S V E U Č I L I Š T E U S P L I T U
FAKULTET ELEKTROTEHNIKE, STROJARSTVA I BRODOGRADNJE

PROJEKTIRANJE DIGITALNIH SUSTAVA

SEMINARSKI RAD

Josip Sanader i Ivan Stojković

Split, kolovoz 2024.

SADRŽAJ

1. UVOD

Razvoj i implementacija digitalnih sustava na programibilnim logičkim uređajima postali su standardna praksa u dizajnu suvremenih elektroničkih sustava. Jedan od najvažnijih alata za dizajn i simulaciju digitalnih sklopova je jezik za opis sklopovlja (*engl. hardware description language)* Verilog. Jezik omogućava razvojnim inženjerima da opišu funkcionalnost i strukturu digitalnih sustava na visokoj razini apstrakcije. U ovom seminaru fokusirat ćemo se na realizaciju 8-bitnog procesora sa cjevovodom (*engl. pipeline*) koristeći Verilog, implementiranog na FPGA pločici Spartan-3E.

Procesori s cjevovodom su osnovni građevni blokovi modernih mikroprocesora, omogućujući paralelnu obradu instrukcija radi povećanja performansi sustava. Cjevovodna arhitektura omogućava istovremeno izvršavanje više instrukcija tako što razdvaja izvršenje svake instrukcije u više faza koje se izvršavaju paralelno. Na taj način povećava se propusnost sustava, jer nova instrukcija može započeti izvršenje prije nego što se prethodna završi.

Cilj ovo seminarskog rada je bolje razumijevanje osnovnih principa cjevovodne arhitekture, objasnit će se njena funkcionalnost i na koji način ova arhitektura omogućava poboljšanje performansi procesora, kao i izazove koji se javljaju prilikom dizajniranja takvih sustava. Također, jedan od ciljeva rada je da se realizira funkcionalni model 8-bitnog procesora. Koristeći Verilog bit će opisana struktura i funkcionalnost osnovnih komponenti procesora, uključujući glavni modul za upravljanje, ALU (aritmetičko-logička jedinica), registre i memoriju. Nakon opisa sklopa, biti će izvršena implementacija i testiranje na samoj FPGA pločici.

Kroz ovaj seminar, sudionici će steći duboko razumijevanje dizajna procesora sa cjevovodom, kao i praktično iskustvo u implementaciji digitalnih sustava koristeći Verilog i FPGA tehnologiju. Dodatno će biti obrađeni izazovi i rješenja koja su se pojavila tijekom razvoja, pružajući tako sveobuhvatan uvid u praktičnu primjenu teorijskih znanja iz područja projektiranja digitalnih sustava.

2. CJEVOVOD

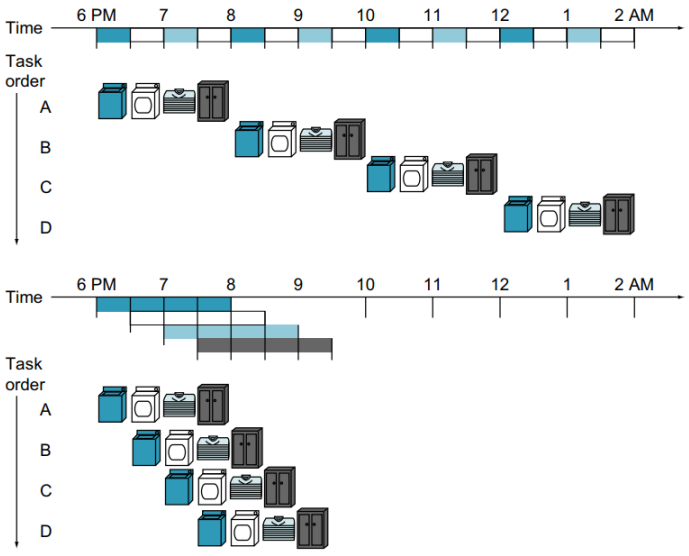
U suvremenim računalnim arhitekturama, cjevovod (eng. *pipeline*) predstavlja jednu od ključnih tehnika koja omogućava povećanje učinkovitosti procesora. Njegovom se implementacijom omogućuje istodobno izvođenje više uputa, čime se postiže veća brzina obrade podataka i povećava ukupna računalna snaga. Cjevovod se temelji na načelu podjele izvršavanja uputa u nekoliko faza, pri čemu svaka faza obavlja dio ukupnog zadatka.

Princip cjevovoda se može objasniti na principu perilice rublja. Princip perilice bez cjevovoda bi bio sljedeći:

- Postavi komad prljave odjeće u perilicu
- Kada perilica završi, postavi mokru odjeću u sušilicu
- Kada sušilica završi, postavi odjeću na radnu ploču i složi je
- Kada je složiš, spremi je u ormar

Kada se završi četvrti korak, ide sljedeći ciklus.

