Izvedba aritmetičko-logičke jedinice 8-bitnog računala na FPGA sučelju

Rogić, Vid

Undergraduate thesis / Završni rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka, Faculty of Engineering / Sveučilište u Rijeci, Tehnički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:190:030908

Rights / Prava: Attribution 4.0 International/Imenovanje 4.0 međunarodna

Download date / Datum preuzimanja: **2024-05-17**

Repository / Repozitorij:

Repository of the University of Rijeka, Faculty of Engineering

SVEUČILIŠTE U RIJECI

TEHNIČKI FAKULTET

Preddiplomski sveučilišni studij elektrotehnike

Završni rad

Izvedba aritmetičko-logičke jedinice 8-bitnog računala na FPGA sučelju

Rijeka, rujan 2023.

Vid Rogić

0069086881

SVEUČILIŠTE U RIJECI

TEHNIČKI FAKULTET

Preddiplomski sveučilišni studij elektrotehnike

Završni rad

Izvedba aritmetičko-logičke jedinice 8-bitnog računala na FPGA sučelju

Mentor: doc.dr.sc. Ivan Volarić

Rijeka, rujan 2023.

Vid Rogić

0069086881

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
POVJERENSTVO ZA ZAVRŠNE ISPITE

Rijeka, 21. ožujka 2022.

Zavod: **Zavod za automatiku i elektroniku**
Predmet: **Digitalna elektronika**
Grana: **2.03.03 elektronika**

ZADATAK ZA ZAVRŠNI RAD

Pristupnik: **Vid Rogić (0069086881)**
Studij: Preddiplomski sveučilišni studij elektrotehnike

Zadatak: **Izvedba aritmetičko-logičke jedinice 8-bitnog računala na FPGA sučelju / Implementation of the 8-bit arithmetic-logic unit on the FPGA board**

Opis zadatka:

Aritmetičko-logička jedinica ima dva 8-bitna podatkovna ulaza i jedan 8-bitni podatkovni izlaz. Jedinica će izvršavati aritmetičke operacije (zbrajanje i oduzimanje) te logičke operacije (I, ILI, XILI, NE) nad ulazima. Odabir operacije se vrši pomoću dodatna tri kontrolna ulaza. Osim podatkovnih izlaza jedinica mora imati dva dodatna izlaza za zastavice. Prva zastavica će poprimiti vrijednost logičke jedinice ako je rezultat izvršene operacije '00000000', dok će druga
zastavica poprimiti vrijednost logičke jedinice ako se prilikom izvršavanja operacije dogodio prelijev u 9 bit.
Zadani sklop potrebno je realizirati na FPGA sklopu pomoću VHDL jezika. Osnovne kombinacijske i sekvencijske sklopove potrebno je ponašajno modelirati, dok složenije sklopove kao i cijelu jedinicu strukturno. Potrebno je dizajnirati sklop koji će se preko GPIO konektora spojiti sa FPGA sklopom na kojemu će se preko DIP prekidača spojenih u pull-down konfiguraciju simulirati podatkovni i kontrolni ulazi. Izlaz iz aritmetičko-logičke jedinice potrebno je prikazati na 7-segmentnim pokazivačima.

Rad mora biti napisan prema Uputama za pisanje diplomskih / završnih radova koje su objavljene na mrežnim stranicama studija.

Vid Rogić

Zadatak uručen pristupniku: 21. ožujka 2022.

Mentor:

Doc. dr. sc. Ivan Volarić

Predsjednik povjerenstva za završni ispit:

Prof. dr. sc. Viktor Sučić

Izjava o samostalnoj izradi završnog rada

Ovom izjavom potvrđujem da sam završni rad s temom: Izvedba aritmetičko-logičke jedinice 8-bitnog računala na FPGA sučelju napravio samostalno uz pomoć mentora doc.dr.sc. Ivana Volarića koristeći literaturu referiranu u ovom završnom radu.

Vid Rogić

_____

Zahvala

Želim se zahvaliti ovim putem profesoru i mentoru doc.dr.sc. Ivanu Volariću na ideji vrlo zanimljivog i informativnog završnog rada te pomoći i strpljenju iskazanom tijekom izvršavanja završnog rada.

Također želim se zahvaliti na ustupljenoj razvojnoj pločici korištenoj za simuliranje programa.

