SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

ZAVRŠNI RAD br. 174

# Ugradnja instrukcije množenja na procesoru P16 s minimalnim skupom instrukcija

Luka Karlović

Zagreb, lipanj 2008.

# Sadržaj

# 1. Uvod

Programabilna logika uvela je svijet elektronike u novo doba.
Razvoj na razini ploče (*eng. board-level design*) dobio je moderniju, prihvatljiviju i bolju alternativu – razvoj na razini integriranog kruga (*eng. chip-level design*).
Takav pristup omogućio je brži razvoj proizvoda, mnogo opsežnije mogućnosti rekonfiguracije uređaja i razvoj vlastitih kompleksnih rješenja po prihvatljivim cijenama.

Razvoj programabilne logike bio je postupan, od najjednostavnijih programabilnih sklopova do vrlo kompleksne programabilne logike sposobne za implementaciju više različitih složenih sustava na jednom integriranom krugu.

Najnaprednija programabilna tehnologija današnjice je FPGA (*eng. Field Programmable Gate Array*) porodica sklopova.
Ova tehnologija se danas najbrže razvija i sve više koristi u mnogim segmentima elektronike, pa tako nalazi primjenu i u nekim područjima do sada rezerviranim isključivo za analognu domenu (npr. SDR, *eng. Software Defined Radio*).

Programabilna logika je u početku korištena isključivo za implementaciju logičkih funkcija, kombinatoričkih ili sekvencijalnih, manje složenih ili više složenih.
Porastom kompleksnosti ovih sklopova (CPLD, *eng. Complex Programmable Logic Device*, a posebice FPGA) omogućena je implementacija kompletnih računalnih sustava na jednom programabilnom sklopu.
Više nije potrebno razvijenu mikroprocesorsku jezgru proizvoditi tehnološkim postupkom u pogonu za proizvodnju silicijskih integriranih krugova, već je dovoljno projektirani dizajn mikroprocesora prenijeti u programabilni logički sklop visoke kompleksnosti.
Ovakav pristup razvoju računalnih sustava omogućuje jeftiniji razvoj, a ne sprječava kasnije prebacivanje identičnog dizajna na klasični silicijski integrirani krug u proizvodnom pogonu (za potrebe masovne proizvodnje).

Procesori koji su potpuno programski opisani (a fizičke implementacije postoje isključivo na programabilnom sklopu) se često nazivaju procesorima sa mekom jezgrom (*eng. soft-core processor*). U nastavku će biti korišten engleski naziv. Izvorni kod ovakvih procesora (pisan najčešće u jezicima VHDL ili Verilog) je intelektualno vlasništvo njegovog autora, a sukladno tome se javlja i engleski naziv *IP Core* (*eng. Intellectual Property Core*).

Predmet ovog dokumenta je implementacija procesora P16 na Spartan-3 FPGA porodici programabilnih sklopova.
Procesor P16 je 16-bitovni stogovni procesor niskog stupnja kompleksnosti (procesor sa minimalnim skupom instrukcija).
Implementacija procesora izvedena je u jeziku za opis sklopovlja VHDL.

U sljedećim poglavljima biti će opisana Spartan-3 porodica programabilnih sklopova, arhitektura procesora P16, te razna arhitekturna proširenja procesora P16 sa ciljem ugradnje instrukcije množenja na taj procesor.

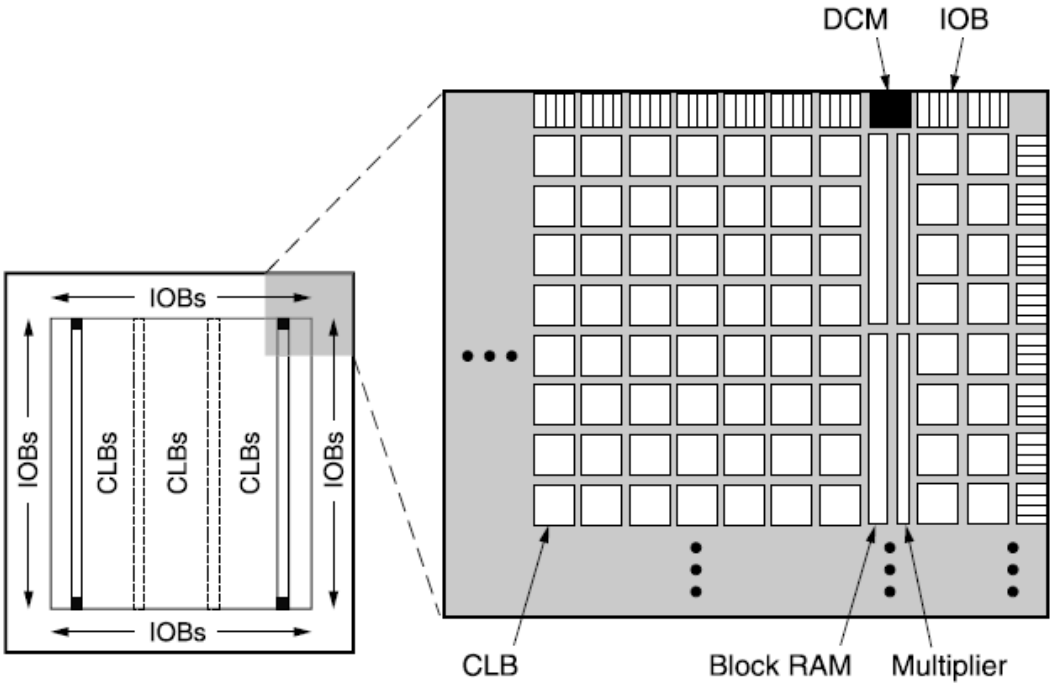
# 2. Programabilni logički sklopovi porodice FPGA

FPGA sklopovi su građeni od logičkih blokova međusobno povezanih poveznicama i prospojnom mrežom.
Detalji implementacije FPGA sklopa ovise o proizvođaču i o porodici sklopa.
U nastavku će biti opisana Xilinx Spartan-3 porodica FPGA sklopova, koja služi kao ciljna platforma za implementaciju procesora P16 opisanog u sljedećim poglavljima.
Programiranje FPGA sklopovlja se vrši pomoću specijaliziranih jezika za opis sklopovlja (HDL, *eng. Hardware Description Language*). Najpoznatiji HDL jezici su Verilog i VHDL (*eng. Very High Speed Integrated Circuit HDL*).

Svi konfiguracijski elementi FPGA sklopa su izvedeni u SRAM (*eng. Static RAM*) tehnici, pa se konfiguracija sklopa briše gubitkom napajanja. Zbog toga svi FPGA sklopovi u svojoj okolini imaju konfiguracijsku ROM memoriju sa cjelovitom konfiguracijom sklopa, koja se učitava (puni) pri uključenju FPGA sklopa.

## 2.1 Organizacija Spartan-3 FPGA sklopa

Svaki Spartan-3 FPGA sklop je izgrađen od nekoliko različitih tipova blokova. Ovi blokovi mogu biti: konfigurabilni logički blokovi, ulazno izlazni blokovi, ugrađena množila, RAM blokovi, te blokovi za digitalnu kontrolu signala vremenskog vođenja.



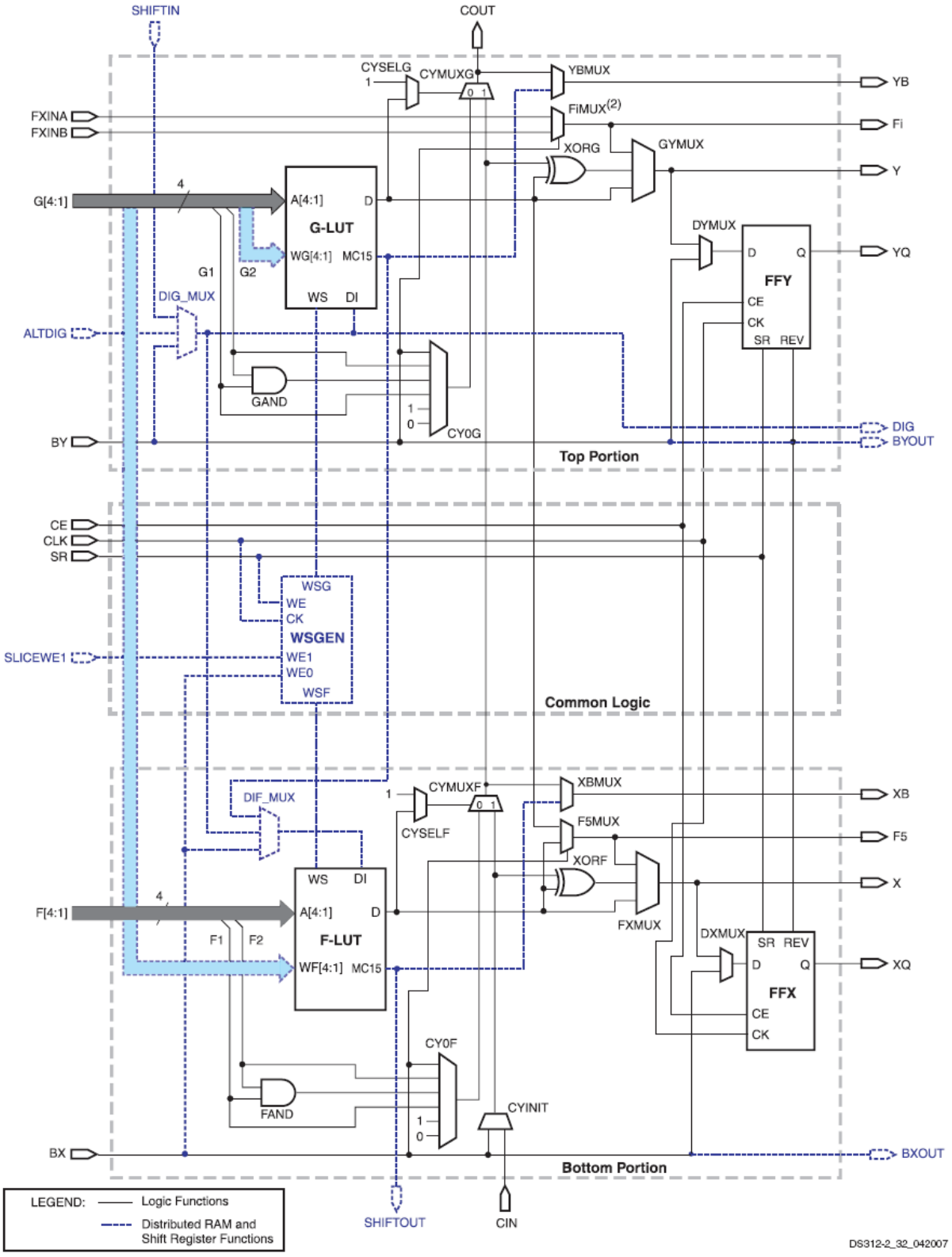

Slika 2.1.1: Spartan-3 FPGA sklop, [2], *Architectural Overview*

Slika 2.1.1 prikazuje unutarnju organizaciju Spartan-3 FPGA sklopa. Konfigurabilni logički blokovi (CLB) su najbrojniji po tipu (oni su glavna programabilna komponenta za ostvarenje željene logičke funkcije).
Uz vanjske rubove sklopa su smješteni ulazno izlazni blokovi (IOB). Ulazno izlazni blokovi su direktno vezani na vanjske priključke sklopa i omogućuju razmjenu informacija sa okolinom sklopa.
U unutrašnjosti sklopa se nalaze posebni stupci sa sklopovski implementiranim komponentama specijaliziranim za posebne operacije. Broj takvih stupaca ovisi o veličini Spartan-3 komponente. Stupci se sastoje od RAM blokova za pohranu većih količina podataka i od množila. RAM blokovi i množila su smješteni fizički u neposrednoj blizini kako bi bilo moguće brzo izvođenje matematički intenzivnih operacija množenja nad većom količinom podataka u RAM blokovima.

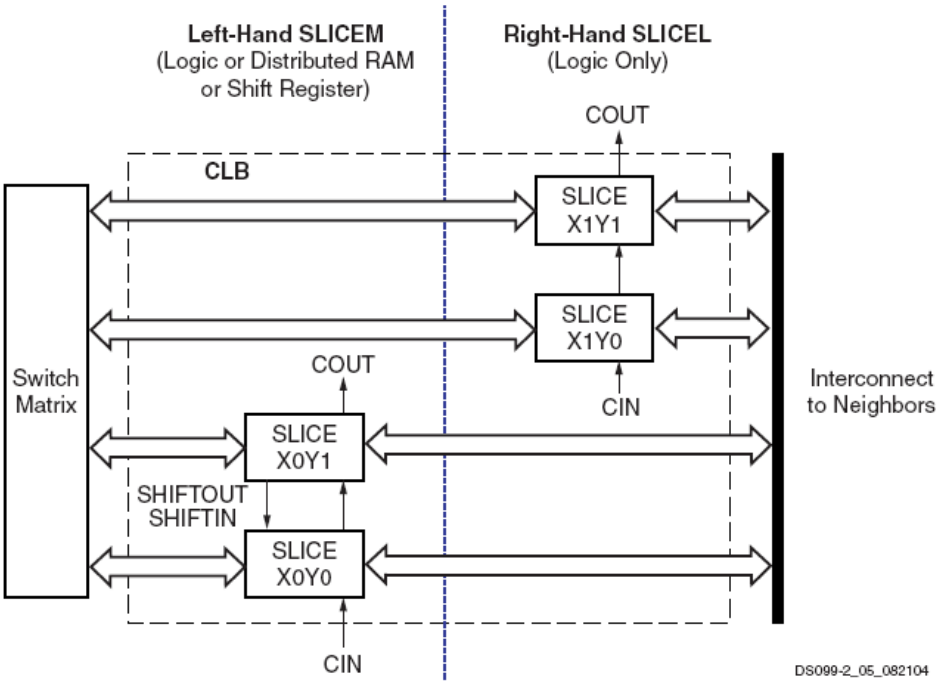
Stupci su uglavnom smješteni u blizini ulazno izlaznih blokova, kako bi bila moguća brza razmjena većih količina podataka sa okolinom.