„Cjevovodni“ pristup bi bio takav da čim perilica završi sa prvim punjenjem odjeće i stavi ga u sušilicu, perilica se napuni novim setom prljave odjeće. Kada se prvi set osuši i složi, drugi izlazi iz perilice i stavlja se na sušenje itd. Ova četiri koraka se zovu faze cjevovoda i u različitim procesorima imamo različiti broj faza koje se izvode istovremeno. Dakle, kod cjevovoda potrebno je isto vrijeme za pranje odjeće odnosno izvođenje operacije, ali kod velikog broja operacija ima jako velik značaj. U ovom slučaju perilica sa cjevovodom je potencijalno četiri puta brža od obične perilice.



Slika 2.1 Princip cjevovoda

Kao što je već rečeno, tipičan cjevovod procesora sastoji se od nekoliko osnovnih faza koje mogu varirati ovisno o specifičnoj arhitekturi procesora.

Najčešće se izdvajaju sljedeće faze:

- Dohvat naredbe (*eng. Instruction Fetch – IF)*: u ovoj fazi procesor dohvaća naredbu iz memorije, obično iz glavne ili programske memorije. Dohvaćena naredba se prenosi u odgovarajući registar kako bi bila spremna za daljnju obradu.
- Dekodiranje naredbe (*eng. Instruction Decode - ID*): nakon dohvaćanja, uputa se dekodira kako bi procesor razumio što treba napraviti. U ovoj fazi se također određuje koji operandi su potrebni i iz kojih registara ih treba dohvatiti.
- Izvršavanje (*eng. Execution - EX*): u ovoj fazi procesor izvršava dekodiranu uputu. To može uključivati aritmetičke i logičke operacije, pomake, skokove i druge vrste operacija. Ovdje se često koristi aritmetičko-logička jedinica (ALU).
- Pristup memoriji (*eng. Memory Access - MEM*): ako uputa zahtijeva pristup memoriji (npr. učitavanje podataka iz memorije ili spremanje podataka u nju), ta se operacija odvija u ovoj fazi.
- Zapis rezultata (*eng. Write Back - WB*): na kraju cjevovoda, rezultati izvršavanja upisuju se natrag u odgovarajući registar procesora ili memoriju, ovisno o tipu upute.

U slučaju ovog seminarskog rada nisu uključene sve faze, već je cjevovod podijeljen u četiri faze što će biti detaljnije opisano u jednom od sljedećih poglavlja.

Glavna prednost cjevovoda je mogućnost paralelnog izvršavanja uputa. Dok se jedna uputa dekodira, druga se već može dohvaćati, a treća izvršavati. Ova tehnika omogućuje značajno povećanje broja uputa koje se mogu obraditi u jedinici vremena.

U optimalnim uvjetima, procesor s cjevovodom može završiti jednu uputu po svakom ciklusu takta, što znači da će, primjerice, u procesoru s pet faza cjevovoda u svakom trenutku biti pet uputa u različitim fazama obrade. Ovaj oblik paralelizma poznat je kao *paralelizam na razini uputa* (ILP - *Instruction Level Parallelism*).

Međutim, cjevovod nije bez svojih izazova. Jedan od glavnih problema su *hazardi* ili sukobi koji mogu nastati zbog međusobne ovisnosti uputa. Postoje tri osnovne vrste hazarda:

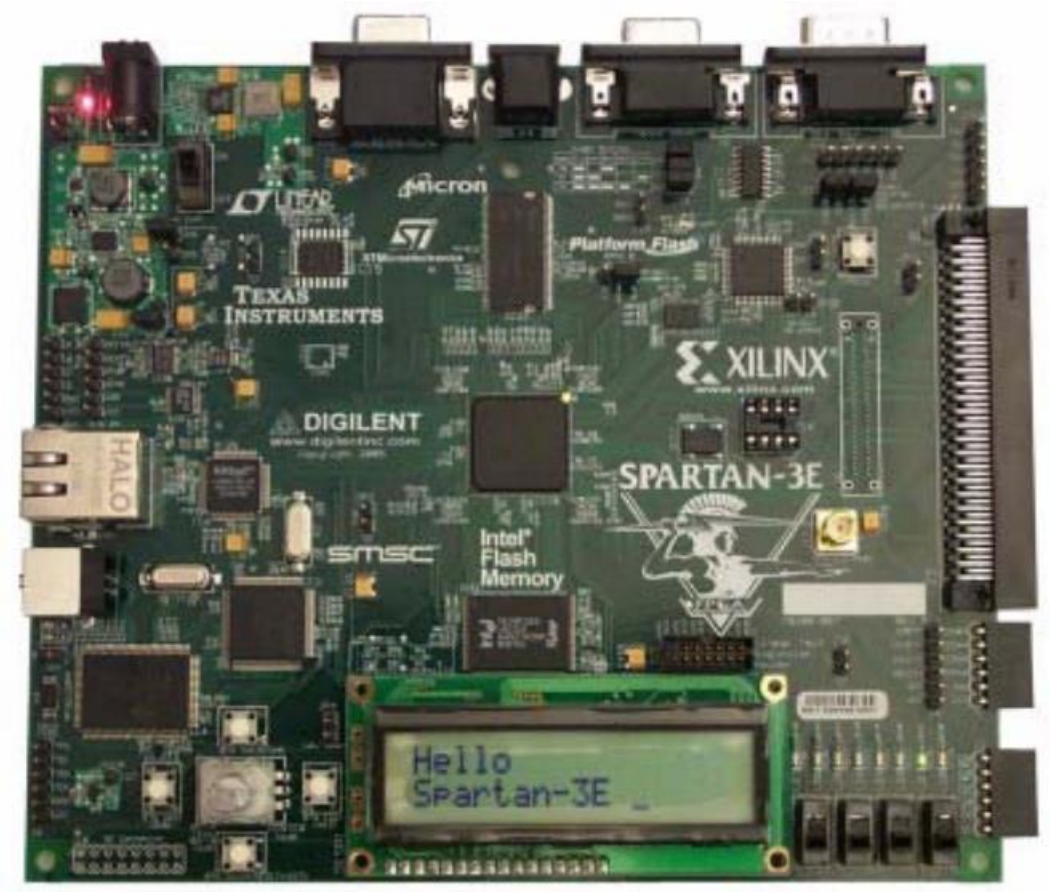
- Strukturni hazard: Nastaje kada dvije upute zahtijevaju istu hardversku komponentu u isto vrijeme. Na primjer, dvije upute koje žele pristupiti memoriji mogu uzrokovati konflikt ako procesor nema dovoljnu količinu memorijskih jedinica.

