatkovnog stoga na adresni (*sta*) i povratni (*str*) koriste sve resurse vezane uz sve stogove, umanjujući pokazivače podatkovnog i uvećavajući pokazivače na adresni odnosno povratni stog. S druge strane, instrukcije za upis na podatkovni stog s povratnog (*ldr*) ili adresnog (*lda*) umanjuju pokazivače na izvorne stogove, te uvećavaju pokazivač na odredišni stog.

```
when sta =>
       aw_en <= '1'; data_pointer_operation <= "10"; data_oper <= "0011";
       d_in_oper <= "10"; address_pointer_operation <= "01";
       a_in_oper <= "01";
when lda =>
       dw_en <= '1'; data_pointer_operation <= "01"; data_oper <= "0001";
       d_in_oper <= "01"; address_pointer_operation <= "10";
       a_in_oper <= "11";
when str =>
       rw_en <= '1'; data_pointer_operation <= "10"; data_oper <= "0011";
       d_in_oper <= "10"; return_pointer_operation <= "01";
       r_in_oper <= "01";
when ldr =>
       dw_en <= '1'; data_pointer_operation <= "01"; data_oper <= "0010";
       d_in_oper <= "01"; return_pointer_operation <= "10";
       r_in_oper <= "11";
```

Instrukcije za pohranu i čitanje iz podatkovne memorije

Poznato je da procesor uz programsku memoriju sadrži i podatkovnu memoriju u koju možemo upisivati rezultate operacija. Često u toku izvršavanja programa imamo potrebu za stalnijim pohranjivanjem nekih podataka, te nam stog ne pruža dovoljno kapaciteta. Zbog toga ćemo uposliti dodatno blok RAM s FPGA sklopa, te ćemo ga nazvati DTM (podatkovna memorija). Za uključivanje blok RAM-a u dizajn koristit ćemo slijedni opis, po predlošku proizvođača [7]. Postoje dvije instrukcije za svaku od ovih akcija, instrukcija koja čita/upisuje podatak i inkrementira adresu (*ldi*, *sti*), te instrukcija koja samo čita ili piše podatak u memoriju (*load*, *store*). Ove dvije inačice naizgled istih instrukcija mogu imati

potpuno različite primjene. Primjerice, korištenjem instrukcije *store*, možemo na određeno mjesto u memoriji pohranjivati varijable koje se često mijenjaju. S druge strane, ukoliko računamo uzorke nekog signala, te je taj signal potrebno pohraniti u memoriju, koristimo instrukciju *sti*, koja automatski povećava adresu na koju se podaci zapisuju.

Ove instrukcije moraju aktivirati kontrolne podatke za podatkovnu memoriju, te raditi stavljanje ili uzimanje podataka s podatkovnog stoga, uz opciju uvećavanja podatka na vrhu adresnog stoga, koji, kako smo već rekli, služi kao adresa za pristup podatkovnoj memoriji.

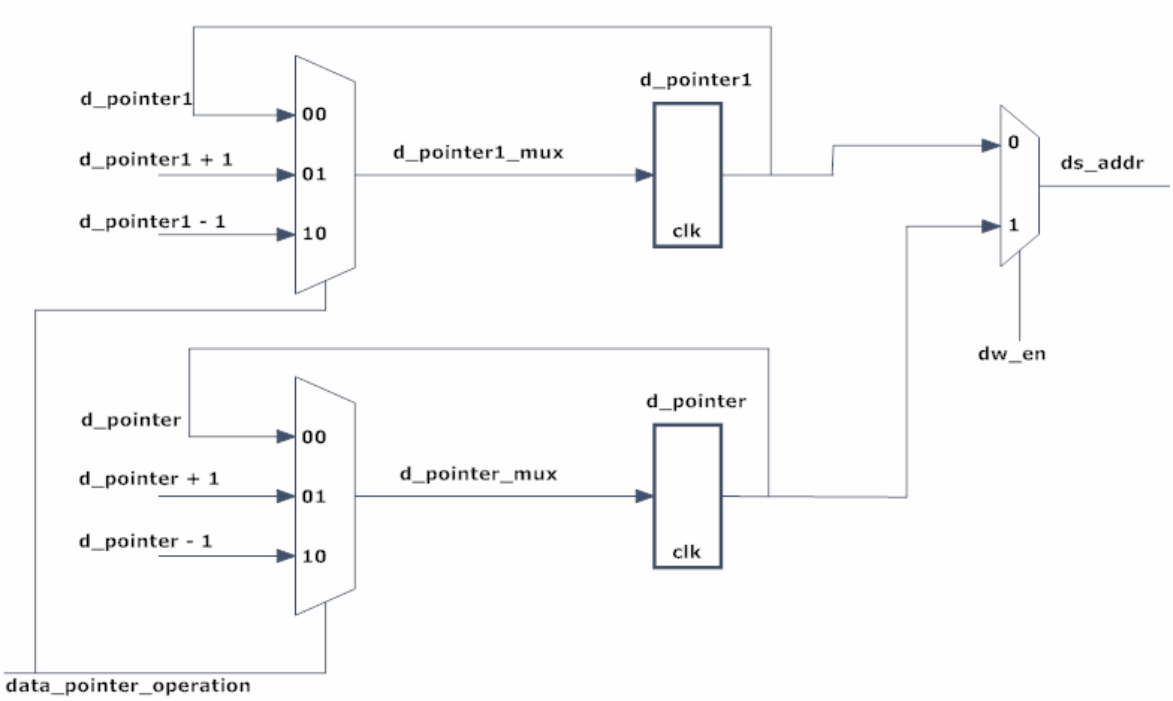
```
when store =>
       dtm_en <= '1'; dtm_we <= '1';
       data_pointer_operation <= "10"; data_oper <= "0100";
       d_in_oper <= "01";
when sti =>
       dtm_en <= '1'; dtm_we <= '1'; a_in_oper <= "10";
       data_pointer_operation <= "10"; data_oper <= "0100";
       d_in_oper <= "01";
when load =>
       dtm_en <= '1'; dw_en <= '1';
       data_pointer_operation <= "01"; data_oper <= "0011";
       d_in_oper <= "10";
when ldi =>
       dtm_en <= '1'; dw_en <= '1'; a_in_oper <= "10";
       data_pointer_operation <= "01"; data_oper <= "0011";
       d_in_oper <= "10";
```

Ovime smo pokazali dekodiranje svih instrukcija osim instrukcije *nop*, čije samo ime kaže kako ne radi nikakvu operaciju (*no operation*). Cjelokupna faza dekodiranja instrukcija se može pronaći u poglavlju *9. Dodatak A : VHDL kod procesora*.

4.1.7. Izvođenje instrukcija

Do sada smo objasnili tri od četiri faze naše protočne strukture: *dohvat instrukcijskog spremnika iz programske memorije*, *dohvat instrukcije iz spremnika* i *dekodiranje instrukcija*. Sljedeća, ujedno i posljednja faza protočne strukture jest faza izvođenja instrukcija. U ovoj se fazi, ovisno o kontrolnim signalima kojima su pridružene vrijednosti tijekom faze dekodiranja, nad određenim podacima vrše aritmetičke ili logičke operacije, operacije posmaka ili jednostavne operacije prebacivanja podataka. Kao što je već rečeno u prethodnom poglavlju, postoji ukupno 5 mogućih odredišta instrukcija, s time da neke instrukcije koriste samo

jedan od njih, a neke dva ili više. Pregled resursa počinjemo s pokazivačima na podatkovni stog. Vidjeli smo da su nam potrebna dva pokazivača, koja su implementirana preko registara. Nadalje, svaki od tih registara ima ulazni multipleksor, koji ovisno o kontrolnom signalu, prema registru propušta neku vrijednost, koja se u taj registar upisuje na rastući brid takta. Budući da tijekom jedne instrukcije nad oba pokazivača provodimo istu operaciju (umanjivanje ili uvećavanje), možemo koristiti isti kontrolni signal za oba pokazivača. Na sljedećoj slici je prikazana blok shema mehanizma za ažuriranje pokazivača, s registrima i pripadnim multipleksorima.

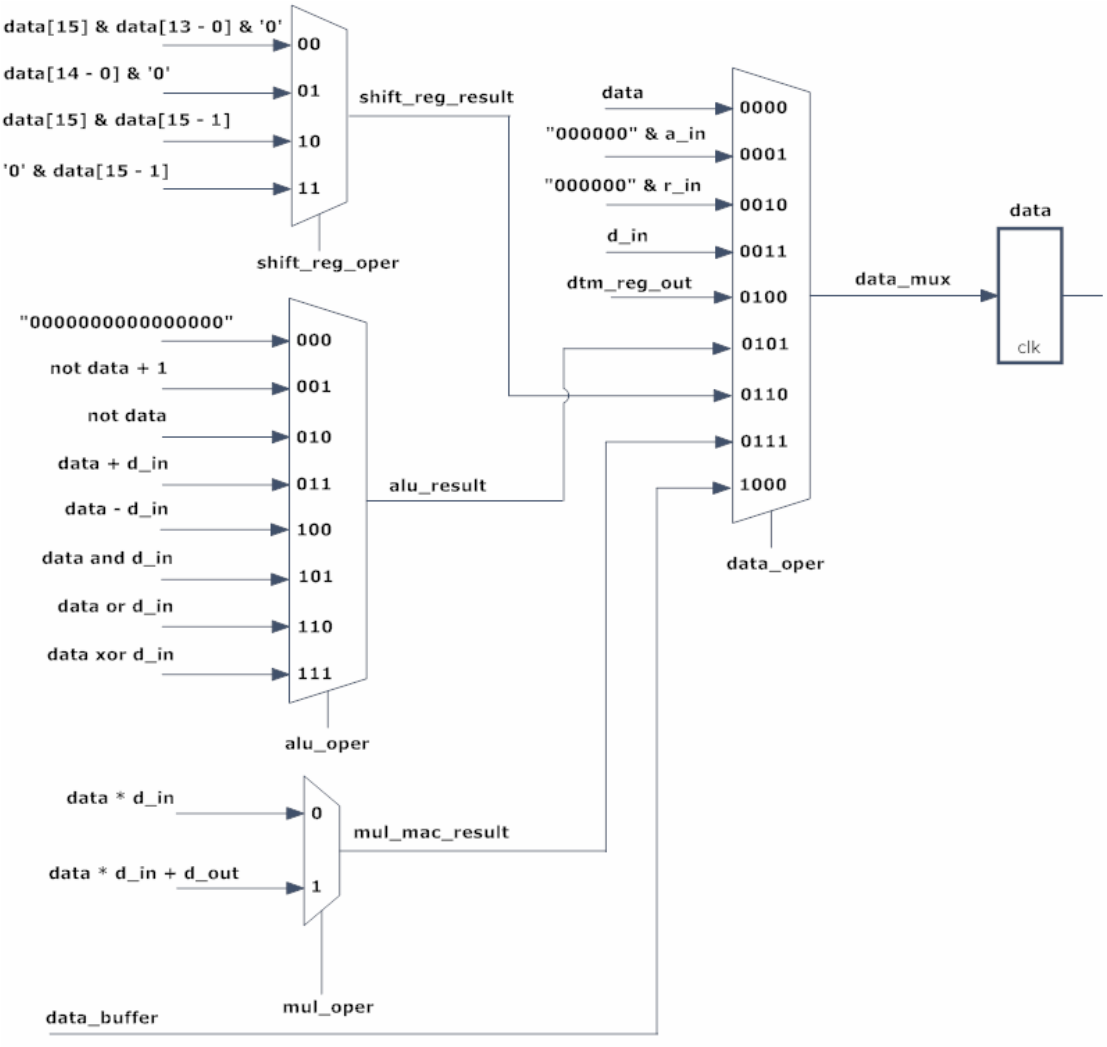


Slika 21. Multipleksori i registri pokazivača na podatkovni stog

Način djelovanja multipleksora i registara vezanih uz pokazivače na podatkovni stog prikazan je u sljedećem odsječku koda (zbog jednostavnosti prikazan je samo jedan multipleksor i jedan registar pokazivača):

```
with dw_en select ds_addr <= d_pointer when "01", d_pointer1 when others;

process(data_pointer_operation, d_pointer)
begin
	case data_pointer_operation is
		when "00" => data_pointer_mux <= d_pointer;
		when "01" => data_pointer_mux <= d_pointer + 1;
		when "10" => data_pointer_mux <= d_pointer - 1;
		when others => null;
	end case;
end process;
process(clk)
begin
	if rising_edge(clk) then
		if reset = '1' then
			d_pointer <= (others => '0');
		else
			d_pointer <= data_pointer_mux;
		end if;
	end if;
end process;
```

Budući da smo ažurirali pokazivače na stog, potrebno je na stog staviti ili s njega uzeti podatak, ovisno o izvođenoj instrukciji. Kako je već prije spomenuto, iznad podatkovnog stoga izvedenog distribuiranim RAM-om postoje dva registra, koja na omogućuju operacije nad dvije ili tri varijable. Stoga se na sljedećoj slici nalazi put signala do najvišeg registra, registra data. Vidimo da postoji velik broj mogućnosti ulaznih signala za ovaj registar, te su oni zbog toga podijeljeni u 4 grupe: signali koji su vezani uz instrukcije posmaka, signali vezani uz ALU instrukcije, instrukcije množila, te signali vezani uz instrukcije ažuriranja stogova. Tako smo podijelili i multipleksore, na način koji je prikazan sljedećom slikom:



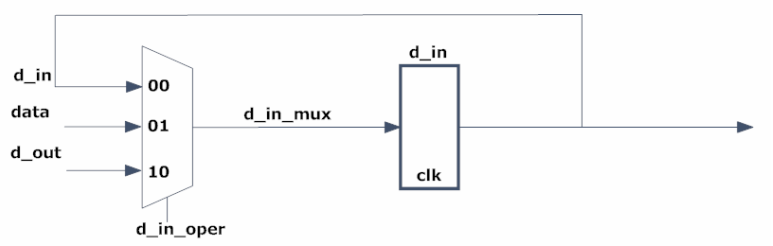
Slika 22. Put signala do najvišeg elementa podatkovnog stoga

Slično kao i u prethodnom primjeru, u nastavku slijedi odsječak koda koji opisuje rad multipleksora i odredišnog registra:

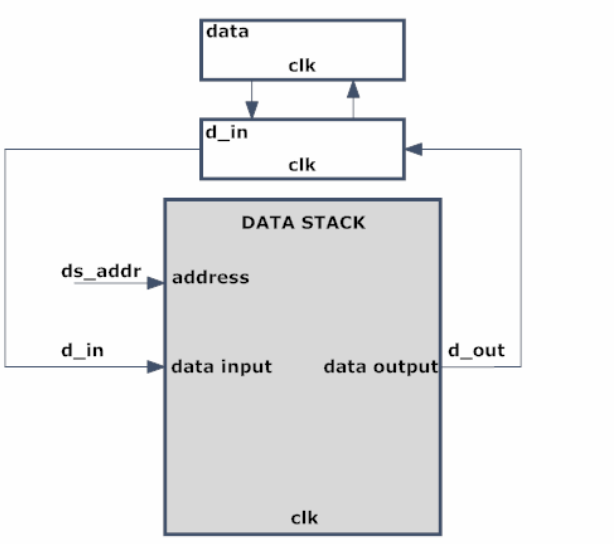
```
mul_result_long     <= data * d_in;
mul_result          <= mul_result_long(15 downto 0);
mac_result          <= mul_result + d_out;
mul_mac_result      <= mul_result when mul_oper='1' else mac_result;
-- shift register
process(shift_reg_oper, data)
begin
  case shift_reg_oper is
   when "00" =>   shift_reg_result <= data(15) & data(13 downto 0) & '0';
   when "01" =>   shift_reg_result <= data(14 downto 0) & '0';
   when "10" =>   shift_reg_result <= data(15) & data(15 downto 1);
   when "11" =>   shift_reg_result <= '0' & data(15 downto 1);
   when others => shift_reg_result <= (others => '0');
  end case;
end process;
```

```
-- ALU
process(alu_oper, data, d_in)
begin
  case alu_oper is
   when "000" =>  alu_result <= (others => '0');
   when "001" =>  alu_result <= not data + 1;
   when "010" =>  alu_result <= not data;
   when "011" =>  alu_result <= data + d_in;
   when "100" =>  alu_result <= data - d_in;
   when "101" =>  alu_result <= data and d_in;
   when "110" =>  alu_result <= data or d_in;
   when "111" =>  alu_result <= data xor d_in;
   when others => alu_result <= (others => '0');
  end case;
end process;
-- data input multiplexer
process(data_oper, data, data_buffer, a_in, r_in, d_in, dtm_reg_out,
             alu_result, shift_reg_result, mul_mac_result)
begin
  case data_oper is
   when "0000" => data_mux <= data;
   when "0001" => data_mux <= "000000" & a_in;
   when "0010" => data_mux <= "000000" & r_in;
   when "0011" => data_mux <= d_in;
   when "0100" => data_mux <= dtm_reg_out;
   when "0101" => data_mux <= alu_result;
   when "0110" => data_mux <= shift_reg_result;
   when "0111" => data_mux <= mul_mac_result;
   when "1000" => data_mux <= data_buffer;
   when others => data_mux <= (others => '0');
  end case;
end process;
-- data register
process(clk)
begin
   if rising_edge(clk) then
      if reset = '1' then
         data <= (others => '0');
      else
         data <= data_mux;
      end if;
   end if;
end process;
```

Na kraju, kako bismo završili cjelinu podatkovnog stoga, moramo odrediti put signala za registar koji se nalazi neposredno iznad stoga (registar *d_in*). Taj put je prikazan na slici 23. Njegov je izlaz spojen izravno na ulaz podatkovnog stoga (hardverske implementacije distribuiranim RAM-ovima) kao što se vidi na slici 24, dok se na njegov ulaz spaja multipleksor, koji ovisno o instrukciji određuje koji signal se upisuje u registar na sljedeći rastući brid signala takta.



Slika 23. Put signala za registar *d_in*

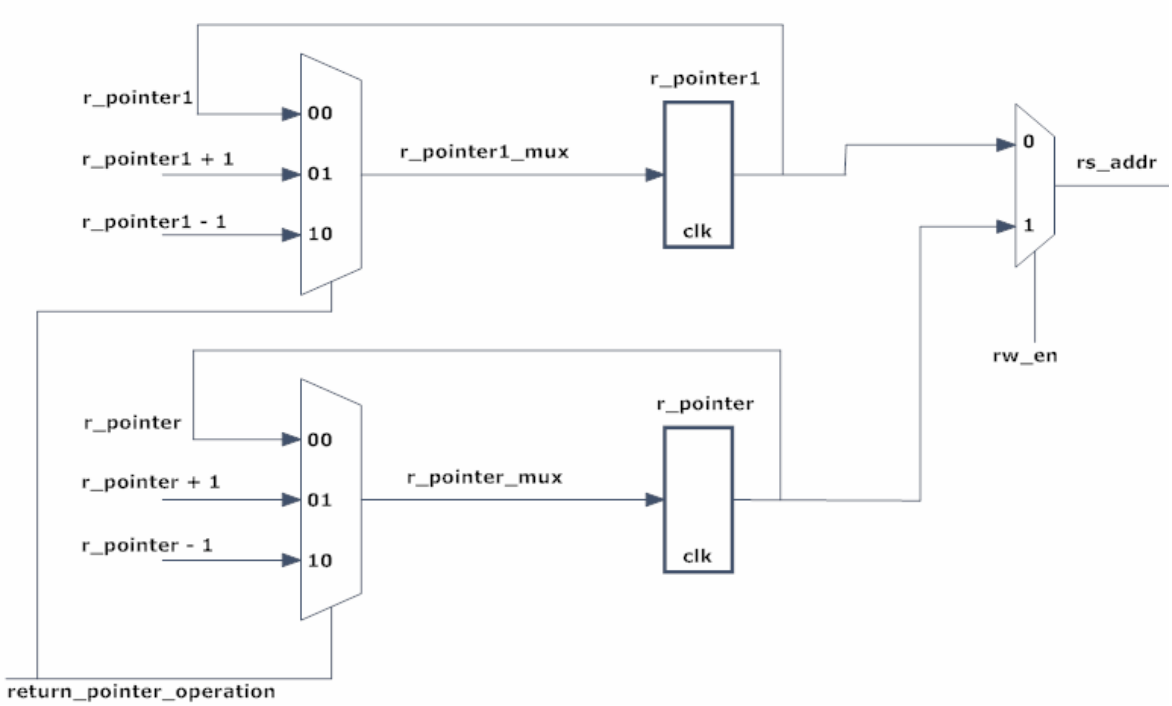


Slika 24. Podatkovni stog s pripadnim registrima

Opis rada multipleksora i registra *d_in* prikazan je u sljedećem odsječku koda:

```
process(d_in_oper, d_in, data, d_out)
begin
	case d_in_oper is
		when "00" => d_in_mux <= d_in;
		when "01" => d_in_mux <= data;
		when "10" => d_in_mux <= d_out;
		when others => null;
	end case;
end process;
process(clk)
begin
	if rising_edge(clk) then
		if reset = '1' then
			d_in <= (others => '0');
		else
			d_in <= d_in_mux;
		end if;
	end if;
end process;
```

Nakon što smo opisali podatkovni stog, slijede opisi ostalih stogova, koje započinjemo opisom stoga povratne adrese, ili jednostavnije, povratnog stoga. Ovaj stog je poprilično jednostavniji od podatkovnog, zbog toga što ima mali broj mogućih ulaznih signala, te samo jedan registar iznad stoga izvedenog distribuiranim RAM-ovima. Kod ovog stoga, kao i kod ostalih imamo dva pokazivača koja istovremeno umanjujemo ili uvećavamo, te multipleksor koji ih provodi na adresni ulaz stoga (slika 25).