## 2.2 Konfigurabilni logički blokovi

Konfigurabilni logički blok (CLB, *eng. Configurable Logic Block*) je osnovna programabilna komponenta za ostvarenje željene kombinatoričke ili sekvencijalne funkcije.
Jedan CLB blok se sastoji od četiri podcjeline (*eng. slice*) grupirane u parove podcjelina: lijevi i desni par (*eng. slicem, slicel*).
Svaka od četiri podcjeline sadrži dva generatora logičkih funkcija, dva elementa za pohranu, multipleksore za višestruku namjenu, logiku za zbrajanje i bit prijenosa.
Podcjeline (parovi podcjelina) nisu po funkcionalnosti u potpunosti identični. Lijevi par (*slicem*) ima dodatne mogućnosti koje mu omogućuju tablice generatora logičkih funkcija (*eng. lookup table*), realizirane u tehnici statičkog RAM-a.
Dodatne mogućnosti lijevog para podcjelina uključuju korištenje tablica generatora logičkih funkcija kao distribuirane RAM memorije za pohranu podataka ili kao posmačnog registra.
Slika 2.2.1 prikazuje shematski prikaz lijevog para podcjelina (*eng. slicem*) unutar CLB bloka.
Slika 2.2.2 prikazuje blok shemu CLB bloka, sa prikazom svih podcjelina (*eng. slices*) CLB bloka.




Slika 2.2.1: Shematski prikaz lijevog para podcjelina (*slicem*) CLB bloka, [2], *CLB Overview*




Slika 2.2.2: Blokovski prikaz podcjelina (*slices*) unutar CLB bloka, [2], *CLB Overview*

## 2.3 Ulazno izlazni blokovi

Ulazno izlazni blokovi (IOB, *eng. Input Output Block*) su programabilni blokovi za upravljanje razmjenom informacija sa okruženjem sklopa. IOB blokovi su grupirani u ulazno izlazne banke (svaka strana sklopa je podijeljena na dvije banke), te se sastoje od tri grane: ulazne, izlazne i grane visoke impedancije. Svaka od tri grane ima svoj par elemenata za pohranu.

Ulazna grana prenosi podatak sa priključka sklopa preko opcionalne linije za kašnjenje do interne logike. Izlazna grana prenosi podatak sa interne logike preko multipleksora i izlaznog pojačala sa tri stanja prema priključku sklopa.

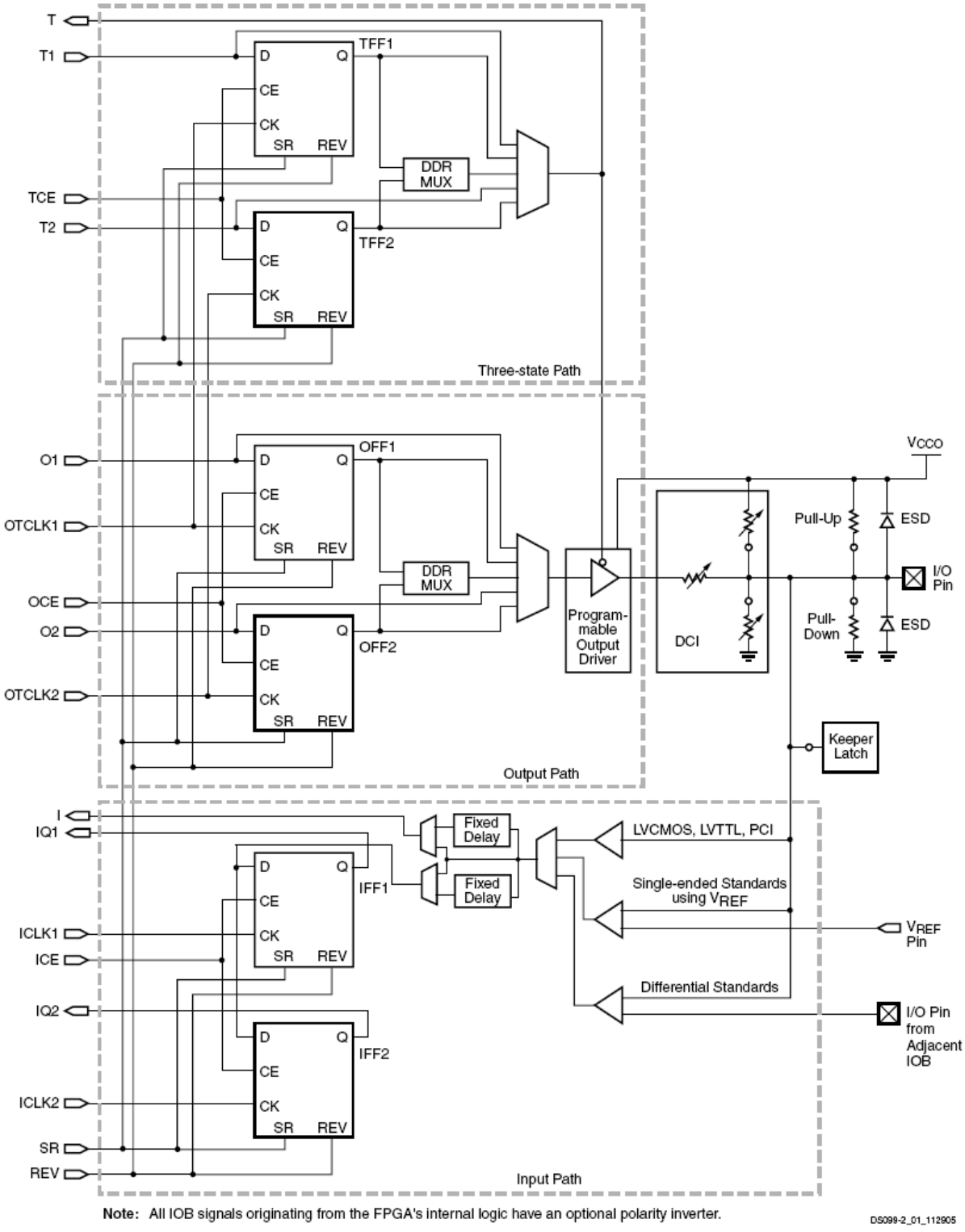
Grana visoke impedancije određuje kada će se izlazno pojačalo nalaziti u stanju visoke impedancije.

IOB blok podržava prijenos podataka na oba brida signala vremenskog vođenja (DDR, *eng. Double Data Rate*) korištenjem parova elemenata za pohranu u izlaznoj ili grani visoke impedancije.

IOB blokovi imaju ugrađenu podršku za mnoštvo naponskih standarda, standarda za diferencijalni prijenos podataka, digitalnu kontrolu impedancije,

zaštitu protiv oštećenja zbog elektrostatskog pražnjenja, invertiranje bilo koje linije za podatke, te mnoštvo dodatnih mogućnosti.

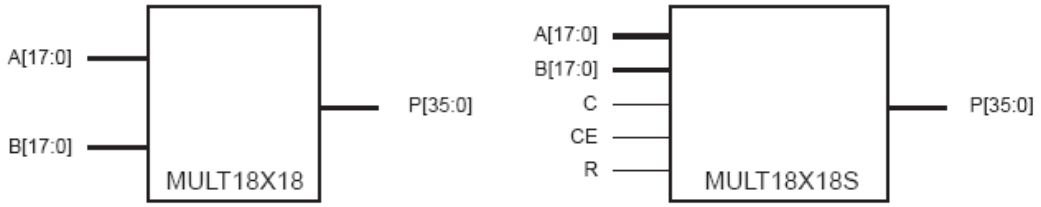
Slika 2.3.1 prikazuje pojednostavljeni shematski prikaz IOB bloka.




Slika 2.3.1: Shematski prikaz ulazno izlaznog bloka, [2], *IOB Overview*

## 2.4 Ugrađena množila

Ugrađena množila (*eng. embedded multipliers*) su sklopovski implementirani blokovi specijalizirani za predznačno ili množenje bez predznaka dvaju ulaznih operanada  širine do 18 bita, gdje se rezultat vraća u preciznosti od 36 bita. Množila su usko povezana sa RAM blokovima. Netipična širina ulaznih operanada od 18 bita odgovara širini RAM blokova, sa bitovima pariteta. Arhitektura sabirnica i smještaj množila u neposrednoj blizini RAM blokova omogućava brze i intenzivne operacije množenja nad većim količinama podataka.
Množila mogu raditi u asinkronom i sinkronom načinu rada. Prikaz blok sheme množila sa ulazima i izlazima u asinkronoj i sinkronoj izvedbi je prikazan na slici 2.4.1.




Slika 2.4.1 : Ugrađeno množilo u asinkronoj i sinkronoj izvedbi, [3], *Data Flow*

Ugrađena množila se mogu koristiti i za alternativne primjene (sklop za posmak, sklop za računanje apsolutne vrijednosti broja, sklop za pretvorbu u oblik dvojnog komplementa), a moguća su i ulančavanja kako bi se postiglo množenje operanada većih širina od 18 bita.
Kako bi se u jeziku VHDL omogućilo korištenje ugrađenog množila, potrebno je instancirati komponentu MULT18X18 (u slučaju asinkronog množila), odnosno komponentu MULT18X18S u slučaju asinkronog množila (kako je prikazano na slici 2.4.1).
Odsječak koda 2.4.1 prikazuje primjer instanciranja asinkronog množila MULT18X18 u jeziku za opis sklopovlja VHDL.

```
-- Component Declaration for MULT18X18 should be placed
-- after architecture statement but before begin keyword

component MULT18X18
port ( P : out STD_LOGIC_VECTOR (35 downto 0);
       A : in STD_LOGIC_VECTOR (17 downto 0);
       B : in STD_LOGIC_VECTOR (17 downto 0));
end component;

-- Component Attribute specification for MULT18X18
-- should be placed after architecture declaration but
-- before the begin keyword
-- Attributes should be placed here
-- Component Instantiation for MULT18X18 should be placed
-- in architecture after the begin keyword

MULT18X18_INSTANCE_NAME : MULT18X18
port map (
             P => user_P,
             A => user_A,
             B => user_B
             );
```

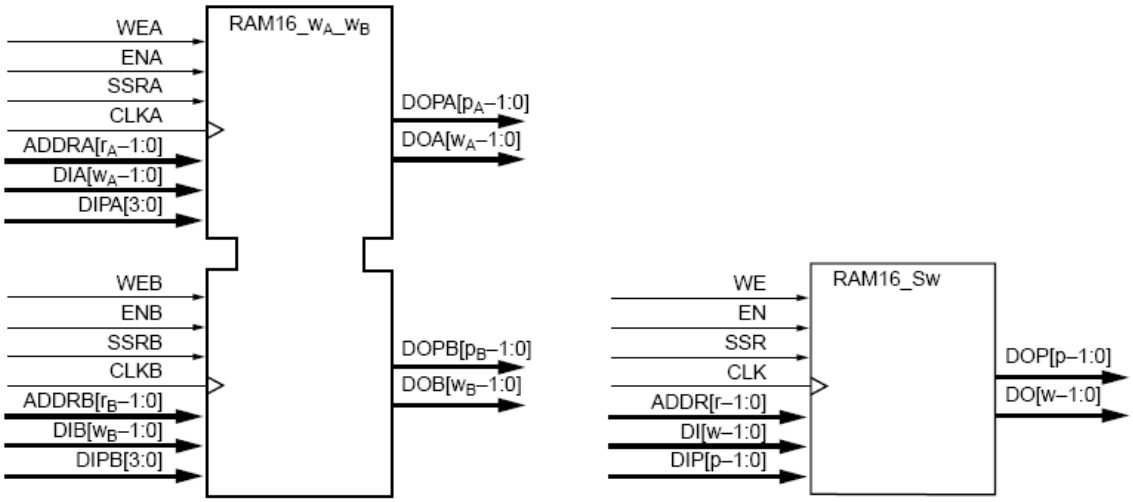
Odsječak koda 2.4.1: Instanciranje množila MULT18X18 u jeziku VHDL, [3], *VHDL Instantiation Template*

Kao i većina blokova u FPGA sklopovima, ugrađeno množilo posjeduje mogućnost korisnički definiranog lokacijskog ograničenja, pomoću kojeg korisnik može instancirati točno određeno množilo u sklopu, umjesto prepuštanja odabira alatu za sintezu. Također je omogućena i implementacija ekvivalentnog množila korištenjem CLB blokova, umjesto instanciranja sklopovski implementiranog množila, no ovakav tip množila nema gotovo nikakvih prednosti nad sklopovski implementiranim ekvivalentom.
Ugrađena množila koriste prilagođeni Boothov algoritam za izračun produkta.

## 2.5 RAM blokovi

RAM blokovi su sklopovski implementirane komponente specijalizirane za pohranu većih količina podataka. Realizirani su kao statički RAM (SRAM) blokovi.
Kapacitet svakog RAM bloka je 18 Kib, od čega je 16 Kib namijenjeno za pohranu podataka, a 2 Kib za pohranu paritetnih informacija (alternativno iskoristivih za pohranu podataka).
Broj RAM blokova unutar jednog FPGA sklopa je ovisan o veličini sklopa.
Svakom RAM bloku je pridruženo ugrađeno množilo.
Pristup RAM bloku je moguć preko dva potpuno nezavisna sinkrona priključka.
Uporaba oba priključka nije nužna, moguće je korištenje samo jednog od njih.

Slika 2.5.1 prikazuje priključke RAM blokova u oba slučaja.




Slika 2.5.1: RAM blok u izvedbi sa dva i jednim priključkom, [5], *Introduction*

Moguće su različite konfiguracije broja memorijskih riječi ovisno o dužini memorijske riječi. Popis mogućih konfiguracija je prikazan u tablici 2.5.1 za memorijsku konfiguraciju sa jednim priključkom, odnosno u tablici 2.5.2 za memorijsku konfiguraciju sa dva priključka. Svaki priključak može imati svoju konfiguraciju, a moguća su i ulančavanja više RAM blokova zbog postizanja alternativnih memorijskih konfiguracija sa većom širinom riječi ili većim brojem lokacija. Instanciranje RAM bloka se vrši ovisno o željenoj konfiguraciji priključaka (broj priključaka, dužina riječi, broj memorijskih lokacija).

Pri instanciranju RAM bloka sa jednim priključkom koristi se ime RAM16_Sw, gdje je w širina memorijske riječi tog priključka (uključujući paritetne bitove).

Pri instanciranju RAM bloka sa dva priključka koristi se ime RAM16_$w_A$_$w_B$ gdje su $w_A$ i $w_B$ širine memorijskih riječi na priključcima A i B, uključujući paritetne bitove.