- Podatkovni hazard: Javlja se kada jedna uputa treba rezultat prethodne upute koja još nije završena. Ovo se rješava tehnikama poput *forwardinga*, gdje se rezultat preusmjerava iz faze izvršenja prije nego što se formalno upiše natrag u registar.

- Kontrolni hazard: Povezan je s uputama koje mijenjaju tok izvršavanja programa, poput skokova i grananja. Kako bi se smanjio utjecaj ovih hazard, koriste se tehnike predviđanja grananja (*branch prediction*).

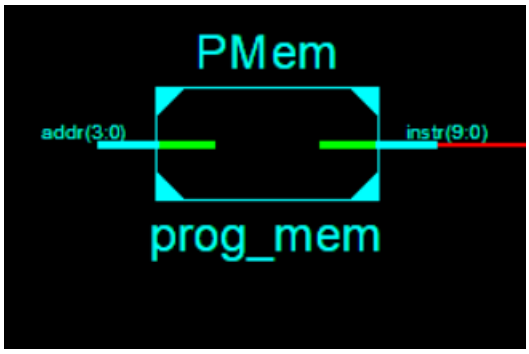
3. REALIZACIJA SKLOPA

Kao što je rečeno u uvodu, 8-bitni procesor s cjevovodom je realiziran na Spartan-3E pločici. Opis sklopa sastoji se od pet modula, četiri „sporedna“ i jedan glavni, odnosno Top Module u kojem su svi povezani. Prvi korak u dizajnu je bio odrediti arhitektu i operacije koje će se realizirati. Nakon toga krenulo se u opis sklopovlja. Prvo su opisane programska i podatkovna memorija.



Slika 3.1. Pločica Spartan-3E

Programska memorija je potrebna odmah u prvoj fazi, odnosno u fazi dohvata instrukcije. U toj fazi se sa 4-bitne adrese, koja se nalazi u programskom brojilu PC, dohvaća naredba iz programske memorije i dovodi na ulaz 10-bitnog instrukcijskog registra (IR). Instrukcijski registri su 10-bitni zbog operacijskog koda naredbi, za kojeg smo se sami odlučili i biti će opisan u sljedećem poglavlju. Programska memorija se sastoji od 16 10-bitnih memorijskih lokacija, zbog čega je programsko brojilo 4-bitno i instrukcijski registri 10-bitni. Po završetku svake faze sadržaj instrukcijskog registra se prenosi u sljedeći IR jer u svakoj fazi procesor mora znati što radi. Nakon faze dohvata naredbe dolazi faza dekodiranja naredbe.



Slika 3.2. Programska memorija

Ulaz programske memorije addr spojen je na registar PC iz top module-a, odnosno predstavlja adresu sa koje se čita instrukcija, a izlaz programske memorije instr predstavlja instrukciju i povezan je na varijablu ir1_wire iz top module-a.

Program koji se izvodi zapisan je u programskoj memoriji u tekstualnoj datoteci, a u memoriju se učitava pomoću $readmemb sistemske funkcije u Verilogu koja se koristi za inicijalizaciju memorijskih blokova iz neke vanjske datoteke. Naredba čita binarne podatke iz tekstualne datoteke i upisuje ih u memorijski blok unutar Verilog koda.

```
0010000110
0010011101
0010101000
0100111010
0111100011
0000000000
0000000000
0000000000
0011101111
0110000111
0000000000
0000000000
0000000000
0011000000
0011110111
0011010010
```