SADRŽAJ

1. UVOD 6

2. FPGA 7

2.1. Opis rada FPGA sklopa 7

2.2. Programiranje FPGA-a 9

2.3. Primjena i povijest FPGA-a 9

3.VHDL 10

3.1.Kratki opis VHDL-a 10

3.2. Glavni dijelovi koda 10

3.3. Strukturalni dizajn 12

3.4. Ponašajno modeliranje 13

3.5. Varijable, portovi i signali 15

4. Aritmetičko-logička jedinica 17

4.1. Opis aritmetičko-logičke jedinice 17

4.2. Operacije aritmetičko-logičke jedinice 17

4.3. Integrirani krug 74181 18

4.4. Zastavice 21

5. Eksperimentalna pločica Altera DE10-Lite i programsko sučelje Quartus 22

5.1. Opis sučelja i pločice 22

5.2. Quartus prime 23

6. Opisivanje rada ALU jedinice kroz komponente 24

6.1. NE sklop 24

6.2. I sklop 25

6.3. ILI sklop 27

6.4. XILI sklop 28

6.5. Poluzbrajalo 30

6.6. Potpuno zbrajalo 31

6.7. Paralelno zbrajalo 34
6.8. Drugi komplement 36
6.9. Oduzimanje 40
6.10. Spajanje sklopa u cjelinu i multipleksor 43
7. Izrada fizičkog sklopa 46
8. Zaključak 48
9. Literatura 49
10. Prilog 50
11. Sažetak 59

1. UVOD

U ovom radu na temelju dobivenog zadatka za završni rad objašnjen je rad *FPGA (eng. Field programmable gate array)* sučelja te njegovo programiranje pomoću programskog jezika *VHDL* (*eng. VHSIC Hardware Description Language*) kroz implementaciju sklopa aritmetičko-logičke jedinice te kasnijim testiranjem izrađenog sklopa na prototipnoj pločici.

Aritmetičko-logička jedinica sastavni je dio svakog računalnog procesora, kako onih razvijenih u početku računalne revolucije tako i u modernim računalima uz proširenje njene funkcionalnosti. Njen zadatak je izvođenje operacija nad binarnim brojevima koji dolaze sa sabirnica na njene ulaze te na izlaznoj sabirnici vraća rezultat operacije. Kako samo ime kaže, aritmetičko-logička jedinica dizajnirana je da provodi više operacija, prvenstveno one aritmetičke: zbrajanje, oduzimanje, množenje i dijeljenje te logičke operacije nad binarnim brojevima što su logičke operacije NE, I, ILI, itd.

FPGA je uređaj namijenjen implementaciji koda koji opisuje rad logičkih sklopova te ih simulira koristeći unutarnje logičke ćelije koje se mogu reprogramirati. Koristan je u mnogim granama industrije zbog svog brzog izvršavanja te kod dizajniranja digitalnih integriranih krugova prilikom testiranja. Za njegovo programiranje koriste se programski jezici za opisivanje sklopovlja poput *VHDL*-a i *Verilog*-a. U ovom radu korišten je Alterin *FPGA* DE10Lite koji se koristi za podučavanje pri programiranju sklopova te je korišten *VHDL* jezik.

2. FPGA

2.1. Opis rada FPGA sklopa

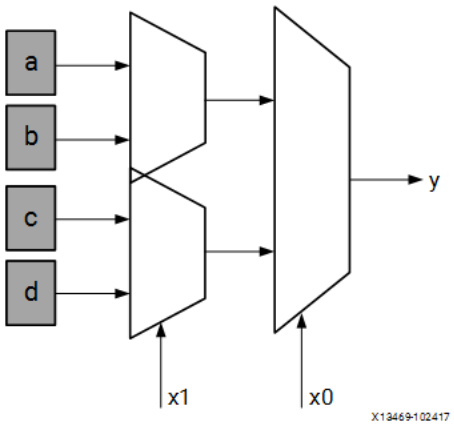
Kao što je rečeno u uvodu *FPGA* je programibilni poluvodički sklop koji daje dobar kompromis između mikroprocesora i klasičnih integriranih krugova. Sklop se sastoji od mnogo logičkih blokova koji su međusobno povezani preko programibilne sabirnice, kao i sa ulazno-izlaznim pinovima. Broj logičkih blokova određuje mogućnost *FPGA*-a za implementaciju složenijih logičkih krugova. Na slici 2.1. je prikazan *FPGA* u *SMD* (*eng. surface mount device*) izvedbi.

Slika 2.1. Primjer modernog FPGA čipa u SMD izvedbi.

FPGA je u suštini sklop s paralelnim izvođenjem naredbi što ga čini najbržim načinom za procesuiranje naredbi. Umjesto pohranjivanja informacija u registre i radnu memoriju, *FPGA* može iskoristiti veći broj logičkih blokova za primanje i slanje podataka te zaobići tradicionalnu obradu informacija koja ovisi o radnom taktu procesora.