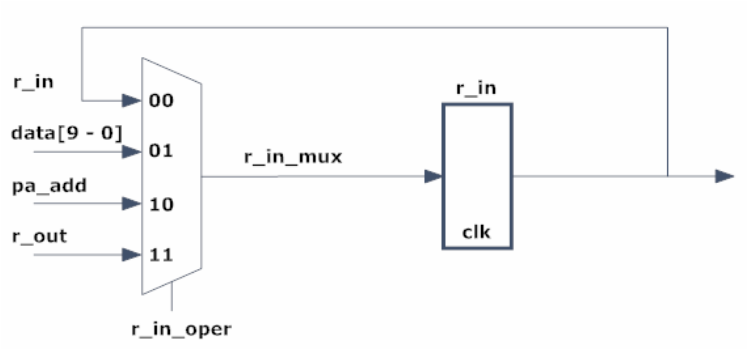


Slika 25. Pokazivači na vrh adresnog stoga i pripadni multipleksori

U sljedećem odsječku koda prikazan je način djelovanja multipleksora i registara vezanih uz pokazivače na povratni stog:

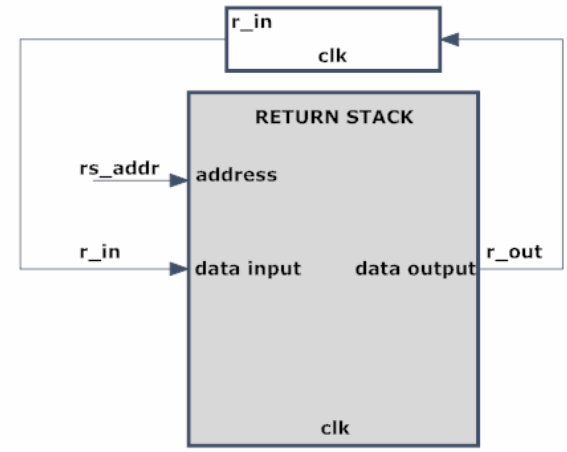
```
with rw_en select rs_addr <= r_pointer when "01", r_pointer1 when others;

process(return_pointer_operation, r_pointer)
begin
	case return_pointer_operation is
		when "00" => r_pointer_mux <= r_pointer;
		when "01" => r_pointer_mux <= r_pointer + 1;
		when "10" => r_pointer_mux <= r_pointer - 1;
		when others => null;
	end case;
end process;
process(clk)
begin
	if rising_edge(clk) then
		if reset = '1' then
			r_pointer <= (others => '0');
		else
			r_pointer <= r_pointer_mux;
		end if;
	end if;
end process;
```

Kako smo već spomenuli, iznad povratnog stoga je samo jedan registar, te je put signala za taj registar uz pripadni odsječak koda prikazan u nastavku:



Slika 26. Multipleksor i registar koji predstavlja vrh povratnog stoga

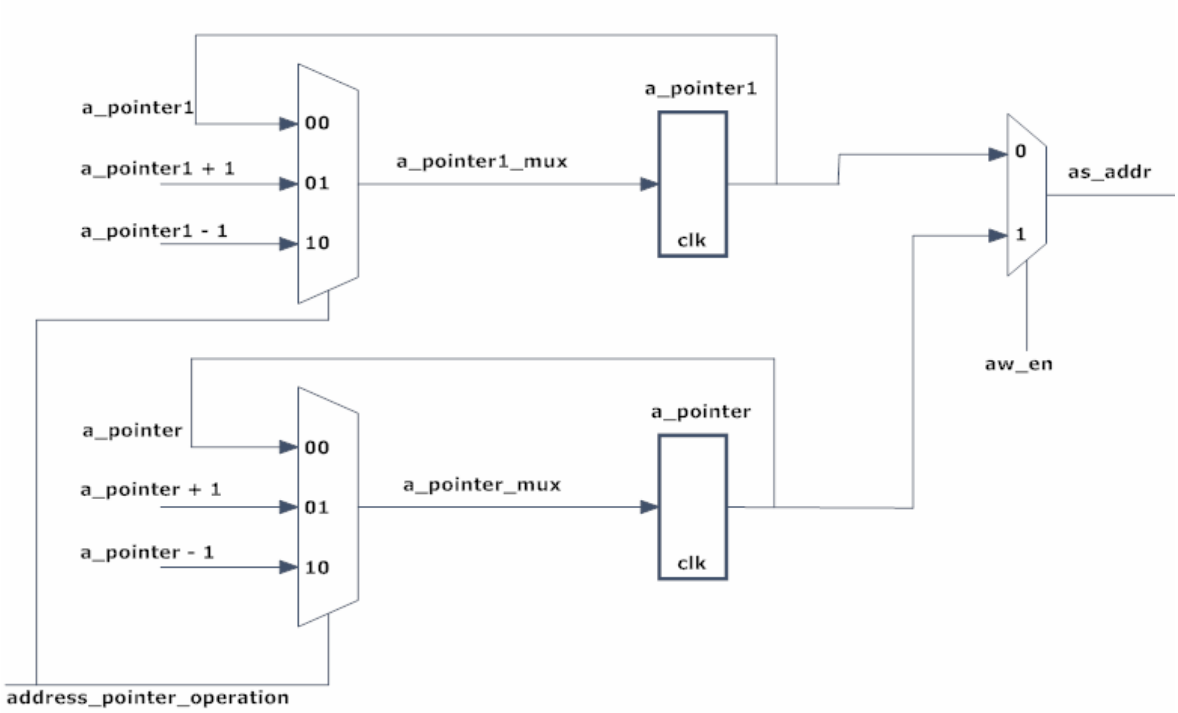
```
process(r_in_oper, r_in, data, pa_add, r_out)
begin
	case r_in_oper is
		when "00" => r_in_mux <= r_in;
		when "01" => r_in_mux <= data(9 downto 0);
		when "10" => r_in_mux <= pa_add;
		when "11" => r_in_mux <= r_out;
		when others => null;
	end case;
end process;
process(clk)
begin
	if rising_edge(clk) then
		if reset = '1' then
			r_in <= (others => '0');
		else
			r_in <= r_in_mux;
		end if;
	end if;
end process;
```



Slika 27. Povratni stog s pripadnim registrom

Posljednji stog koji nam je preostao jest adresni stog. Kako već znamo, iznad adresnog stoga postoji registar koji služi za pristup podatkovnoj memoriji. Stoga je

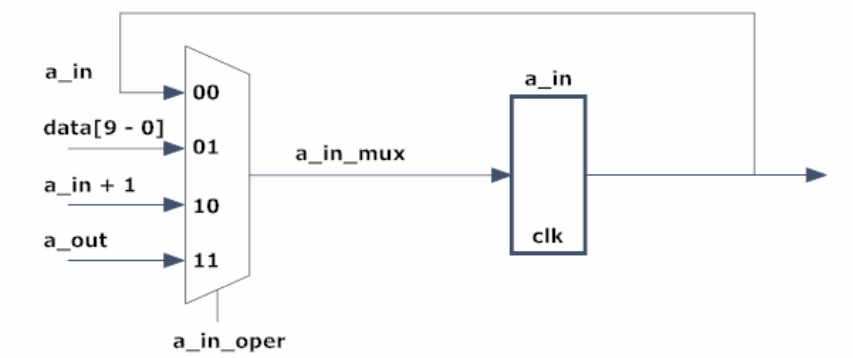
ovaj stog vrlo sličan povratnom, uz malu razliku mogućih ulaza u najviši element stoga. Dakako, i za ovaj stog su nam potrebna dva pokazivača:



Slika 28. Pokazivači na vrh adresnog stoga s pripadnim multipleksorima

```
with aw_en select as_addr <= a_pointer when "01", a_pointer1 when others;

process(address_pointer_operation, a_pointer)
begin
	case address_pointer_operation is
		when "00" => a_pointer_mux <= a_pointer;
		when "01" => a_pointer_mux <= a_pointer + 1;
		when "10" => a_pointer_mux <= a_pointer - 1;
		when others => null;
	end case;
end process;
process(clk)
begin
	if rising_edge(clk) then
		if reset = '1' then
			a_pointer <= (others => '0');
		else
			a_pointer <= a_pointer_mux;
		end if;
	end if;
end process;
```

Na sljedećoj je slici prikazan put signala prema registru na vrhu adresnog stoga, s pripadnim multipleksorom:

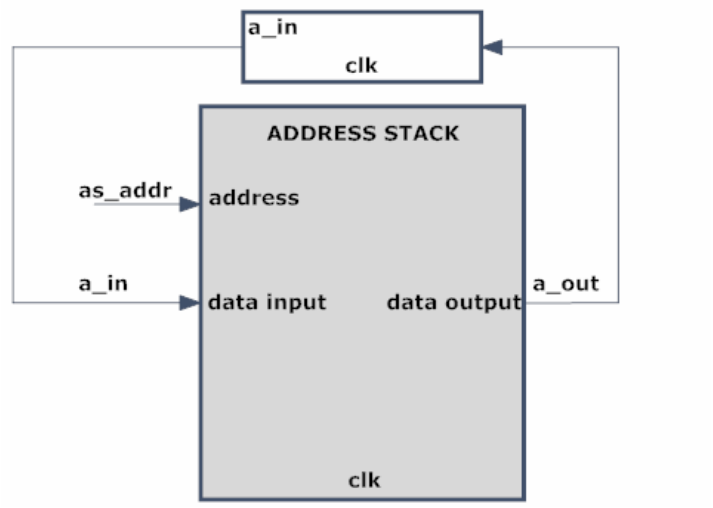


Slika 29. Adresni registar, vrh adresnog stoga

Opis ovog dijela sklopa u VHDL-u je prikazan u sljedećem odsječku koda:

```
process(a_in_oper, a_in, data, pa_add, a_out)
begin
	case a_in_oper is
		when "00" => a_in_mux <= a_in;
		when "01" => a_in_mux <= data(9 downto 0);
		when "10" => a_in_mux <= a_in + 1;
		when "11" => a_in_mux <= a_out;
		when others => null;
	end case;
end process;
process(clk)
begin
	if rising_edge(clk) then
		if reset = '1' then
			a_in <= (others => '0');
		else
			a_in <= a_in_mux;
		end if;
	end if;
end process;
```

Kao i kod ostalih stogova, registar *a_in* predstavlja vrh adresnog stoga, te preko njega podaci idu u i iz adresnog stoga, kao što je prikazano na slici 30.

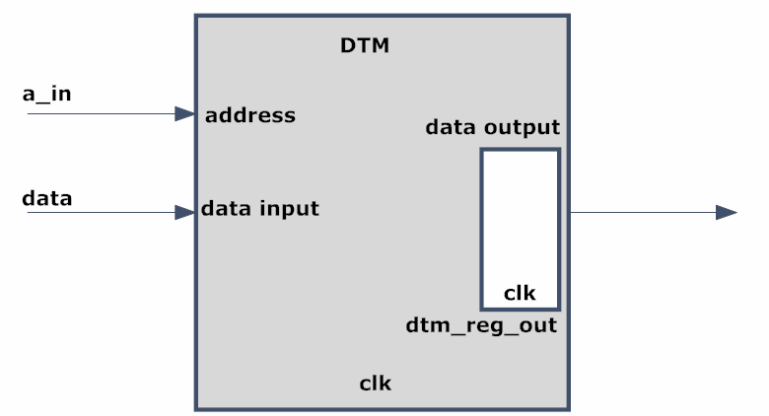


Slika 30. Adresni stog s pripadnim registrom

Iz već prethodno navedenih razloga, procesor koristi i podatkovnu memoriju izvedenu blok RAM-om, te je ta memorija opisana slijednim opisom, kako slijedi u sljedećem odsječku koda:

```
process(clk)
begin
	if rising_edge(clk) then
		if dtm_en = '1' then
			if dtm_we = '1' then
				DTM(conv_integer(a_in)) <= data;
			else
				dtm_reg_out <= DTM(conv_integer(a_in));
			end if;
		end if;
	end if;
end process;
```

Ovdje vidimo kako na izlazu iz blok RAM-a postoji izlazni registar, kao što je to slučaj kod programske memorije. Zbog ovog registra postoji ograničenje vezano za korištenje podatkovne memorije, a više će riječi o tome biti u sljedećem poglavlju kod funkcijske simulacije instrukcija koje koriste podatkovnu memoriju.



Slika 31. Podatkovna memorija izvedena blok RAM-om

Ovime je završen pregled svih resursa procesora, te sada sve instrukcije imaju svoje 4 faze protočne strukture. U prethodnom smo poglavlju vidjeli kako se pri dekodiranju instrukcija signalima daju određene vrijednosti, dok smo u ovom poglavlju na temelju tih vrijednosti različite signale pridruživali odredišnim resursima, odnosno registrima na vrhu stogova, stogovima i podatkovnoj memoriji. Cjelokupni programski kod može se pronaći u poglavlju *8. Privitak A – VHDL kod procesora*.

4.2. Funkcijska simulacija

Nakon što je dizajn gotov, potrebno je provesti funkcijsku simulaciju kako bi se uvjerili da radi ispravno. U ovom slučaju možemo zadati sadržaj programske memorije te na taj način ispitati ispravnost rada procesora. Za funkcijsku simulaciju potrebna nam je nova izvorna datoteka, no ovaj put u nju ne pišemo kôd nego ju formiramo kao tzv. *test-bench* datoteku, te ju povežemo s originalnom datotekom (u kojoj je napisan kôd za procesor). Nova datoteka mora imati naziv *<ime_vhdl_datoteke>_tb.vhd* (*Project => New Source => VHDL Test Bench*). Nakon povezivanja s originalnom datotekom, programski alat će sam dodati potrebne stvari u datoteku za simulaciju. Ovo se prvenstveno odnosi na instanciranje komponente koju testiramo (u našem slučaju dizajn procesora), koji dobiva labelu *uut* (*eng. unit under test*) te nam samo preostaje dodati definicije ulaznih signala u sklop. Kod ovog sklopa rad ovisi o programskoj memoriji, koja se, kako znamo, zadaje putem datoteke. Tako da ovdje moramo definirati i pridružiti vrijednosti samo dvama signalima, signalu takta (*clk*) i signalu za reset.

Nakon što je datoteka za simulaciju kreirana i dopunjena, simulacija se može pokrenuti na dva načina. Jedan je način automatsko pokretanje simulatora (*ModelSim PE*), pri čemu Xilinx ISE sam pokreće simulator i prikazuje prozor s valnim oblicima. Ovdje imamo ograničenje u vidu prikazanih signala, jer simulator prikazuje samo ulazne i izlazne signale iz implementiranog dizajna. Zbog toga se češće koristi ručno pokretanje simulatora, jer tako imamo mogućnost podešavanja prikaza, te odabira signala za prikaz. Naredbe za podešavanje se mogu unositi ručno u prozor simulatora, ili putem komandnih datoteka. Pristup podešavanja prikaza putem komandnih datoteka je češće korišten jer omogućava ponovljivost i jednostavnu izmjenu. Komandne datoteke se mogu pisati u okviru programskog alata ModelSim, a imaju ekstenziju *.do* (*File => New => Source => Do*). Prije pisanja komandne datoteke potrebno se pozicionirati u mapu projekta u kojoj se nalaze dizajn i datoteka za simulaciju. To se radi na način da se u prozor simulatora upiše:

```
ModelSim> cd c:/xilinx92i/<mapa_projekta>/
```

Nakon toga možemo početi pisanje komandne datoteke, pri čemu nam može pomoći priručnik za ModelSim[14], a koji se može pronaći u instalacijskoj mapi programskog okruženja ModelSim.

4.2.1. Komandna datoteka

Kod pisanja komandne datoteke potrebno je pratiti sintaksu pisanja *.do* datoteka [14]. U komandnoj datoteci potrebno je kreirati biblioteku u kojoj se će se nalaziti svi potrebni modeli. Nadalje, potrebno je prevesti sve VHDL modele u oblik koji se koristi u simulatoru, te pokretanje simulatora. Kod prevođenja je bitno da se najprije prevedu modeli koji su najniže u hijerarhiji. Slijedi popis i objašnjenje korištenih naredbi:

```
# naredbe se grupiraju po funkciji, # označava komentar
#------------------------ upravljačke naredbe --------------------------
quit -sim     # izlazi iz trenutno izvođene simulacije

vlib work     # kreiranje biblioteke gdje se nalaze svi potrebni modeli

vcom -explicit  -93 "proc.vhd"          # prevođenje VHDL modela u oblik

vcom -explicit  -93 "proc_tb.vhd"       #      koji koristi simulator
      # prvo se prevode datoteke koje su najniže u hijerarhiji

vsim -t 10ps   -lib work proc_tb_vhd    # pokretanje simulatora
      # 10 ps je vremenska rezolucija simulatora
#------------------------ dodavanje valnih oblika ------------------------
add wave            # dodaje valni oblik

-noupdate           # ne osvježava Wave prozor nakon serije add wave naredbi

-divider            # vertikalni razmak u Wave prozoru
      {ime_razmaka}

-format             # specificira format prikaza objekta
      Logic         # std_logic : '0', '1', 'U', 'X', ...
      Literal       # pravokutnik s vrijednošću objekta
      Analog

-color              # boja prikaza
      green, blue, red, ...

-expand             # proširuje složeni signal, npr. std_logic_vector
-radix              # format brojeva
      Binary, ASCII, unsigned, decimal, octal, hexadecimal, ...
-label              # alternativno ime signala
      {ime_signala}

#primjer retka za dodavanje valnog oblika
#BITNO: sve naredbe moraju biti u jednom retku
add wave -noupdate -format Logic -label {CLK} /proc_tb_vhd/clk
#-------------------------------------------------------------------------
run 400ns     # pokreće simulaciju, definira trajanje simulacije
```

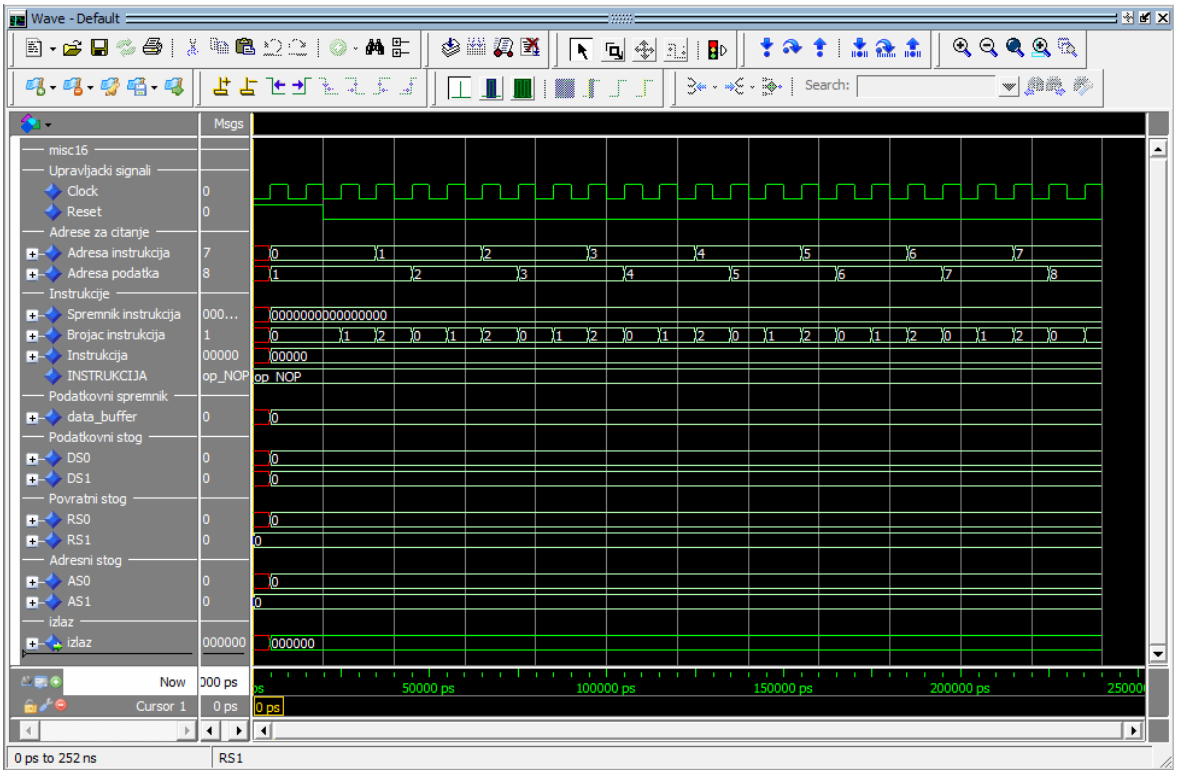
Ovisno o potrebi, postoje dodatne naredbe i mogućnosti simulatora [14], no navedene su dovoljne za potrebe simulacije rada procesora. Cijela komandna datoteka dostupna je u poglavlju *9. Privitak B*. Za potpunu verifikaciju rada procesora, potrebno je prikazati ulazne kontrolne signale (dakle signal takta *clk* i *reset*), zatim programsku i podatkovnu adresu, instrukcijski i podatkovni spremnik, te instrukciju koja se dekodira. Na kraju, potrebno je prikazati sva tri stoga, odnosno njihova dva najviša elementa. Dodatno, kako bi se pojednostavio proces verifikacije, te u skladu s mogućnostima ModelSim programskog alata, u dizajn procesora je umetnut enumeracijski tip koji označava trenutno dekodiranu instrukciju. Ovo je napravljeno kako bi se u prozoru simulatora jednostavnije identificirala instrukcija koja se dekodira, odnosno izvodi. U nastavku slijedi popis svih valnih oblika prikazanih tijekom simulacije:

```
-divider {Upravljacki signali}
-label {Clock}                  /fetch_data_tb_vhd/clk
-label {Reset}                  /fetch_data_tb_vhd/reset

-divider {Adrese za citanje}
-label {Adresa instrukcija}             /fetch_data_tb_vhd/uut/prog_addr
-label {Adresa podatka}                 /fetch_data_tb_vhd/uut/data_addr

-divider {Instrukcije}
-label {Spremnik instrukcija}   /fetch_data_tb_vhd/uut/instr_buffer
-label {Brojac instrukcija}     /fetch_data_tb_vhd/uut/counter
-label {Instrukcija}            /fetch_data_tb_vhd/uut/code
-label {INSTRUKCIJA}            /fetch_data_tb_vhd/uut/instrukcija_enum

-divider {Podatkovni spremnik}
-label {data_buffer}            /fetch_data_tb_vhd/uut/data_buffer

-divider {Podatkovni stog}
-label {DS0}                    /fetch_data_tb_vhd/uut/data
-label {DS1}                    /fetch_data_tb_vhd/uut/d_in

-divider {Povratni stog}
-label {RS0}                    /fetch_data_tb_vhd/uut/r_in
-label {RS1}                    /fetch_data_tb_vhd/uut/r_out

-divider {Adresni stog}
-label {AS0}                    /fetch_data_tb_vhd/uut/a_in
-label {AS1}                    /fetch_data_tb_vhd/uut/a_out
```

Kao što vidimo, zbog jednostavnosti su izostavljene naredbe vezane uz format prikaza, te su prikazani samo nazivi valnih oblika i signali kojima su oni pridruženi. Pokretanje datoteke se obavlja naredbom *do* u prozoru simulatora:

```
ModelSim> do <ime_komandne_datoteke>.do
```

Nakon pokretanja komandne datoteke, ovisno o sadržaju programske memorije, dobijemo sljedeći prikaz (ovdje je zbog jednostavnosti programska memorija napunjena logičkim nulama što odgovara instrukciji *nop*):



Slika 32. Izgled prozora za prikaz valnih oblika, ModelSim simulator

4.2.2. Simulacija instrukcija procesora

Provjera rada procesora podrazumijeva simulaciju svake instrukcije, i provjeru njenog ispravnog rada te utjecaja na resurse sustava, odnosno stogove i memorije. Budući da se programska memorija inicijalizira pomoću tekstualne datoteke, editiranjem te datoteke možemo zadati instrukcije za rad procesora. Programsku memoriju smo inicijalizirali tako da ima 1024 redaka po 16 bitova, te smo funkcijom učitavali te retke u programsku memoriju. Stoga svaki redak u programskoj memoriji mora imati minimalno 16 bitova, nakon čega mogu uslijediti komentar i slično, te isto tako mora biti minimalno 1024 retka. Višak znakova u retku ili višak redaka funkcija zanemaruje, pošto koristi funkciju iz biblioteke TEXTIO, koja čita samo zadani broj znakova iz datoteke, te tada prelazi u sljedeći redak, i tako u petlji, zadani broj puta. U nastavku slijede simulacije rada implementiranih instrukcija, s pripadajućim odsječcima programske memorije. Kod

nekih instrukcija, kao što ćemo vidjeti postoje određena ograničenja vezana uz mjesto te instrukcije u instrukcijskom spremniku i slično.