Slika 3.3. Primjer sadržaja programske memorije

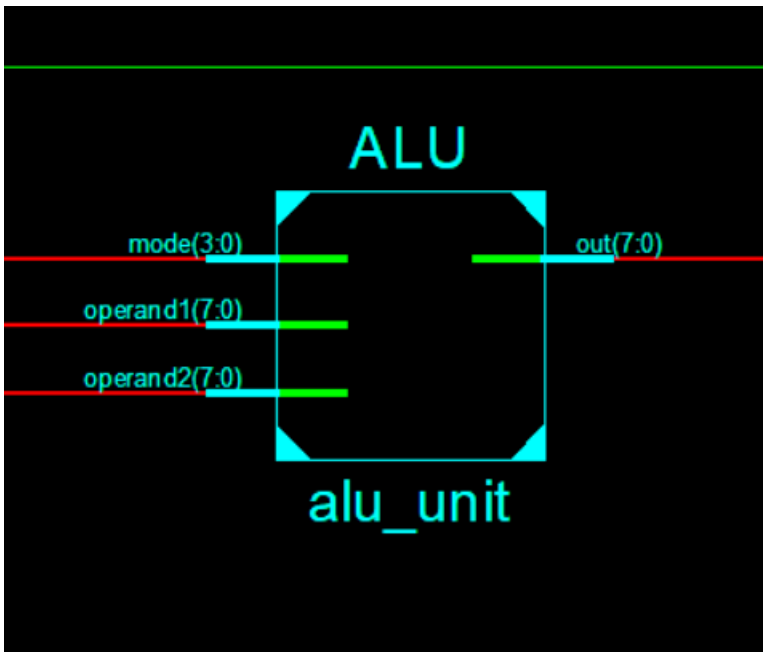
U fazi dekodiranja se na temelju operacijskog koda dohvaćaju operandi. Operacijski kod je za svaku naredbu drugačiji.

```
LOADC :  begin
            OP1 <= IR1[3:0];
            IR2 <= IR1;
         end
ADD :    begin
            case(IR1[3:2])
               0 : OP1 <= REG0;
               1 : OP1 <= REG1;
               2 : OP1 <= REG2;
               3 : OP1 <= REG3;
            endcase
            case(IR1[1:0])
               0 : OP2 <= REG0;
               1 : OP2 <= REG1;
               2 : OP2 <= REG2;
               3 : OP2 <= REG3;
            endcase
            IR2 <= IR1;
         end
```

Slika 3.4. Primjer dohvata operanda

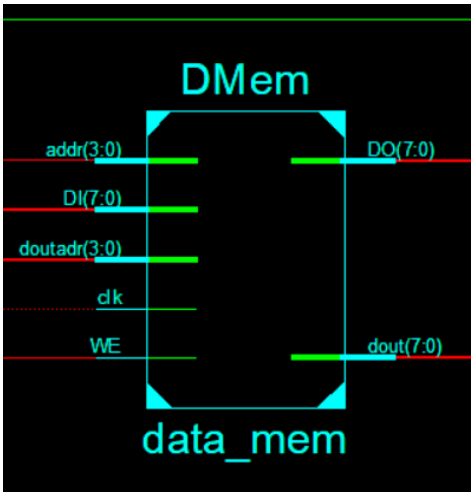
Dakle, sadržaj IR1 se prenosi u IR2 za sljedeću fazu, a na temelju određenih bitova u IR1 se dohvaćaju operandi iz instrukcije, kao što se vidi na slici 3.4.

U fazi izvođenja operacije (*engl. execute*) dohvaćaju se rezultati iz aritmetičko-logičke jedinice i u slučaju naredbe STORE se rezultat sprema u podatkovnu memoriju.



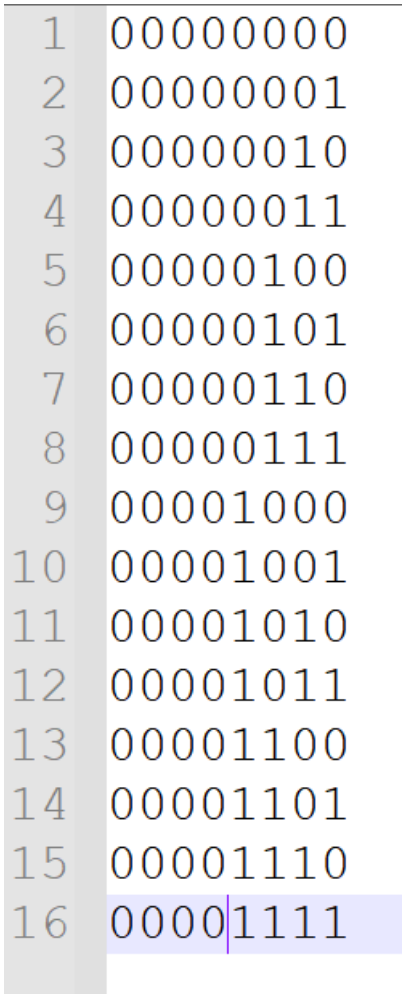
Slika 3.5. Aritmetičko-logička jedinica

Modul ALU ima tri ulaza: 4-bitni ulaz mode, koji je povezan sa IR2[9:6], ponovno zbog operacijskog koda, te na temelju ta 4 bita određuje koju operaciju treba izvesti. Ulazi operand1 i operand2 su 8-bitni. povezani su sa OP1 i OP2 iz top module-a i to su operandi za operaciju, čiji se rezultat dobiva na izlazu out koji je povezan sa varijablom acc iz glavnog modula. U ovoj fazi se izlaz sa ALU sprema u varijablu iz glavnog modula ACC. U slučaju naredbe STORE, aktivira se signal data_WE kako bi se omogućio upis podataka u podatkovnu memoriju. Podaci se upisuju iz jednog od odgovarajućih registara opće namjene (REG0….REG3) na temelju operacijskog koda.