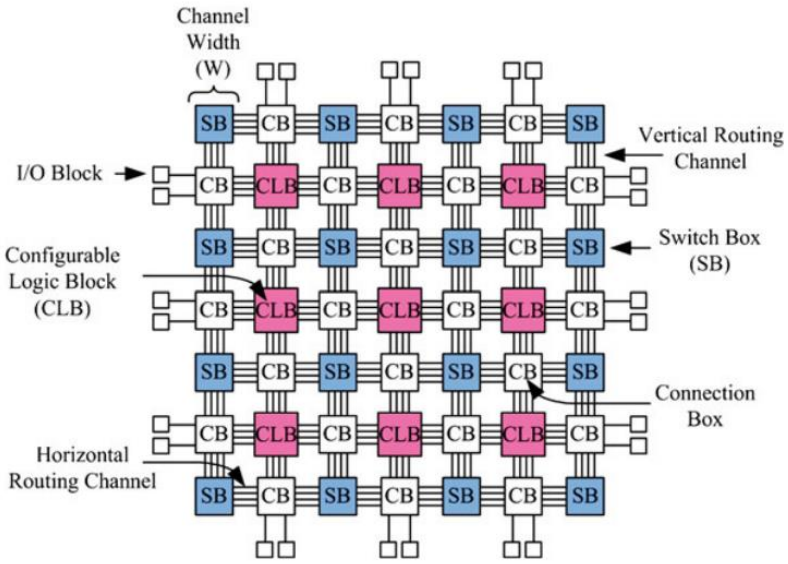
Logički blokovi sastoje se od *LUT (eng. look-lp table)* sklopa što bi u prijevodu bile tablice istinitosti i bistabila. *LUT* sklopovi mogu izvršiti bilo koju logičku funkciju s N ulaznih varijabli. To postižu spajanjem sličnom multiplekseru (Slika 2.2.) za kojega znamo da se pomoću dovoljno ulaza te invertiranjem ili neinvertiranjem ulaznih bitova može postići bilo koja booleanova jednadžba.

Bistabili u logičkim blokovima imaju svrhu memorije jednog ili više bitova kroz više radnih taktova, što ih čini idealne za izvedbu sekvencijalnih sklopova.



Slika 2.2. Primjer povezivanja LUT.

Interna programibilna sabirnica (Slika 2.3.) u *FPGA*-u raspoređena je u mrežu (matricu), dok se na čvorištima sabirnice nalaze programibilni sklopovi koji omogućuju spajanje bilo koje kombinacije prethodno opisanih logičkih blokova.



Slika 2.3. Programibilna sabirnica.

Po pitanju ulazno-izlaznih pinova *FPGA* je sličan mikrokontrolerima gdje se pinovima mogu dodijeliti ulazne ili izlazne varijable, ali u puno većem broju.

2.2. Programiranje FPGA-a

Programiranje *FPGA* sklopa vrši se u više koraka:

1. Prvi korak je samo programiranje odnosno određivanje strukture, ponašanja te spoja komponenti unutar sklopa, pisanog u programskom jeziku.
2. Drugi korak je sinteza koda koju obavljaju softwareska sučelja. Ona izvode pretvaranje koda visoke razine u takozvanu „netlistu“ koju *FPGA* može obraditi za realizaciju interkonekcija.
3. U „place-and-route“ fazi software realizira prethodno izrađenu netlistu odnosno mapira ju na specifični unutarnji raspored *FPGA* sklopa kojeg programiramo.
4. Generacija konfiguracijskog bitstreama služi generaciji završne datoteke koju šaljemo s računala prema samom sklopu.
5. U koraku konfiguracije nakon slanja bitstreama, memorija samog sklopa prima naredbe i distribuira ih *FPGA*-u putem *JTAG* (*eng. Joint Test Action Group*) protokola za simulaciju, otkrivanje pogrešaka i programiranje čime završavamo proces te je sučelje spremno za korištenje.

2.3. Primjena i povijest FPGA-a

Ideja za *FPGA* dolazi od njegovih prethodnika PLD (*eng. Programmable logic device*) i PLA (eng. *programmable logic array*) odnosno programibilnih logičkih sklopova koji su koristili I i ILI vrata spojenih na programibilnu matricu slično kao i kod *FPGA* sklopa. Bitna razlika između dva sklopa sastoji se u tome što se PLA može programirati samo jednom dovođenjem određenog napona na spojeve poveznica te trajno spajanje prethodno dizajniranog sklopa korištenjem programibilnih logičkih vrata. Ubrzani razvoj *FPGA*-a kreće 80-ih i 90-ih godina prošlog stoljeća kada se u integrirane krugove ugrađuje sve veća količina blokova, te se interpretacija koda prepušta softwareu koji sastavlja konačan raspored. Predvodnici koji se ističu u ovom polju su tvrtke Altera i Xilinx.