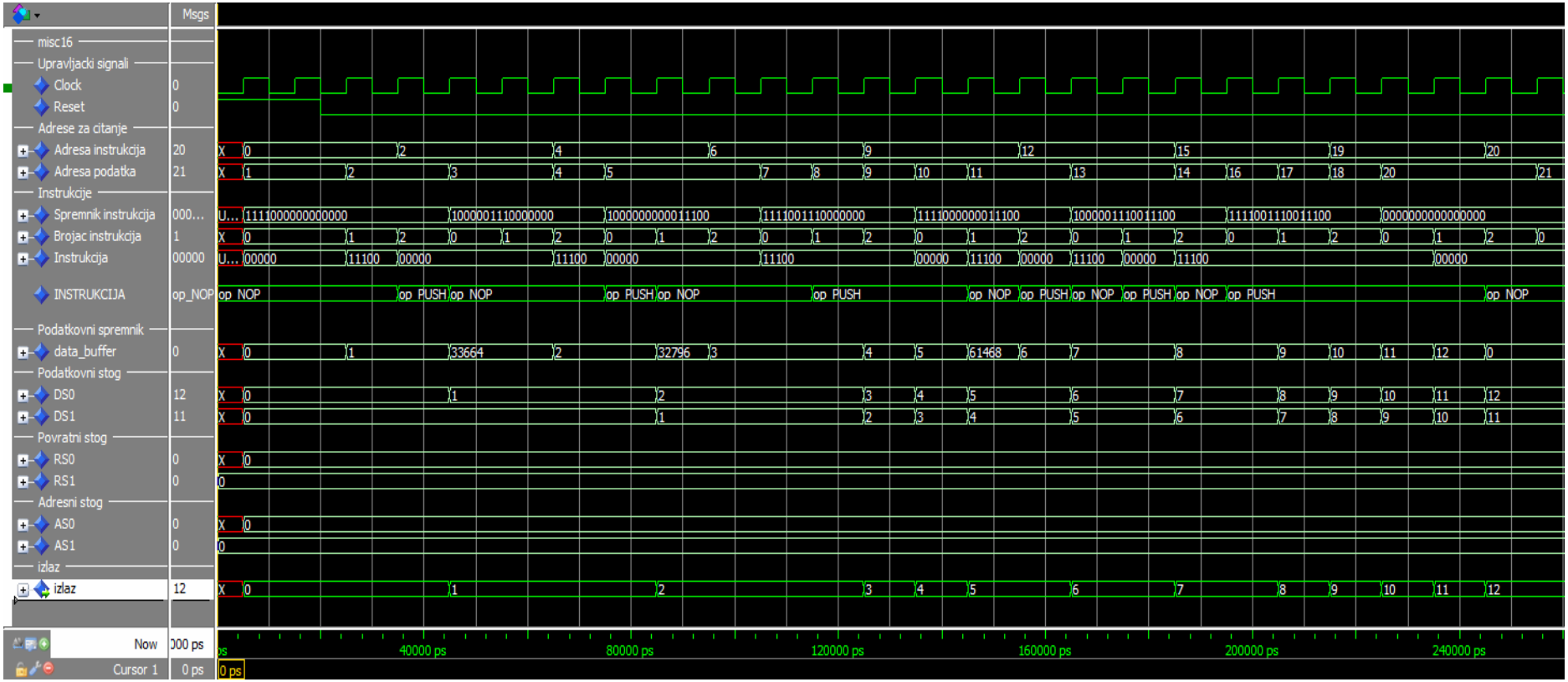
Simulacija instrukcija vezanih uz podatkovni stog

Za početak ćemo simulirati rad instrukcija koje čitaju podatke iz programske memorije. Budući da su s pogleda hardverske izvedbe sve, takozvane podatkovne, instrukcije jednake, dovoljno je samo testirati rad jedne od tih instrukcija, a ovdje će to biti *push* instrukcija. Ovime ćemo provjeriti i rade li adresni generatori kako treba, tako da variramo broj podatkovnih instrukcija u pojedinom retku memorije. U nastavku slijedi sadržaj programske memorije (tekstualna datoteka *misc16_PGM.txt*):

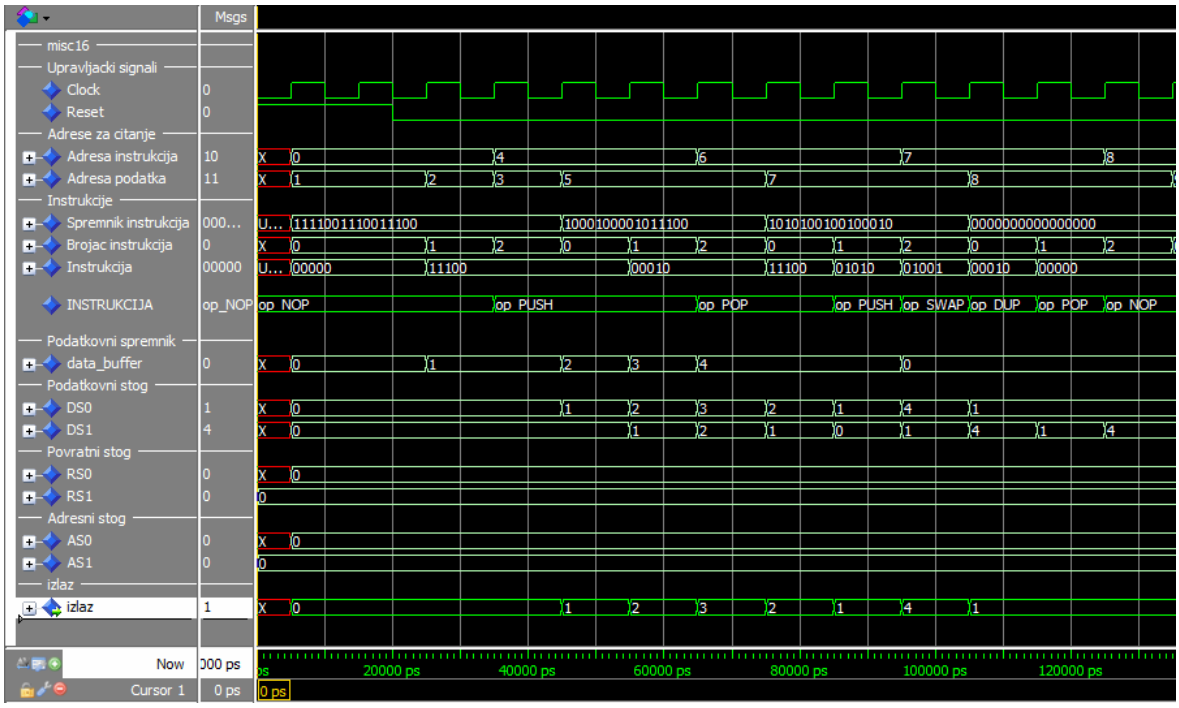
```
1111000000000000    PUSH NOP NOP
0000000000000001    1
1000001110000000    NOP PUSH NOP
0000000000000010    2
1000000000011100    NOP NOP PUSH
0000000000000011    3
1111001110000000    PUSH PUSH NOP
0000000000000100    4
0000000000000101    5
1111000000011100    PUSH NOP PUSH
0000000000000110    6
0000000000000111    7
1000001110011100    NOP PUSH PUSH
0000000000001000    8
0000000000001001    9
1111001110011100    PUSH PUSH PUSH
0000000000001010    10
0000000000001011    11
0000000000001100    12
0000000000000000
```

Na slici 33. vidimo moguće položaje podatkovne instrukcije unutar jednog retka programske memorije. Primjećujemo ispravno punjenje podatkovnog stoga neovisno o položaju *push* instrukcije, te ispravno generiranje adresa instrukcija ovisno o broju podatkovnih instrukcije, i ispravno generiranje podatkovnih adresa za svaku podatkovnu instrukciju. Na sličan ćemo način testirati rad ostalih instrukcija vezanih samo uz podatkovni stog, a uz instrukciju *push*, to su instrukcije *pop*, *dup* i *swap*. Rad ovih instrukcija objašnjen je u poglavlju *4.1.6. Dekodiranje instrukcija*, a simulaciju njihovog rada možemo vidjeti na slici 34.

```
1111001110011100    PUSH PUSH PUSH
0000000000000001    1
0000000000000010    2
0000000000000011    3
1000100001011100    POP POP PUSH
0000000000000100    4
1010100100100010    SWAP DUP POP
```



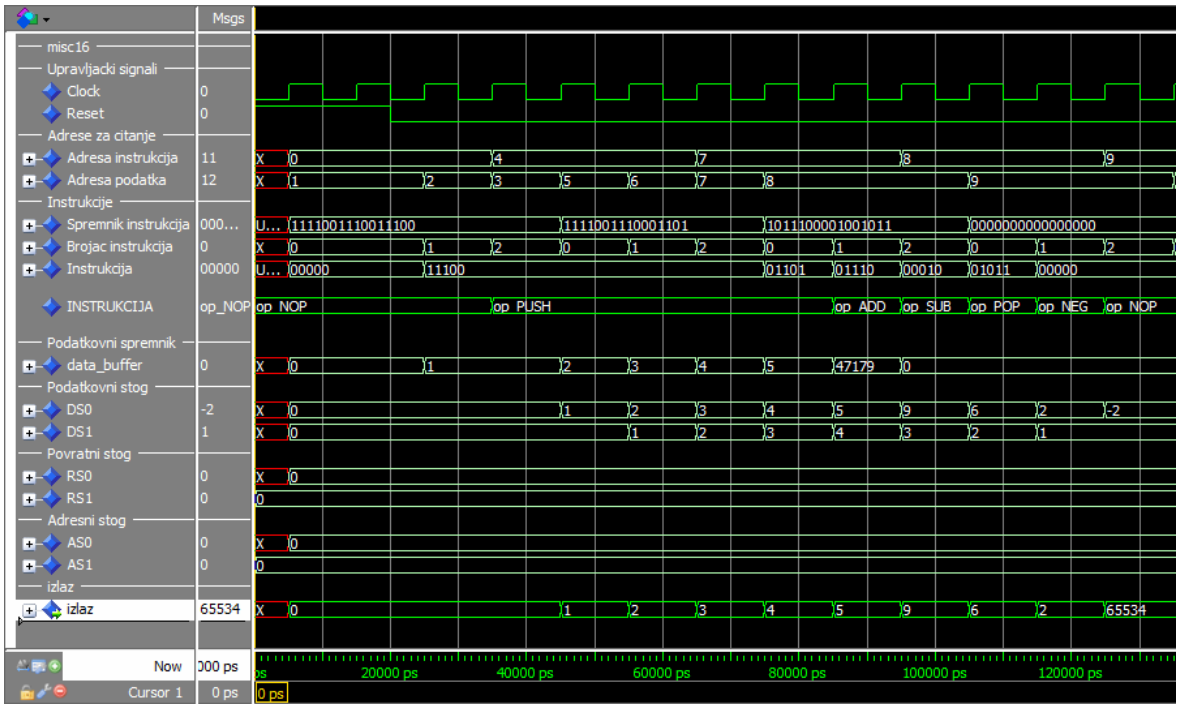
Slika 33. Simulacija rada podatkovne instrukcije i adresnih generatora



Slika 34. Rad instrukcija *push*, *pop*, *swap* i *dup*

Nastavkom simulacija instrukcija vezanih uz podatkovni stog dolazimo do aritmetičkih i logičkih instrukcija. Rezultati ovih instrukcija prolaze kroz dva multipleksora na putu do vrha podatkovnog stoga, kako je pokazano u poglavlju 4.1.7. Izvođenje instrukcija. Neke od tih instrukcija koriste najviši podatak s vrha stoga, a neke dva najviša. Budući da je za logičke instrukcije poželjno promatrati binarni oblik podatka, simulaciju ALU instrukcija ćemo podijeliti na dva dijela. Slijedi programska memorija kojom se simulira rad ALU instrukcija, te zatim slika s prikazima valnih oblika:

```
1111001110011100    PUSH PUSH PUSH
0000000000000001    1
0000000000000010    2
0000000000000011    3
1111001110001101    PUSH PUSH ADD
0000000000000100    4
0000000000000101    5
1011100001001011    SUB POP NEG
```



Slika 35. Simulacija rada instrukcija *add*, *sub* i *neg*

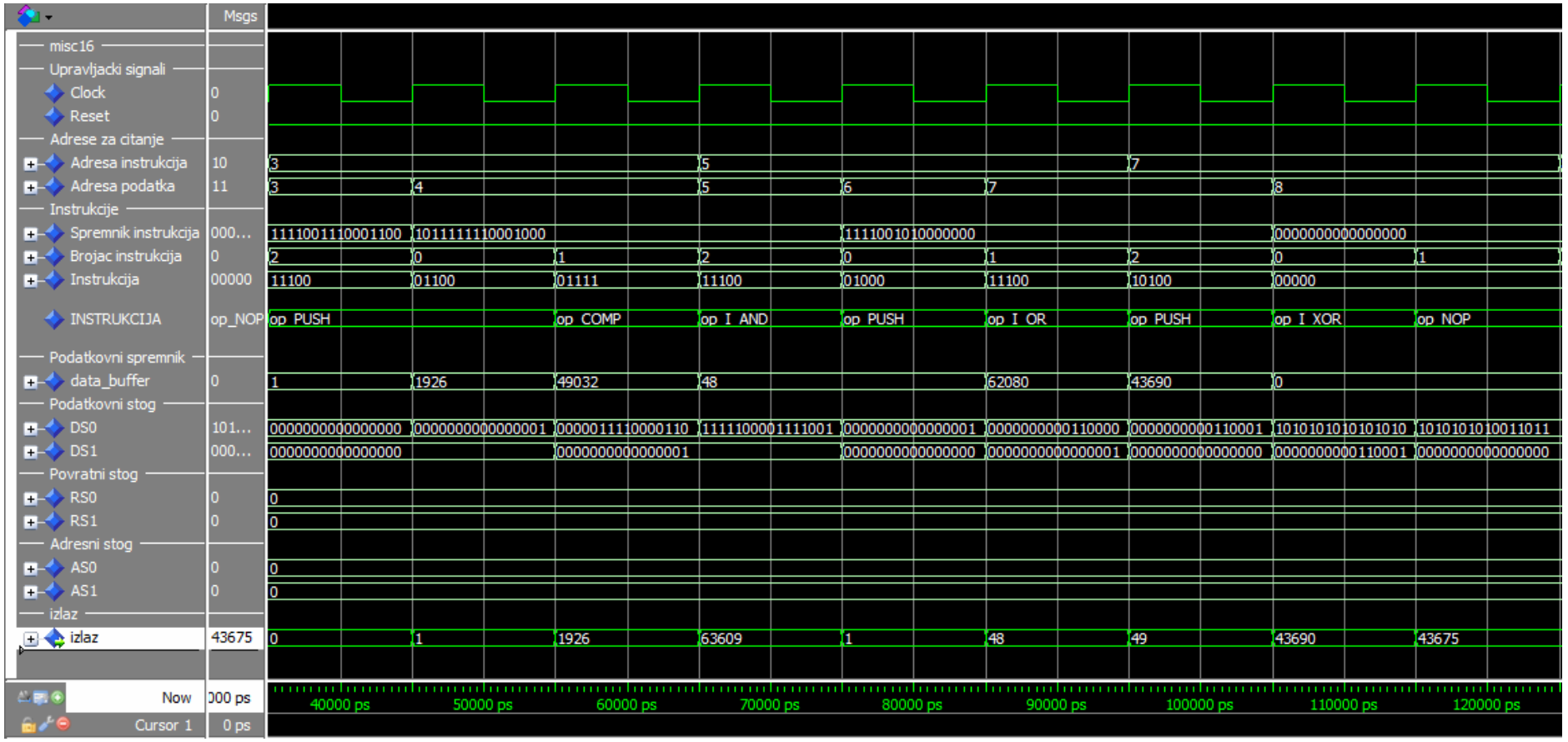
Logičke instrukcije simuliramo sljedećim odsječkom programske memorije:

```
1111001110001100    PUSH PUSH COMP
0000000000000001    1
0000011110000110
1011111110001000    I_AND PUSH I_OR
0000000000110000
1111001010000000    PUSH I_XOR NOP
1010101010101010
```

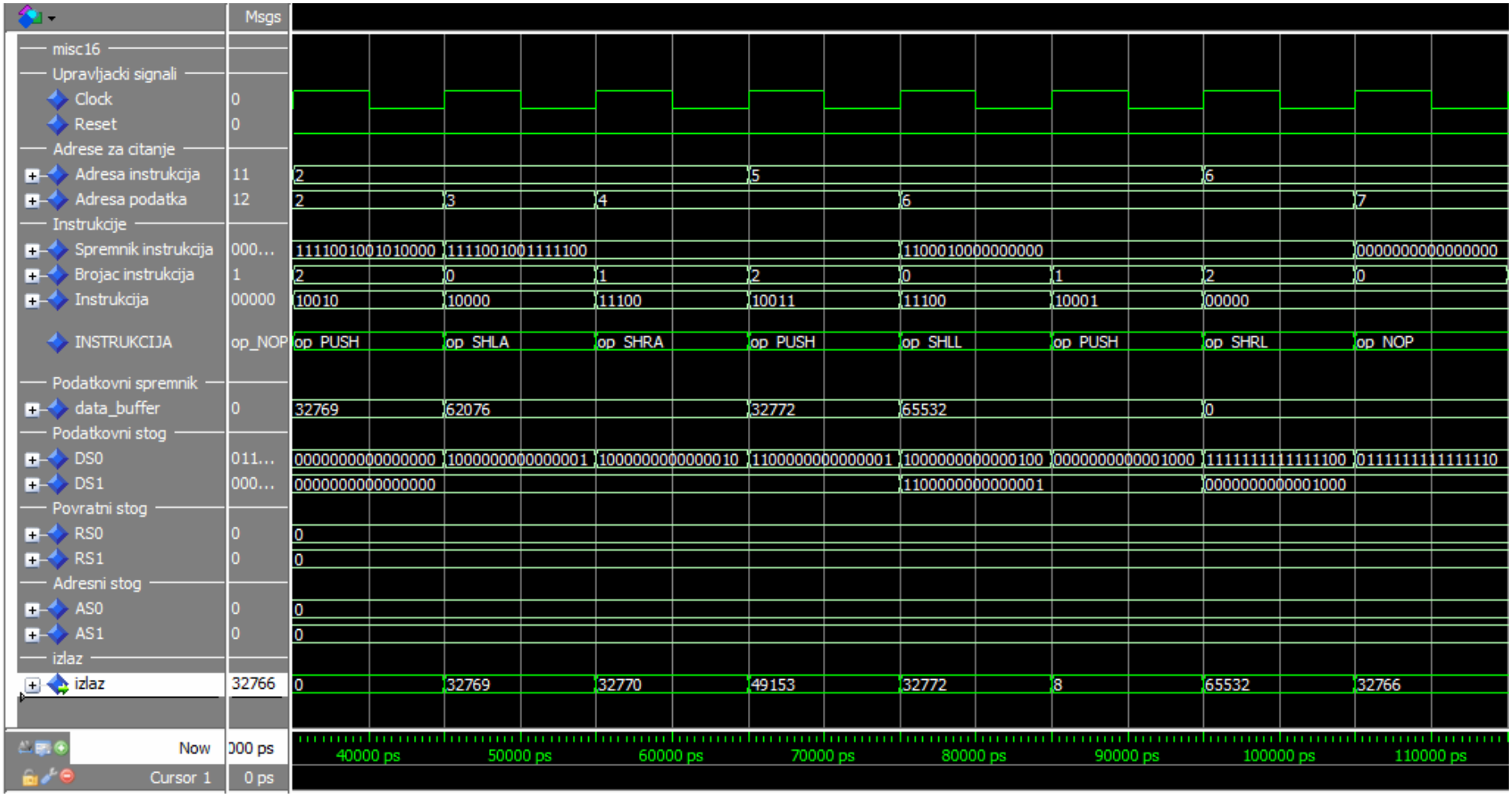
Na slici 36 možemo vidjeti simulaciju rada logičkih instrukcija. Zbog jednostavnije provjere, ovdje je format podataka podatkovnog stoga postavljen kao binarni.

Sljedeće instrukcije čiji rad je potrebno simulirati su instrukcije posmaka. Rezultati instrukcija posmaka, kao i ALU instrukcija i instrukcija množila prolaze kroz dodatni multipleksor na putu prema vrhu podatkovnog stoga. Implementirane su dvije vrste posmaka, aritmetički, koji uzima u obzir predznak signala, te logički, koji vrši osnovnu verziju posmaka s postavljanjem logičke nule na upražnjeno mjesto. U nastavku slijede odsječak programske memorije, te prikaz rezultirajućih valnih oblika (slika 36):

```
1111001001010000    PUSH SHLA SHRA
1000000000000001
1111001001111100    PUSH SHLL PUSH
1000000000000100
1111111111111100
1100010000000000    SHRL
```



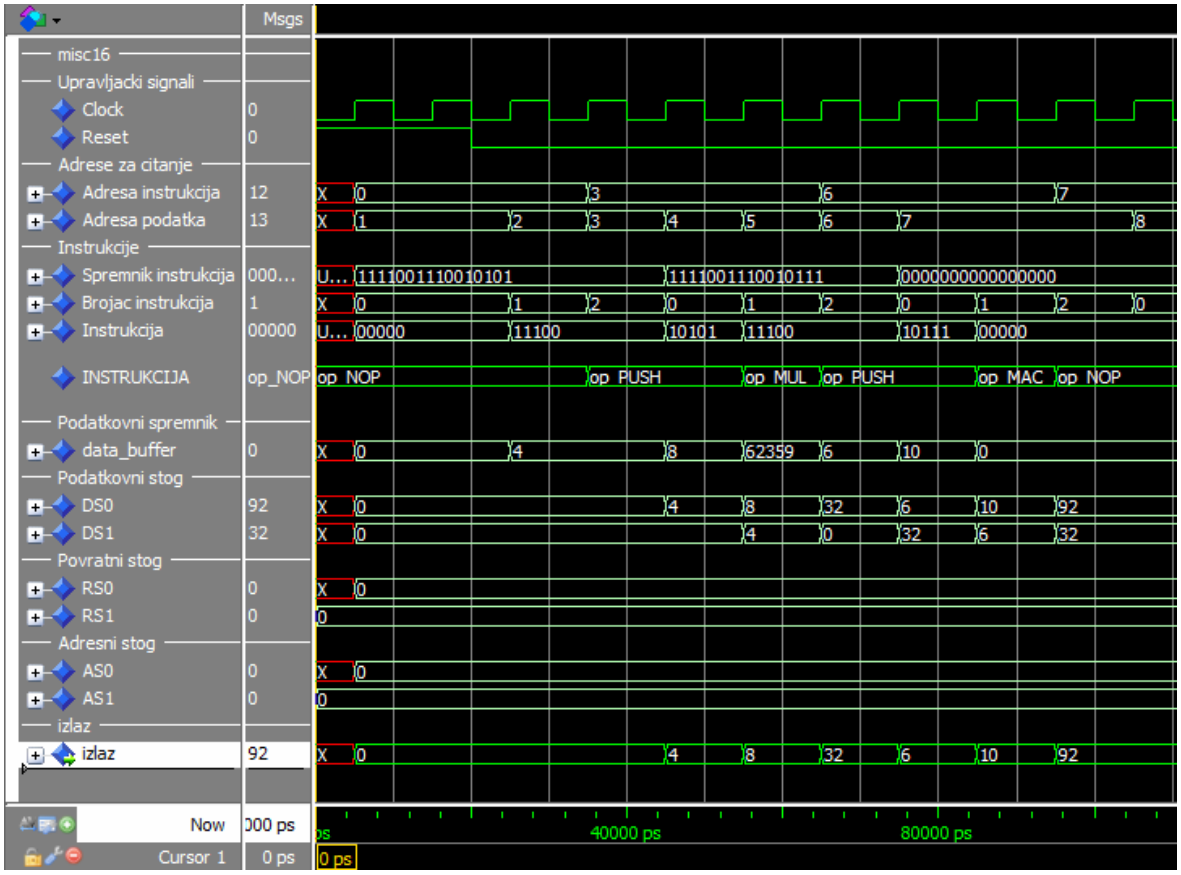
Slika 36. Simulacija logičkih instrukcija (*comp*, *i_and*, *i_or*, *i_xor*)



Slika 37. Prikaz rezultata instrukcija posmaka

Posljednji skup instrukcija vezanih uz podatkovni stog su instrukcije vezane uz množilo. Kako je već navedeno, odabrano je kombinacijsko množilo zbog jednostavnije izvedbe procesora, budući da je na taj način ostvarena funkcionalnost množenja i množenja s akumulacijom u jednom ciklusu signala takta. Drugim riječima, sinkrono registarsko brojilo bi zahtijevalo dodatan ciklus signala takta za računanje i pohranjivanje rezultata na podatkovni stog. Iako je takva izvedba množila brža, pokazalo se kroz postupak sinteze (poglavlje *4.3. Sinteza korištenjem XST alata*) kako su razlike brzine sklopa u ova dva slučaja minimalne, te je stoga bolja opcija imati jednociklusno množenje budući da je time očuvana protočna struktura. Programski odsječak i prikaz valnih oblika za funkcije množila prikazane su u nastavku:

```
1111001110010101    PUSH PUSH MUL
0000000000000100    4
0000000000001000    8
1111001110010111    PUSH PUSH MAC
0000000000000110    6
0000000000001010    10
```



Slika 38. Simulacija instrukcija množila (*mul* i *mac*)

Možemo primijetiti kako ovdje kao drugi najviši element stoga ostaje podatak koji je iskorišten za akumulaciju. Ovo je neizbježna posljedica načina rada stoga čiji se pokazivači uvijek uvećavaju ili umanjuju u koracima od 1. Ovime smo zaključili instrukcije vezane uz podatkovni stog, a slijedi simulacija rada instrukcija vezanih uz povratni stog.