Slika 3.5 Podatkovna memorija

Podatkovna memorija se sastoji od 16 8-bitnih memorijskih lokacija. Prije izvođenja programa u podatkovnu memoriju su ponovno inicijalizirani podaci pomoću sistemske naredbe $readmemb.



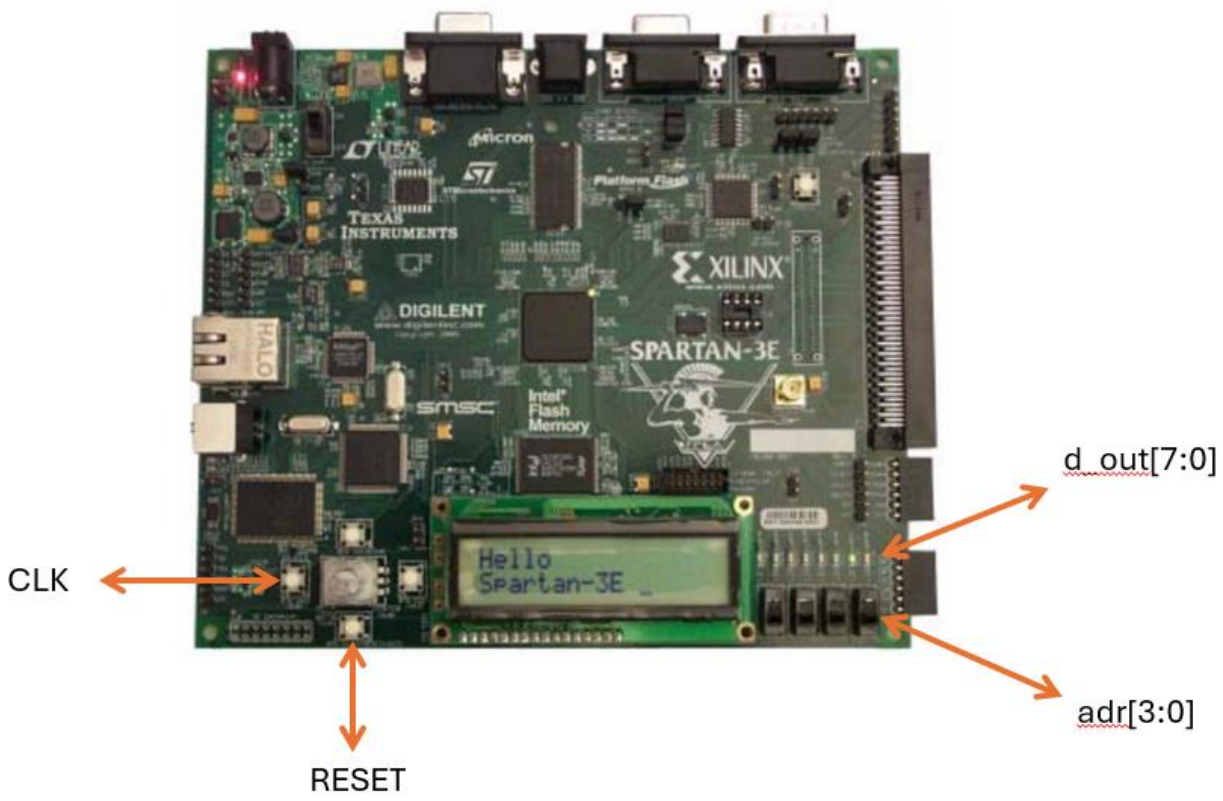
Slika 3.6. Sadržaj podatkovne memorije na početku

Ulaz addr podatkovne memorije povezan je sa varijablom ACC iz top module-a, to je zapravo adresa na koju se trebaju spremiti podaci ili s koje se čitaju podaci, signal WE je povezan sa data_WE i zapravo omogućuje pristup memoriji kad je u stanju logičke jedinice. Ulaz DI predstavlja podatke koji se upisuju u memoriju u slučaju da je WE aktivan i povezan je sa dr_in iz glavnog modula. Doutadr predstavlja adresu koja se namješta ručno na pločici pomoću switch-eva od strane korisnika i spojen je sa ulazom adr glavnog modula koji zapravo predstavlja sklopke. Sadržaj memorije se prikazuje na varijabli dout koji je povezan sa d_out glavnog modula i to su zapravo LED-ice na pločici koje u prirodnom binarnom kodu prikazuju sadržaj memorije. To je zapravo i jedini način na koji se može testirati ispravnost sklopa, a to je praćenje sadržaja memorije nakon svakog takta. Izlaz DO predstavlja podatke koji se vraćaju sa memorijske lokacije određene signalom addr. DO je povezan sa signalom dr_out glavnog modula.

Takt je svim modulima zajednički i u ovom slučaju takt je upravljan od strane korisnika korištenjem gumba. Dakle, svaki pritisak gumba na pločici predstavlja jedan takt, odnosno jedan pozitivni brid clk signala.