Tablica 2.5.1: Podržane konfiguracije RAM bloka sa jednom priključnicom, [5], *Memory Organization/Aspect Ratio*

| Organizacija | Broj riječi | Širina riječi | Paritetni bitovi | Podatkovne linije | Paritetne linije | Adresne linije | Ime instance | Sveukupan broj bitova |
|---|---|---|---|---|---|---|---|---|
| **512bit x36** | 512 | 32 | 4 | (31:0) | (3:0) | (8:0) | RAMB16_S36 | 18Kbit |
| **1Kbit x18** | 1024 | 16 | 2 | (15:0) | (1:0) | (9:0) | RAMB16_S18 | 18Kbit |
| **2Kbit x9** | 2048 | 8 | 1 | (7:0) | (0:0) | (10:0) | RAMB16_S9 | 18Kbit |
| **4Kbit x4** | 4096 | 4 | - | (3:0) | - | (11:0) | RAMB16_S4 | 18Kbit |
| **8Kbit x2** | 8192 | 2 | - | (1:0) | - | (12:0) | RAMB16_S2 | 18Kbit |
| **16Kbit x1** | 16384 | 1 | - | (0:0) | - | (13:0) | RAMB16_S1 | 18Kbit |

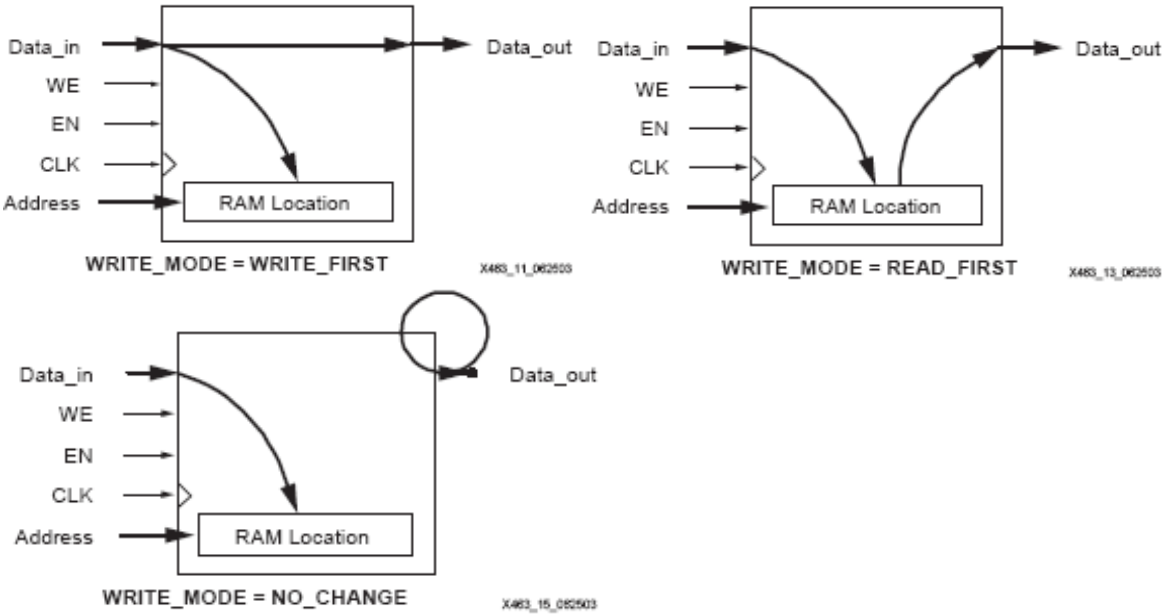
Tablica 2.5.2: Podržane konfiguracije RAM bloka sa dva priključka
(ispred imena instance se dodaje prefiks RAMB16), [5], *Memory Organization/Aspect Ratio*

| | | **Priključak A** | | | | | |
|---|---|---|---|---|---|---|---|
| | | 16Kbit x1 | 8Kbit x2 | 4Kbit x4 | 2Kbit x9 | 1Kbit x18 | 512bit x36 |
| **Priključak B** | 16Kbit x1 | _S1_S1 | | | | | |
| | 8Kbit x2 | _S1_S2 | _S2_S2 | | | | |
| | 4Kbit x4 | _S1_S4 | _S2_S4 | _S4_S4 | | | |
| | 2Kbit x9 | _S1_S9 | _S2_S9 | _S4_S9 | _S9_S9 | | |
| | 1Kbit x18 | _S1_S18 | _S2_S18 | _S4_S18 | _S9_S18 | _S18_S18 | |
| | 512bit x36 | _S1_S36 | _S2_S36 | _S4_S36 | _S9_S36 | _S18_S36 | _S36_S36 |

Kako se operacije na priključku memorije odvijaju sinkrono, moguće je definirati tri različita ponašanja čitanja u tijeku pisanja (*eng. read while write*) pomoću atributa WRITE_MODE. Ukoliko je atribut postavljen na WRITE_FIRST, u trenutku zapisivanja novog podatka u memoriju taj isti podatak se odmah prosljeđuje na izlaz tog priključka. Ukoliko je atribut postavljen na READ_FIRST, u trenutku zapisivanja novog podatka u memoriju stari podatak sa adresirane lokacije se prosljeđuje na izlaz tog priključka, a novi podatak se upisuje na adresiranu lokaciju. Postavljanjem atributa na NO_CHANGE neće doći do nikakve promjene podatka na izlazu priključka koji vrši zapisivanje. Opisane situacije su ilustrirane na slici 2.5.2.

Ukoliko se u istom trenutku sa oba priključka pokuša pisati na istu lokaciju, interno sklopovlje će spriječiti oštećenja sklopovlja, a podatak na adresiranoj lokaciji će postati neispravan.

Kao i većina blokova u FPGA sklopovima, RAM blok posjeduje mogućnost korisnički definiranog lokacijskog ograničenja, pomoću kojeg korisnik može instancirati točno određeni RAM blok u sklopu, umjesto prepuštanja odabira alatu za sintezu.




Slika 2.5.2: Čitanje za vrijeme pisanja sa različitim postavkama atributa WRITE_MODE (WRITE_FIRST, READ_FIRST, NO_CHANGE), [5], *Read Behavior During Simultaneous Write*

## 2.6 Distribuirani RAM

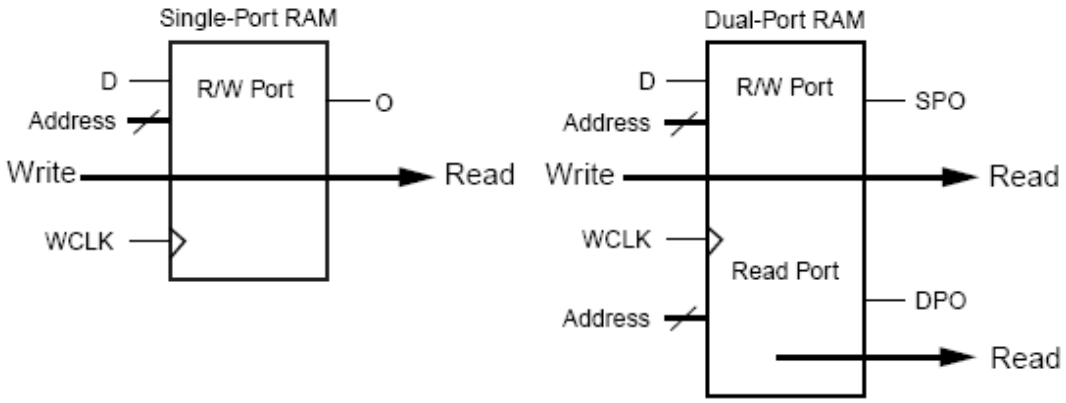
Svaka od četiri podcjeline u konfigurabilnom logičkom bloku (CLB) može realizirati dvije logičke funkcije sa 4 ulazne varijable. Takve logičke funkcije se implementiraju pomoću tablice generatora logičkih funkcija (*eng. lookup table*). Tablica pojedinog generatora logičke funkcija je realizirana kao statički RAM kapaciteta 16 bitova (logička funkcija sa 4 ulaza).

Distribuirani RAM omogućuje iskorištavanje tablica generatora logičkih funkcija kao brze, lokalizirane RAM memorije (malog kapaciteta).

Svaki konfigurabilni logički blok ima 8 generatora logičkih funkcija, no samo generatori u lijevom paru podcjelina (*eng. slicem*) podržavaju funkcionalnost distribuirane RAM memorije, što daje ukupni kapacitet distribuirane RAM memorije u jednom CLB-u od 64 bita.

Dodatno ograničenje kapaciteta proizlazi iz odabira načina rada distribuirane RAM memorije, koja može raditi kao memorija sa dva priključka ili kao memorija sa jednim priključkom.

Ukoliko memorija radi sa dva priključka, moguće je korištenje samo 32 bita distribuirane RAM memorije unutar jednog CLB bloka. Sa jednim priključkom je moguće iskoristiti 64 bita distribuirane memorije.
Korištenje dva priključka podrazumijeva jedan priključak za sinkrono zapisivanje i asinkrono čitanje, te jedan asinkroni priključak za čitanje.
Korištenje jednog priključka podrazumijeva jedan priključak za sinkrono zapisivanje i asinkrono čitanje.
U oba slučaja moguće je postići sinkrono čitanje uz pomoć komponente za pohranu u svakoj podcjelini (*slice*).
Ulančavanje CLB blokova je moguće za dobivanje distribuirane RAM memorije veće širine riječi i većeg broja riječi.
Slika 2.6.1 prikazuje distribuirani RAM u izvedbi sa jednim i dva priključka.
Tablica 2.6.1 prikazuje podržane načine rada distribuirane RAM memorije unutar jednog CLB bloka.




Slika 2.6.1: Distribuirani RAM sa jednim i dva priključka, [4], *Single-Port and Dual-Port RAMs*

Tablica 2.6.1: Podržani načini rada distribuirane RAM memorije, [4], *Library Primitives*

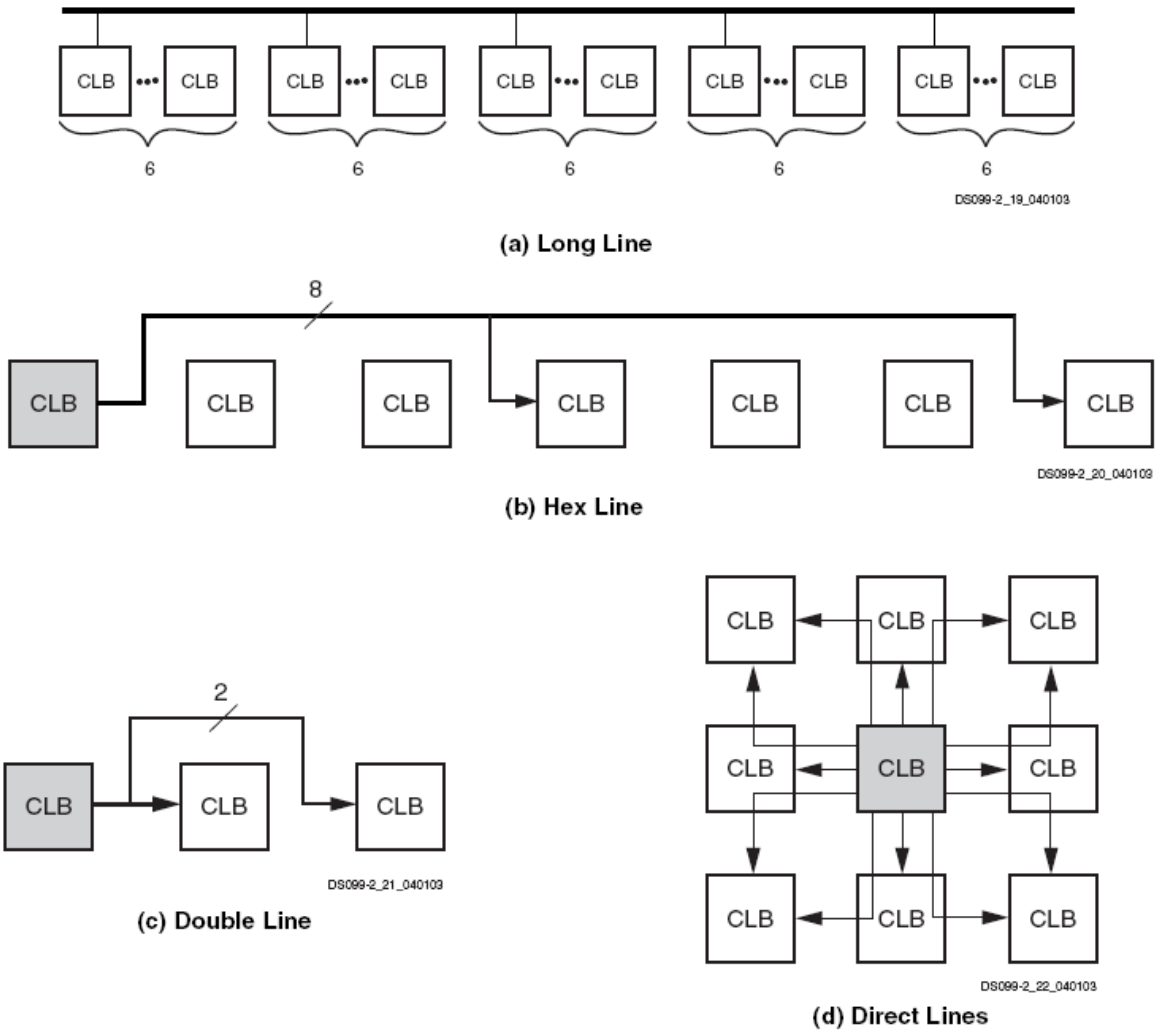
| Ime instance | Broj riječi x dužina riječi | Broj priključaka | Adresne linije |
|---|---|---|---|
| **RAM16X1S** | 16x1 | Jedan | A3,A2,A1,A0 |
| **RAM32X1S** | 32x1 | Jedan | A4,A3,A2,A1,A0 |
| **RAM64X1S** | 64x1 | Jedan | A5,A4,A3,A2,A1,A0 |
| **RAM16X1D** | 16x1 | Dva | A3,A2,A1,A0 |

## 2.7 Blokovi za digitalnu kontrolu signala vremenskog vođenja

Većina sklopova porodice Spartan-3 posjeduje 4 bloka za digitalnu kontrolu signala vremenskog vođenja (DCM). Ovi blokovi omogućavaju sintezu različitih frekvencija, unošenje faznog pomaka i stabilizaciju signala vremenskog vođenja. Funkcionalnost DCM sklopa je izvedena pomoću povratne petlje sa kašnjenjem (*eng. delay-locked loop*).
Mreža za distribuciju signala vremenskog vođenja se sastoji od osam nezavisnih mreža vrlo niskog kapaciteta koje omogućavaju dovođenje vrlo brzih signala vremenskog vođenja iz bilo kojeg DCM sklopa do bilo kojeg konfigurabilnog logičkog bloka.