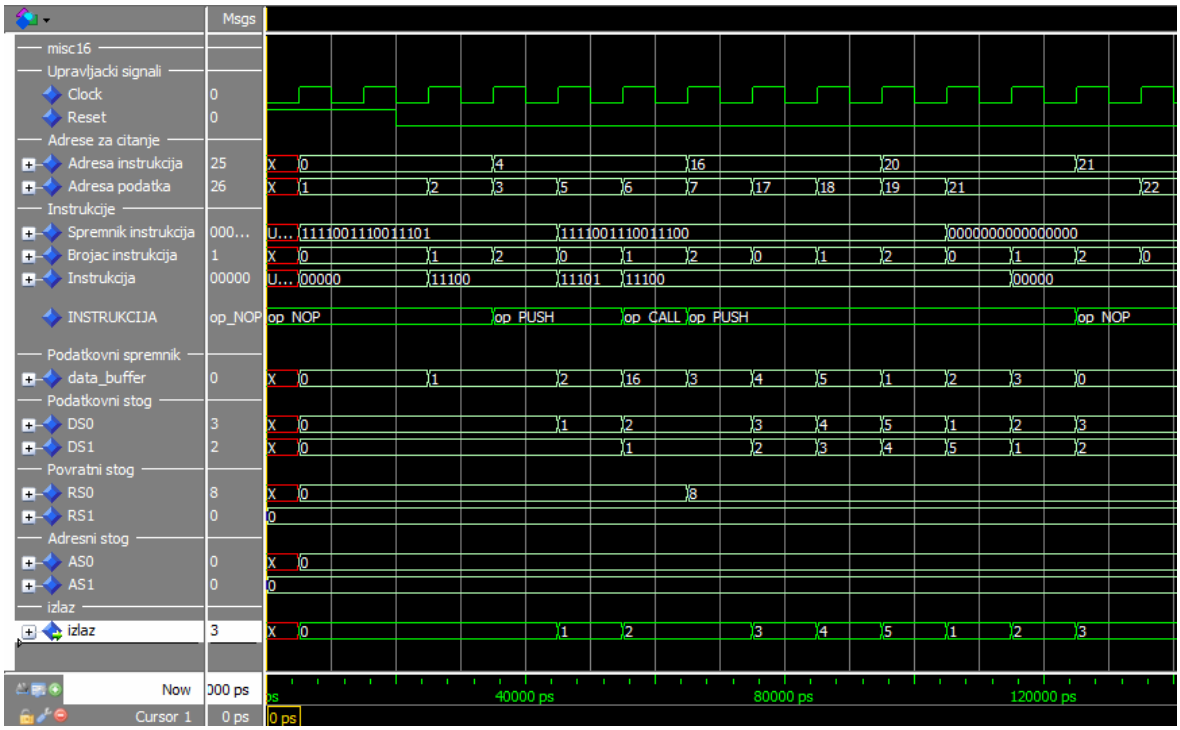
Instrukcije vezane uz povratni stog

Povratni stog se, kako mu samo ime kaže, koristi za pohranjivanje povratne adrese pri pozivima potprograma. Kao što je već i rečeno, postoje instrukcije koje prosljeđuju podatke između povratnog i podatkovnog stoga, kako bi omogućile manipulaciju povratnim adresama iz bilo kojeg razloga. Stoga su prve instrukcije čiji rad ćemo simulirati instrukcije *call* i *ret*, odnosno instrukcije poziva i povrataka iz potprograma.

```
1111001110011101    PUSH PUSH CALL
0000000000000001    1
0000000000000010    2
0000000000010000    16
1111001110011100    PUSH PUSH PUSH
0000000000000011    3
0000000000000100    4
0000000000000101    5
-- LOKACIJA 16
1111001110011100    PUSH PUSH PUSH
0000000000000001    1
0000000000000010    2
0000000000000011    3
```

Na slici 39 možemo vidjeti simulaciju gore navedenog programskog odsječka. Primjećujemo kako se zbog četverostupanjske protočne strukture izvršavaju još tri instrukcije nakon instrukcije *call*, odnosno nakon „najave“ skoka u toku programa. Kod ovakvih situacija postoje dvije opcije: poništiti izvođenje instrukcija nakon instrukcije za poziv skoka u programu, ili pustiti da se izvede određen broj sljedećih instrukcija (koji je određen brojem stupnjeva protočne strukture). Prva opcija ima nekoliko mana. Prvenstveno zahtjeva dodatnu logiku za poništavanje dohvata, dekodiranja i izvođenja instrukcija, što bi dodatno zakompliciralo dizajn procesora i zasigurno dovelo do smanjenja brzine rada sustava. Isto tako, u tom slučaju imamo dohvat instrukcija koje se ponište, te efektivno imamo tri prazna ciklusa u kojima procesor ne obavlja koristan rad. Ove poteškoće možemo izbjeći time što svjesno dozvoljavamo izvođenje triju sljedećih instrukcija. Time smo u nekim slučajevima izbjegli „prazne“ cikluse procesora, jer možemo programsku

memoriju izvesti na taj način da prebacimo instrukciju *call* tri naredbe unaprijed znajući da će se i ostale izvesti. Dakako, u nekim slučajevima takvo premještanje instrukcija neće biti moguće, te je tada potrebno nakon instrukcije *call* staviti tri *nop* instrukcije, kako bi omogućili procesoru vrijeme za promjenu adrese za dohvat novog bloka instrukcija. Taj je pristup praćen u izvedbi ovog procesora.



Slika 39. Simulacija rada instrukcije *call*

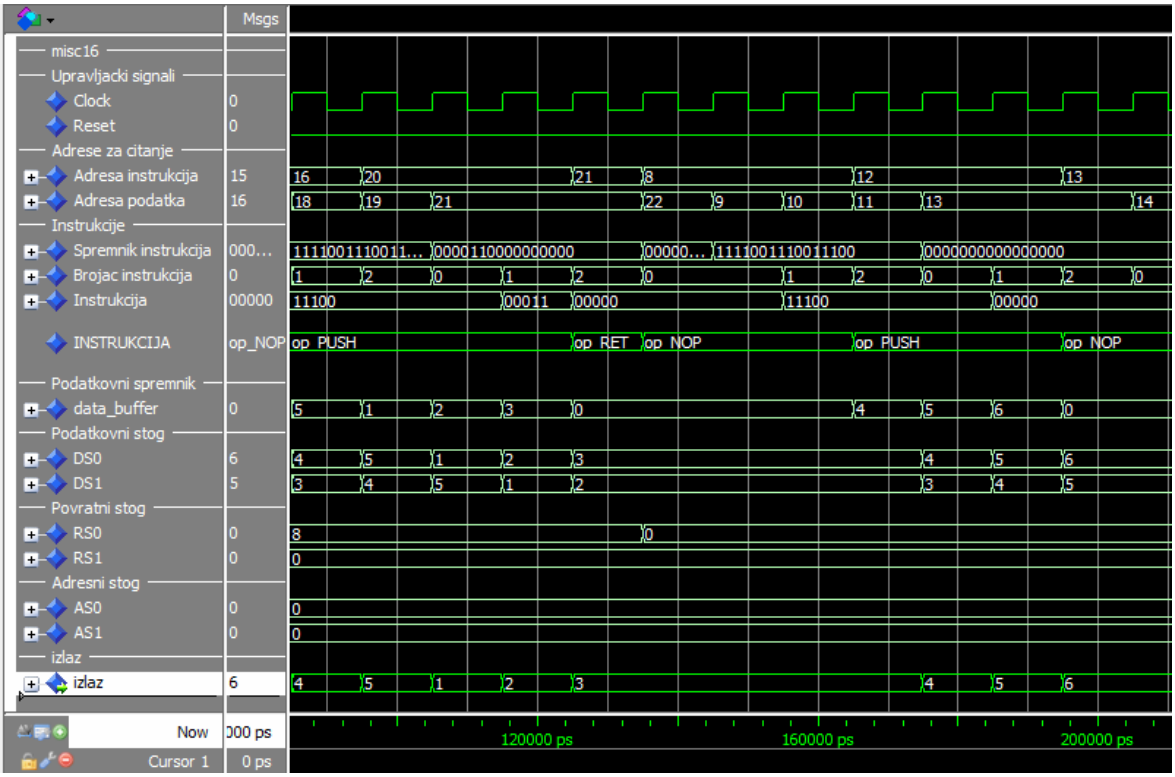
No, možemo primijetiti kako instrukcija *call* ima ograničenje u vidu pozicije unutar bloka instrukcija. Budući da znamo da se izvode tri sljedeće instrukcije, postavlja se pitanje pohranjivanja prave adrese za povratak na povratni stog. Kako se skokovi uvijek izvode na prvu instrukciju u bloku, na povratni stog bi se trebala pohraniti adresa instrukcijskog bloka koji slijedi tri „prijelazne“ instrukcije. U ovom konkretnom slučaju možemo vidjeti kako su sve te tri instrukcije podatkovne, te se sljedeća instrukcija nalazi na adresi 8. Ovdje moramo uzeti u obzir i pravovremenost ispravnog podatka na ulazu u multipleksor registra *r_in*. Naime, pri izvođenju instrukcije *call*, jedinica za dohvat je već dohvatila sljedeći spremnik, te se na zbrajalu pojavila adresa sljedećeg spremnika. Kada bi, primjerice, instrukcija *call* bila na nekom drugom mjestu u bloku, prilikom izvođenja te instrukcije, postojala bi mogućnost da se krivi podatak nađe na izlazu iz zbrajala, te bi se samim time taj krivi podatak pohranio na povratni stog. Dodatni razlog

predstavlja činjenica kako se uvijek skače na prvu instrukciju u bloku, te ako bi instrukcija call bila na bilo kojem drugom mjestu osim na trećem, postoji opasnost da se neke instrukcije ili izvedu dva puta, ili uopće ne izvedu. Iz tih razloga, instrukcija *call* mora biti na posljednjem mjestu u bloku. I ovdje, dakako postoji mogućnost ubacivanja *nop* instrukcija ukoliko se kod ne može tako napisati da instrukcija *call* dođe na posljednje mjesto u bloku, ali nažalost te slučajeve nije moguće izbjeći s postojećim jedinicama ovog procesora.

Sljedeća instrukcija za verifikaciju je instrukcija za povratak iz potprograma, odnosno instrukcija *ret*. Kod ove instrukcije vrijede iste stvari kao i kod instrukcije *call*, točnije, zbog protočne strukture se izvode tri instrukcije koje slijede instrukciju *ret*, te ju je korisno, ukoliko imamo mogućnost, premjestiti tri mjesta unaprijed. Ako tako nešto nije moguće, potrebno je nakon instrukcije ret ubaciti tri nop instrukcije koje služe za promjenu adrese za dohvat instrukcija uvjetovanu instrukcijom *ret*. Ova instrukcija, međutim, nema nikakva ograničenja vezana uz poziciju u bloku instrukcija, budući da se nikakvi podaci ne pohranjuju na povratni stog. U nastavku slijedi odsječak koda vezan uz instrukciju ret. Ovaj odsječak je isti kao i u prethodnom primjeru, odnosno proširena je verzija prethodnog primjera kako bi se omogućio povratak iz potprograma:

```
1111001110011101    PUSH PUSH CALL
0000000000000001    1
0000000000000010    2
0000000000010000    16
1111001110011100    PUSH PUSH PUSH
0000000000000011    3
0000000000000100    4
0000000000000101    5
1111001110011100    PUSH PUSH PUSH
0000000000000100    4
0000000000000101    5
0000000000000110    6
0000000000000000
0000000000000000
0000000000000000
0000000000000000
1111001110011100    PUSH PUSH PUSH
0000000000000001    1
0000000000000010    2
0000000000000011    3
0000110000000000    RET NOP NOP
0000000000000000    NOP
```

Na sljedećoj slici vidimo situaciju u kojoj se izvode instrukcije potprograma, te je povratna adresa pohranjena na povratni stog. Ovo je preslika situacije s prethodne slike, te je ovdje radi jednostavnosti prikazana od trenutka početka izvođenja instrukcija potprograma.



Slika 40. Simulacija rada instrukcije *ret*

Osim instrukcija call i ret, kojima je „osnovna“ svrha promjena toka programa, te povratni stog koriste kao međusobnu komunikaciju, postoje još dvije instrukcije, kojima je jedina svrha punjenje i pražnjenje povratnog stoga, a to su instrukcije *str* i *ldr*. Ove instrukcije rade na način da vrh podatkovnog stoga postave na vrh adresnog stoga (instrukcija *str*), ili obratno (instrukcija *ldr*). Programski odsječak za simulaciju rada tih instrukcija i prikaz rezultirajućih valnih oblika slijedi u nastavku:

```
1111001110011100    PUSH PUSH PUSH
0000000000000011    3
0000000000000100    4
0000000000000101    5
1111001110000100    PUSH PUSH STR
0000000000000110    6
0000000000000111    7
1001000010000100    STR STR STR
1001010010100101    LDR LDR LDR
1001010000000000    LDR NOP NOP
```

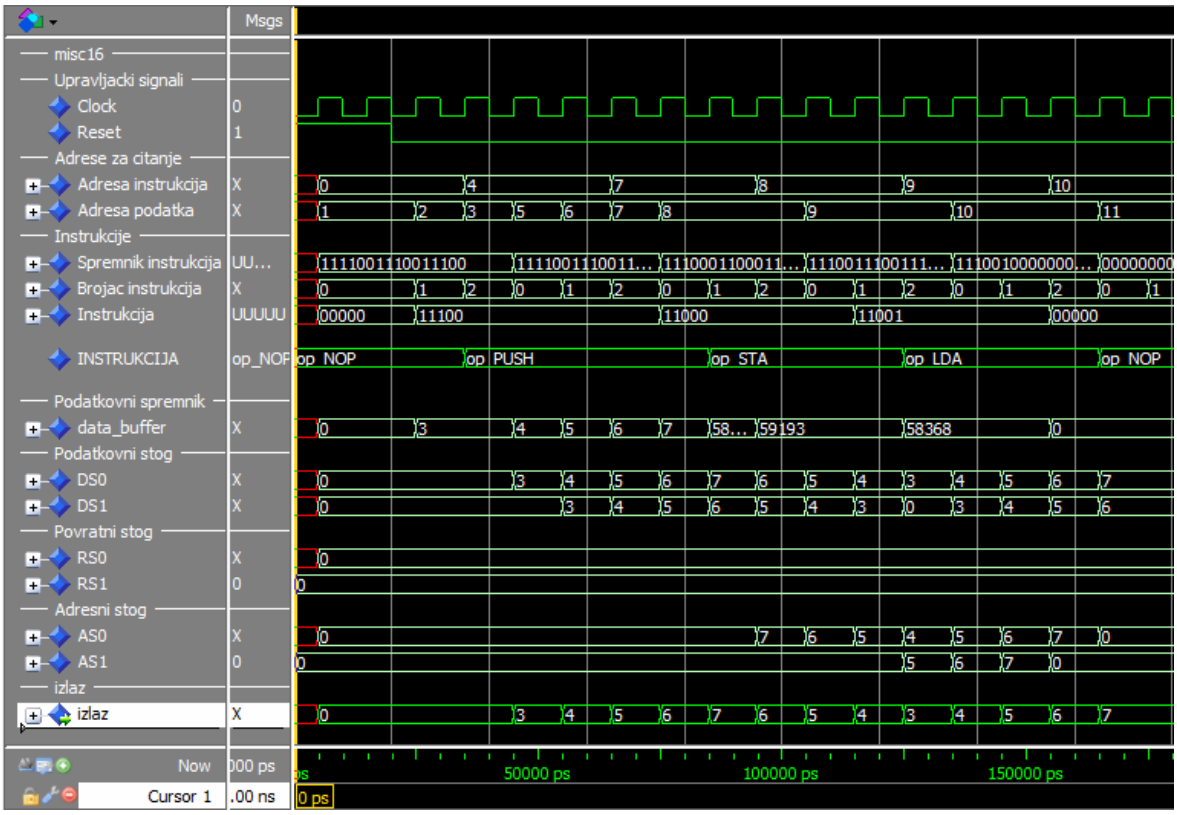
Slika 41. Simulacija rada instrukcija *str* i *ldr*

Iz priložene slike primjećujemo jednu karakteristiku distribuiranog RAM-a. Naime, dok god traje punjenje stoga, odnosno dok se god na adresni ulaz distribuiranih RAM-ova dovodi pokazivač na prvi prazni element stoga, nema vrijednosti na izlazu distribuiranih RAM-ova. Ova značajka komponente distribuiranog RAM-a nam ne smeta, pošto, kao što vidimo sa slike, kod pražnjenja stoga se ispravne vrijednosti javljaju na izlazu distribuiranog RAM-a, odnosno na signalu *r_out*. Isti efekt možemo primijetiti kod adresnog stoga, čije instrukcije slijede.

Instrukcije vezane uz adresni stog

Slično kao i kod povratnog stoga, postoje dvije instrukcije koje prosljeđuju podatke između podatkovnog i adresnog stoga, i to na isti način kao i kod povratnog stoga. Radi se o instrukcijama *sta* i *lda*, čija simulacija slijedi u nastavku:

```
1111001110011100    PUSH PUSH PUSH
0000000000000011    3
0000000000000100    4
0000000000000101    5
1111001110011000    PUSH PUSH STA
0000000000000110    6
0000000000000111    7
1110001100011000    STA STA STA
1110011100111001    LDA LDA LDA
1110010000000000    LDA NOP NOP
```



Slika 42. Simulacija rada instrukcija *sta* i *lda*

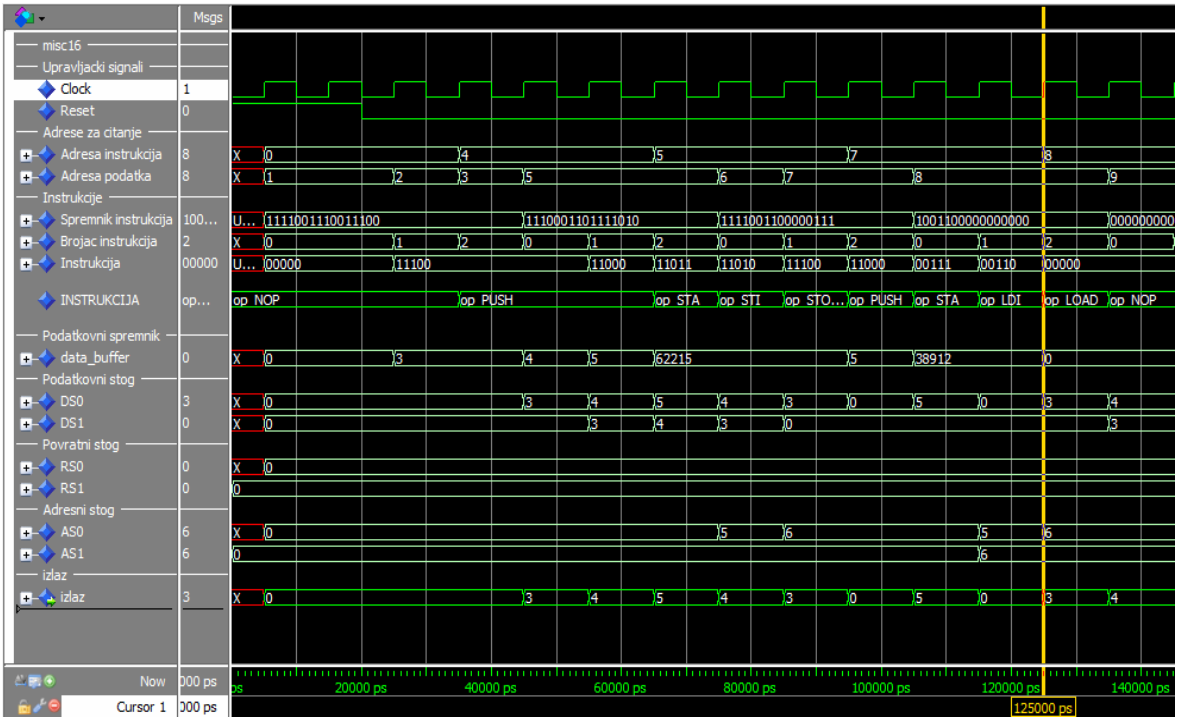
Uz ove, postoje još neke instrukcije koje provode jednostavne operacije nad vrhom adresnog stoga. Radi se o instrukcijama koje pišu i čitaju iz programske memorije te inkrementiraju adresu za pristup podatkovnoj memoriji, koja je ujedno i vrh adresnog stoga.