U posljednjoj fazi se na osnovu operacijskog koda određuje u koji registar opće namjene se treba spremiti rezultat te se on sprema.

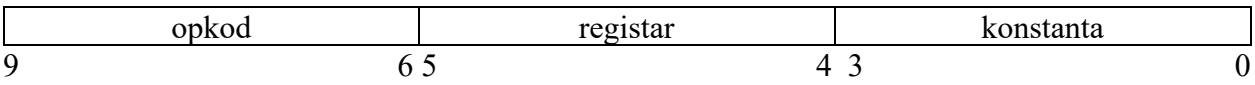
Na slici 4.1. prikazani su korišteni elementi na Spartan-3E pločici. Kao clk signal uzeta je tipka BTN_WEST (D18), i funkcionira na način da jedan pritisak tipke označava jedan pozitivni brid clock signala. Kao reset signal uzeta je tipka BTN_SOUTH (K17), a funkcionira na način da se pritiskom tipke adresa u programskom brojilu resetira na nulu. Klizni prekidači (switch-evi) SW0 (L13), SW1 (L14), SW2 (H18), SW3 (N17) predstavljaju adresu u podatkovnoj memoriji čiji sadržaj se želi prikazati na LED diodama: LED0 (F12), LED1 (E12), LED2 (E11), LED3 (F11), LED4 (C11), LED5 (D11), LED6 (E9), LED7 (F9). Dakle, SW1 predstavlja adr[0], SW2 predstavlja adr[1] i tako redom, a LED0 predstavlja d_out[0], LED1 predstavlja d_out[1] i tako redom. Način spajanja vidljiv je u datoteci *procesor.ucf* koja je priložena.



Slika 4.1. Korišteni elementi na pločici

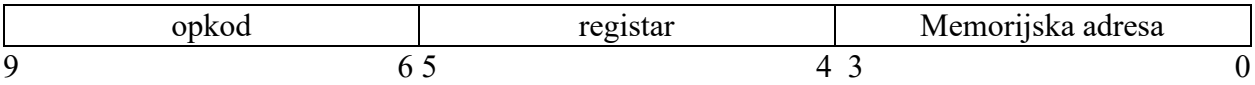
4. NAREDBE

Prva naredba koja je realizirana je učitavanje konstante u registar LOADC. Ova naredba se koristi kada se želi postaviti određena vrijednost u registar bez pristupanja memoriji.



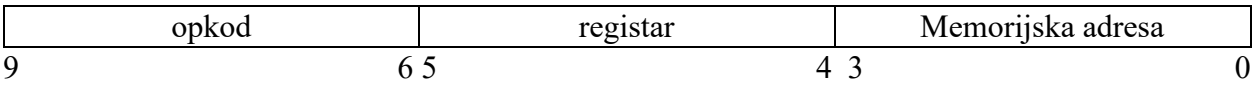
Dakle, četiri najvažnija bita služe za kodiranje naredbe, a LOADC je kodiran kao 0100. Iduća dva bita služe za određivanje registra u koji se upisuje konstanta, npr. R3 je 11, R2 10 itd. Imamo četiri registra pa su dovoljna dva bita. Četiri najmanje važna bita predstavljaju konstantu koja se učitava u registar. Primjer naredbe bi bio LOADC R1 7, u binarnom zapisu 0100_10_0111. Dakle, ovaj binarni broj se nalazi na određenoj adresi u programskoj memoriji.

Iduća je LOAD. Ova naredba služi za učitavanje podatka iz podatkovne memorije u željeni registar.



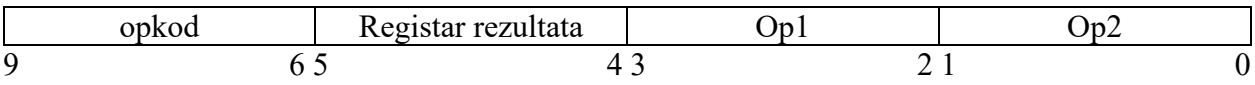
Kao i u prethodnoj, i u ovoj naredbi četiri najvažnija bita služe za kodiranje naredbe. LOAD je kodiran sa 0010, iduća četiri bita predstavljaju registar u koji se učitava podatak iz memorije, a četiri najmanje važna bita predstavljaju memorijsku adresu podatkovne memorije sa koje se učitava rezultat. Primjer ove naredbe: LOAD R0 13, u binarnom zapisu 0010_00_1101.

Naredba STORE predstavlja naredbu u kojoj se rezultat sprema na željenu memorijsku lokaciju.



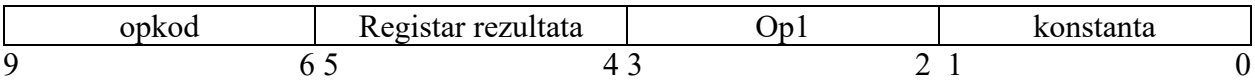
Naredba je kodirana sa 0011, što predstavljaju četiri najvažnija bita, sljedeća dva bita predstavljaju registar čiji se sadržaj želi spremiti u podatkovnu memoriju, a četiri najmanje važna bita predstavljaju adresu u podatkovnoj memoriji na koju se podaci žele spremiti. Primjer: STORE R1 16, u binarnom zapisu 0011_01_1111.