FPGA sklopovi našli su svoju primjenu u svim granama elektrotehnike posebice u vojnoj industriji, medicini, istraživačkom području, itd. Služe za obradu signala, provođenje kompleksnih izračuna (npr. mrežna oprema), obradu podataka senzora, itd.

3.VHDL

3.1.Kratki opis VHDL-a

VHDL kao što njegovo ime u prijevodu govori je programski jezik namijenjen opisivanju sklopovlja, razvijen od strane američkog ministarstva obrane kroz 1980-e godine za potrebe razvitka integriranih krugova velike brzine. Iako sam program nije uspio u razvitku novih tehnologija, *VHDL* je postao industrijski standard. Baziran je na još jednom programskom jeziku razvijenom od strane američke vlade, „ADA“ . Ova dva jezika slična su u sintaksi i općim pravilima za pisanje koda. Institut IEEE podržao je *VHDL* standardizirajući ga 1987. godine od kada kreće njegova široka upotreba izvan vojnog programa.

Program napisan u *VHDL* kodu je prenosiv (s uređaja na uređaj) i njegovi dijelovi mogu se prenamijeniti za uporabu u drugim programima. Kod ne razlikuje velika i mala slova, te zahtjeva završavanje linija interpunkcijama (, ;). Ovaj programski jezik možemo nazvati programskim jezikom za simuliranje toka podataka što se očituje paralelizmom izvršavanja operacija kao što i sam *FPGA* tako izvršava svoje naredbe. *VHDL* možemo nazvati strogo tipiziranim jezikom zbog potrebe za pridodavanjem vrste podatka varijabli pri njenoj deklaraciji.

3.2. Glavni dijelovi koda

VHDL kod možemo podijeliti u više dijelova koje svaki program kojeg mislimo simulirati ili sintetizirati mora imati. Prije pisanja samog koda vrši se pozivanje knjižnica u kojima su sadržane sve potrebne naredbe koje koristimo kroz program.

Prvi dio se naziva e*ntity* koji opisuje sučelje komponente. U *entity* opisu, sklopu dajemo naziv, te opisujemo sučelje povezivanja s ostalim komponentama ili izlaznim pinovima. To obično podrazumijeva deklariranje ulaznih i izlaznih varijabli, ali i ostalih značajki. U nastavku je pokazan primjer *entity* bloka project1 koji sadrži dva ulazna i jedan izlazni signal:

```
library ieee;
use ieee.std_logic_1164.all;

entity project1 is

    port(
        c : out std_logic;
        b : in std_logic_vector (0 to 7);
        a : in std_logic_vector (0 to 7);
        );

end entity project1;
```

Drugi dio koda koji obavezno moramo opisati naziva se *Architecture* (*eng. arhitektura*). Ako je *entity* opisao izgled sklopa prema vanjskom svijetu i drugim komponentama, arhitektura sklopa opisuje kako se sklop ponaša. U nastavku je prikazan primjer arhitekture sklopa koji obavlja I i ILI funkciju:

```
architecture structural of and_or is
        begin

            izlaz1 <= a and b;
            izlaz2 <= a or b;


end structural;
```

U ovom programskom jeziku *entity* i *arhitecture* su razdijeljeni za razliku od ostalih jezika što pruža mogućnost da pojedini *entity* ima više arhitektura. To omogućuje odabir odgovarajuće arhitekture za iskorištavanje sklopa u različitim situacijama. Deklaracija odabira arhitekture vrši se naredbom *Configuration*. U sljedećem primjeru *entitiy*-u „mux“ pridodjeljujemo prethodno napisanu arhitekturu *dataflow*.

```
CONFIGURATION mux2 OF mux IS

    FOR dataflow
    END FOR;

END mux2;
```

3.3. Strukturalni dizajn

Funkcionalnost sklopa možemo opisati na više načina u *VHDL*-u koristeći različite metode modeliranja. U sklopu izrađenom u sklopu ovog završnog rada najviše dolazi do izraza upravo strukturalni dizajn sklopa. On je baziran na činjenici da već napravljene dijelove sklopa koristimo kao komponente u složenijem sklopu. Ovaj pristup uspoređujemo s modularnim dizajnom gdje module povezujemo u cjelinu. Možemo reći da time razmišljamo kao i kod dizajna fizičkih sklopova gdje komponente spajamo vodičima i sabirnicama za ostvarenje konačnog cilja.