Instrukcije za pristup podatkovnoj memoriji

Kao što je već navedeno, podatkovna memorija je na sklopu realizirana jednopristupnim blok RAM-om. Zbog karakteristika blok RAM-a, neke od ovih instrukcija će imati određena ograničenja u upotrebi. Naime, kako je čitanje iz blok RAM resursa sinkrono signalu takta, postoji izlazni registar koji je neizbježan. Nadalje budući da se adresa za pristup podatkovnoj memoriji postavlja u fazi

izvođenja instrukcije vezane uz adresni stog, u sljedećem ciklusu se traženi podatak javlja na izlaznom registru, te je potreban dodatni registar kako bi se on sproveo na vrh podatkovnog stoga. Iz tog razloga, između postavljanja adrese za pristup podatkovnoj memoriji i samog čitanja iz nje, potreban je jedan ciklus takta u kojem se traženi podatak dohvati iz memorije. Stoga, ako je moguće, treba premjestiti neku drugu instrukciju između generiranja adrese za dohvat i samog dohvata. Ukoliko zbog karakteristika programa tako nešto nije moguće, potrebno je ubaciti jednu „praznu“ instrukciju (*nop*) kako bi se osigurao ispravan rad procesora. U sljedećem primjeru prikazan je pogrešan dohvat iz podatkovne memorije i poteškoće koje se tada javljaju:

```
1111001110011100    PUSH PUSH PUSH
0000000000000011    3
0000000000000100    4
0000000000000101    5
1110001101111010    STA STI STORE
1111001100000111    PUSH STA LDI
0000000000000101    5
1001100000000000    LOAD
```

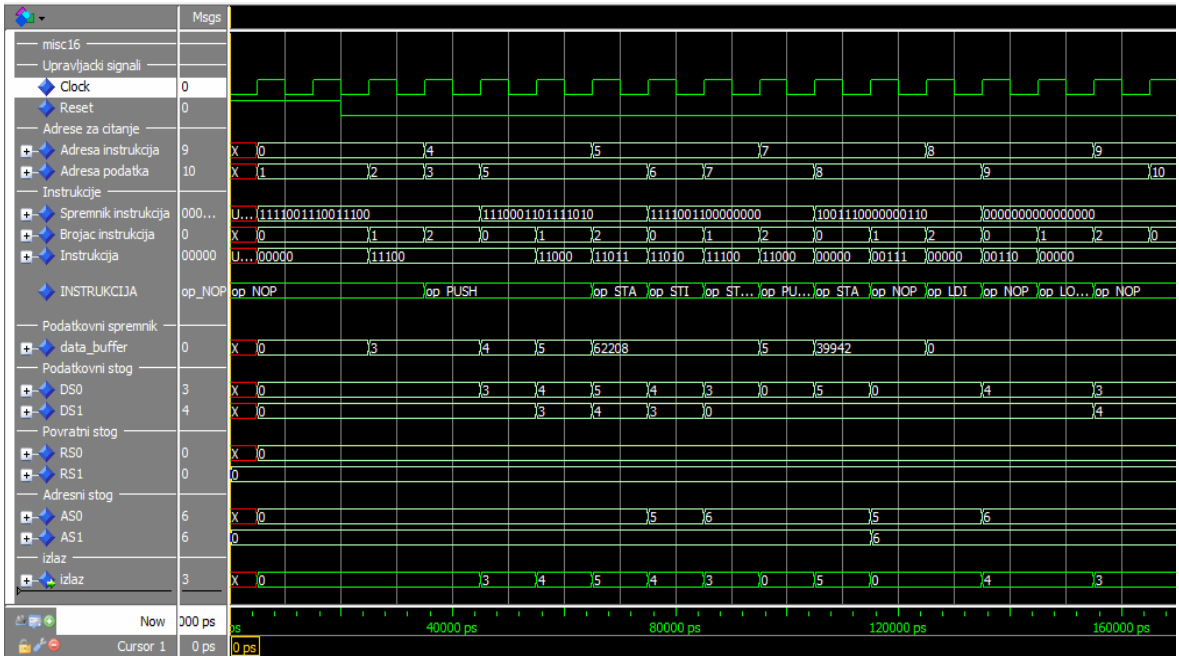


Slika 43. Neispravno čitanje iz podatkovne memorije

S prethodne slike možemo primijetiti spominjani efekt blok RAM-a. Naime, prilikom čitanja memorije instrukcijom *ldi*, koja neposredno slijedi instrukciju *sta* kojom se na adresni stog stavlja nova adresa, podatak koji čitamo je *3*, iako znamo iz prethodno implementiranog pohranjivanja kako se na toj adresi mora nalaziti

podatak *4*. Instrukcijom *ldi* se automatski inkrementira adrese, te se u sljedećem ciklusu čita novi podatak, prividno s adrese *6*, dok se zapravo tek u ovom trenutku na izlaznom registru pojavio podatak s adrese *5*. Kako bismo osigurali ispravno čitanje iz podatkovne memorije između postavljanja adrese i čitanja podatka umećemo instrukciju *nop*:

```
1111001110011100    PUSH PUSH PUSH
0000000000000011    3
0000000000000100    4
0000000000000101    5
1110001101111010    STA STI STORE
1111001100000000    PUSH STA NOP
0000000000000101    5
1001110000000110    LDI NOP LOAD
```



Slika 44. Ispravno čitanje iz podatkovne memorije

Na slici 44 vidimo ispravan način čitanja iz podatkovne memorije. Između postavljanja adrese i samog čitanja umetnut je jedan „prazni“ ciklus, te nam taj ciklus omogućava pojavljivanje traženog podatka na izlaznom registru blok RAM-a. Možemo vidjeti kako s adrese *5* učitavamo podatak *4*, a s adrese *6* podatak *3*, kako je i upisano u memoriju. Isto tako, primjećujemo kako kod instrukcija upisa nije potrebno „čekanje“ na adresu, te je upis moguće vršiti bez ograničenja.

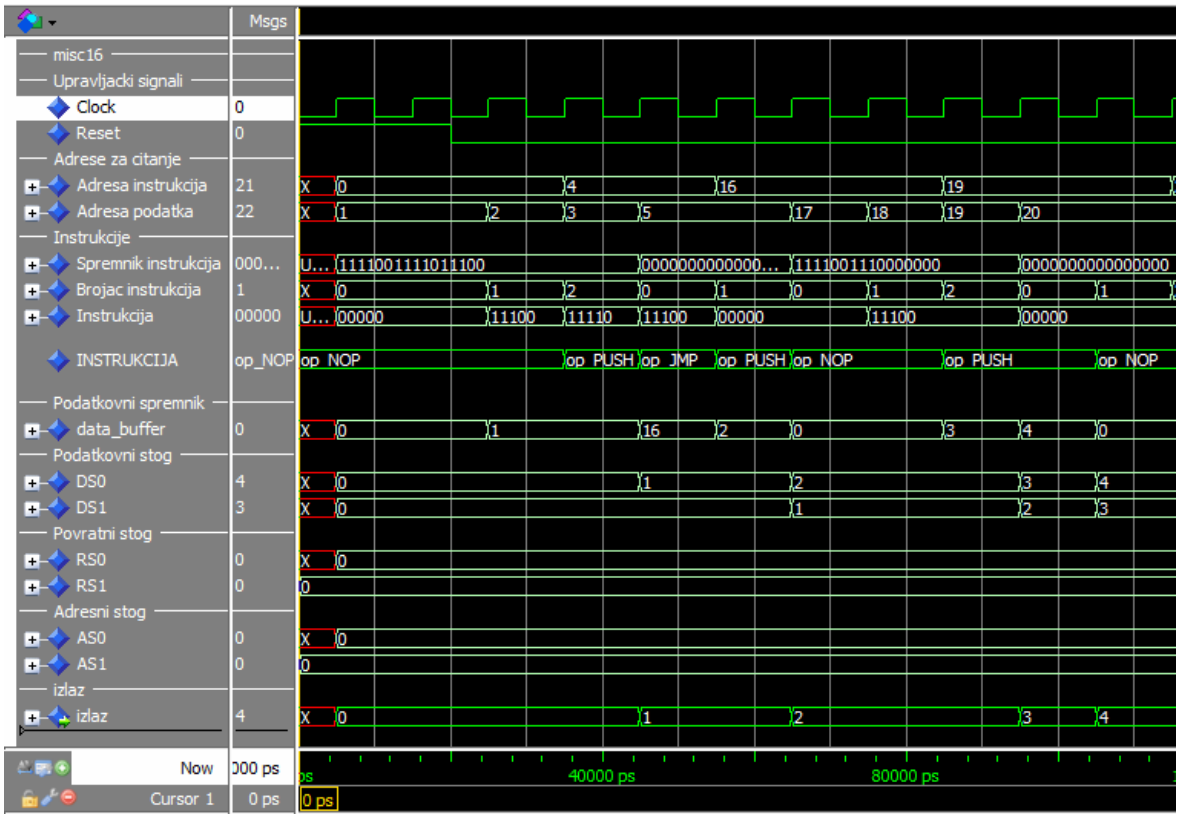
Ovime smo simulirali i rad ostalih instrukcija vezanih uz adresni stog, konkretno instrukcija *sti* i *ldi*, koje automatski uvećavaju podatak na vrhu adresnog stoga, to jest adresu za pristup podatkovnoj memoriji. Kako bismo završili

verifikaciju procesora nedostaju nam simulacije dviju instrukcija, a to su instrukcije skokova: *jmp* i *jz*.

Instrukcije skokova

Instrukciju bezuvjetnog skoka možemo jednostavno simulirati, imajući na umu karakteristike procesora kada su u pitanju skokovi. Ovo se odnosi na činjenicu da se zbog protočne strukture tri instrukcije nakon instrukcije za skok izvode bez ograničenja ili poništavanja. Slijedi prikaz odsječka programske memorije i rezultirajućih valnih oblika:

```
1111001111011100    PUSH PUSH JMP
0000000000000001    1
0000000000010000    16
0000000000000010    2
0000000000000000    NOP
-- LOKACIJA 16
1111001110000000    PUSH PUSH NOP
0000000000000011    3
0000000000000100    4
```

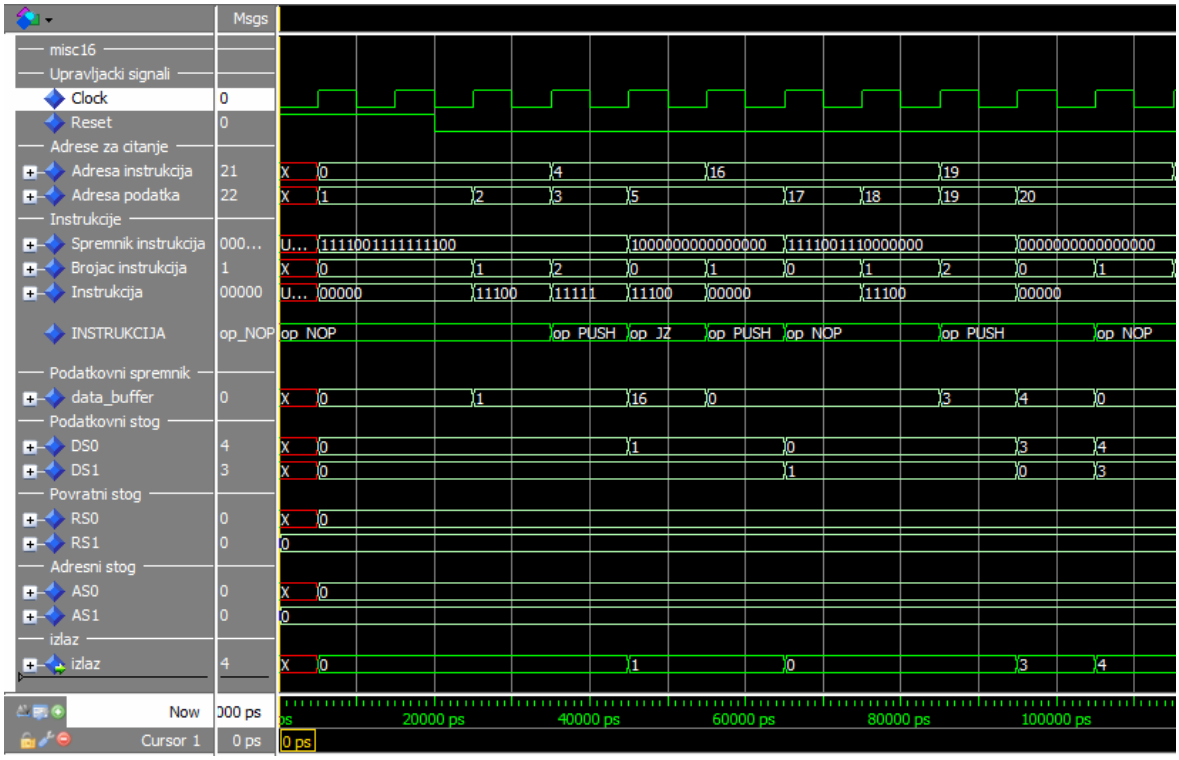


Slika 45. Simulacija instrukcije za bezuvjetni skok

Na slici vidimo kako se nakon „najave“ instrukcije skoka izvode 3 sljedeće instrukcije. Slično kao i kod instrukcije *ret*, instrukcije za skok nemaju ograničenje

vezano uz položaj te instrukcije u instrukcijskom bloku. No, kako ćemo vidjeti, instrukcija jz ima slično ograničenje kao i instrukcije za čitanje iz programske memorije. Naime, tijekom faze dekodiranja instrukcije za skok se generira signal skoka koji upravlja multipleksorom sljedeće adrese za dohvat instrukcija. Budući da se provjera uvjeta odvija u fazi dekodiranja koja se poklapa s fazom izvođenja prethodne instrukcije, upitna je pravovremenost podatka na vrhu podatkovnog stoga. Dakle, ukoliko se radi o instrukciji koja stavlja podatak na vrh podatkovnog stoga, potreban je ciklus kašnjenja kako bi se ispravan podatak uzeo u obzir pri testiranju uvjeta. Tako u sljedećem odsječku koda i na sljedećoj slici možemo vidjeti simulaciju instrukcije *jz*, koja dozvoljava skok ukoliko je podatak na vrhu podatkovnog stoga jednak nuli:

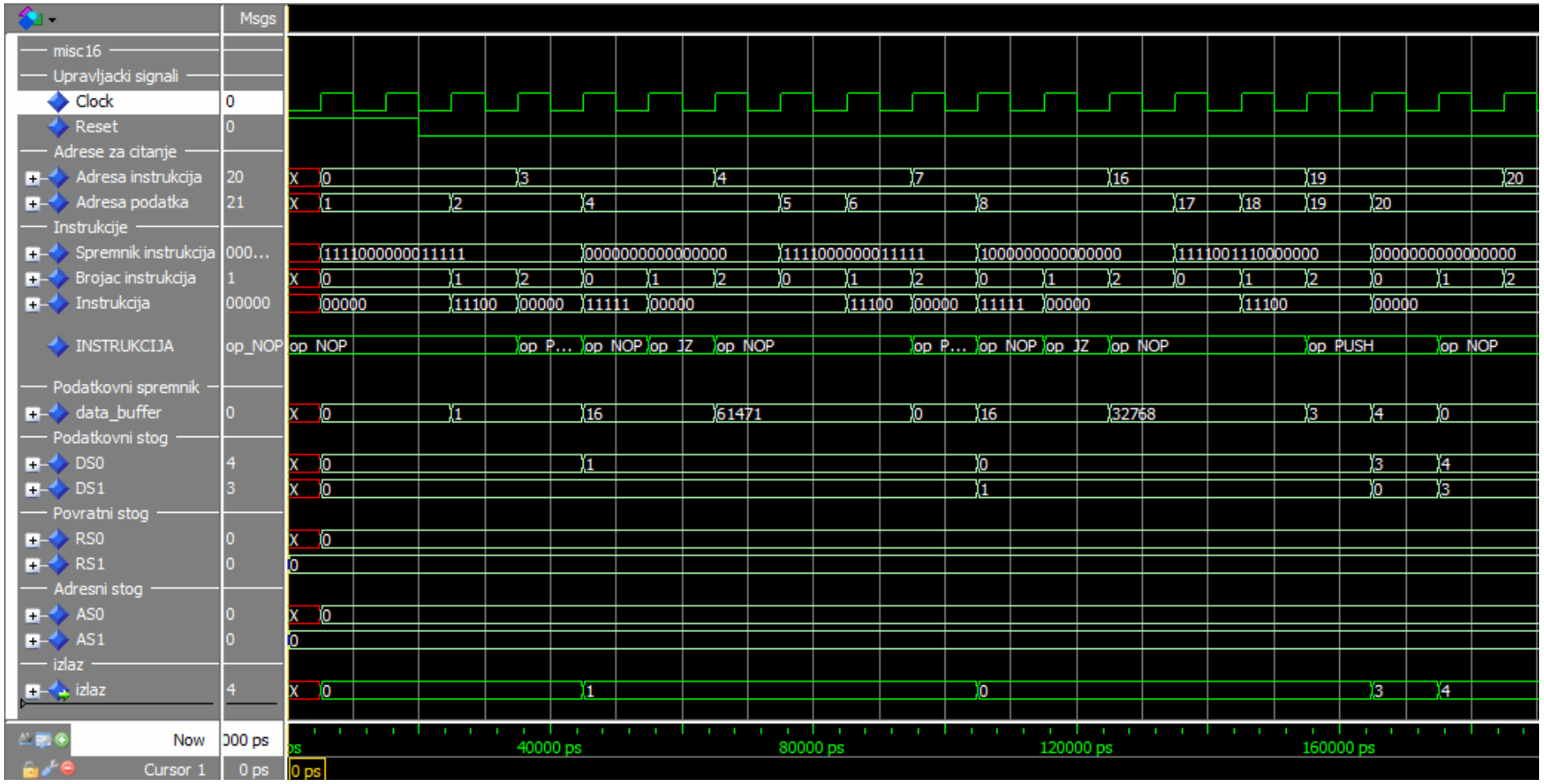
```
1111001111111100    PUSH JZ PUSH
0000000000000001    1
0000000000010000    16
0000000000000000    0
1000000000000000    NOP NOP NOP
0000000000000000    NOP
-- LOKACIJA 16
1111001110000000    PUSH PUSH NOP
0000000000000011    3
0000000000000100    4
```



Slika 46. Neispravno testiranje uvjeta instrukcije za uvjetni skok, *jz*

Slijedi da ukoliko dozvolimo jedan ciklus odgode testiranja uvjeta (kao u situaciji prikazanoj sljedećim odsječkom koda), imamo ispravno testiranje uvjeta i ispravan rad sustava za instrukciju *jz*:

```
1111000000011111    PUSH NOP JZ
0000000000000001    1
0000000000010000    16
0000000000000000    NOP NOP NOP
1111000000011111    PUSH NOP JZ
0000000000000000    0
0000000000010000    16
1000000000000000    NOP NOP NOP
1000000000000000    NOP
-- LOKACIJA 16
1111001110000000    PUSH PUSH NOP
0000000000000011    3
0000000000000100    4
```



Slika 47. Simulacija rada instrukcije *jz*, s ispravnim testiranjem uvjeta

Budući da smo ovime završili verifikaciju rada sustava tako da smo simulirali i ispitali ispravnost rada svih instrukcija, još ćemo jednom navesti sva ograničenja u radu pojedinih instrukcija:

- kod instrukcije call – bitno je da instrukcija bude na posljednjem mjestu u instrukcijskom bloku
- kod instrukcija za čitanje iz podatkovne memorije – potrebno je omogućiti ažuriranje podatka na izlaznom registru dodavanjem jednog ciklusa (prazne instrukcije ili instrukcije koja koristi preostale resurse procesora) nakon postavljanja adrese
- kod instrukcije uvjetnog skoka – potrebno je omogućiti ispravno testiranje uvjeta dodavanjem jednog ciklusa (prazne instrukcije ili instrukcije koja koristi preostale resurse procesora) nakon postavljanja podatka na podatkovni stog

Ovime završava proces funkcijske simulacije rada sustava, te je sustav potrebno sintetizirati i implementirati na ciljanom sustavu, o čemu se govori u sljedećem poglavlju.

4.3. Sinteza sustava

Nakon dizajna sustava (i njegove verifikacije preko funkcijske simulacije) imamo RTL model napisan HDL jezikom koji je potrebno pretvoriti u oblik pogodan za implementaciju na ciljanom sustavu, te nam za to služi sinteza. Sinteza sustava je proces koji se provodi kako bi se RTL model pretvorio u listu povezanih standardnih logičkih sklopova, pri čemu se koristi baza koju daje proizvođač. Ovdje je korišten integrirani alat *Xilinx Synthesis Tool* (XST). XST kao izlaz daje datoteku koja uključuje logički dizajn i moguća korisnička ograničenja vezana uz sam proces sinteze. Kao i ostali integrirani alati, XST nudi mnoštvo opcija u koje u ovom trenutku nećemo detaljno ulaziti, samo je potrebno reći kako se zbog boljih performansi predlaže isključenje opcije dijeljenja resursa (*Resource Sharing*). Kako bi se pokrenula sinteza sustava, potrebno je, uz označen željeni model za sintezu pokrenuti proces Synthesize (XST). Tijekom sinteze XST nam javlja koje je komponente iskoristio u stvaranju logičkog modela sustava, uz moguća upozorenja ili prijave grešaka. Isto tako, provodi se analiza putova signala, te se

na temelju najgoreg puta definira minimalni takt na kojem sustav ispravno radi. Budući da kod ovog procesora postoje četiri faze protočne strukture, potrebno je ispitati njihove vremenske odnose kako bi se ustanovila ispravnost odabranih faza, te po potrebi izmijenile pojedine faze ili dodale dodatne. U tablici 1 vidimo vremenske značajke pojedinih faza:

Broj faze	Ime faze	Minimalni period	Maksimalna frekvencija
1.	Dohvat instrukcijskog spremnika	-	-
2.	Dohvat instrukcije iz spremnika	3.148 ns	317.628 MHz
3.	Dekodiranje instrukcije	-	-
4.	Izvođenje instrukcije	8.118 ns	123.185 MHz

Tablica 1. Faze protočne strukture i vremenski odnosi

Iz tablice možemo zaključiti kao su faze 1 i 3 protočne strukture jednostavne, jer se radi o čitanju blok RAM resursa, i o logičkim funkcijama s 5 ulaznih varijabli. S druge strane, kompleksne su faze 2 i 4. Kod faze 2 imamo mnoštvo logike i nekoliko razina multipleksora, kao što se vidi slikom 18, dok kod faze 4 imamo velik broj ulaznih varijabli te prolazak signala kroz nekoliko multipleksora do odredišta. Iako je dohvat rastavljen na dvije faze, funkcijski ga obavlja jedna jedinica, zbog povezanosti ulaznih signala, te je stoga potrebno promatrati minimalan period te jedinice, kao što se vidi u sljedećoj tablici:

Broj faze	Faza	Minimalni period	Maksimalna frekvencija
1.	Dohvat instrukcije	6.138 ns	162.912 MHz
2.	Dekodiranje instrukcije	-	-
3.	Izvođenje instrukcije	8.118 ns	123.185 MHz

Tablica 2. Vremenski odnosi pojedinih jedinica za realizaciju protočne strukture

Budući da se vidi kako su faze približno jednake, možemo zaključiti kako su faze ispravno odabrane, te možemo pristupiti sintezi cijelog sustava.

Inačica procesora *misc16*

Kod sinteze cijelog sustava ograničavajući faktor brzine sustava je najsporiji dio. Kao što se može vidjeti iz tablice, ograničava nas faza izvođenja instrukcije.