## 2.8 Poveznice i prospojna mreža

Sve komponente Spartan-3 arhitekture se povezuju korištenjem poveznica i prospojnih mreža.
Prospojne mreže (*eng. switch matrix*) su konfigurabilna spojna mjesta koja povezuju razne poveznice i blokove.
Poveznice su spojni vodovi, grupirani u 4 kategorije.
Poveznice tipa *long* su grupe od 24 linije koje se rasprostiru horizontalno i vertikalno, a povezuju svaki šesti blok. Zbog malog kapaciteta su idealne za brze signale.
Poveznice tipa *hex* su grupe od 8 linija koje se rasprostiru horizontalno i vertikalno a povezuju svaki treći blok. Samo blok na kraju linije može upravljati signalima na liniji.
Poveznice tipa *double* su grupe od 2 linije koje se rasprostiru u svim smjerovima (horizontalno, vertikalno, dijagonalno) a povezuju svaki drugi blok.
Poveznice tipa *direct* su vodovi koji povezuju blok sa svim susjednim blokovima.
Slika 2.8.1 prikazuje sve tipove poveznica i način povezivanja CLB blokova.




Slika 2.8.1: Poveznice (*long*, *hex*, *double* i *direct* poveznice), [2], *Interconnect*

# 3. Procesor P16 s minimalnim skupom instrukcija

Procesor P16 je *soft-core* procesor.

Ciljna platforma za implementaciju procesora je Xilinx Spartan-3 FPGA sklop.

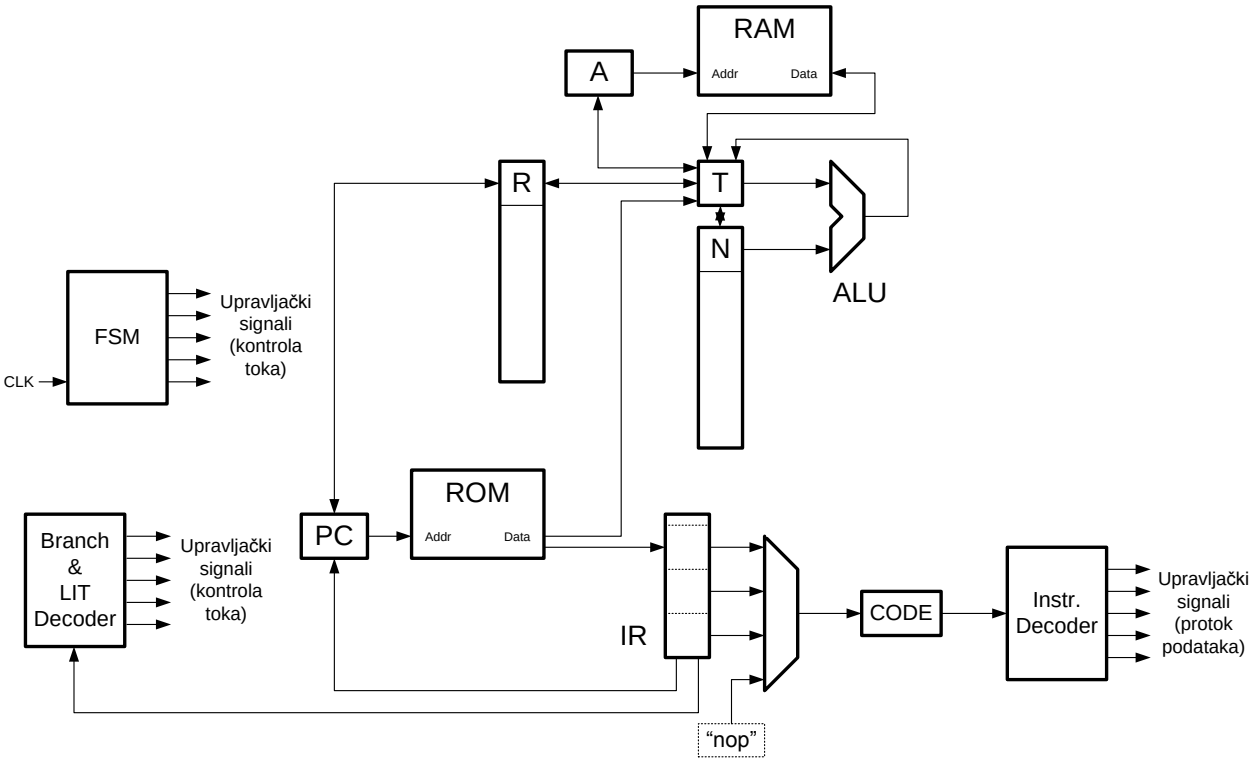
Procesor izvorno podržava samo 26 instrukcija i svrstava se u skupinu procesora sa minimalnim skupom instrukcija (MISC, *eng. Minimal Instruction Set Computer*).

Izvorni skup instrukcija procesora P16 je prikazan u tablici 3.1.

Tablica 3.1: Izvorni skup instrukcija procesora P16, [1], *Set instrukcija*

| Instrukcija | Kod | Opis |
|---|---|---|
| ***jmp*** | 00000 | bezuvjetni skok s 10 bitnim argumentom |
| ***ret*** | 00100 | povratak iz potprograma, povratna adresa iz R se upisuje u P |
| ***jz*** | 00010 | uvjetni skok s 10 bitnim argumentom ako je registar T=0 |
| ***jnc*** | 00011 | uvjetni skok s 10 bitnim argumentom ako je C=0 (ako nije bilo prelijeva) |
| ***call*** | 00001 | poziv potprograma sa 10 bitnim argumentom |
| ***zer*** | 01000 | „brisanje“ T registra, (upisivanje vrijednosti 0) |
| ***lit*** | 01010 | neposredno učitavanje sljedeće programske riječi iz ROM-a |
| ***ld*** | 01100 | dohvat korištenjem adresnog registra; u T se stavlja vrijednost iz RAM-a na adresi iz adresnog registra |
| ***ldp*** | 01001 | dohvat korištenjem adresnog registra uz povećanje vrijednosti adresnog registra za 1 |
| ***st*** | 01111 | pohrana korištenjem adresnog registra; u RAM na adresu iz adresnog registra sprema se vrijednost iz registra T |
| ***stp*** | 01101 | pohrana korištenjem adresnog registra uz povećanje vrijednosti adresnog registra za 1 |
| ***shl*** | 10001 | pomak vrijednosti T registra za jedan bit u lijevo |
| ***shr*** | 10010 | pomak vrijednosti T registra za jedan bit u desno, uključujući bit za preljev |
| ***addc*** | 10011 | uvjetno nedestruktivno (vrijednost na vrhu stoga N ostaje nepromijenjena) zbrajanje vrijednosti iz T i N ako LSB od T ima vrijednost 1 |
| ***xorr*** | 10100 | ekskluzivno ILI vrijednosti iz T i N |
| ***andd*** | 10101 | logičko I vrijednosti iz T i N |
| ***addd*** | 10111 | zbrajanje vrijednosti iz T i N |
| ***subb*** | 10000 | oduzimanje vrijednosti T od N |
| ***lda*** | 11001 | spremanje vrijednosti iz adresnog registra A u registar T |
| ***sta*** | 11101 | spremanje vrijednosti iz registra T u adresnog registra A |
| ***pop*** | 11000 | spremanje vrijednosti iz registra T na vrh stoga R |
| ***push*** | 11100 | spremanje vrijednosti sa vrha stoga R u registar T |
| ***dup*** | 11010 | kopiranje vrijednosti registra T na vrh stoga N |
| ***over*** | 11011 | zamjena vrijednosti T registra i vrijednosti vrha stoga N |
| ***drop*** | 11111 | kopiranje vrijednosti sa vrha stoga N u registar T; stari T se „briše“ |
| ***nop*** | 11110 | bez operacije; kašnjenje za jedan period takta |

Stogovna arhitektura procesora omogućava uporabu relativno malog broja instrukcija i nisku kompleksnost sklopovlja.
Izvorna izvedba procesora sadrži dva stoga, podatkovni stog i stog za povratne adrese pozvanih potprograma. Prikaz izvorne arhitekture procesora P16 je prikazan na slici 3.1. Stogovi su implementirani korištenjem distribuirane RAM memorije Spartan-3 sklopa. Svaki stog ima 16 riječi širine 16 bita, a za izgradnju jednog stoga je potrebno 8 CLB blokova, jer se u implementaciji stoga distribuirani RAM koristi u načinu rada sa dva priključka.



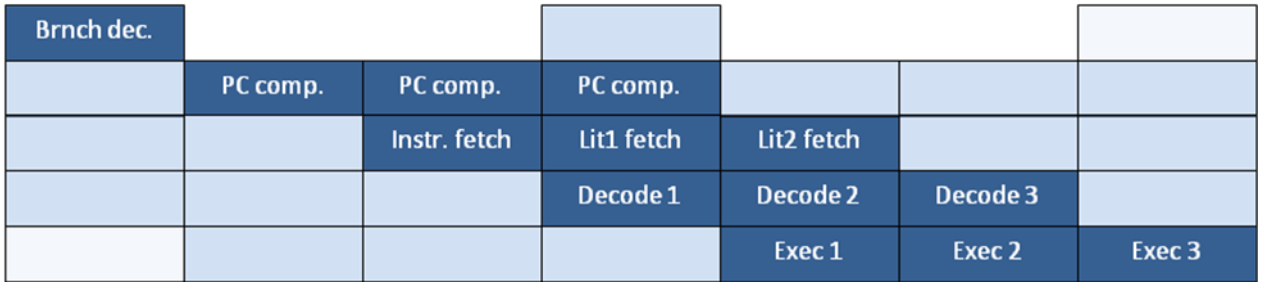

Slika 3.1: Izvorna arhitektura procesora P16, [1], *Organizacija procesora*

Dva RAM bloka Spartan-3 sklopa (u konfiguraciji 16 bita x 1024) su iskorišteni kao podatkovna i programska memorija, svaka kapaciteta 1024 riječi.
Podatkovna memorija kapaciteta 1024 memorijske riječi je adekvatne veličine za primjenu zbog visoke gustoće koda i male veličine izvršnog programa za stogovnu arhitekturu procesora P16.

Širina podatkovne riječi procesora je 16 bita. Širina instrukcijske riječi procesora je 5 bita, a instrukcije su grupirane u skupine od 3 instrukcije.

U fazi dohvata dohvaća se cijela memorijska riječ (16 bita) koja sadrži jednu grupu od 3 instrukcije.

Skokovi u izvođenju programa su dozvoljeni samo na diskretnim mjestima u kodu, na granici između dvije grupe od 3 instrukcije.
Grananje unutar grupe nije dozvoljeno. O strukturi grupe instrukcija i grupiranju ne treba voditi računa programer, već se taj zadatak obavlja automatski za vrijeme prevođenja korištenjem pripadnog prevodioca.
Procesor ima protočnu strukturu sa 5 protočnih segmenata (slika 3.2).
U protočnoj strukturi se mogu javiti različiti hazardi, koji se rješavaju na odgovarajuće načine.
Upravljački hazard je izbjegnut korištenjem detekcije instrukcija koje mijenjaju tijek izvođenja programa još u fazi dohvata grupe instrukcija. Nakon što određena grupa instrukcija krene sa izvođenjem, više nema opasnosti od upravljačkog hazarda jer grananje unutar paketa instrukcija nije dozvoljeno.
Podatkovne i strukturne hazarde rješava prevodilac u fazi prevođenja.
Situacije u kojima dolazi do hazarda se rješavaju pri prevođenju automatskim umetanjem NOP instrukcija u izvršni program.
Sve ostale situacije problematične zbog zahtjeva na strukturu grupe instrukcija također rješava prevodilac umetanjem NOP instrukcija pri prevođenju.




Slika 3.2: Protočna struktura procesora P16, [1], *Protočna arhitektura*

Osim prevodioca, programska podrška procesora uključuje i alat za ugradnju prevedenog izvršnog programa u VHDL kod (u inicijalizacijsku strukturu programskog RAM bloka).

Procesor sa okolinom komunicira preko ulazno izlaznog sučelja *IOPoller.*

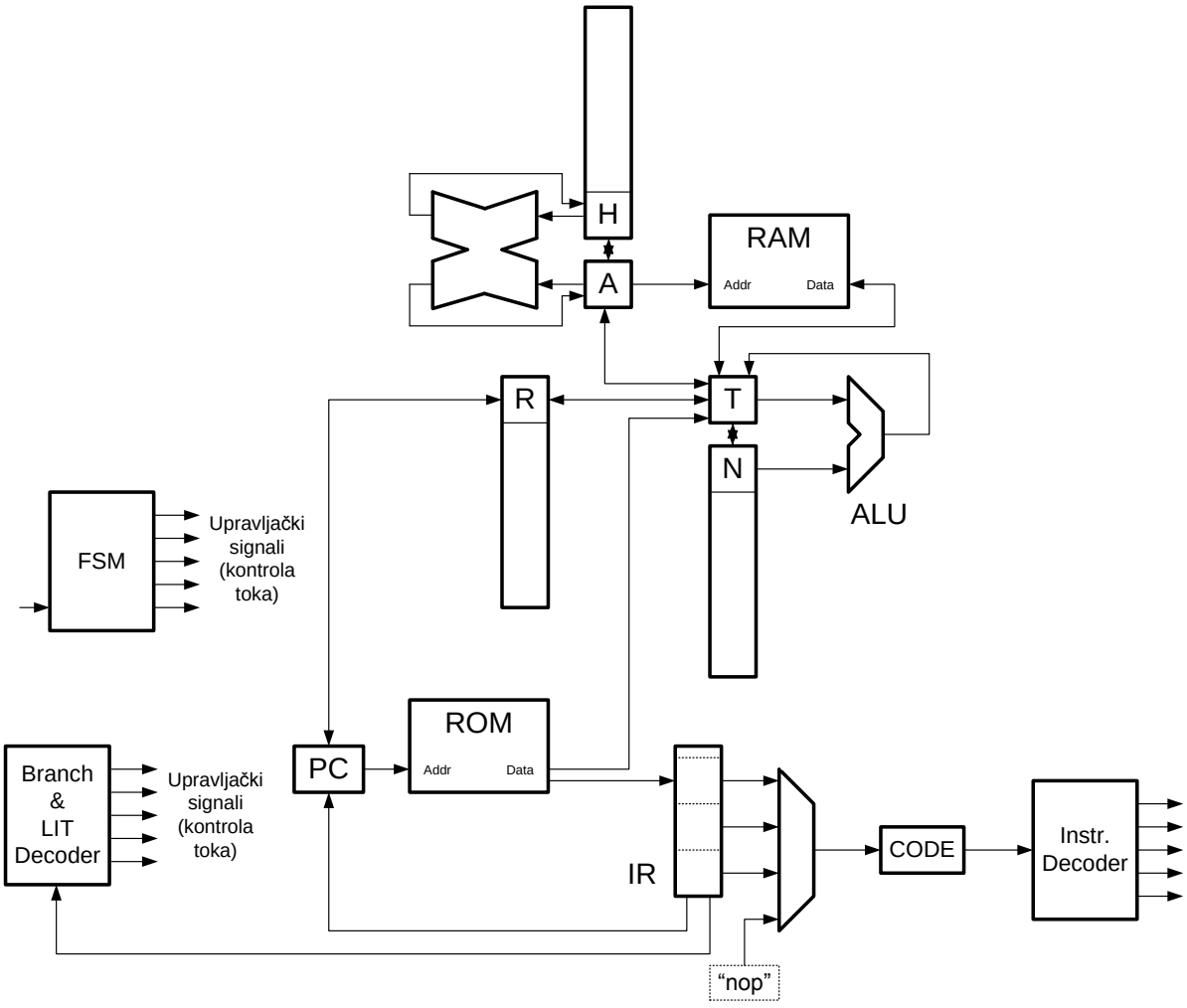
# 4. Ugradnja instrukcije množenja na procesor P16

Operacije množenja na izvornoj izvedbi procesora P16 su resursno i računski intenzivne operacije koje se svode na uzastopno zbrajanje.
Ciljna platforma za implementaciju procesora P16 (Spartan-3 FPGA) posjeduje ugrađena, sklopovski implementirana množila koja se mogu ugraditi u arhitekturu procesora P16 sa ciljem ubrzanja izvršavanja računskih operacija množenja.
Ugradnja instrukcije množenja uključuje nekoliko koraka opisanih u nastavku.