Ovu funkcionalnost izvršavamo naredbom *Component*. Za ispravan rad ove naredbe potrebna su nam tri dijela koda. Prvi dio je ispravno sastavljen prijašnji *entity* kod sa svim potrebnim ulazima i izlazima. Svi ulazi i izlazi ne moraju nužno biti iskorišteni za njegovo korištenje kao komponente, ali u kasnijim upotrebama tog *entity*-a njih ne možemo dodavati jer oni jednostavno nisu opisani i ne možemo predvidjeti njihovu funkcionalnost. Drugi dio sastoji se od pozivanja komponente koji se postavlja na početku arhitekture složenijeg sklopa. Sadrži slični sintaksu kao i deklaracija *entity*-a te označuje uporabu komponente u daljnjoj razradi arhitekture. Završni dio je sama uporaba komponente kojoj dajemo naziv i pozivamo unutar arhitekture. Komponenta se povezuje „port mapom“ gdje se ulazi i izlazi povezuju s ulazima, izlazima ili signalima trenutne arhitekture. U nastavku je prikazan primjer pozivanja komponente u kodu.

```
architecture structural of sklop is

    begin

        component and_or
            port(
                a : in std_logic_vector;
                b : in std_logic_vector;
                izlaz1 : out std_logic_vector;
                izlaz2 : out std_logic_vector
                );
        end component;


        komponenta1 : and_or
            port map(
                a => x,
                b => y,
                izlaz1 => i1,
                izlaz2 => i2
                );

end structural;
```

Strukturalni kod glavna je značajka ovakve vrste modeliranja sklopa gdje varijablama direktno pridodajemo vrijednosti (a <= b). Ove naredbe izvršavaju se istovremeno te omogućuju simuliranje dijelova onakvima kakvi se javljaju u stvarnim integriranim krugovima. Najčešće naredbe u ovakvom modeliranju su logičke operacije, *when else* (naredba za odabir izvora podatka), aritmetičke operacije nad kompleksnim varijablama, itd.

3.4. Ponašajno modeliranje

Ponašajno modeliranje sličnije je standardnim programskim jezicima u tome što se naredbe izvršavaju slijedno. Izvršavanje naredbi ovisi o brzini vanjskog takta i nije trenutno, ali nam omogućava klasične metode programiranja kao npr. petlje ili uvjetovane naredbe.

Ovaj način modeliranja možemo prepoznati po početnoj naredbi *process* unutar arhitekture. Ova naredba može biti opisana svojim imenom te nakon nje slijedi lista osjetljivosti signala. Lista osjetljivosti signala vrlo je bitna u *VHDL*-u zbog pozivanja na izvršavanje svakog odvojenog procesa. Pri promjeni vrijednosti varijable iz liste osjetljivosti proces se pokreće i izvršava naredbe. U deklarativnom dijelu procesa deklariramo varijable koje ćemo koristiti unutar samog procesa koje moramo prenijeti na same signale ili izlaze za njihovu daljnju uporabu. Naposljetku, potrebno je opisati ponašanje sklopa korištenjem sekvencijskih naredbi, od čega one najčešće su:

1. IF ELSE

IF ELSE naredba provjerava istinitost izraza napisanog unutar zagrada te sukladno istinitosti izvršava naredbu ili prelazi na izvršavanje blokova naredbi napisanih u *ELSE* odnosno *ELSIF* blokovima. Ova naredba mora biti napisana unutar *process* bloka zbog toga što svakom promjenom varijable iz liste osjetljivosti ponovno se provjeravaju sve istinitosti u hijerarhijskom nizu od *IF* do svakog sljedećeg *ELSE* ili *ELSIF* bloka što ne možemo obaviti paralelno. U nastavku je prikazan tipičan primjer korištenja *IF* naredbe:

```
if (a > b) then
    a := b + c;
elsif (a < b) then
    a := b - c;
else
    a := 0;

end if;
```

2. LOOP (petlje)

Petlje su još jedna klasična naredba korištena u gotovo svim programskim jezicima. Krucijalne su za automatiziranje koda te slijedno izvršavanje naredbi gdje zadatak izvršavamo u više iteracija koje bi bile previše opsežne za pojedinačno kodiranje ili kada ne znamo koliko iteracija nam je potrebno za dolazak do rješenja. U *FOR* petlji zadajemo varijablu koja se mijenja od početne do krajnje vrijednosti u određenim koracima što deklariramo nakon samog pozivanja petlje te dio koda koji izvršavamo u zadanim iteracijama.