Nadalje, u izvještaju koji se dobije sintezom možemo vidjeti koji je najkritičniji put signala, te ga po mogućnosti možemo optimirati. U nastavku slijedi isječak iz *Synthesis Reporta* koji se tiče iskorištenog hardvera pri sintezi našeg sustava:

```
Device utilization summary:
---------------------------

Selected Device : 3s500efg320-5

 Number of Slices:                      334  out of   4656     7%
 Number of Slice Flip Flops:            163  out of   9312     1%
 Number of 4 input LUTs:                629  out of   9312     6%
    Number used as logic:               593
    Number used as RAMs:                 36
 Number of IOs:                          18
 Number of bonded IOBs:                  18  out of    232     7%
 Number of BRAMs:                         2  out of     20    10%
 Number of MULT18X18SIOs:                 1  out of     20     5%
 Number of GCLKs:                         1  out of     24     4%
```

Nadalje, minimalan period signala takta, odnosno maksimalna frekvencija signala takta jest:

```
Minimum period: 8.409ns (Maximum Frequency: 118.915MHz)
```

Dodatna mogućnost koji nam nudi *Synthesis Report* je, kao što smo već i rekli, mogućnost provjere najsporijeg puta signala:

```
                           Gate     Net
Cell:in->out      fanout   Delay   Delay  Logical Name
----------------------------------------  ------------
FDS:C->Q             11   0.514   0.862  d_pointer1_0
LUT3:I1->O           14   0.612   0.850  ds_addr<0>1_1
RAM16X1S:A0->O        3   1.065   0.520  DATA_STACK[0].DSTACK
LUT2:I1->O            1   0.612   0.000  Madd_mac_result_lut<0>
MUXCY:S->O            1   0.404   0.000  Madd_mac_result_cy<0>
MUXCY:CI->O           1   0.052   0.000  Madd_mac_result_cy<1>
MUXCY:CI->O           1   0.052   0.000  Madd_mac_result_cy<2>
MUXCY:CI->O           1   0.052   0.000  Madd_mac_result_cy<3>
MUXCY:CI->O           1   0.052   0.000  Madd_mac_result_cy<4>
MUXCY:CI->O           1   0.052   0.000  Madd_mac_result_cy<5>
MUXCY:CI->O           1   0.052   0.000  Madd_mac_result_cy<6>
MUXCY:CI->O           1   0.052   0.000  Madd_mac_result_cy<7>
MUXCY:CI->O           1   0.052   0.000  Madd_mac_result_cy<8>
MUXCY:CI->O           1   0.052   0.000  Madd_mac_result_cy<9>
MUXCY:CI->O           1   0.052   0.000  Madd_mac_result_cy<10>
MUXCY:CI->O           1   0.052   0.000  Madd_mac_result_cy<11>
MUXCY:CI->O           1   0.052   0.000  Madd_mac_result_cy<12>
MUXCY:CI->O           1   0.052   0.000  Madd_mac_result_cy<13>
MUXCY:CI->O           0   0.052   0.000  Madd_mac_result_cy<14>
XORCY:CI->O           1   0.699   0.360  Madd_mac_result_xor<15>
LUT4:I3->O            2   0.612   0.380  data_mux0001<15>166
MULT18X18SIO:A15          0.198          Mmult_mul_result_long
----------------------------------------
Total                     8.409ns (5.437ns logic, 2.972ns route)
                                  (64.7% logic, 35.3% route)
```

Primjećujemo kako je propagacija *carry* signala uzročnik velikog dijela kašnjenja zbog prolaska kroz logička vrata (*gate delay*), te zaključujemo kako bismo mogli

izostaviti instrukciju za množenje s akumulacijom (*mac – multiply and accumulate*) kako bismo postigli bolje performanse sustava.

Inačica procesora *misc16_mul*

Inačica procesora bez *mac* instrukcije koristi sljedeće resurse sklopa:

```
Device utilization summary:
---------------------------

Selected Device : 3s500efg320-5

 Number of Slices:                      324  out of   4656     6%
 Number of Slice Flip Flops:            161  out of   9312     1%
 Number of 4 input LUTs:                613  out of   9312     6%
    Number used as logic:               577
    Number used as RAMs:                 36
 Number of IOs:                          18
 Number of bonded IOBs:                  18  out of    232     7%
 Number of BRAMs:                         2  out of     20    10%
 Number of MULT18X18SIOs:                 1  out of     20     5%
 Number of GCLKs:                         1  out of     24     4%
```

Najkritičniji put signala kod inačice misc16_mul jest:

```
                          Gate     Net
Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
----------------------------------------  ------------
FDR:C->Q              13   0.514   0.988  data_0 (data_0)
LUT1:I0->O             1   0.612   0.000  Madd_alu_result_addsub0000_cy<0>_rt
MUXCY:S->O             1   0.404   0.000  Madd_alu_result_addsub0000_cy<0>
MUXCY:CI->O            1   0.051   0.000  Madd_alu_result_addsub0000_cy<1>
MUXCY:CI->O            1   0.051   0.000  Madd_alu_result_addsub0000_cy<2>
MUXCY:CI->O            1   0.051   0.000  Madd_alu_result_addsub0000_cy<3>
MUXCY:CI->O            1   0.051   0.000  Madd_alu_result_addsub0000_cy<4>
MUXCY:CI->O            1   0.051   0.000  Madd_alu_result_addsub0000_cy<5>
MUXCY:CI->O            1   0.051   0.000  Madd_alu_result_addsub0000_cy<6>
MUXCY:CI->O            1   0.051   0.000  Madd_alu_result_addsub0000_cy<7>
MUXCY:CI->O            1   0.051   0.000  Madd_alu_result_addsub0000_cy<8>
MUXCY:CI->O            1   0.051   0.000  Madd_alu_result_addsub0000_cy<9>
MUXCY:CI->O            1   0.051   0.000  Madd_alu_result_addsub0000_cy<10>
MUXCY:CI->O            1   0.051   0.000  Madd_alu_result_addsub0000_cy<11>
MUXCY:CI->O            1   0.051   0.000  Madd_alu_result_addsub0000_cy<12>
MUXCY:CI->O            1   0.051   0.000  Madd_alu_result_addsub0000_cy<13>
MUXCY:CI->O            0   0.051   0.000  Madd_alu_result_addsub0000_cy<14>
XORCY:CI->O            1   0.699   0.360  Madd_alu_result_addsub0000_xor<15>
LUT4:I3->O             1   0.612   0.387  data_mux0001<15>109_SW0 (N2476)
LUT4_L:I2->LO          1   0.612   0.130  data_mux0001<15>135
LUT4:I2->O             2   0.612   0.380  data_mux0001<15>157
MULT18X18SIO:A15           0.198          Mmult_mul_result_long
----------------------------------------
Total                       7.229ns (4.984ns logic, 2.244ns route)
                                    (68.9% logic, 31.1% route)

Minimum period: 7.229ns (Maximum Frequency: 138.341MHz)
```

Iz ovoga vidimo kako smo dobili 20 MHz veću maksimalnu frekvenciju, odnosno povećanje brzine rada od 17 posto, te je sada kritičan dio propagacija *carry* signala kod ALU instrukcija.

Inačica procesora *misc16_f*

Dodatno ubrzanje možemo postići potpunim uklanjanjem množila, te je tada iskorištenje resursa sklopa:

```
Device utilization summary:
---------------------------

Selected Device : 3s500efg320-5

 Number of Slices:                      294  out of   4656     6%
 Number of Slice Flip Flops:            160  out of   9312     1%
 Number of 4 input LUTs:                555  out of   9312     5%
    Number used as logic:               519
    Number used as RAMs:                 36
 Number of IOs:                          18
 Number of bonded IOBs:                  18  out of    232     7%
 Number of BRAMs:                         2  out of     20    10%
 Number of GCLKs:                         1  out of     24     4%
```

Pri verziji bez množila imamo sljedeće karakteristike sustava:

```
                          Gate     Net
Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
----------------------------------------  ------------
FDR:C->Q             14   0.514   0.880  counter_1
LUT3_L:I2->LO         1   0.612   0.103  Mmux_instruction_mux5_SW0
LUT4:I3->O            2   0.612   0.449  Mmux_instruction_mux5
LUT4:I1->O            2   0.612   0.000  Madd_da_adder_lut<0>
MUXCY:S->O            1   0.404   0.000  Madd_da_adder_cy<0>
MUXCY:CI->O           1   0.052   0.000  Madd_da_adder_cy<1>
MUXCY:CI->O           1   0.052   0.000  Madd_da_adder_cy<2>
MUXCY:CI->O           1   0.052   0.000  Madd_da_adder_cy<3>
MUXCY:CI->O           1   0.052   0.000  Madd_da_adder_cy<4>
MUXCY:CI->O           1   0.052   0.000  Madd_da_adder_cy<5>
MUXCY:CI->O           1   0.052   0.000  Madd_da_adder_cy<6>
MUXCY:CI->O           1   0.052   0.000  Madd_da_adder_cy<7>
MUXCY:CI->O           0   0.052   0.000  Madd_da_adder_cy<8>
XORCY:CI->O           1   0.699   0.426  Madd_da_adder_xor<9> (da_adder<9>)
LUT3:I1->O            1   0.612   0.000  data_addr_mux<9>111 (N2326)
FDR:D                     0.268          data_addr_9
----------------------------------------
Total                      6.603ns (4.745ns logic, 1.858ns route)
                                   (71.9% logic, 28.1% route)

Minimum period: 6.603ns (Maximum Frequency: 151.447MHz)
```

Vidimo kako smo uklanjanjem množila ostvarili dodatno povećanje brzine rada sustava (10 posto u odnosu na prethodnu inačicu). Ovdje također možemo primjetiti i jednu bitnu stvar, a to je da ukoliko nemamo množila, najkritičniji put signala više nije u fazi izvođenja instrukcija, već je to propagacija *carry* signala kod zbrajala za adresu za dohvat podataka iz programske memorije. Ovime smo dokazali kako su faze protočne strukture jako bliske pošto je uklanjanjem množila uz razliku perioda od samo oko 0.5 ns najkritičniji put prebačen iz faze izvođenja u fazu dohvata.

Zbog prikazanog su razvijene tri inačice procesora, te se sukladno potrebama može koristiti bilo koja.

- **misc16** – inačica koja sadrži množilo te posjeduje instrukcijski set koji uključuje *mul* i *mac* instrukcije. Ova inačica je i najsporija, a poželjno ju je koristiti u situacijama koje zahtijevaju mnogo množenja s akumulacijama, jer ova inačica to obavlja jednom ciklusu.
- **misc16_mul** – inačica koja sadrži množilo, ali ne posjeduje instrukciju za množenje s akumulacijom. Ova inačica je brža od prethodne, te se predlaže njeno korištenje ukoliko nam treba mnogo operacija množenja, ali mali broj popratnih zbrajanja.
- **misc16_f** – inačica koja ne posjeduje množilo, ali ju odlikuje vrlo visok takt na kojem može raditi

4.4. Implementacija sustava

Sintezom se, kako je već rečeno, RTL model prevodi u listu povezanih logičkih sklopova, te se provodi analiza vremenskih karakteristika modela. No, ova analiza je temeljena na modelu i ne uzimaju u obzir stvarna kašnjenja i poziciju sklopova i njihove međusobne udaljenosti na sklopu. Zbog toga nam je potreban proces implementacije sintetiziranog dizajna na odabrani sklop. Ovaj proces se provodi u tri koraka [3]:

- *mapping* – mapiranje liste generirane sintezom na ciljanu platformu, odnosno prilagođavanje konkretnom sklopovlju na kojem se radi implementacija
- *placing* – razmještanje elemenata iz liste, uz optimizacije u vidu poboljšavanja performansi (pazi se na duljinu vodova i propagaciju) ili smanjenja utrošene površine.
- *routing* – u ovom koraku se stvaraju vodovi koji povezuju elemente liste

Konačan rezultat ovog procesa za FPGA sklopove je datoteka s konfiguracijom sklopa (programming file), koja se zatim može učitati na sam sklop. Budući da se kod implementacije vodi računa i o stvarnim kašnjenjima, vremenska analiza nakon implementacije se razlikuje od one prilikom sinteze. U tablici 3 možemo

vidjeti razlike performansi pojedinih inačica sklopa nakon procesa sinteze i nakon procesa implementacije.

Inačica procesora	Minimalni period		Maksimalni takt	
	sinteza	implementacija	sinteza	implementacija
misc16	8.409 ns	8.662 ns	118.915 MHz	115.447 MHz
misc16_mul	7.229 ns	7.946 ns	138.341 MHz	125.849 MHz
misc16_f	6.603 ns	7.576 ns	151.447 MHz	131.996 MHz

Tablica 3. Razlike vremenskih značajki nakon sinteze i implementacije

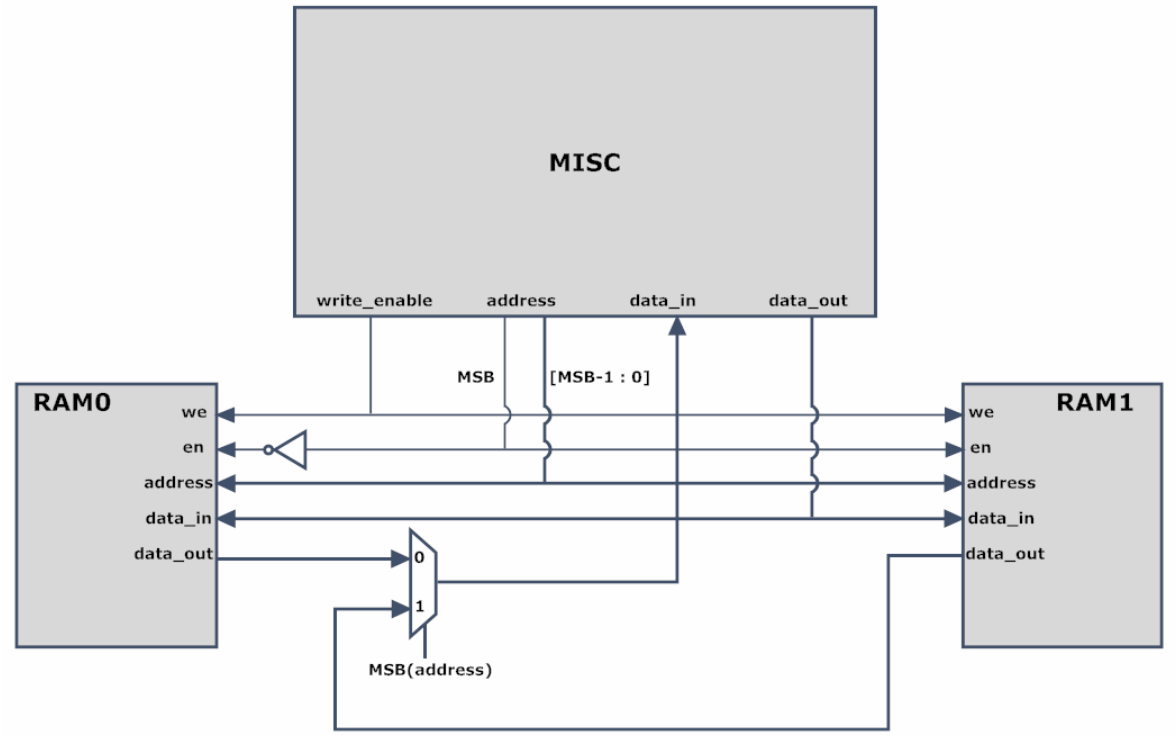
Iz priloženog vidimo kako je razlika frekvencije takta između sinteze i implementacije raste s porastom frekvencije, a najveća je u slučaju bez množila, te je tada razlika 13 posto, odnosno čak 20 MHz.

4.5. Višestruke procesorske jezgre

Dodatna mogućnost poboljšanja performansi procesora leži u činjenici da ovakva izvedba procesora troši samo mali dio sklopa na kojem se implementira. Što se tiče konfigurabilnih logičkih blokova (*CLB*), implementacijom procesora MISC16 se troši oko 7 posto ukupnog broja CLB-a. Nadalje, za implementaciju MISC16 procesora je potrebno oko 6 posto LUT-ova, pa stoga vidimo kako je moguće u isti sklop staviti mnogo ovakvih procesora, čime se mogu ostvariti višestruko bolje performanse cjelokupnog sustava. Ograničavajući faktor broju jezgri procesora unutar ove inačice Spartan-3E sklopa jest ukupni broj blok RAM resursa, koji iznosi 20. Kako znamo da svaki procesor koristi dva blok RAM-a (programska memorija i podatkovna memorija), zaključujemo u ovaj sklop možemo staviti maksimalno 10 ovakvih jezgri. No, ako bismo procesore u ovom obliku postavili u sklop, dobili bismo mogućnost paralelnog izvođenja 10 različitih zadaća, definiranih preko zasebnih programskih memorija. Stoga bi bilo izuzetno korisno povezati jezgre kako bi one mogle dijeliti resurse, a s time i posao. Pri ovome nam olakotnu okolnost predstavlja činjenica da svaki procesor kao podatkovnu memoriju koristi jednopristupni blok RAM. Stoga možemo kao vezu između jezgara koristiti dvopristupne blok RAM resurse, na način da se jezgre

lančano vežu na blok RAM-ove, tako da blok RAM na jednom portu vidi jednu jezgru a na drugom drugu. Kako bi ovo povezivanje bilo moguće, potrebno je na izlaze procesora spojiti vrh adresnog stoga (koji se, kako smo pokazali u poglavlju *4.1. Procesor MISC16*, koristi za pristup podatkovnoj memoriji), te signal za omogućavanje upisa u memoriju (*write enable*). Dodatno je na izlaz procesora potrebno spojiti ulazne i izlazne sabirnice preko kojih se podaci prenose prema i od memorije. Kao izlazni signal koristi se vrh podatkovnog stoga, budući da se taj podatak upisuje u memoriju, a kao ulazni signal se koristi signal *dtm_reg_out* koji se prema slici 22 spaja na ulazni multipleksor za vrh podatkovnog stoga.

Kako se svaka jezgra spaja na dva blok RAM-a, postigli smo udvostručenje memorijskog prostora svake jezgre, te sada pojedina jezgra ima ukupno 2048 lokacija po 16. bitova. Kako bismo povećali adresni prostor, dodajemo jedan bit adresnom registru *a_in*, i adresnom stogu (potreban nam je dodatni distribuirani RAM). Kako svaka memorija ima 10-bitni adresni ulaz, najviši bit adrese možemo koristiti za odabir blok RAM-a, i to na način da ukoliko je najviši bit adrese jednak nuli, pristupa se „lijevom“ RAM-u, a ako je jednak jedinici pristupamo „desnom“ RAM-u. Implementacija ove funkcionalnosti je vrlo jednostavna, pošto najviši bit adrese možemo spojiti na *enable* ulaz RAM-a, kao što je prikazano slikom 48.

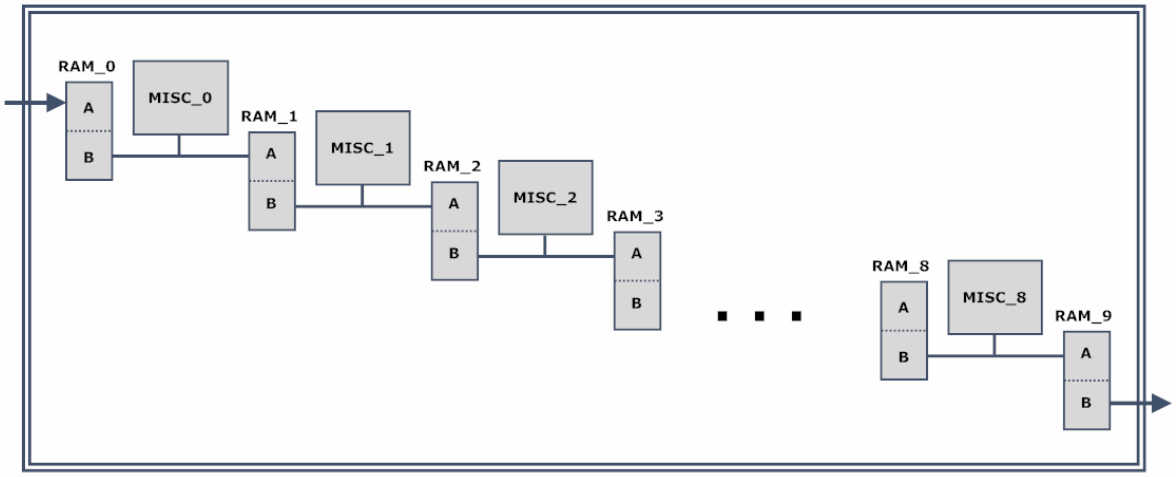


Slika 48. Jezgra MISC procesora spojena na dva blok RAM resursa

Dodatno, u ovisnosti o tom bitu adresni signal i signal za omogućavanje upisa spajamo na odabrani RAM. Kod čitanja podatka iz memorije, situacija je nešto složenija, budući da signali moraju proći kroz multipleksor kojeg isto tako kontrolira najviši bit adrese. U sljedećem odsječku koda vidimo spajanje jedne jezgre procesora na dva blok RAM resursa, pri čemu su signali jezgre označeni s *m0*, a signali vezani uz RAM-ove s *ram0* i *ram1*:

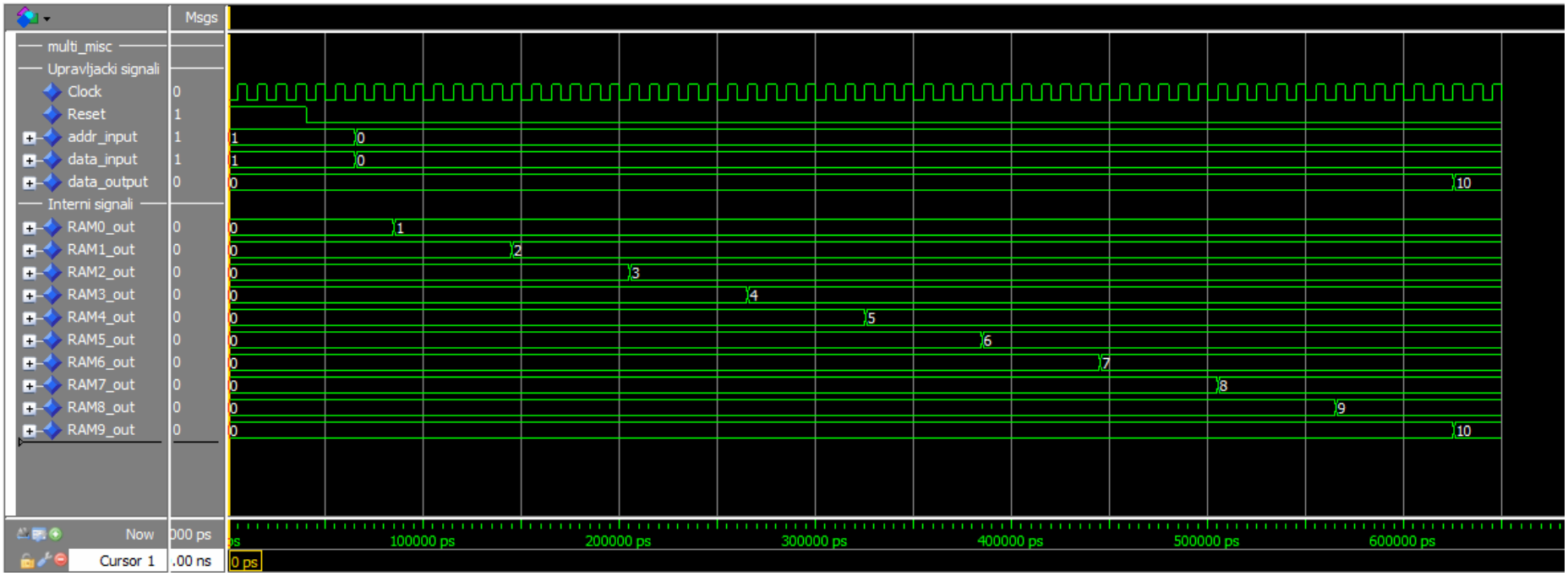
```
------------------------ MISC_0 ----------------------------------------
process(ram_addr_m0, ram_we_m0)
begin
	case ram_addr_m0(10) is
		when '0' => ram0_addr_b <= ram_addr_m0(9 downto 0);
				ram0_we_b <= ram_we_m0;
				ram1_addr_a <= (others => '0');
				ram1_we_a <= '0';
		when '1' => ram1_addr_a <= ram_addr_m0(9 downto 0);
				ram1_we_a <= ram_we_m0;
				ram0_addr_b <= (others => '0');
				ram0_we_b <= '0';
		when others => null;
	end case;
end process;

-- enable signali
ram0_en_b <= not ram_addr_m0(10);
ram1_en_a <= ram_addr_m0(10);

-- multipleksor za čitanje podataka iz memorije
process(ram_addr_m0, data_out_ram0b, data_out_ram1a)
begin
	case ram_addr_m0(10) is
		when '0' => data_in_m0 <= data_out_ram0b;
		when '1' => data_in_m0 <= data_out_ram1a;
		when others => null;
	end case;
end process;
```

Budući da se svaka jezgra veže na dva RAM-a, te sadrži u sebi sadrži još jednog, zaključujemo kako je maksimalan broj ovako formiranih jezgri unutar sklopa jednak 9, čime koristimo ukupno 19 blok RAM-ova. Spajanjem svih jezgara s RAM-ovima na opisani način, pri čemu se drugi portovi prvog i posljednjeg blok RAM-a vežu na izlaze sklopa, dobijemo sustav prikazan blok shemom na slici 49.