Sljedeća naredba je ADD. Naredba za zbrajanje koja zbraja dva operanda učitana iz registara opće namjene i rezultat sprema ponovno u neki od registara.



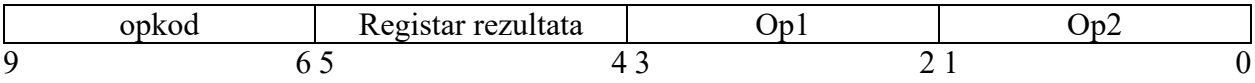
Naredba je kodirana sa 0001, to predstavljaju četiri najznačajnija bita, sljedeća dva bita su registar u koji se sprema rezultat, pa iduća dva prvi operand, te najmanje značajna dva bita predstavljaju drugi operand. Primjer: ADD R2 R1 R0, u binarnom zapisu 0001_10_01_00.

ADDI predstavlja naredbu za zbrajanje sa konstantom, dakle sadržaj željenog registra zbraja sa konstantom.



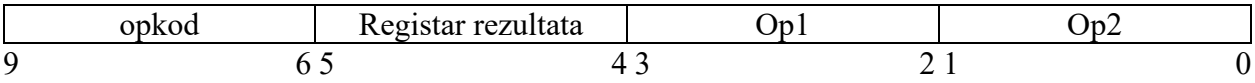
Naredba je kodirana sa 1001, sljedeća dva bita predstavljaju registar rezultata, sljedeća dva operand 1, te zadnja dva konstantu. Primjer: ADDI R3 R0 3, u binarnom zapisu 1001_11_00_11.

Iduća naredba je AND. Predstavlja logičku operaciju AND dva operanda.



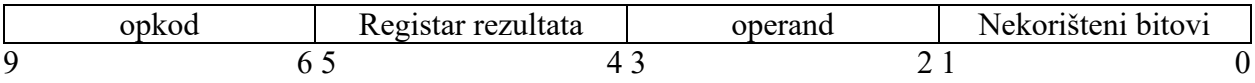
Naredba je kodirana sa 0110 što predstavljaju četiri najvažnija bita, sljedeća dva bita predstavljaju registar rezultata, sljedeća dva predstavljaju operand1, a zadnja dva najmanje važna bita operand2. Primjer: AND R1 R2 R3, u binarnom zapisu 0110_01_10_11.

Sljedeća naredba koja je realizirana je XOR, a predstavlja logičku operaciju XOR dva operanda.



Naredba je kodirana sa 0101 što predstavljaju četiri najvažnija bita, sljedeća dva bita predstavljaju registar rezultata, sljedeća dva predstavljaju operand1, a zadnja dva najmanje važna bita operand2. Primjer: XOR R0 R1 R2, u binarnom zapisu 0101_00_01_10.

Realizirane su također i naredbe SHL i SHR, a predstavlja logičko shiftanje ulijevo (SHL) ili udesno (SHR).



Naredbe su kodirane sa 0111 i 1000, iduća dva bita predstavljaju registar rezultata, iduća dva bita predstavljaju operand, tj. registar čiji se sadržaj namjerava shiftati, a dva najmanje značajna bita ostaju neiskorištena, a u nekoj sljedećoj implementaciji mogu biti iskorišteni u sklopu shiftanja sa konstantom. Primjer ulijevo: SHL R1 R3, u binarnom zapisu 0111_01_11_00; primjer udesno: SHR R0, R2, u binarnom zapisu 1000_00_10_00.

5. DEBOUNCE MODUL

Unutar koda se može vidjeti da se koristi i instancira debounce modul. Debounce modul u Verilogu koristi se za uklanjanje smetnji koje nastaju kada se fizička tipka pritisne ili otpusti. Kada se tipka pritisne, kontakti unutar tipke ne uspostavljaju trenutačni i stabilni kontakt, već dolazi do više trenutnih zatvaranja i otvaranja kontakata zbog mehaničkog podrhtavanja *(eng. bounce*). To uzrokuje pojavu više lažnih signala u kratkom vremenskom razdoblju, što može dovesti do nepravilnog ponašanja u digitalnom sustavu. U našem slučaju morali smo debounce-ati clk i reset signale. Ovo nije predstavljalo veći problem jer je kod dostupan na internetu.