## 4.1 Smještaj množila u P16 arhitekturi

Procesor P16 sve računske operacije izvodi nad podatkovnim stogom (stog N sa vrhom stoga T, slika 3.1).
U idealnom slučaju množilo bi trebalo biti priključeno upravo na ovom mjestu. Operandi za množenje bi se dohvaćali sa podatkovnog stoga, dok bi rezultat u punoj 32-bitnoj preciznosti bio vraćen na podatkovni stog, efektivno prebrisujući operande množenja.
Zbog arhitekturnih ograničenja Spartan-3 arhitekture (nedostatak resursa i poveznica u okolini vrha podatkovnog stoga T, komponente arhitekture procesora P16) opisana izvedba ugradnje množila nije moguća.
Množilo mora biti smješteno na alternativnu lokaciju unutar arhitekture procesora, no izmjene na arhitekturi moraju biti minimalne i ne smiju utjecati na skup postojećih instrukcija.
Podsustav za pristup RAM memoriji je arhitekturna komponenta na kojoj je takvo proširenje najjednostavnije implementirati.
Na slici 3.1 je prikazana podatkovna (RAM) memorija i pomoćni A registar koji adresira lokaciju unutar podatkovne memorije.
Da bi se pristupilo lokaciji u podatkovnoj memoriji, potrebno je u registar A učitati adresu lokacije (prethodno generiranu na vrhu podatkovnog stoga), a zatim pokrenuti pristup podatkovnoj memoriji (razmjenu informacija između adresirane lokacije i vrha podatkovnog stoga).

Dodatno, korištenjem specijalnih instrukcija moguće je automatski povećati adresu u adresnom registru A pri pristupu podatkovnoj memoriji (korištenje pri slijednom pristupu podatkovnim lokacijama).
Izvornu specifikaciju sustava za pristup podatkovnoj memoriji je potrebno proširiti na način da adresni registar A postane vrh novog stoga (adresnog stoga).
Množilo se povezuje na prve dvije lokacije adresnog stoga. Operandi se uzimaju sa vrha adresnog stoga i sa prve lokacije ispod vrha stoga, a rezultat množenja se u punoj 32-bitnoj preciznosti vraća na adresni stog prebrisujući ulazne argumente.
Pojednostavljeni prikaz procesora P16 sa arhitekturnim proširenjem sustava za pristup podatkovnoj memoriji je prikazan na slici 4.1.1.




Slika 4.1.1: Arhitekturna proširenja na procesoru P16, temeljeno na [1], *Organizacija procesora*

Pri zapisivanju rezultata množenja na adresni stog, nižih 16 bita rezultata se zapisuju na vrh adresnog stoga (A), a viših 16 bita se zapisuju na prvu lokaciju ispod vrha (H).
Ovakav način zapisivanja rezultata na adresni stog je povoljniji pri množenju manjih brojeva kako bi bio moguć brži dohvat rezultata (prijenos rezultata sa vrha adresnog stoga na vrh podatkovnog stoga).
Pretpostavlja se da će u većem broju operacija množenja duljina rezultata biti manja od 16 bita, stoga je odabran način koji pogoduje bržem izvođenju takvih operacija.

## 4.2 VHDL implementacija arhitekturnih proširenja

Adresni stog je implementiran pomoću distribuirane RAM memorije.
Cijeli podsustav za množenje i adresni stog su smješteni unutar podsustava procesora za pristup podatkovnoj memoriji.
U arhitekturi procesora P16 se već pojavljuje struktura stog, pa je adresni stog dobiven instanciranjem već postojeće implementacije stoga (odsječak koda 4.2.1).

```
component Stack is
port ( F_CLK, M_CLK   : in std_logic;
         CLR : in std_logic;

         CNTE    : in std_logic;
         PUSH : in std_logic;
         SP_ACTION: in std_logic;

         S_IN   : in std_logic_vector(15 downto 0);
         S1_OUT  : out std_logic_vector(15 downto 0);
         S2_OUT  : out std_logic_vector(15 downto 0)
       );
end component;

A_Stack_inst: component Stack
port map (
         F_CLK => F_CLK,
         M_CLK => M_CLK,
         CLR => CLR,

         CNTE => CNTE,
         PUSH => PUSH,
         SP_ACTION => sp_action,

         S_IN => Stack_in,
         S1_OUT => Stack_out,
         S2_OUT => Stack_out2
);
```

Odsječak koda 4.2.1: Instanciranje adresnog stoga (modules/RAM_Access_Unit.vhd)

Adresni stog se sastoji od vrha adresnog stoga (A) i ostatka adresnog stoga (sa vrhom H).
Postojeći adresni registar A je iskorišten kao vrh adresnog stoga.
Odsječak koda 4.2.1 prikazuje instanciranje ostatka adresnog stoga (H) kao univerzalne komponente stog.
Na univerzalnoj komponenti stog su izvršene prilagodbe kako bi takav stog bio prikladan za operacije množenja. Problem sa univerzalnim stogom se javlja u trenutku množenja, kada je potrebno dohvatiti jedan od operanada sa ostatka adresnog stoga (H) te u istom ciklusu prebrisati taj operand sa viših 16 bita rezultata množenja. Interni pokazivač vrha stoga (*eng. stack pointer*) bi za vrijeme obavljanja ovih operacija trebao zadržati konstantnu vrijednost (bez promjene), što u izvornoj implementaciji nije podržano.
Dodatna komplikacija je postojanje dvostrukog pokazivača vrha stoga u izvornoj implementaciji stoga (iskorištavaju se oba priključka distribuirane RAM memorije stoga). Pokazivač za čitanje uvijek pokazuje na podatak na vrhu stoga, dok pokazivač za pisanje uvijek pokazuje na prvo slobodno mjesto na vrhu stoga, kako bi stog uvijek bio spreman za čitanje i za pisanje.
Problem je riješen nadogradnjom implementacije osnovnog stoga dodavanjem dodatnog ulaznog kontrolnog signala (SP_ACTION, odsječak koda 4.2.1).

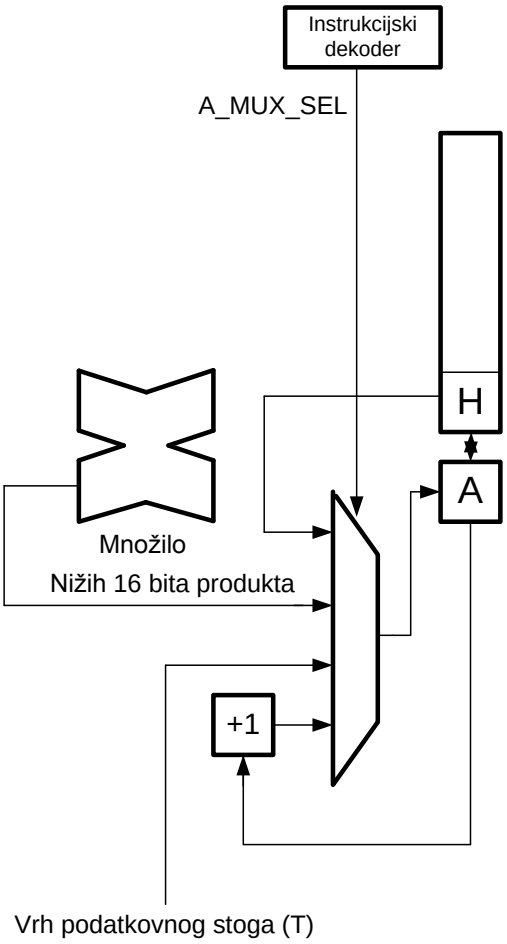
```
-- up/down counteri za generiranje adresa za citanje i pisanje
process (F_CLK)
begin
	if falling_edge(F_CLK) then
		if CLR = '1' then
			AR <= (others => '0');
			AW <= (0 => '1', others => '0');

		else
			if CNTE = '1' and SP_ACTION = '1' then
				if PUSH='1' then
					AR <= AR + 1;
					AW <= AW + 1;
				else
					AR <= AR - 1;
					AW <= AW - 1;
				end if;
			end if;
		end if;
	end if;
end process;

with SP_action select
	AW_out <= AW when '1',
		  AR when others;
```

Odsječak koda 4.2.2: Izmjene u internoj implementaciji stoga (modules/stack.vhd)

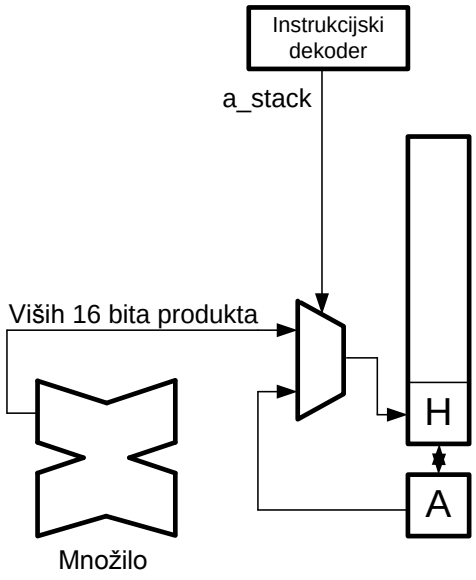
Odsječak koda 4.2.2 prikazuje izmjene u internoj implementaciji stoga.
Ako je ulazni signal SP_ACTION u stanju logičke jedinice, ponašanje stoga je identično ponašanju prije izmjena na implementaciji.
Ako je ulazni signal SP_ACTION u stanju logičke nule, generatori adresa pokazivača vrha stoga miruju (adrese se ne mijenjaju), a dodatno pokazivač za zapisivanje se privremeno izjednačuje sa pokazivačem za čitanje.
Za potrebe izjednačavanja pokazivača stoga za zapisivanje sa pokazivačem stoga za čitanje je uveden novi signal AW_out, koji nosi ulogu efektivnog vrha za pisanje stoga (umjesto prijašnjeg signala AW).
Dodatni ulazni kontrolni signal SP_ACTION se postavlja u podsustavu koji koristi usluge stoga. Svi prethodni sustavi gdje se koristi instanca stoga su prilagođeni na način da signal SP_ACTION trajno spajaju na logičku jedinicu, kako bi ponašanje instance stoga za te sustave bilo nepromijenjeno.
U sustavu za pristup podatkovnoj memoriji (gdje se koristi instanca stoga kao adresni stog) se vrijednost ulaznog signala SP_ACTION definira ovisno o upravljačkom signalu *a_stack*, koji određuje način rada stoga (odsječak koda 4.2.3).
Ako se trenutno izvodi operacija množenja, signal SP_ACTION se postavlja u logičku nulu, inače u logičku jedinicu.
Odsječak koda 4.2.3 prikazuje odabir ulaznih podataka za vrh adresnog stoga (A) i za prvu sljedeću lokaciju adresnog stoga (H). Shematski prikaz odabira ulaznih podataka prikazan je na slici 4.2.1 (ulazni podatak za vrh adresnog stoga A) i slici 4.2.2 (ulazni podatak za prvu sljedeću lokaciju adresnog stoga H).

```
-- MUX koji ulazi u vrh stoga A
with A_MUX_SEL select
       A_IN <= A_plus1_Latched when "00",
                    Stack_out when "01",
                    Produkt_l when "11",
                    T when others;

-- MUX koji ulazi u ostatak stoga(H)
with a_stack select
       Stack_in <= A_local when "01",
                    Produkt_h when "10",
                    T when others;
```

Odsječak koda 4.2.3: Odabir ulaznih vrijednosti vrha i ostatka adresnog stoga (modules/RAM_Access_Unit.vhd)




Slika 4.2.1: Odabir ulazne vrijednosti za A




Slika 4.2.2: Odabir ulazne vrijednosti za H

Pri izvođenju operacije množenja nižih 16 bita rezultata se sprema u vrh adresnog stoga (A), dok se viših 16 bita sprema u ostatak adresnog stoga (H).

Tablica 4.2.1 prikazuje kodove operacija signala A_MUX_SEL, koji određuje ulaznu vrijednost vrha adresnog stoga (A).

Tablica 4.2.2 prikazuje kodove operacija signala *a_stack* koji određuje ulaznu vrijednost ostatka adresnog stoga (H).