Slika 49. Sustav s višestrukim jezgrama povezanim preko blok RAM resursa

Ovako formiran sustav može brže obavljati složenije zadaće prosljeđivanjem podataka i posla ostalim procesorima, pri čemu svaka jezgra ima zasebnu programsku memoriju. U svrhu verifikacije rada ovakvog sustava, osmišljena je jednostavna zadaća čitanja podatka iz „nižeg“ RAM-a, uvećavanje podatka za 1, i prosljeđivanje sljedećoj jezgri putem „višeg“ RAM-a. Simulacija i rezultati obavljanja tih zadaća prikazani su slikom 50, a na slici možemo vidjeti i propagaciju signala kroz RAM-ove na sklopu, budući da su pod *Interni signali* navedeni podatkovni izlazi svih RAM-ova, koji postaju aktivni prilikom čitanja podataka iz tog blok RAM-a.



Slika 50. Simulacija kooperacije više procesorskih jezgri

Na kraju potrebno je kroz provođenje postupaka sinteze i implementacije provjeriti performanse ovog sustava. Za očekivati je kako će maksimalna frekvencija takta biti niža od maksimalnog frekvencije takta zasebnih jezgara, zbog složenog postupka raspoređivanja komponenti i njihovog međusobnog povezivanja budući da se koristi oko 60 posto ukupnog sklopa.

```
Device utilization summary:
---------------------------

Selected Device : 3s500efg320-5

 Number of Slices:                     2875  out of   4656    61%
 Number of Slice Flip Flops:           1440  out of   9312    15%
 Number of 4 input LUTs:               5436  out of   9312    58%
    Number used as logic:              5103
    Number used as RAMs:                333
 Number of IOs:                          45
 Number of bonded IOBs:                  45  out of    232    19%
 Number of BRAMs:                        19  out of     20    95%
 Number of GCLKs:                         1  out of     24     4%
```

Zbog najboljih performansi, pri izgradnji *multi_misc* sustava s višestrukim jezgrama modificirana je i korištena inačica procesora *misc16_f*. U tablici 4 možemo vidjeti rezultate postupaka sinteze i implementacije sklopa *multi_misc*:

Inačica sklopa	Minimalni period		Maksimalni takt	
	sinteza	implementacija	sinteza	implementacija
misc16_f	6.603 ns	7.584 ns	151.447 MHz	131.856 MHz
multi_misc	6.603 ns	8.295 ns	151.447 MHz	120.554 MHz

Tablica 4. Rezultati sinteze i implementacije

Iz tablice 3 možemo primijetiti kako se maksimalna frekvencija signala takta spustila za manje od 10 posto, u odnosu na performanse korištene inačice procesora u slučaju kada je olakšana implementacija, odnosno kada se na sklopu nalazi samo jedna jezgra *misc16_f*.

5. Zaključak

Predstavljena je izvedba procesora MISC16. Procesor posjeduje minimalan skup instrukcija, te je izveden stogovnom arhitekturom, što znači da je, za razliku od RISC i CISC računala, gdje su osnovni mehanizam kontrole podataka i operacija registri, ovdje to stog, odnosno preciznije, nekoliko njih. Osnovni stog, preko kojeg prelaze svi podaci i nad kojim se obavljaju gotovo sve operacije naziva se podatkovni stog. Dodatni stogovi su adresni, čiji najviši element služi za pristup podatkovnoj memoriji, te povratni, na koji se pohranjuje adresa sljedeće instrukcije prilikom pozivanja potprograma. U programskoj memoriji procesora se nalaze pomiješane instrukcije i podaci, i to na način da se u jednom retku memorije nalazi blok od tri instrukcije, a njih slijede podaci vezani uz te instrukcije. Operacija procesora provodi se kroz četiri nivoa protočne strukture, dohvat instrukcijskog bloka, dohvat instrukcije iz bloka, dekodiranje i izvođenje. Uz stogove i progamsku memoriju, procesor sadrži i podatkovnu memoriju. Implementirano je ukupno 30 instrukcija, a broj instrukcija je ograničen na 32 zbog duljine operacijskog koda instrukcije od 5 bitova. Procesor je implementiran na FPGA sklopu porodice Xilinx Spartan-3E, pri čemu su iskorišteni neki namjenski resursi sklopa. U prvom redu radi se o memorijama, pa su tako stogovi izvedeni pomoću paralelnih blokova distribuiranog RAM-a, dok su programska i podatkovna memorija izvedene pomoću blok RAM-a. Prilikom funkcijske simulacije identificirana su neka ograničenja određenih funkcija, te bi stoga bilo izuzetno korisno razviti prevodilac koji bi uzimao u obzir ta ograničenja i time olakšao posao programeru. Nadalje, prilikom sinteze i implementacije sustava identificirani su kritični putevi signala koji ograničavaju brzinu rada cijelog sustava. Stoga su razvijene tri inačice procesora, pri čemu prva inačica, *misc16* sadrži množilo te ima mogućnost obavljanja množenja s akumulacijom u jednom ciklusu signala takta. Druga inačica, *misc16_mul*, isto tako sadrži množilo ali ne posjeduje mogućnost obavljanja množenja s akumulacijom. Uklanjanjem instrukcije za množenje s akumulacijom postignuto je povećanje maksimalne frekvencije signala takta od 17 posto. Treća inačica procesora, *misc16_f*, nema implementirano množilo, no ima mogućnost vrlo brzog rada i poželjno ju je koristiti u slučajevima koji ne zahtjevaju množenje. Uklanjanjem množila postignuto je povećanje brzine

rada od čal 28 posto u odnosu na osnovnu inačicu, *misc16*. Na kraju, budući da jezgra ovog procesora koristi samo oko 6 posto hardvera na sklopu, razvijena je inačica procesora *misc*, koja ima dvostruko veći adresni prostor za podatkovnu memoriju, te se spaja na dva blok RAM resursa na sklopu, čime je omogućeno povezivanje više takvih jezgri u jedinstveni i moćniji sustav koji objedinjuje ukupno 9 ovakvih jezgri. Budući da je time iskorišteno oko 60 posto sklopa, razmještanje i spajanje komponenti je složenije, te se gubi oko 10 posto brzine u odnosu na jednu jezgru.

Potpis: _____

6. Literatura

[1] Minimal instruction set computer, 6. travnja 2012.,
http://en.wikipedia.org/wiki/Minimal_instruction_set_computer
[2] Koopman, P.J., Stack Computers: the new wave, Ellis Horwood, 1989., elektronička verzija,
http://www.ece.cmu.edu/~koopman/stack_computers/index.html
[3] Vučić, M., Molnar, G., Alati za razvoj digitalnih sustava – materijali za predavanja, Zagreb, 2009.
[4] Ashenden, P.J., The Designer's Guide to VHDL, Morgan Kaufmann, 2008.
[5] Xilinx, Spartan-3E FPGA Family: Data Sheet, 2009.
[6] Xilinx, Spartan-3 Generation FPGA User Guide
[7] Xilinx, XST User Guide (for Xilinx ISE 9.2i)
[8] P16 processor in VHDL, 10. rujna 2000.,
http://www.ultratechnology.com/p16vhdl.htm
[9] MSL16 processor,
http://www.cse.cuhk.edu.hk/~phwl/mt/public/archives/old/msl16/msl16.html
[10] Forth (programming language), 24. travnja 2012.,
http://en.wikipedia.org/wiki/Forth_%28programming_language%29#History
[11] Xilinx Training: Basic HDL Coding Techiniques,
http://www.xilinx.com/csi/training/basic-hdl-coding-techniques-part1.htm
[12] Design Assistant for XST: Reasons why a latch is inferred,
http://www.xilinx.com/support/answers/39136.htm
[13] XST – „WARNING:Xst:737“ – Latch found,
http://www.xilinx.com/support/answers/13979.htm
[14] Mentor Graphics, ModelSim Reference Manual (Software version 10.1a)
[15] Xilinx, Spartan-3E Libraries Guide for HDL Designs, 16. rujna, 2009.

7. Sažetak

FPGA izvedba procesora s minimalnim skupom instrukcija

Predstavljena je FPGA implementacija procesora s minimalnim skupom instrukcija (MISC – minimal instruction set computer), ostvarena stogovnom arhitekturom. Platforma za implementaciju je porodica FPGA sklopova Spartan-3, firme Xilinx. Širina instrukcijske riječi je 5 bitova, čime je broj instrukcija ograničen na 32, dok je širina podataka 16 bitova. Zbog karakteristika resursa na sklopu i postizanja veće gustoće koda, u jednom retku programske memorije su upakirane tri instrukcije. Procesor koristi tri stoga: podatkovni, povratni i adresni, i svi su dubine 16 riječi. Memorijski elementi izvedeni su pomoću namjenskih resursa sklopa, na način da su stogovi izvedeni kao distribuirani RAM, a programska i podatkovna memorija kao blok RAM. Implementirana je protočna struktura sa četiri faze: dohvat instrukcijskog bloka, dohvat instrukcije iz bloka, dekodiranje i izvođenje. Analizom pojedinih faza identificirali su se kritični putevi, te su zbog toga razvijene tri inačice kako bi se dobila ravnoteža između opsega performansi i brzine izvođenja. Dodatno, više ovakvih jezgri je povezano kroz dijeljeni memorijski prostor čime je omogućena kooperacija tih jezgri i realiziran je sustav s mnogo većom procesnom moći.

Ključne riječi: MISC, stogovno računalo, Spartan-3, distribuirani RAM, blok RAM, instrukcijski spremnik, protočna struktura, dijeljeni memorijski prostor, kooperacija višestrukih jezgri

FPGA implementation of a minimal instruction set computer

An FPGA implementation of minimal instruction set computer (MISC) is presented. The processor is developed as a stack machine, and the implementation platform is Xilinx's Spartan-3 family. The instruction word is 5 bits wide, and the data is 16 bits in width. Due to the characteristics of Spartan's resources and better code density, one line of program memory contains a package of three instructions. The processor uses three stacks: data, address and return, all of which are 16 words in depth. Memory elements are implemented using Spartan's dedicated resources, so that the stacks are implemented as distributed RAMs, and the program and data memories are implemented as block RAMs. A four-stage pipeline has been implemented: instruction-block fetch, instruction fetch, decode and execute. Through the analysis of the individual stages, critical paths were identified, and therefore three versions of processor have been developed, so that there could be a balance between the scope of performance and executin speed. Additionally, a number of these processor cores have been interconnected through a shared memory space which allows their cooperation and the implementation of a system with much greater processing capabilities.

Keywords: MISC, stack machine, Spartan-3, distributed RAM, block RAM, instruction block, pipeline, shared memory space, cooperation of multiple cores