Osim *FOR* petlje postoji i *WHILE* petlje gdje umjesto preddefiniranog broja ponavljanja, naredbe izvršavamo sve dok se početna tvrdnja ne zadovolji. U nastavku su prikazane obije najčešće petlje na jednostavnom primjeru:

```
--FOR petlja
for i in 0 to 7 loop
  if a(0) > a(i) then
    a := a+1;
  end if;
 end loop;
```

```
--WHILE petlja
while a > b loop
  if a(0) > a(1) then
        a := a+1;
    end if;
    b := b-1;
end loop;
```

3.5. Varijable, portovi i signali

U svim dijelovima koda susrećemo se s različitim varijablama te ovisno o njihovom mjestu deklaracije dijelimo ih na tri skupine. U početku koda ulaze i izlaze nazivamo „Portovima“, oni imaju mogućnost povezivanja s vanjskim svijetom te je bitno odabrati dovoljan broj i vrstu portova koje možemo kasnije povezati na hardwareu. U portovima moramo naznačiti dali su oni ulaznog (in) ili izlaznog (out) tipa. Tu još postoje i inout portovi za korištenje u složenijim sustavima koji su dvosmjerni.

Kada u arhitekturi moramo povezati naredbe ili komponente koristimo *„signal“*. *Signal* možemo smatrati vodom (žicom) ili sabirnicom. Posljednju skupinu pronalazimo u *process* naredbi i nazivamo je *variable*. Ova varijabla je slična signalu utoliko što ne može direktno kontrolirati izlaze te se mijenja i izvodi samo kada se pokreće *process* naredba.

Svakoj varijabli, portu i signalu moramo pridodati tip i veličinu. Najjednostavniji tip jest *std_logic* tip koji možemo usporediti s „Boolean“ tipom varijable drugih programskih jezika što označava jedan bit nad kojim vršimo operacije. Polje *std_logic* tipa naziva se *std_logic_vector* koji određuje skup ili vektor binarnih brojeva. Ovdje govorimo o vektoru iz razloga što ga možemo koristiti kao cjelinu, ali i pojedinačno svaki od njegovih bitova. Deklarira se zajedno sa svojim rasponom odnosno brojem bitova te je potrebno odrediti način zapisa *MSB*-a (bita najveće težine) i *LSB*-a (bita najmanje težine). Ukoliko je korištena deklaracija (x *downto* y) za dodjeljivanje raspona *MSB* biti će a(y) bit, a ako je korištena (x *to* y) *MSB* će tada biti a(x).

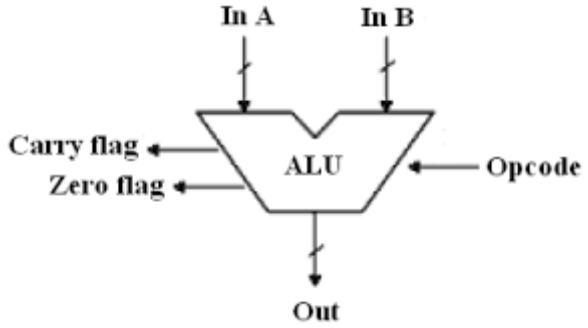
Pored najčešćih tipova varijabli postoje i *signed* i *unsigned* tipovi opisane u knjižnici *numeric_std.* Ovi tipove varijabli opisuju binarne brojeve na isti način kao i *std_logic_vector* no bitno se razlikuju po tome što se nad *signed* i *unsigned* varijablama mogu vršiti osim logičkih i aritmetičke operacije direktno u kodu (+, -, *,…) te će ih *FPGA* procesuirati na točan način. *Signed* varijabla za razliku od *unsigned* varijable može razlikovati pozitivne i negativne brojeve (sustav s drugim komplementom). Zadnji učestali tip varijable su realni i cijeli brojevi (*real* i *integer*). Oni predstavljaju kao što samo ime kaže brojeve u njihovom standardnom decimalnom formatu, opisuju se s 32 bita, te su korisni u procesnim naredbama. U nastavku je prikazan primjer deklaracije varijabli:

```
a : in std_logic_vector (2 downto 0) := "110";

--a(0)=0 a(1)=1 a(2)=1

b : in std_logic_vector (0 to 2) := "110";

--b(0)=1 b(1)=1 b(2)=0

c : out unsigned(7 downto 0);
d : out signed(7 downto 0);

signal e : integer;
```

4. Aritmetičko-logička jedinica

4.1. Opis aritmetičko-logičke jedinice

ALU (eng. *Arithmetic Logic Unit*) je centralni i sastavni dio procesora zaslužan za aritmetičke i logičke operacije izvršene na operandima dovedenima na njen ulaz. U suštini, to je složeni digitalni sklop jedinstvene namjene specifičan za svaki procesor posebno. Količina operacija te veličina ulaza ovise o procesorskoj arhitekturi. Ovaj sklop može se opisati kao sklop visoke brzine te namjenski sklop za obavljanje više radnji. Na sljedećoj slici (Slika 4.1.) prikazan je simbol aritmetičko-logičke jedinice s njenim ulazima i izlazima.