Tablica 4.2.1: Ulazna vrijednost vrha adresnog stoga (A) ovisna o upravljačkom signalu A_MUX_SEL

| Kod | Ulazna vrijednost |
|---|---|
| **00** | Vrijednost vrha adresnog stoga uvećana za 1 |
| **01** | Vrh ostatka stoga (H) |
| **10** | Vrijednost vrha podatkovnog stoga (T) |
| **11** | Nižih 16 bita rezultata množenja |

Tablica 4.2.2: Ulazna vrijednost ostatka adresnog stoga (H) ovisna o upravljačkom signalu a_stack

| Kod | Ulazna vrijednost |
|---|---|
| **01** | Vrh adresnog stoga (A) |
| **10** | Viših 16 bitova rezultata množenja |

Sukladno definiranim kodovima, signali A_MUX_SEL i *a_stack* poprimaju vrijednosti ovisno o operaciji čije izvođenje je u tijeku.
Pri izvođenju instrukcije STA (stavljanje podatka na adresni stog), signal A_MUX_SEL poprima vrijednost 10 i time se kao ulazna vrijednost vrha adresnog stoga odabire vrijednost vrha podatkovnog stoga, dok signal *a_stack* poprima vrijednost 01 čime se odabire prethodna vrijednost vrha adresnog stoga (A) kao nova vrijednost vrha ostatka stoga (H).
Pri izvođenju instrukcije LDA (uzimanje podatka sa adresnog stoga), signal A_MUX_SEL poprima vrijednost 01 i time se kao ulazna vrijednost vrha adresnog stoga odabire vrijednost vrha ostatka adresnog stoga (H), dok vrijednost signala *a_stack* nije relevantna, jer se ne vrši upis u ostatak adresnog stoga (H) .
Pri izvođenju instrukcija ST i LD (pisanje i čitanje podatkovne memorije) nema utjecaja na vrijednost vrha adresnog stoga ni ostatka adresnog stoga.
Pri izvođenju instrukcija STP i LDP vrijednost ostatka adresnog stoga (H) nije promijenjena, no vrijednost vrha adresnog stoga (A) se povećava za 1, stoga je vrijednost upravljačkog signala A_MUX_SEL u tom slučaju jednaka 00.
Pri izvođenju instrukcije množenja, vrijednost signala A_MUX_SEL je 11 čime se u vrh adresnog stoga (A) sprema nižih 16 bitova produkta, a vrijednost signala *a_stack* je 10 čime se u vrh ostatka adresnog stoga (H) sprema viših 16 bitova produkta, bez promjene vrijednosti pokazivača vrha adresnog stoga (kako je prikazano u odsječku koda 4.2.4).

```
with a_stack select
      sp_action <= '0' when "10",
                   '1' when others;
```

Odsječak koda 4.2.4: Upravljanje promjenom vrijednosti pokazivača vrha adresnog stoga u ovisnosti o upravljačkom signalu a_stack (modules/RAM_Access_Unit.vhd)

Upravljački signal *a_stack* ima višestruku namjenu. Osim za odabir ulazne vrijednosti ostatka adresnog stoga (H), pomoću njega se indicira izvršavanje instrukcije množenja. Taj uvjet se koristi za postavljanje signala SP_ACTION za upravljanje pokazivačima vrha adresnog stoga (odsječak koda 4.2.4) i za upravljanje operacijama čitanja i pisanja na ostatku adresnog stoga (H) kako je prikazano u odsječku koda 4.2.5.

```
if rising_edge(F_CLK) then
              if(a_stack = "01") then
              CNTE <= '1';
              PUSH <= '1';
       end if;

       if (a_stack = "10") then
              CNTE <= '1';
              PUSH <= '1';
       end if;

       if (a_stack = "11") then
              CNTE <= '1';
              PUSH <= '0';
       end if;

       if (a_stack = "00") then
              CNTE <= '0';
              PUSH <= '0';
       end if;
end if;
```

Odsječak koda 4.2.5: Upravljanje operacijama čitanja i pisanja na ostatku adresnog stoga (H) ovisno o upravljačkom signalu a_stack (modules/RAM_Access_Unit.vhd)

Tablica 4.2.3 prikazuje odabir operacija čitanja i pisanja nad ostatkom adresnog stoga (H) ovisno o vrijednosti upravljačkog signala *a_stack*.

Tablica 4.2.3: Operacija nad ostatkom adresnog stoga (H) ovisna o upravljačkom signalu a_stack

| Kod | Operacija |
|---|---|
| **00** | Nema operacije |
| **01** | Pisanje |
| **10** | Pisanje |
| **11** | Čitanje |

Signale *a_stack*, A_MUX_SEL i mnoge druge generira upravljačka jedinica procesora.

Upravljačka jedinica dekodira učitani operacijski kod i generira mnoštvo upravljačkih signala koji upravljaju radom cijelog sustava.

Signali generirani nakon dekodiranja instrukcije u upravljačkoj jedinici procesora P16 su aktivni tek u protočnom segmentu izvršavanja. Svi prethodni signali potrebni za rad prethodnih protočnih segmenata su generirani automatizmom i nisu povezani sa signalima generiranim u protočnom segmentu dekodiranja instrukcije. Specijalne instrukcije (grananje, literali) su dekodirani u prethodnim protočnim segmentima od strane specijaliziranog sklopovlja.

Potrebno je nadograditi upravljačku jedinicu da bi instrukcija množenja bila ispravno dekodirana i poduzete odgovarajuće mjere kako bi se operacija množenja izvršila.
Prije ugradnje potrebno je odabrati operacijski kod nove instrukcije.
Odabrani operacijski kod za instrukciju množenja je 0x16. Razlozi odabira upravo ovog operacijskog koda su navedeni u sljedećim poglavljima.
Odsječak koda 4.2.6 prikazuje dekodiranje instrukcije množenja i generiranje upravljačkih signala u upravljačkoj jedinici.
Zbog razlika u nomenklaturi signal A_MUX_SEL se u upravljačkoj jedinici naziva *a_sel*. Njegova vrijednost je postavljena na 11 kako bi nižih 16 bita rezultata operacije množenja bilo odabrano kao vrijednost za upis u vrh adresnog stoga (A).
Signal *a_stack* ima identično ime u upravljačkoj jedinici, a postavljena mu je vrijednost 10 kako bi viših 16 bita rezultata operacije množenja bilo odabrano kao vrijednost za upis u vrh ostatka adresnog stoga (H), te da bi se onemogućila izmjena vrijednosti pokazivača vrha ostatka adresnog stoga i da bi se aktivirali odgovarajući signali za upis u ostatak adresnog stoga (H).
Signal *a_ce* je postavljen u stanje logičke jedinice i time je omogućen upis u vrh adresnog stoga (A) za vrijeme izvršavanja ove instrukcije.
Ostale operacije unutar procesora su postavljene u stanje mirovanja za vrijeme izvršavanja instrukcije množenja.

```
constant     mul :  std_logic_vector(4 downto 0) :="10110";


                    when mul => data_unit_sel <= "1000";
                                alu_sel <= "00";
                                n_push <= '0';
                                n_cnte <= '0';
                                r_push <= '0';
                                r_cnte <= '0';
                                r_sel <= '0';
                                a_sel <= "11";
                                ram_sel <= "00";
                                a_ce <= '1';
                                ram_ce <= '0';
                                ram_we <= '0';
                                isADDC <= '0';
                                a_stack <= "10";
```

Odsječak koda 4.2.6: Dekodiranje instrukcije množenja i generiranje upravljačkih signala (modules/instruction_decoder.vhd)

Ugrađeno množilo je potrebno instancirati i povezati na odgovarajuće signale. Pri instanciranju ugrađenog množila potrebno je u razvojnoj okolini (Xilinx) postaviti MULT_STYLE ograničenje sa podrazumijevane postavke *Auto* na postavku *Block*, kako alat za sintezu ne bi izgradio množilo korištenjem CLB blokova već iskoristio jedno od ugrađenih množila.

Koristi se asinkrona izvedba ugrađenog množila. Množilo je enkapsulirano u zasebnu komponentu koja podržava dodatne mogućnosti, poput odabira predznačnog ili množenja bez predznaka.

Primjer instanciranja (enkapsuliranog) množila prikazan je u odsječku koda 4.2.7. Operandi množila su vezani na vrh adresnog stoga (A) i vrh ostatka adresnog stoga (H). Asinkrona operacija množenja je aktivna za cijelo vrijeme rada sklopa. Trenutak pohrane rezultata se naziva izvođenjem operacije množenja, iako je konkretno množenje ulaznih argumenata već asinkrono obavljeno i prije nego krene izvođenje instrukcije množenja.

```
component Multiplier_Unit is
port (       Produkt : out STD_LOGIC_VECTOR (31 downto 0);
             Operand1 : in STD_LOGIC_VECTOR (15 downto 0);
             Operand2 : in STD_LOGIC_VECTOR (15 downto 0);
             MultiplierMode : in std_logic_vector(2 downto 0));
end component;

Multiplier_Unit_inst: component Multiplier_Unit
port map (
                    Operand1 => Op1,
                    Operand2 => Op2,
                    Produkt(31 downto 16) => Produkt_h(15 downto 0),
                    Produkt(15 downto 0) => Produkt_l(15 downto 0),
                    MultiplierMode => MultiplierMode
             );

Op1 <= A_local;
Op2 <= Stack_out2;
```

Odsječak koda 4.2.7: Instanciranje enkapsuliranog množila (modules/RAM_Access_Unit.vhd)

Registar (sinkroni signal) *MultiplierMode* priključen na enkapsulirano množilo (odsječak koda 4.2.7) određuje način rada množila. Podržani načini rada su predznačno množenje (operandi su u formatu dvojnog komplementa) i množenje bez predznaka.

Način rada množila se odabire zapisivanjem konfiguracijskog broja u podatkovnu memoriju na adresu 0x3FF.

Implementacija podsustava za konfiguraciju množila je prikazana u odsječku koda 4.2.8 i na slici 4.2.3. Ukoliko se aktivira signal za zapisivanje u podatkovnu memoriju, a adresa na vrhu adresnog stoga je 0x3FF, podatak sa podatkovne sabirnice se pohranjuje u registar *MultiplierMode*.

```
-- RAM Latch
process (RAM_DATA, M_CLK) is
begin
       if M_CLK = '1' then
              RAM_OUT <= RAM_DATA;
              if SAVE_MULTMODE = '1' and RAM_ADDR_internal = "1111111111" then
                     MultiplierMode(2 downto 0) <= RAM_DATA(2 downto 0);
              end if;
       end if;
end process;
```

Odsječak koda 4.2.8: Pohranjivanje konfiguracije množila (modules/RAM_Access_Unit.vhd)

Signal SAVE_MULTMODE je spojen na signal za pisanje u podatkovnu memoriju, te je aktivan u logičkoj jedinici.

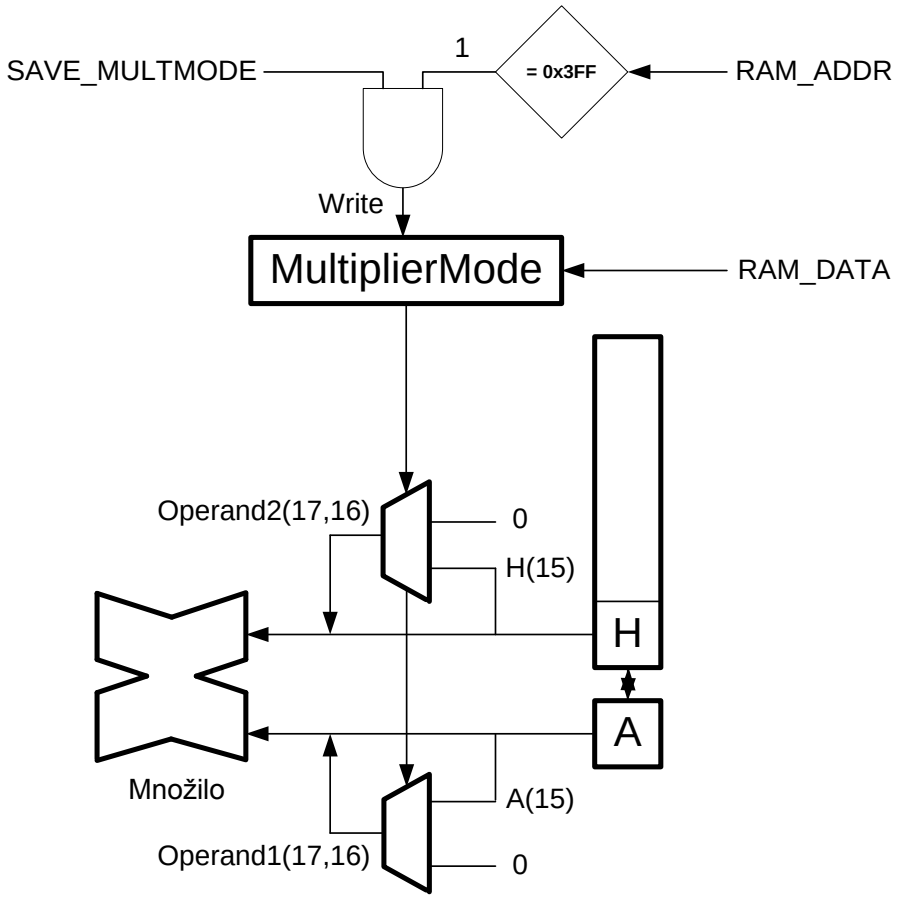
Konfiguracijski broj je trobitni podatak. Ako je konfiguracijski broj 1, množilo je postavljeno u način rada bez predznaka. Konfiguracijski broj 2 postavlja množilo u predznačni način rada. Tablica 4.2.4 prikazuje način rada množila ovisno o konfiguracijskom broju. Trobitni podatak je odabran za buduća proširenja.
Ukoliko se nakon uključenja sklopa ne odabire način rada množila, automatski je postavljen način rada bez predznaka, što je postignuto korištenjem inicijalizacije signala na vrijednost 1 nakon globalnog reseta.
Korištenje memorijske lokacije 0x3FF kao konfiguracijske lokacije za množilo smanjuje kapacitet podatkovne memorije sa 1024 riječi na 1023 riječi.