8. Dodatak A – Procesor *misc16*

```
-- ================================================================================================
--      PROCESOR MISC16 - diplomski rad br. 535, Fakultet elektrotehnike i računarstva
--             Luka Dejanović
--             Mentor: prof. dr. sc. Davor Petrinović
--             Inačica procesora s množilom i MUL i MAC instrukcijama
-- ================================================================================================
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library UNISIM;
use UNISIM.VComponents.all;

library std;
use std.textio.all;

entity misc16 is
       port (clk : in std_logic;
                      reset : in std_logic;
                      podatak : out std_logic_vector(15 downto 0)
                      );
end misc16;

architecture rtl of misc16 is
-- ================================================================================================
--      PROGRAMSKA MEMORIJA - komponenta dvopristupnog ROM-a
-- ================================================================================================
   component ram01 is
      generic  (
               AWidth               : integer := 10;
               rom_content_file     : string := "rom_content.txt"
               );
      port  (
            clk         : in std_logic;
            en          : in std_logic;
            we          : in std_logic;
            addr1       : in std_logic_vector(AWidth-1 downto 0);
            addr2       : in std_logic_vector(AWidth-1 downto 0);
            din         : in std_logic_vector(15 downto 0);
            dout1       : out std_logic_vector(15 downto 0);
            dout2       : out std_logic_vector(15 downto 0)
            );
   end component;

-- ================================================================================================
--                            INSTRUKCIJE I OPERACIJSKI KODOVI
-- ================================================================================================
       constant nop   : std_logic_vector(4 downto 0) := "00000";
       constant push  : std_logic_vector(4 downto 0) := "11100";
       constant call  : std_logic_vector(4 downto 0) := "11101";
       constant jmp   : std_logic_vector(4 downto 0) := "11110";
       constant jz    : std_logic_vector(4 downto 0) := "11111";
       constant pop   : std_logic_vector(4 downto 0) := "00010";
       constant dup   : std_logic_vector(4 downto 0) := "01001";
       constant swap  : std_logic_vector(4 downto 0) := "01010";
       constant ret   : std_logic_vector(4 downto 0) := "00011";
       constant neg   : std_logic_vector(4 downto 0) := "01011";
       constant comp  : std_logic_vector(4 downto 0) := "01100";
       constant add   : std_logic_vector(4 downto 0) := "01101";
       constant sub   : std_logic_vector(4 downto 0) := "01110";
       constant i_and : std_logic_vector(4 downto 0) := "01111";
       constant i_or  : std_logic_vector(4 downto 0) := "01000";
       constant i_xor : std_logic_vector(4 downto 0) := "10100";
       constant shla  : std_logic_vector(4 downto 0) := "10010";
       constant shll  : std_logic_vector(4 downto 0) := "10011";
       constant shra  : std_logic_vector(4 downto 0) := "10000";
       constant shrl  : std_logic_vector(4 downto 0) := "10001";
       constant mul   : std_logic_vector(4 downto 0) := "10101";
       constant mac   : std_logic_vector(4 downto 0) := "10111";
       constant sta   : std_logic_vector(4 downto 0) := "11000";
       constant lda   : std_logic_vector(4 downto 0) := "11001";
```

```
        constant store : std_logic_vector(4 downto 0) := "11010";
        constant sti   : std_logic_vector(4 downto 0) := "11011";
        constant load  : std_logic_vector(4 downto 0) := "00110";
        constant ldi   : std_logic_vector(4 downto 0) := "00111";
        constant str   : std_logic_vector(4 downto 0) := "00100";
        constant ldr   : std_logic_vector(4 downto 0) := "00101";
-- ================================================================================
--             enumeracijski tip i signali vezani uz instrukcije - za simulaciju
-- ================================================================================
        type code_enum is (   op_NOP,  op_PUSH,  op_CALL,  op_JMP,  op_JZ,  op_RET,  op_POP,
op_SWAP,  op_DUP,  op_COMP,  op_NEG,  op_ADD,  op_SUB,  op_I_AND,  op_I_OR,  op_I_XOR,  op_MUL,
op_MAC,  op_SHLA,  op_SHLL,  op_SHRA,  op_SHRL,  op_STA,  op_LDA,  op_STR,  op_LDR,  op_STI,
op_STORE, op_LDI, op_LOAD);
        signal instrukcija_enum : code_enum;
-- ================================================================================
--                                        signali vezani uz programskoj memoriji
-- ================================================================================
        signal en,we          : std_logic;
        signal din            : std_logic_vector(15 downto 0);
        signal instr_buffer   : std_logic_vector(15 downto 0);
        signal data_buffer    : std_logic_vector(15 downto 0);
        signal prog_addr      : std_logic_vector(9 downto 0);
        signal data_addr      : std_logic_vector(9 downto 0);
-- ================================================================================
--             signali za generatore programske i podatkovne adrese
-- ================================================================================
        signal counter               : std_logic_vector(1 downto 0);
        signal instruction_mux       : std_logic_vector(4 downto 0);
        signal code                  : std_logic_vector(4 downto 0);
        signal a, b, c               : std_logic;
        signal L0, L1, L2            : std_logic;
        signal pa_adder              : std_logic_vector(2 downto 0);
        signal pa_add                : std_logic_vector(9 downto 0);
        signal pa_mux                : std_logic_vector(9 downto 0);
        signal prog_addr_mux         : std_logic_vector(9 downto 0);
        signal jump                  : std_logic;
        signal data_inc              : std_logic;
        signal da_adder              : std_logic_vector(9 downto 0);
        signal data_addr_mux         : std_logic_vector(9 downto 0);
        signal counter_rst           : std_logic;
        signal instr_fetch           : std_logic;
        signal data_buffer1          : std_logic_vector(15 downto 0);
        signal ret_yes               : std_logic;
        signal jump_source           : std_logic_vector(9 downto 0);
        signal cnt_rst_jump          : std_logic;
-- ================================================================================
--                            podatkovna memorija
--================================================================================
        type data_mem is array (2**10-1 downto 0) of std_logic_vector(15 downto 0);
        signal DTM                   : data_mem;
        signal dtm_en                : std_logic := '0';
        signal dtm_we                : std_logic := '0';
        signal dtm_reg_out           : std_logic_vector(15 downto 0);
-- ================================================================================
--                            signal vezani uz stogove
-- ================================================================================
        signal data                  : std_logic_vector(15 downto 0);
        signal d_in                  : std_logic_vector(15 downto 0);
        signal d_out                 : std_logic_vector(15 downto 0);
        signal ds_addr               : std_logic_vector(3 downto 0);
        signal d_pointer             : std_logic_vector(3 downto 0);
        signal d_pointer1            : std_logic_vector(3 downto 0);
        signal dw_en                 : std_logic := '0';
--------------------------------------------------------------------------------
        signal r_in                  : std_logic_vector(9 downto 0);
        signal r_out                 : std_logic_vector(9 downto 0);
        signal rs_addr               : std_logic_vector(3 downto 0);
        signal r_pointer             : std_logic_vector(3 downto 0);
        signal r_pointer1            : std_logic_vector(3 downto 0);
        signal rw_en                 : std_logic := '0';
--------------------------------------------------------------------------------
        signal a_in                  : std_logic_vector(9 downto 0);
        signal a_out                 : std_logic_vector(9 downto 0);
        signal as_addr               : std_logic_vector(3 downto 0);
        signal a_pointer             : std_logic_vector(3 downto 0);
        signal a_pointer1            : std_logic_vector(3 downto 0);
        signal aw_en                 : std_logic := '0';
```

```
-- ================================================================================================
--                     signali vezani uz multipleksore i registre
-- ================================================================================================
------------------------------------------------------------------------------------------
       signal data_pointer_operation : std_logic_vector(1 downto 0);
       signal data_pointer_mux       : std_logic_vector(3 downto 0);
       signal data_pointer_mux1      : std_logic_vector(3 downto 0);
       signal data_oper              : std_logic_vector(3 downto 0);
       signal data_mux               : std_logic_vector(15 downto 0);
       signal d_in_oper              : std_logic_vector(1 downto 0);
       signal d_in_mux               : std_logic_vector(15 downto 0);
       signal shift_reg_oper         : std_logic_vector(1 downto 0);
       signal shift_reg_result       : std_logic_vector(15 downto 0);
       signal alu_oper               : std_logic_vector(2 downto 0);
       signal alu_result             : std_logic_vector(15 downto 0);
       signal mul_oper               : std_logic;
       signal mac_result             : std_logic_vector(15 downto 0);
       signal mul_result_long        : std_logic_vector(31 downto 0);
       signal mul_result             : std_logic_vector(15 downto 0);
       signal mul_mac_result         : std_logic_vector(15 downto 0);
------------------------------------------------------------------------------------------
       signal a_in_oper              : std_logic_vector(1 downto 0);
       signal a_in_mux               : std_logic_vector(9 downto 0);
       signal address_pointer_operaton : std_logic_vector(1 downto 0);
       signal address_pointer_mux    : std_logic_vector(3 downto 0);
       signal address_pointer_mux1   : std_logic_vector(3 downto 0);
------------------------------------------------------------------------------------------
       signal r_in_oper              : std_logic_vector(1 downto 0);
       signal r_in_mux               : std_logic_vector(9 downto 0);
       signal return_pointer_operation : std_logic_vector(1 downto 0);
       signal return_pointer_mux     : std_logic_vector(3 downto 0);
       signal return_pointer_mux1    : std_logic_vector(3 downto 0);
       signal zero                   : std_logic;
------------------------------------------------------------------------------------------
begin
-- ================================================================================================
--                                  instanciranje potrebnih komponenti: ROM i stogovi
-- ================================================================================================
rom_c: ram01
      generic map (
                  AWidth                          => 10,
                  rom_content_file          => "misc16_PGM.txt")
      port map (
                                  clk                                 => clk,
                                  en                                  => '1',
                                  we                                  => '0',
                                  addr1                               => prog_addr,
                                  addr2                               => data_addr,
                                  din                                 => din,
                                  dout1                               => instr_buffer,
                                  dout2                               => data_buffer1
               );

-------------------  PODATKOVNI STOG  ----------------------------------------------------
       DATA_STACK : for I in 15 downto 0 generate
              DSTACK : RAM16X1S
                     port map (
                            O => d_out(I),
                            A0 => ds_addr(0),
                            A1 => ds_addr(1),
                            A2 => ds_addr(2),
                            A3 => ds_addr(3),
                            D => d_in(I),
                            WCLK => clk,
                            WE => dw_en
                     );
       end generate DATA_STACK;

-------------------  POVRATNI STOG  ------------------------------------------------------
       RETURN_STACK : for I in 9 downto 0 generate
              RSTACK : RAM16X1S
                     port map (
                            O => r_out(I),
                            A0 => rs_addr(0),
                            A1 => rs_addr(1),
                            A2 => rs_addr(2),
```

```
                        A3 => rs_addr(3),
                        D => r_in(I),
                        WCLK => clk,
                        WE => rw_en
                );
        end generate RETURN_STACK;

-------------------  ADRESNI STOG  ---------------------------------------------------
        ADDRESS_STACK : for I in 9 downto 0 generate
                ASTACK : RAM16X1S
                        port map (
                                O => a_out(I),
                                A0 => as_addr(0),
                                A1 => as_addr(1),
                                A2 => as_addr(2),
                                A3 => as_addr(3),
                                D => a_in(I),
                                WCLK => clk,
                                WE => aw_en
                        );
        end generate ADDRESS_STACK;

-- ============================================================================================
--      FETCH JEDINICA - jedinica za dohvat instrukcija i podataka iz programske memorije
-- ============================================================================================
-- brojac instrukcija u bloku
        process(clk)
        begin
                if rising_edge(clk) then
                        if reset = '1' or counter_rst = '1' then
                                counter <= "00";
                        else
                                counter <= counter + 1;
                        end if;
                end if;
        end process;

-- counter_rst signal - resetira brojac instrukcija
        counter_rst <=  (counter(1) and not counter(0)) or cnt_rst_jump;

-- counter reset zbog skoka
        process(clk)
        begin
                if rising_edge(clk) then
                        if reset = '1' then
                                cnt_rst_jump <= '0';
                        else
                                cnt_rst_jump <= jump;
                        end if;
                end if;
        end process;

-- instrukcijski multipleksor - u ovisnosti o brojacu propusta odredene dijelove
instr_buffera
        process(counter, instr_buffer)
        begin
                case counter is
                        when "00" => instruction_mux <= instr_buffer(14 downto 10);
                        when "01" => instruction_mux <= instr_buffer(9 downto 5);
                        when "10" => instruction_mux <= instr_buffer(4 downto 0);
                        when "11" => instruction_mux <= "00000";
                        when others => null;
                end case;
        end process;

-- code registar - sadrzi instrukciju koja se trenutno izvodi (odnosno dekodira)
        process(clk)
        begin
                if rising_edge(clk) then
                        if reset = '1' then
                                code <= (others => '0');
                        else
                                code <= instruction_mux;
                        end if;
                end if;
        end process;
```

```
-- data_buffer delay 1 cycle
	process(clk)
	begin
		if rising_edge(clk) then
			if reset = '1' then
				data_buffer <= (others => '0');
			else
				data_buffer <= data_buffer1;
			end if;
		end if;
	end process;

-- ================================================================================
--                          generator podatkovne adrese za programsku memoriju
-- ================================================================================
-- data_inc signal - odreduje je li trenutno (u sljedecem ciklusu) izvodena instrukcija, i
-- to provjerom 3 MSB-a instrukcijskog multipleksora
	data_inc <= instruction_mux(4) and instruction_mux(3) and instruction_mux(2);

-- da_adder signal - pribraja data_inc trenutnoj adresi u registru podatkovne adrese
	da_adder <= data_addr + data_inc;

-- data_addr_mux - kod reseta brojila se programska adresa uvecana za 1 stavlja u
-- multipleksor podatkovne adrese
	process(counter_rst, da_adder, prog_addr)
	begin
		case counter_rst is
			when '0' => data_addr_mux <= da_adder;
			when '1' => data_addr_mux <= prog_addr + 1; --prog_addr_mux
			when others => null;
		end case;
	end process;

-- registar podatkovne adrese
	process(clk)
	begin
		if rising_edge(clk) then
			if reset = '1' then
				data_addr <= (0 => '1', others => '0');
			else
				data_addr <= data_addr_mux;
			end if;
		end if;
	end process;

-- ================================================================================
--                          generator instrukcijske adrese za programsku memoriju
-- ================================================================================
-- signali koji oznacavaju "podatkovne" instrukcije u instr_bufferu
	a <= instr_buffer(14) and instr_buffer(13) and instr_buffer(12);
	b <= instr_buffer(9) and instr_buffer(8) and instr_buffer(7);
	c <= instr_buffer(4) and instr_buffer(3) and instr_buffer(2);

-- generiranje signala koji označava broj podataka u bloku instrukcije - suma minterma iz
-- tablice istinitosti
	L2 <= a and b and c;
	L1 <= (not a and not b and c) or
			(not a and b and not c) or
			(not a and b and c) or
			(a and not b and not c) or
			(a and not b and c)   or
			(a and b and not c);
	L0 <= (not a and not b and not c) or
			(not a and b and c) or
			(a and not b and c) or
			(a and b and not c);

-- pa_adder signal - konkatenacija signala koji oznacavaju broj podataka koji slijede
-- instrukcijski blok, uvecan za 1
	pa_adder <= L2 & L1 & L0;

-- pa add signal - zbroj trenutne programske adrese i broja podataka koje ju slijede,
-- uvecan za 1
	pa_add <= prog_addr + pa_adder;

-- Amux0 - treba dodati kada se omoguci povratak iz potprograma
	process(ret_yes, data_buffer, r_in)
```

```
	begin
		case ret_yes is
			when '0' => jump_source <= data_buffer(9 downto 0);
			when '1' => jump_source <= r_in;
			when others => null;
		end case;
	end process;

-- Amux1 - prvi adresni multipleksor, odreduje radi li se o skoku ili o slijednom citanju
	process(jump, pa_add, jump_source)
	begin
		case jump is
			when '0' => pa_mux <= pa_add;
			when '1' => pa_mux <= jump_source;
			when others => null;
		end case;
	end process;

-- instr_fetch signal - signal je jednak 1 kada je brojac instrukcija jednak 1, zbog toga
-- sto prolaskom kroz adresni registar imamo kasnjenje
-- ovog signala za jedan ciklus, odnosno adresa se postavi ciklus kasnije, sto omogucava
-- pravovremeno citanje novog spremnika instrukcija
	instr_fetch <= ((not counter(1) and counter(0)) or jump) and not cnt_rst_jump;

-- Amux2 - drugi adresni multipleksor, odreduje radi li se o citanju novog spremnika iz
-- programske memorije
	process(instr_fetch, prog_addr, pa_mux)
	begin
		case instr_fetch is
			when '0' => prog_addr_mux <= prog_addr;
			when '1' => prog_addr_mux <= pa_mux;
			when others => null;
		end case;
	end process;

-- registar programske adrese
	process(clk)
	begin
		if rising_edge(clk) then
			if reset = '1' then
				prog_addr <= (others => '0');
			else
				prog_addr <= prog_addr_mux;
			end if;
		end if;
	end process;
-- ================================================================================
--			DEKODIRANJE INSTRUKCIJA
-- ================================================================================
	process(clk)
	begin
		if rising_edge(clk) then
			if reset = '1' then
				rw_en <= '0';
				dw_en <= '0';
				aw_en <= '0';
				dtm_en <= '1';
				dtm_we <= '0';
				data_pointer_operation <= (others => '0');
				address_pointer_operation <= (others => '0');
				return_pointer_operation <= (others => '0');
				data_oper <= (others => '0');
				alu_oper <= (others => '0');
				shift_reg_oper <= (others => '0');
				d_in_oper <= (others => '0');
				a_in_oper <= (others => '0');
				r_in_oper <= (others => '0');
				mul_oper <= '0';
				jump <= '0';
				ret_yes <= '0';
			else
				rw_en <= '0';
				dw_en <= '0';
				aw_en <= '0';
				dtm_en <= '1';
				dtm_we <= '0';
				data_pointer_operation <= (others => '0');
```

```
address_pointer_operation <= (others => '0');
return_pointer_operation <= (others => '0');
data_oper <= (others => '0');
alu_oper <= (others => '0');
shift_reg_oper <= (others => '0');
d_in_oper <= (others => '0');
a_in_oper <= (others => '0');
r_in_oper <= (others => '0');
mul_oper <= '0';
jump <= '0';
ret_yes <= '0';
case code is
	when push =>
		dw_en <= '1'; data_pointer_operation <= "01";
		data_oper <= "1000"; d_in_oper <= "01";
		instrukcija_enum <= op_PUSH;
	when pop =>
		data_pointer_operation <= "10";
		data_oper <= "0011"; d_in_oper <= "10";
		instrukcija_enum <= op_POP;
	when call =>
		rw_en <= '1'; return_pointer_operation <= "01";
		r_in_oper <= "10"; jump <= '1';
		instrukcija_enum <= op_CALL;
	when ret =>
		return_pointer_operation <= "10";
		r_in_oper <= "11"; ret_yes <= '1'; jump <= '1';
		instrukcija_enum <= op_RET;
	when neg =>
		data_oper <= "0101"; alu_oper <= "001";
		instrukcija_enum <= op_NEG;
	when comp =>
		data_oper <= "0101"; alu_oper <= "010";
		instrukcija_enum <= op_COMP;
	when dup =>
		dw_en <= '1'; data_pointer_operation <= "01";
		d_in_oper <= "01";
		instrukcija_enum <= op_DUP;
	when swap =>
		data_oper <= "0011"; d_in_oper <= "01";
		instrukcija_enum <= op_SWAP;
	when add =>
		data_oper <= "0101"; alu_oper <= "011";
		data_pointer_operation <= "10";
		d_in_oper <= "10";
		instrukcija_enum <= op_ADD;
	when sub =>
		data_oper <= "0101"; alu_oper <= "100";
		data_pointer_operation <= "10";
		d_in_oper <= "10";
		instrukcija_enum <= op_SUB;
	when i_and =>
		data_oper <= "0101"; alu_oper <= "101";
		data_pointer_operation <= "10";
		d_in_oper <= "10";
		instrukcija_enum <= op_I_AND;
	when i_or =>
		data_oper <= "0101"; alu_oper <= "110";
		data_pointer_operation <= "10";
		d_in_oper <= "10";
		instrukcija_enum <= op_I_OR;
	when i_xor =>
		data_oper <= "0101"; alu_oper <= "111";
		data_pointer_operation <= "10";
		d_in_oper <= "10";
		instrukcija_enum <= op_I_XOR;
	when shla =>
		data_oper <= "0110"; shift_reg_oper <= "00";
		instrukcija_enum <= op_SHLA;
	when shll =>
		data_oper <= "0110"; shift_reg_oper <= "01";
		instrukcija_enum <= op_SHLL;
	when shra =>
		data_oper <= "0110"; shift_reg_oper <= "10";
		instrukcija_enum <= op_SHRA;
	when shrl =>
		data_oper <= "0110"; shift_reg_oper <= "11";
```

```
                                        instrukcija_enum <= op_SHRL;
                                    when mul =>
                                        data_oper <= "0111"; mul_oper <= '1';
                                        data_pointer_operation <= "10";
                                        d_in_oper <= "10";
                                        instrukcija_enum <= op_MUL;
                                    when mac =>
                                        data_oper <= "0111";
                                        data_pointer_operation <= "11";
                                        d_in_oper <= "10";
                                        instrukcija_enum <= op_MAC;
                                    when jmp =>
                                        jump <= '1';
                                        instrukcija_enum <= op_JMP;
                                    when jz =>
                                        jump <= zero;
                                        instrukcija_enum <= op_JZ;
                                    when nop =>
                                        instrukcija_enum <= op_NOP;
                                        --null;
                                    when sta =>
                                        aw_en <= '1'; data_pointer_operation <= "10";
                                        data_oper <= "0011"; d_in_oper <= "10";
                                        address_pointer_operation <= "01";
                                        a_in_oper <= "01";
                                        instrukcija_enum <= op_STA;
                                    when lda =>
                                        dw_en <= '1'; data_pointer_operation <= "01";
                                        data_oper <= "0001"; d_in_oper <= "01";
                                        address_pointer_operation <= "10";
                                        a_in_oper <= "11";
                                        instrukcija_enum <= op_LDA;
                                    when store =>
                                        dtm_en <= '1'; dtm_we <= '1';
                                        data_pointer_operation <= "10";
                                        data_oper <= "0011"; d_in_oper <= "10";
                                        instrukcija_enum <= op_STORE;
                                    when sti =>
                                        dtm_en <= '1'; dtm_we <= '1';
                                        a_in_oper <= "10";
                                        data_pointer_operation <= "10";
                                        data_oper <= "0011"; d_in_oper <= "10";
                                        instrukcija_enum <= op_STI;
                                    when load =>
                                        dtm_en <= '1'; dw_en <= '1';
                                        data_pointer_operation <= "01";
                                        data_oper <= "0100"; d_in_oper <= "01";
                                        instrukcija_enum <= op_LOAD;
                                    when ldi =>
                                        dtm_en <= '1'; dw_en <= '1'; a_in_oper <= "10";
                                        data_pointer_operation <= "01";
                                        data_oper <= "0100"; d_in_oper <= "01";
                                        instrukcija_enum <= op_LDI;
                                    when str =>
                                        rw_en <= '1'; data_pointer_operation <= "10";
                                        data_oper <= "0011"; d_in_oper <= "10";
                                        return_pointer_operation <= "01";
                                        r_in_oper <= "01";
                                        instrukcija_enum <= op_STR;
                                    when ldr =>
                                        dw_en <= '1'; data_pointer_operation <= "01";
                                        data_oper <= "0010"; d_in_oper <= "01";
                                        return_pointer_operation <= "10";
                                        r_in_oper <= "11";
                                        instrukcija_enum <= op_LDR;
                                    when others => null;
                                end case;
                            end if;
                        end if;
                end process;

-- ================================================================================
--                              IZVODENJE INSTRUKCIJA
-- ================================================================================
-- stack pointer select - ovisno radi li se o push ili pop naredbi odreduje koji signal
-- spojiti na adresni ulaz stoga
```

```
	with data_pointer_operation		select	ds_addr	<=	d_pointer	when	"01",
	d_pointer1 when others;
	with return_pointer_operation		select	rs_addr	<=	r_pointer	when	"01",
	r_pointer1 when others;
	with address_pointer_operation		select	as_addr	<=	a_pointer	when	"01",
	a_pointer1 when others;

-- ================================================================================
--                                        generator adresa za pokazivače na stog
-- ================================================================================
-- DATA STACK POINTERS
-- ds_pointer
	process(data_pointer_operation, d_pointer)
	begin
		case data_pointer_operation is
			when "00" => data_pointer_mux <= d_pointer;
			when "01" => data_pointer_mux <= d_pointer + 1;
			when "10" => data_pointer_mux <= d_pointer - 1;
			when "11" => data_pointer_mux <= d_pointer - 2;
			when others => null;
		end case;
	end process;

-- ds_pointer1
	process(data_pointer_operation, d_pointer1)
	begin
		case data_pointer_operation is
			when "00" => data_pointer_mux1 <= d_pointer1;
			when "01" => data_pointer_mux1 <= d_pointer1 + 1;
			when "10" => data_pointer_mux1 <= d_pointer1 - 1;
			when "11" => data_pointer_mux1 <= d_pointer1 - 2;
			when others => null;
		end case;
	end process;

-- synchronous registers
	process(clk)
	begin
		if rising_edge(clk) then
			if reset = '1' then
				d_pointer <= (others => '0');
			else
				d_pointer <= data_pointer_mux;
			end if;
		end if;
	end process;

	process(clk)
	begin
		if rising_edge(clk) then
			if reset = '1' then
				d_pointer1 <= (others => '1');
			else
				d_pointer1 <= data_pointer_mux1;
			end if;
		end if;
	end process;

-- ADDRESS STACK POINTERS
-- as_pointer
	process(address_pointer_operation, a_pointer)
	begin
		case address_pointer_operation is
			when "00" => address_pointer_mux <= a_pointer;
			when "01" => address_pointer_mux <= a_pointer + 1;
			when "10" => address_pointer_mux <= a_pointer - 1;
			when others => null;
		end case;
	end process;

-- as_pointer1
	process(address_pointer_operation, a_pointer1)
	begin
		case address_pointer_operation is
			when "00" => address_pointer_mux1 <= a_pointer1;
			when "01" => address_pointer_mux1 <= a_pointer1 + 1;
			when "10" => address_pointer_mux1 <= a_pointer1 - 1;
```

```
                        when others => null;
                end case;
        end process;

-- synchronous registers
        process(clk)
        begin
                if rising_edge(clk) then
                        if reset = '1' then
                                a_pointer <= (others => '0');
                        else
                                a_pointer <= address_pointer_mux;
                        end if;
                end if;
        end process;

        process(clk)
        begin
                if rising_edge(clk) then
                        if reset = '1' then
                                a_pointer1 <= (others => '1');
                        else
                                a_pointer1 <= address_pointer_mux1;
                        end if;
                end if;
        end process;

-- RETURN STACK POINTERS
-- rs_pointer
        process(return_pointer_operation, r_pointer)
        begin
                case return_pointer_operation is
                        when "00" => return_pointer_mux <= r_pointer;
                        when "01" => return_pointer_mux <= r_pointer + 1;
                        when "10" => return_pointer_mux <= r_pointer - 1;
                        when others => null;
                end case;
        end process;

-- rs_pointer1
        process(return_pointer_operation, r_pointer1)
        begin
                case return_pointer_operation is
                        when "00" => return_pointer_mux1 <= r_pointer1;
                        when "01" => return_pointer_mux1 <= r_pointer1 + 1;
                        when "10" => return_pointer_mux1 <= r_pointer1 - 1;
                        when others => null;
                end case;
        end process;

-- synchronous registers
        process(clk)
        begin
                if rising_edge(clk) then
                        if reset = '1' then
                                r_pointer <= (others => '0');
                        else
                                r_pointer <= return_pointer_mux;
                        end if;
                end if;
        end process;

        process(clk)
        begin
                if rising_edge(clk) then
                        if reset = '1' then
                                r_pointer1 <= (others => '1');
                        else
                                r_pointer1 <= return_pointer_mux1;
                        end if;
                end if;
        end process;
-- ================================================================================
--                      ulazi u podatkovni stog, i pripadni registri
-- ================================================================================
-- multiplier
        mul_result_long <= data * d_in;
```

```
	mul_result <= mul_result_long(15 downto 0);

-- MAC unit
	mac_result <= mul_result + d_out;
	mul_mac_result <= mul_result when mul_oper='1' else mac_result;

-- shift register
	process(shift_reg_oper, data)
	begin
		case shift_reg_oper is
			when "00" =>  shift_reg_result <= data(15) & data(13 downto 0) & '0';
			when "01" =>   shift_reg_result <= data(14 downto 0) & '0';
			when "10" =>   shift_reg_result <= data(15) & data(15 downto 1);
			when "11" =>   shift_reg_result <= '0' & data(15 downto 1);
			when others => shift_reg_result <= (others => '0');
		end case;
	end process;

-- ALU
	process(alu_oper, data, d_in)
	begin
		case alu_oper is
			when "000" =>  alu_result <= (others => '0');
			when "001" =>  alu_result <= not data + 1;
			when "010" =>  alu_result <= not data;
			when "011" =>  alu_result <= data + d_in;
			when "100" =>  alu_result <= data - d_in;
			when "101" =>  alu_result <= data and d_in;
			when "110" =>  alu_result <= data or d_in;
			when "111" =>  alu_result <= data xor d_in;
			when others => alu_result <= (others => '0');
		end case;
	end process;

-- DATAMEM
	process(clk)
	begin
		if rising_edge(clk) then
			if dtm_en = '1' then
				if dtm_we = '1' then
					DTM(conv_integer(a_in)) <= data;
				else
					dtm_reg_out <= DTM(conv_integer(a_in));
				end if;
			end if;
		end if;
	end process;


-- data input multiplexer
	process(data_oper, data, data_buffer, a_in, r_in, d_in, dtm_reg_out, alu_result,
		shift_reg_result, mul_mac_result)
	begin
		case data_oper is
			when "0000" => data_mux <= data;--null;--data_mux <= (others => '0');
			when "0001" => data_mux <= "000000" & a_in;
			when "0010" => data_mux <= "000000" & r_in;
			when "0011" => data_mux <= d_in;
			when "0100" => data_mux <= dtm_reg_out;
			when "0101" => data_mux <= alu_result;
			when "0110" => data_mux <= shift_reg_result;
			when "0111" => data_mux <= mul_mac_result;
			when "1000" => data_mux <= data_buffer;
			when others => data_mux <= (others => '0');
		end case;
	end process;

-- data register
	process(clk)
	begin
		if rising_edge(clk) then
			if reset = '1' then
				data <= (others => '0');
			else
				data <= data_mux;
			end if;
		end if;
```

```
	end process;

-- d_in register
	process(d_in_oper, d_in, data, d_out)
	begin
		case d_in_oper is
			when "00" => d_in_mux <= d_in;
			when "01" => d_in_mux <= data;
			when "10" => d_in_mux <= d_out;
			when others => null;
		end case;
	end process;

	process(clk)
	begin
		if rising_edge(clk) then
			if reset = '1' then
				d_in <= (others => '0');
			else
				d_in <= d_in_mux;
			end if;
		end if;
	end process;

-- ================================================================================
--		ulazi u adresni stog, i pripadni registri
-- ================================================================================
	process(a_in_oper, a_in, data, a_out)
	begin
		case a_in_oper is
			when "00" => a_in_mux <= a_in;
			when "01" => a_in_mux <= data(9 downto 0);
			when "10" => a_in_mux <= a_in + 1;
			when "11" => a_in_mux <= a_out;
			when others => null;
		end case;
	end process;

	process(clk)
	begin
		if rising_edge(clk) then
			if reset = '1' then
				a_in <= (others => '0');
			else
				a_in <= a_in_mux;
			end if;
		end if;
	end process;

-- ================================================================================
--			ulazi u povratni stog, i pripadni registri
-- ================================================================================

	process(r_in_oper, r_in, data, pa_add, r_out)
	begin
		case r_in_oper is
			when "00" => r_in_mux <= r_in;
			when "01" => r_in_mux <= data(9 downto 0);
			when "10" => r_in_mux <= pa_add;--reg;
			when "11" => r_in_mux <= r_out;
			when others => null;
		end case;
	end process;

	process(clk)
	begin
		if rising_edge(clk) then
			if reset = '1' then
				r_in <= (others => '0');
			else
				r_in <= r_in_mux;
			end if;
		end if;
	end process;

-- zero zastavica
```

```
        zero <= not ((data(15) or data(14) or data(13) or data(12)) or (data(11) or data(10)
                or data(9) or data(8)) or (data(7) or data(6) or data(5) or data(4)) or
                (data(3) or data(2) or data(1) or data(0)));

-- izlaz
        podatak <= data;

end rtl;
```

9. Dodatak B – Datoteka za funkcijsku simulaciju rada procesora

```
quit -sim

vlib work
vcom -explicit  -93 "ram01.vhd"
vcom -explicit  -93 "misc16.vhd"
vcom -explicit  -93 "misc16_tb.vhd"
vsim -t 10ps   -lib work misc16_tb_vhd


add wave -noupdate -divider {misc16}

add wave -noupdate -divider {Upravljacki signali}
add wave -noupdate -format Logic -color green -label {Clock} /misc16_tb_vhd/clk
add wave -noupdate -format Logic -color orange -label {Reset} /misc16_tb_vhd/reset

add wave -noupdate -divider {Adrese za citanje}
add wave -noupdate -format Literal -radix unsigned -label {Adresa instrukcija}
       /misc16_tb_vhd/uut/prog_addr
add wave -noupdate -format Literal -radix unsigned -label {Adresa podatka}
       /misc16_tb_vhd/uut/data_addr

add wave -noupdate -divider {Instrukcije}
add wave -noupdate -format Logic -label {Spremnik instrukcija}
       /misc16_tb_vhd/uut/instr_buffer
add wave -noupdate -format Literal -radix unsigned -label {Brojac instrukcija}
       /misc16_tb_vhd/uut/counter
add wave -noupdate -format Logic -label {Instrukcija} /misc16_tb_vhd/uut/code
add wave -noupdate -height 40 -color #00ff00 -label {INSTRUKCIJA}
       /misc16_tb_vhd/uut/instrukcija_enum

add wave -noupdate -divider {Podatkovni spremnik}
add wave -noupdate -format Literal -radix unsigned -label {data_buffer}
       /misc16_tb_vhd/uut/data_buffer

add wave -noupdate -divider {Podatkovni stog}
add wave -noupdate -format Literal -radix unsigned -label {DS0} /misc16_tb_vhd/uut/data
add wave -noupdate -format Literal -radix unsigned -label {DS1} /misc16_tb_vhd/uut/d_in

add wave -noupdate -divider {Povratni stog}
add wave -noupdate -format Literal -radix unsigned -label {RS0} /misc16_tb_vhd/uut/r_in
add wave -noupdate -format Literal -radix unsigned -label {RS1} /misc16_tb_vhd/uut/r_out

add wave -noupdate -divider {Adresni stog}
add wave -noupdate -format Literal -radix unsigned -label {AS0} /misc16_tb_vhd/uut/a_in
add wave -noupdate -format Literal -radix unsigned -label {AS1} /misc16_tb_vhd/uut/a_out

add wave -noupdate -divider {izlaz}
add wave -noupdate -format Logic -color white -label {izlaz} /misc16_tb_vhd/uut/output

run 200ns
```