6. PRIMJER PROGRAMA

U ovom poglavlju će biti objašnjen jedan program sa sadržajem memorije na početku i na kraju izvršavanja. Na primjeru će biti objašnjena potreba za korištenje tzv. NOP-a, odnosno ubacivanja praznog hoda u izvođenje radi mana cjevovodne arhitekture.

```
0010000110//LOAD R0 6, U RO UPIŠI 61
0010011101//LOAD R1 13, U R1 UPIŠI 19
0010101000//LOAD R2 8, U R2 UPIŠI 2
0100111010//LOADC R3 10, U R3 UPIŠI 10
0111100011//LSH R2 R0, SHIFTAJ RO ULIVO I SPREMI U R2(01111010)
0000000000//NOP
0000000000//NOP
0000000000//NOP
0011101111//STORE R2 15, NA ADRESU 15 SPREMI R2(01111010)
0110000111//AND RO R1 R3, AND R1 R3 I SPREMU U R0(00000010)
0000000000//NOP
0000000000//NOP
0000000000//NOP
0011000000//STORE R0 0(2)
0011110111//STORE R3 7 (10)
0011010010//STORE R1 2(19)

SADRŽAJA DATA MEM NA POČETKU: 1 27 43 18 5 7 61 111 2 87 44 152 101 19 72 59
                            [0 1  2  3  4 5  6  7  8 9  10 11  12  13 14 15]

SADRAŽAJ DATA MEM NA KRAJU: 2 27 19 18 5 7 61 10 2 87 44 152 101 19 72 122
                          [0 1  2  3  4 5 6  7  8 9  10 11  12  13 14 15]
```

Slika 6.1. Primjer programa: p53

Prilikom prva 4 pritiska clk signala obavlja se prvi ciklus koji obuhvaća sve faze procesora, ali nema vidljivih promjena u sadržaju memorije registara. Nakon petog pritiska se u registar R0 upisuje broj sa adrese 6 (broj 61), ali se to također ne vidi. Nakon sljedećeg pritiska se u R1 upisuje broj sa adrese 13 (broj 19), zatim na idući takt se u R2 upisuje broj sa adrese 8 (broj 2) i na idućem taktu se u R3 upisuje konstanta 10. Nakon što su svi registri "napunjeni", u sljedećem taktu izvodi se naredba SHL (shift-left) registra R0 i rezultat se sprema u R2 (rezultat je 122).

Zatim je vidljivo 3 takta izvođenja NOP operacije. Mana cjevovodne arhitekture je ta da je pojedinoj operaciji potrebno 3 takta za izvršavanje. Da bi bili sigurni da je u željenom registru ispravna vrijednost koju mi dalje želimo koristiti, ubacuje se operacija NOP kako bi procesor imao vremena završiti tu instrukciju.

Na idući takt se sadržaj registra R2 sprema na adresu 15 i to je prva promjena u podatkovnoj memoriji koja je vidljiva na LED-icama, na način da se na kliznim prekidačima (switch-evima) namjesti adresa 15 (1111) i provjeri se da li se na LED-icama pojavljuje broj 122, tj 01111010.

Dalje ponovno slijede 3 takta operacije NOP pa se na idućem taktu izvršava spremanje sadržaja R0 na adresu 0 (broj 2). U idućem taktu sadržaj registra R3 (broj 10) se sprema na adresu 7, i u zadnjem taktu se sadržaj registra R1 (broj 19) sprema na adresu 2. Time završava izvođenje ovog programa. Testiranjem rada na pločici može se uvidjeti da je rad procesora ispravan. Sadržaj memorije prije i nakon izvođenja prikazan je na slici 6.1. Prilikom rada testirano je više primjera programa koji su slični navedenom.

7. ZAKLJUČAK

U sklopu ovog seminarskog rad realiziran je jednostavan 8-bitni procesor s cjevovodom koji podržava osnovne instrukcije. Analizirani su ključni aspekti procesora s cjevovodom, od teorijskih osnova do konkretne primjene u FPGA okruženju. Pokazano je kako cjevovodna arhitektura omogućava paralelizaciju instrukcija, čime se postiže značajno povećanje brzine obrade podataka.

Projekt je u početku bio vrlo zahtjevan jer je jedan od najtežih koraka bio osmisliti arhitekturu, operacije te način kodiranja operacija. Nakon što je osmišljena arhitektura, dalje se razvoj projekta temeljio na principu pokušaj-pogreška sve dok ne proradi te je za realizaciju ovog sklopa jako bitno teorijsko znanje principa rada procesora sa cjevovodom, kao i Veriloga. Da bi se ispravno napisao kod, bilo je potrebno do detalja poznavati što se odvija u svakoj fazi rada. Nakon što je kod bio napisan sa par osnovnih operacija, još teži korak je bio testiranje sklopa jer je bilo potrebno osmisliti ispravan program upravo takav da se ne dogodi strukturni sukob takav da je potrebno korištenje NOP-a, zbog čega je također važno poznavanje rada procesora. Nakon što je kod bio uspješno simuliran, dodavanje više operacija i sama implementacija na pločici nije bila teška jer je unaprijed postojala ideja kako da se sklop testira. Iz svakog problema na kojeg smo naišli i kojeg smo riješili je proizašla ogromna količina znanja kako o radu procesora tako i o kodiranju u Verilogu.

Budući da je procesor s cjevovodom sam po sebi jako kompliciran tako otvara i brojne mogućnosti za poboljšanja u budućnosti. Prvo poboljšanje bi svakako bilo dodavanje više operacija, tipa operacije za uvjetno grananje JUMP, koja može stvarati brojne probleme.

U ovoj verziji je implementirana verzija logičkog pomaka za jedan, a u sljedećoj verziji se može implementirati pomak za neki određeni broj. Ogromno poboljšanje bi bilo osmisliti kako da procesor sam shvati kada treba implementirati NOP itd.