Odsječak koda 4.2.9 prikazuje internu implementaciju enkapsuliranog množila (instanciranje ugrađenog asinkronog množila Spartan-3 arhitekture i odabir predznačnog načina rada ili načina rada bez predznaka).

```
component MULT18X18

             port (       P : out STD_LOGIC_VECTOR (35 downto 0);
                          A : in STD_LOGIC_VECTOR (17 downto 0);
                          B : in STD_LOGIC_VECTOR (17 downto 0));
end component;

signal ostatak_rez : std_logic_vector(3 downto 0);
signal ostatak_op_a : std_logic_vector(1 downto 0);
signal ostatak_op_b: std_logic_vector(1 downto 0);

begin

MULT18X18_Multiplier1: MULT18X18
       port map (          P (31 downto 0) => Produkt(31 downto 0),
                           P (35 downto 32) => ostatak_rez(3 downto 0),
                           A (15 downto 0) => Operand1(15 downto 0),
                           A (17 downto 16) => ostatak_op_a(1 downto 0),
                           B (15 downto 0) => Operand2(15 downto 0),
                           B (17 downto 16) => ostatak_op_b(1 downto 0));


       with MultiplierMode select
              ostatak_op_a <=     "00" when "001",
                           (others => Operand1(15)) when "010",
                                  "00" when others;


       with MultiplierMode select
              ostatak_op_b <=     "00" when "001",
                           (others => Operand2(15)) when "010",
                                  "00" when others;
end RTL;
```

Odsječak koda 4.2.9: Interna struktura enkapsuliranog množila (modules/multiplier.vhd)

Podsustav za konfiguraciju načina rada množila i za pripremu operanada množila ovisno o odabranom načinu prikazan je na slici 4.2.3.




Slika 4.2.3: Podsustav za odabir načina rada množila i pripremu operanada množila ovisno o odabranom načinu rada

U načinu rada bez predznaka bitovi 16 i 17 oba operanda množila su postavljeni u logičku nulu.

U predznačnom načinu rada bitovi 16 i 17 prvog operanda množila su postavljeni na vrijednost bita 15 prvog operanda, a bitovi 16 i 17 drugog operanda su postavljeni na vrijednost bita 15 drugog operanda.

Na taj način je omogućeno predznačno množenje u tehnici dvojnog komplementa.

Tablica 4.2.4: Odabir načina rada množila

| Konfiguracijski broj | Način rada množila |
|---|---|
| **01** | Množenje bez predznaka (podrazumijevano) |
| **02** | Predznačno množenje |

## 4.3 Odabir operacijskog koda i prilagodba prevodioca

Operacijski kod procesora P16 se sastoji od 5 bita.
Minimalni skup instrukcija ovog procesora uključuje 26 instrukcija (od 32 mogućih).
Od preostalih kodnih riječi nije moguće odabrati proizvoljnu zbog dodatnih ograničenja.
Neiskorištene kodne riječi su 0x05, 0x06, 0x07, 0x0B, 0x0E, 0x16.
Kodne riječi 0x05 do 0x07 su zbog svog posebnog oblika rezervirane za buduće instrukcije grananja.
Kodna riječ 0x0B je zabranjena jer se zbog svog oblika tumači kao LIT instrukcija (od strane specijaliziranog sklopovlja za detekciju literala i instrukcija grananja).
Preostale slobodne riječi su 0x0E i 0x16.
Proizvoljno je odabrana kodna riječ 0x16 kao operacijski kod instrukcije množenja.
Pridruženi mnemonik za zbirni jezik je MUL.

Prilagodba prevodioca za zbirni jezik uključuje dodavanje operacijskog koda i mnemonika u strukture definirane u izvornom kodu prevodioca.
Primjer ugradnje MUL instrukcije je prikazan u odsječku koda 4.3.1.
Prevodilac je pisan u programskom jeziku C++.

```
num eInstrukcija {
        /*…*/
        andd = 0x015,
        mul = 0x016, //dodan operacijski kod
        addd = 0x17,
        pop = 0x18,
        /*…*/
};

const string Mnemonic[] = {"jmp", "call", "jz", "jnc", "ret", ".", ".", ".",
                           "zer", "ldp", "lit", ".", "ld", "stp", ".", "st",
                           "subb", "shl", "shr", "addc", "xorr", "andd.", "mul",
                           "addd", "pop", "lda", "dup", "over", "push", "sta",
                           "nop", "drop"};
        //definiran mnemonik
```

Odsječak koda 4.3.1: Dodavanje MUL instrukcije u zbirni prevodilac (assembler/src/P16_Assembler/Opcode.h)

## 4.4 Ispitivanje

Izvođenjem ispitnog programa se utvrđuje da li je ugradnja instrukcije množenja na procesor P16 uspješno obavljena.
Za jednostavan ispitni program sa množenjem bez predznaka su odabrani brojevi dovoljno veliki kako bi rezultat operacije množenja bio veći od 16 bita (kako bi se utvrdila ispravnost cijelog podsustava za množenje), a dovoljno mali kako bi stali u 16 bita grupe instrukcija kao argument LIT instrukcije (arhitekturno ograničenje procesora P16).
Proizvoljno odabrani operandi su 0x3FF i 0x3FE,  a očekivani rezultat operacije množenja je 0xFF402.
Operacija se izvodi kao množenje bez predznaka. Na adresni stog se stavljaju oba operanda, izvodi se instrukcija množenja, te se sa adresnog stoga skidaju redom nižih 16 bita rezultata (očekivano 0xF402) i viših 16 bita rezultata (očekivano 0x000F).
Jednostavan ispitni program je prikazan u odsječku koda 4.4.1.
Važno je napomenuti da je inicijalni blok programa (NOP i STA instrukcije) nefunkcionalni dio programa koji osigurava ispravan start procesora nakon uključenja napajanja.
Procesor ne podržava instrukciju za zaustavljanje izvođenja programa (HALT), stoga se za završetak programa koristi beskonačna petlja.

```
            ; Ispitni program množenja

            ; Blok za stabilizaciju rada procesora
            nop
            nop
            nop
            nop
            nop
            nop
            zer
            sta
            zer
            ; Kraj bloka za stabilizaciju rada procesora

            ; Učitavanje operanada (0x3FE i 0x3FF) iz programske memorije na
            ; podatkovni stog
            lit 1022
            lit 1023

            ; Premiještanje operanada sa podatkovnog stoga na adresni stog
            sta
            sta

            ; Obavljanje operacije množenja
            mul

            ; Premiještanje rezultata operacije množenja sa adresnog stoga na
            ; podatkovni stog
            lda
            lda

            ; Zaustavljanje rada procesora beskonačnom petljom
      kraj:  jmp kraj
```

Odsječak koda 4.4.1: Zbirni program za ispitivanje MUL instrukcije (assembler/bin/mul_test.p16)

Prevođenje se obavlja pokretanjem prevodioca za P16 zbirni jezik naredbom

*P16_Assembler.exe mul_test.p16 mul_test.bin*

Prevodilac će od ulazne datoteke (*mul_test.p16*) sa programom u zbirnom jeziku (odsječak koda 4.4.1) generirati izlaznu datoteku (*mul_test.bin*) sa binarnim izvršnim kodom za P16 procesor.

Način grupiranja instrukcija u skupine od 3 instrukcije koje obavlja prevodilac zbirnog jezika je prikazan u odsječku koda 4.4.2.

Nakon prevođenja je potrebno binarni izvršni kod ugraditi u VHDL inicijalizacijsku strukturu programske memorije.

Ugradnja se obavlja izvršavanjem programa *bin2vhd* naredbom

*bin2vhd.exe mul_test.bin program_rom.vhd program_rom_Template.vhd*

```
Program in memory:

0:      nop     nop     nop
1:      nop     nop     nop
2:      zer     sta     zer
3:      lit     lit     sta
4:      #1022
5:      #1023
6:      sta     mul     lda
7:      lda     nop     nop
8:      jmp     8
```

Odsječak koda 4.4.2: Način grupiranja instrukcija iz odsječka 4.4.1 nakon prevođenja

Izlaznu VHD datoteku (*program_rom.vhd*) je potrebno postaviti u kazalo *modules* i pokrenuti sintezu i implementaciju dizajna u Xilinx alatu.

Nakon završetka implementacije potrebno je pokrenuti simulaciju pokretanjem ModelSim skriptne datoteke *Processor_Sim.do* u kazalu *simulation*.

Napomena: zbog problema sa skriptnom datotekom i simulacijom sadržaj vrha podatkovnog stoga T je u ModelSim simulatoru moguće pratiti pod imenom skupa signala RAM_ADDR. Zbog ograničenog opsega signala RAM_ADDR na 10 bita, gornjih 6 bitova sadržaja vrha podatkovnog stoga T će biti prikazani kao logička nula.

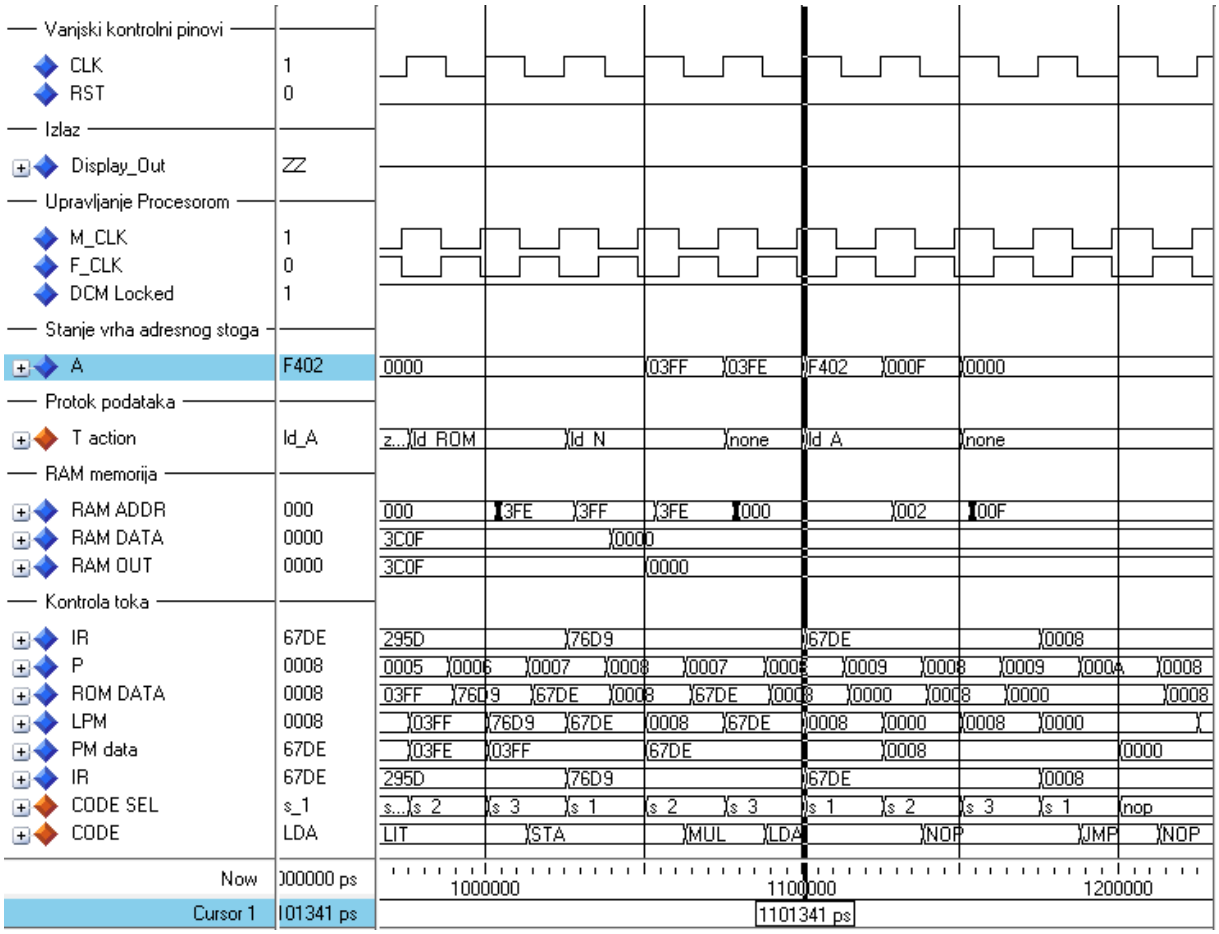
Rezultat simulacije odsječka koda 4.4.1 je prikazan na slici 4.4.1.

Za praćenje simulacije je dovoljno promatrati dva ključna signala, signal A (stanje vrha adresnog stoga) i signal CODE (tekuća instrukcija, operacijski kod zamijenjen sa mnemoničkim imenom). Pri tom treba obratiti pažnju na to da se instrukcija izvršava tek 3 poluperiode signala CLK nakon što se mnemonik te instrukcije pojavio u prikazu signala CODE.

Za jednostavnije praćenje vertikalni kursor je postavljen točno u trenutak izvršenja instrukcije MUL (3 poluperiode signala CLK nakon pojavljivanja instrukcije MUL na signalu CODE).

Praćenjem stanja vrha adresnog stoga (A) moguće je vidjeti inicijalno postavljanje operanada (0x3FF i 0x3FE) na adresni stog korištenjem instrukcija STA, prebrisivanje operanda 0x3FE sa nižih 16 bitova rezultata operacije množenja (0xF402), te viših 16 bitova rezultata operacije množenja (0x000F) nakon izvršenja prve instrukcije LDA (skidanja nižih 16 bitova rezultata sa adresnog stoga).

Time je utvrđeno da je instrukcija množenja uspješno izvedena.




Slika 4.4.1: Rezultat simulacije odsječka koda 4.4.1

Odsječak koda 4.4.3 prikazuje zbirni program za ispitivanje predznačnog množenja.

Postupak pripreme simulacije (prevođenje, sinteza…) se izvodi na jednak način opisan u prethodnom ispitivanju, uz izuzetak što se u ovom testiranju koristi datoteka sa izvornim kodom *sutest.p16* u kazalu *assembler/bin*.

Ispitivanje se vrši pomoću dva množenja.

U prvoj operaciji množenja obavlja se množenje bez predznaka operanada 0xFFFF i 0xFFFF koji predstavljaju brojeve 65535 i 65535.

Očekivani rezultat množenja bez predznaka je 4294836225 (0xFFFE0001).