Slika 4.1. Prepoznatljivi simbol ALU.

ALU nije nužno samo dio procesora već se može naći i kao odvojeni sastavni dio ranijih računala. Napreci u tehnologiji sveli su *ALU* na svega komadić procesorske pločice. Kompleksnost i operacije koje *ALU* jedinice mogu izvršiti razvijale su se zajedno s *CMOS* (*eng. complementary metal-oxide semiconductor*) tehnologijom.

4.2. Operacije aritmetičko-logičke jedinice

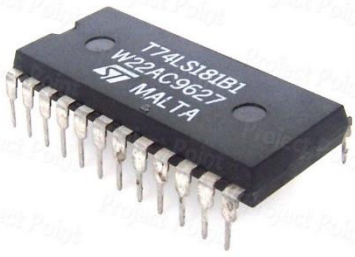
ALU izvršava više logičkih i aritmetičkih funkcija nad ulaznim varijablama ovisno o vrijednosti adresnog ulaza. Dok su se u početcima *ALU* jedinica realizirali sklopovi za operacije I, ILI, NE, XILI te samo zbrajanje brojeva, u novijim procesorima koriste se adresni ulazi 8 bitne širine što omogućuje do 64 operacije provedene nad operandima.

Ovdje možemo navesti istaknute operacije koje štede na procesorskom taktu kao što su oduzimanje, dijeljenje, pomicanje ulijevo i udesno (shift-right, shift-left), rotiranje (rotate-right, rotate-left), kvadriranje, korjenovanje itd. Spominjući dijeljenje i korjenovanje podrazumijeva se kako moderni procesori *ALU* jedinicom mogu manipulirati osim cijelih i realnim brojevima. U tome im pomaže i veći broj *ALU* jedinica koje se pojavljuju razvojem višejezgrenih procesora.

ALU jedinicom u procesorima upravlja kontrolna jedinica procesora (CU – control unit) koja je zaslužna za postavljanje adresnog ulaza te dovođenja operanada u registre kako bi ih *ALU* mogla procesuirati. Nakon izvršene operacije rezultat se sprema u bistabile sve dok se rezultat ne upotrijebi odnosno prenese na izlaznu sabirnicu.

4.3. Integrirani krug 74181

U razvoju *ALU* jedinica ističe se integrirani krug 74181 (Slika 4.2.) tvrtke Texas Instruments. Razvijen je 1970. godine te je postao prvi integrirani krug koji je potpuno integrirao *ALU* logiku. Do tada su se koristile zasebne komponente za izradu mikroračunala. Složen je od ukupno 75 logičkih vrata te omogućuje rad s 4 bitnim ulaznim vrijednostima.