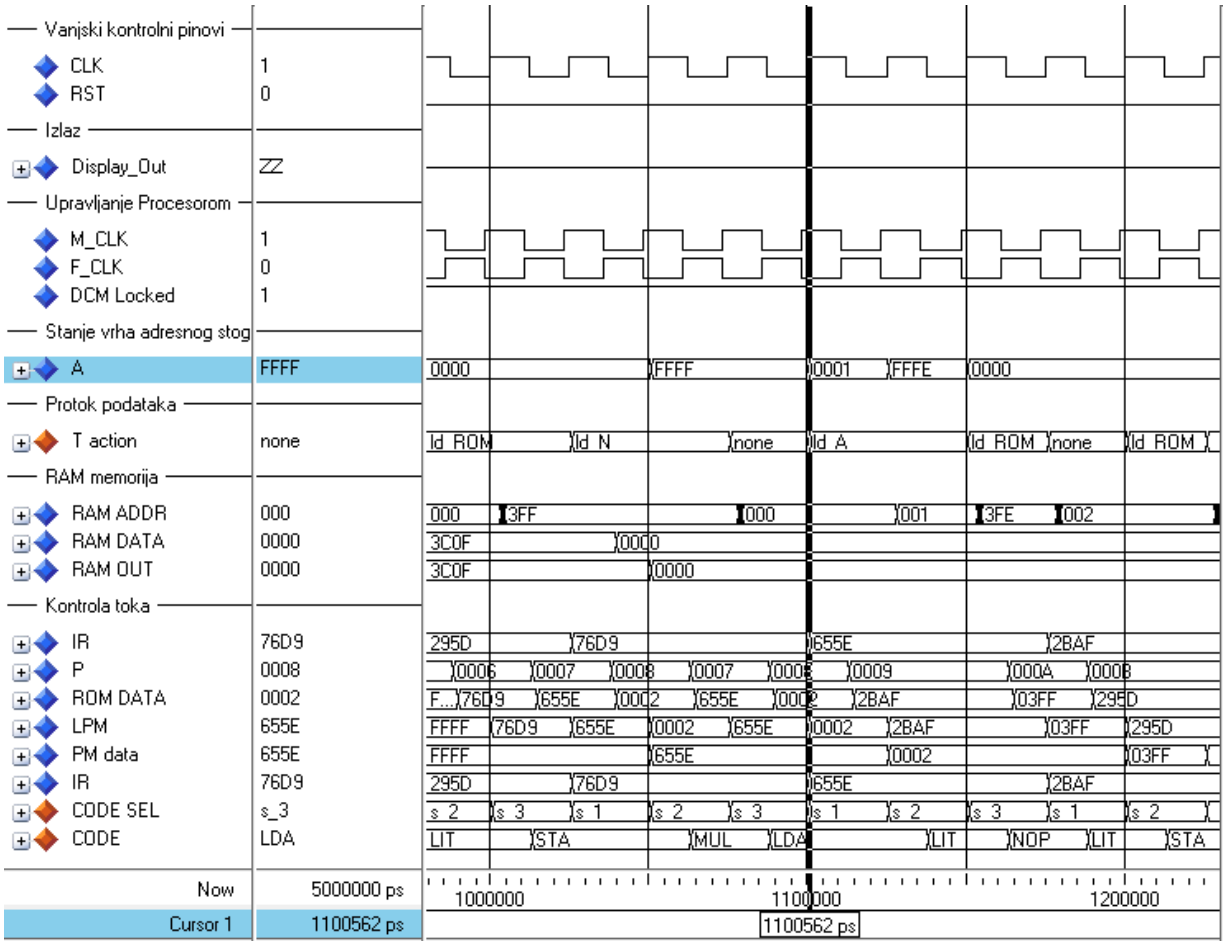
Nakon prvog množenja i skidanja rezultata množenja sa adresnog stoga množilo se postavlja u predznačni način rada zapisivanjem konfiguracijskog broja 2 na lokaciju 0x3FF u podatkovnoj memoriji. U drugoj operaciji množenja obavlja se predznačno množenje operanada 0xFFFF i 0xFFFF koji predstavljaju brojeve -1 i -1.

Očekivani rezultat predznačnog množenja je 1 (0x00000001).

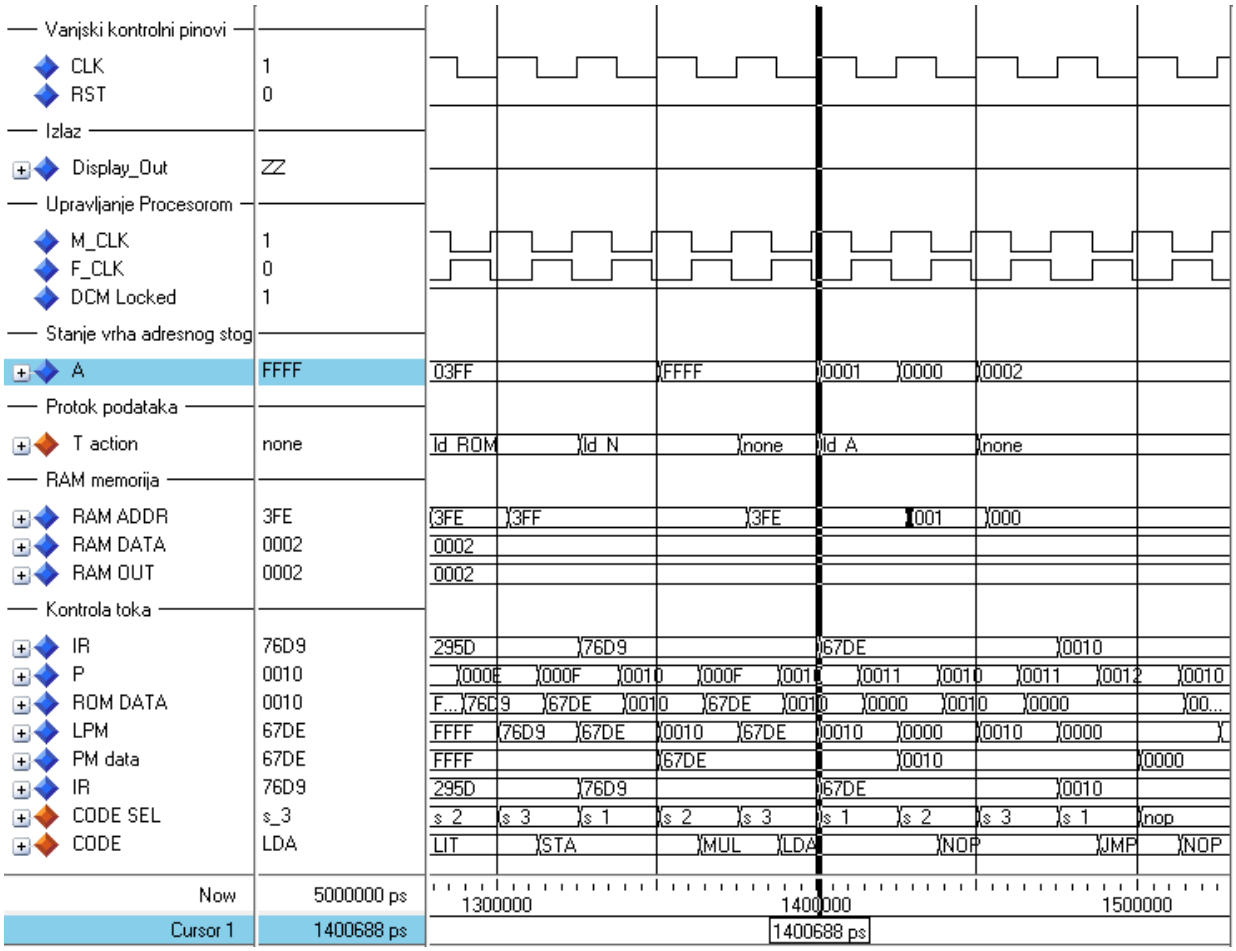
```
        ; Ispitni program predznačnog množenja

        ; Blok za stabilizaciju rada procesora
        nop
        nop
        nop
        nop
        nop
        nop
        zer
        sta
        zer
        ; Kraj bloka za stabilizaciju rada procesora

        ; Učitavanje operanada (65535 i 65535) iz programske memorije na
        ; podatkovni stog
        lit 65535
        lit 65535

        ; Premiještanje operanada sa podatkovnog stoga na adresni stog
        sta
        sta

        ; Obavljanje operacije množenja bez predznaka
        mul

        ; Premiještanje rezultata operacije množenja bez predznaka sa
        ; adresnog stoga na podatkovni stog
        lda
        lda

        ; Postavljanje množila u predznačni način rada
        lit 2
        lit 1023
        ; Pohrana adrese 0x3FF na adresni stog
        sta
        ; Pohrana konfiguracijskog broja 2 (predznačni način rada) na
        ; lokaciju 0x3FF podatkovne memorije
        st

        ; Učitavanje operanada (-1 i -1) iz programske memorije na
        ; podatkovni stog
        lit 65535
        lit 65535

        ; Premiještanje operanada sa podatkovnog stoga na adresni stog
        sta
        sta

        ; Obavljanje operacije predznačnog množenja
        mul

        ; Premiještanje rezultata operacije predznačnog množenja sa adresnog
        ; stoga na podatkovni stog
        lda
        lda

        ; Zaustavljanje rada procesora beskonačnom petljom
kraj:   jmp kraj
```

Odsječak koda 4.4.3: Ispitni zbirni program predznačnog množenja

Analiza rezultata množenja se vrši kao u prethodnom ispitivanju.
Potrebno je pratiti stanje na vrhu adresnog stoga i ime tekuće instrukcije.
Zbog jednostavnijeg praćenja simulacija je prikazana na dvije slike.
Slika 4.4.2 prikazuje vertikalni kursor u trenutku nakon prvog množenja (bez predznaka), a slika 4.4.3 prikazuje vertikalni kursor u trenutku nakon drugog množenja (predznačnog).




Slika 4.4.2: Rezultat prvog dijela simulacije odsječka koda 4.4.3




Slika 4.4.3: Rezultat drugog dijela simulacije odsječka koda 4.4.3

Iz rezultata simulacije se vidi stanje vrha adresnog stoga nakon prvog i drugog množenja. Rezultat prvog množenja (bez predznaka) je očekivanih 0xFFFE0001 (4294836225) dok je rezultat drugog množenja (predznačnog) očekivanih 0x00000001 (1).

Oba operanda obje operacije množenja su 0xFFFF, pa u dijagramu nisu vidljivo razdvojeni prije svake od operacija množenja.

Ovim ispitivanjem je potvrđen ispravan rad podsustava za predznačno množenje.

# 4. Zaključak

Procesor P16 je školski procesor relativno male računske snage.
Arhitekturna proširenja (poput ugradnje instrukcije množenja) podižu računsku snagu procesora i omogućavaju potencijalnu primjenu na stvarnim problemima.
Dodatno podizanje performansi je moguće potencijalnim radom više procesora P16 na jednom sklopu.
Uz popularnu ciljnu platformu Spartan-3 FPGA i uz navedena proširenja može se izgraditi sustav adekvatan za rješavanje složenijih računski zahtjevnih zadataka u realnom vremenu.
Jedno od većih ograničenja pri dodatnim proširenjima procesora P16 predstavlja ograničen broj slobodnih kodnih riječi za ugradnju dodatnih instrukcija. Ovaj problem nije neočekivan, jer procesor P16 ima minimalni skup instrukcija, a svako dodatno proširenje tog skupa narušava ovo svojstvo.
Ugrađena instrukcija množenja omogućava višestruko ubrzanje u izvođenju računskih operacija množenja.
Mogućnost konfiguriranja predznačnog ili množenja bez predznaka dodatno proširuje iskoristivost instrukcije množenja.
Dodatna arhitekturna proširenja su moguća uvođenjem dodatnih načina rada množila (poput množenja decimalnih brojeva ili množenja sa akumuliranjem).
Instrukcija množenja je ugrađena u podsustav procesora koji nije predviđen za računske operacije i time je narušena logička organizacija arhitekture procesora, no takvi problemi su neizbježni pri radu sa ograničenim resursima FPGA sklopovlja.
Stogovna arhitektura procesora P16 je relativno niske razine kompleksnosti, a korištenje sklopovskih rješenja poput ugrađenih množila ne podiže razinu kompleksnosti na previsoki stupanj i ne ograničava skalabilnost procesora te mogućnost jednostavnih arhitekturnih izmjena.

Luka Karlović

# 5. Literatura

[1] Goran Horak, Josip Mihel: *Implementacija MISC stogovnog procesora P16 u Spartan3 FPGA tehnologiji*, prezentacija, FER-ZESOI 2005.

[2] *Spartan-3 FPGA Family: Complete Data Sheet*, Xilinx, DS099, studeni 2007. http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf (20.04.2008.)

[3] *Using Embedded Multipliers in Spartan-3 FPGAs*, Xilinx, XAPP467 (v1.1), svibanj 2003. http://www.xilinx.com/support/documentation/application_notes/xapp467.pdf (20.04.2008.)

[4] *Using Look-Up Tables as Distributed RAM in Spartan-3 Generation FPGAs*, Xilinx, XAPP464 (v2.0,) ožujak 2005. http://www.xilinx.com/support/documentation/application_notes/xapp464.pdf (20.04.2008.)

[5] *Using Block RAM in Spartan-3 Generation FPGAs*, Xilinx, XAPP463 (v2.0) ožujak 2005. http://www.xilinx.com/support/documentation/application_notes/xapp463.pdf (20.04.2008.)

# Dodatak A: hrvatski naslov, ključne riječi, sažetak

## Naslov

Ugradnja instrukcije množenja na procesoru P16 s minimalnim skupom instrukcija

## Ključne riječi

MISC, FPGA, Xilinx, VHDL, P16, instrukcija množenja, procesor sa mekom jezgrom, *soft-core* procesor

## Sažetak

Tema rada je ugradnja instrukcije množenja na procesor P16.
Procesor P16 ima minimalni skup instrukcija, stogovnu arhitekturu i *soft-core* jezgru. Ciljna platforma za implementaciju procesora je Xilinx Spartan-3 FPGA. Instrukcija množenja je implementirana korištenjem ugrađenog množila Spartan-3 arhitekture. Za potrebe rada množila dodan je dodatan stog u arhitekturu procesora baziran na distribuiranoj RAM memoriji konfigurabilnih logičkih blokova. Osim fizičke implementacije instrukcije u radu je obrađen kratki teoretski uvod u FPGA sklopovlje porodice Spartan-3 i opis izvorne implementacije procesora P16.

# Dodatak B: engleski naslov, ključne riječi, sažetak

## Title

Implementation of multiplication instruction on minimal instruction set processor P16


## Keywords

MISC, FPGA, Xilinx, VHDL, P16, multiplication instruction, soft-core processor



## Summary

Main topic of this document is implementation of multiplication instruction on P16 processor. Minimal instruction set, stack-based architecture and soft core implementation are key characteristics of P16 processor. Target platform for P16 implementation is Xilinx Spartan-3 FPGA family. Multiplication instruction is implemented by using an embedded multiplier found in Spartan-3 architecture. Additional stack based on distributed RAM feature of configurable logic blocks is built for purpose of multiplication instruction execution. Document also describes short theoretical basis of Spartan-3 FPGA architecture and original P16 processor implementation.