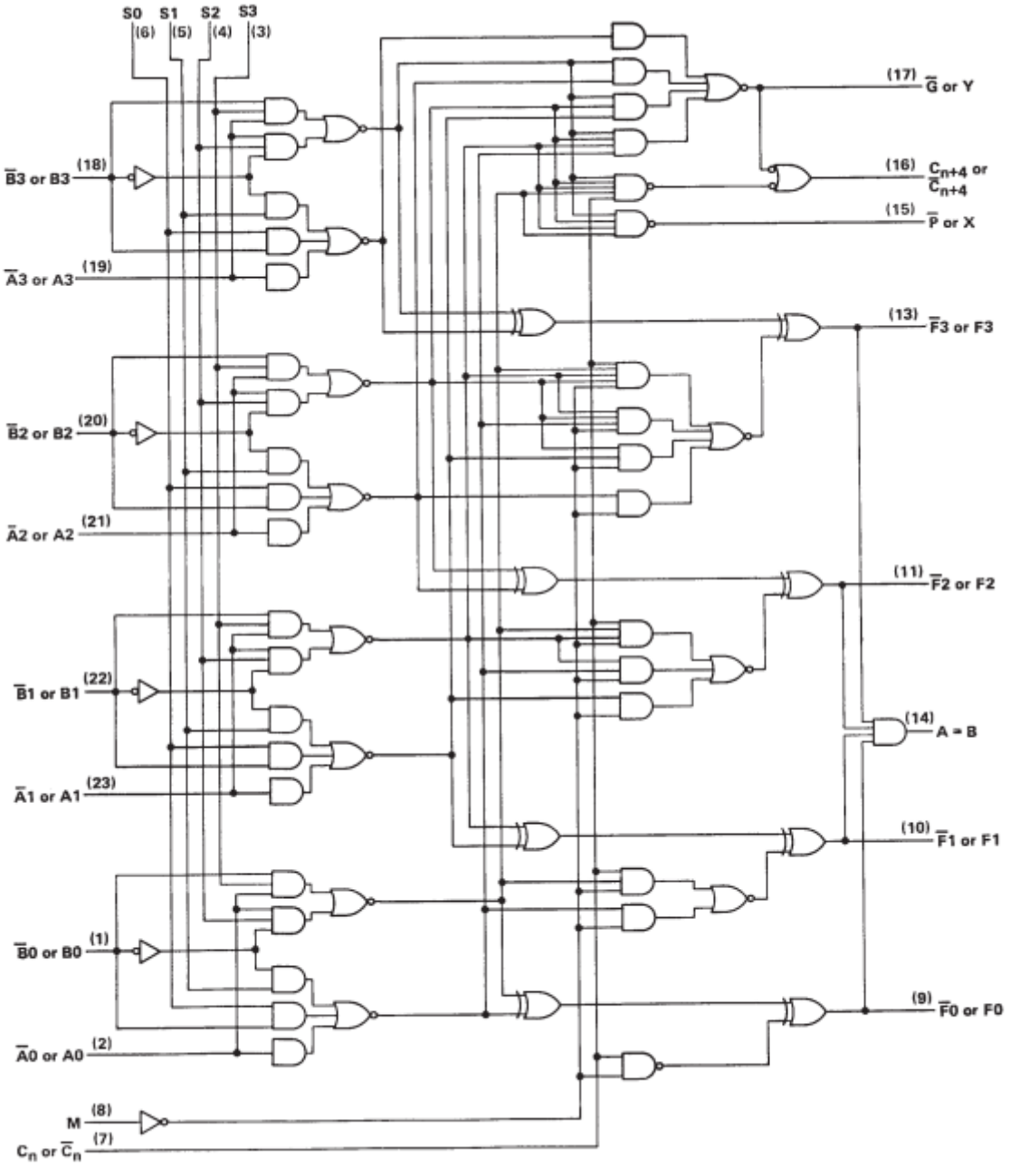
Slika 4.2. 74LS181 u DIP8 pakiranju.

Sa svojih 4 adresnih ulaza obavlja 16 logičkih i 16 aritmetičkih funkcija. One su: I, NI, ILI, NILI, XNILI, zbrajanje, oduzimanje, dekrementacija, itd. Odabir načina rada vrši se promjenom jednog od kontrolnih ulaza (C_n). U tablici (Tablica 4.1.) prikazane su sve operacije koje ovaj čip može izvršiti nad ulazima.

Tablica 4.1. Sve funkcije čipa 74181.

Adresni ulaz	Logičke funkcije C_n=0	Matematičke funkcije C_n=1
0000	$\overline{A}$	A
0001	$\overline{A+B}$	AB
0010	$\overline{A}$B	$A\overline{B}$
0011	0	0
0100	$\overline{AB}$	A plus (A+$\overline{B}$) plus 1
0101	$\overline{B}$	AB plus (A+$\overline{B}$) plus 1
0110	A$\oplus$B	A minus B
0111	$A\overline{B}$	(A+$\overline{B}$) plus 1
1000	$\overline{A}$+B	A plus (A+B) plus 1
1001	$\overline{\text{A}\oplus\text{B}}$	A plus B plus 1
1010	B	A$\overline{B}$ plus (A+B) plus 1
1011	AB	(A+B) plus 1
1100	1	A plus A plus 1
1101	A+$\overline{B}$	AB plus A plus 1
1110	A+B	$A\overline{B}$ plus A plus 1
1111	A	A plus 1

Shema ovog čipa priložena je u nastavku (Slika 4.3.). Ovaj sklop je maksimalno minimiziran što osigurava korištenje minimalnog broja logičkih vrata i jednostavnost proizvodnje.



Slika 4.3. Shema 74181 čipa

4.4. Zastavice

Zastavice ili *flags* koriste se u svim arhitekturama procesora te su neke od izlaza *ALU* jedinice. Zastavice su jedno ili višebitni izlazi iz *ALU* jedinice koji govore kada je izlaz došao u određeno stanje. Uobičajeno se koristi zastavica kada dođe do *overflow*-a odnosno kada operacijom zbrajanja pređemo u n+1 bit te rezultat više nije točan. Još neke od zastavica mogu biti: izlaz koji se sastoji od nula, vrijednost MSB bita koji označava predznak, itd. U tablici 4.2. je prikazan registar s zastavicama moderne x86 arhitekture procesora.

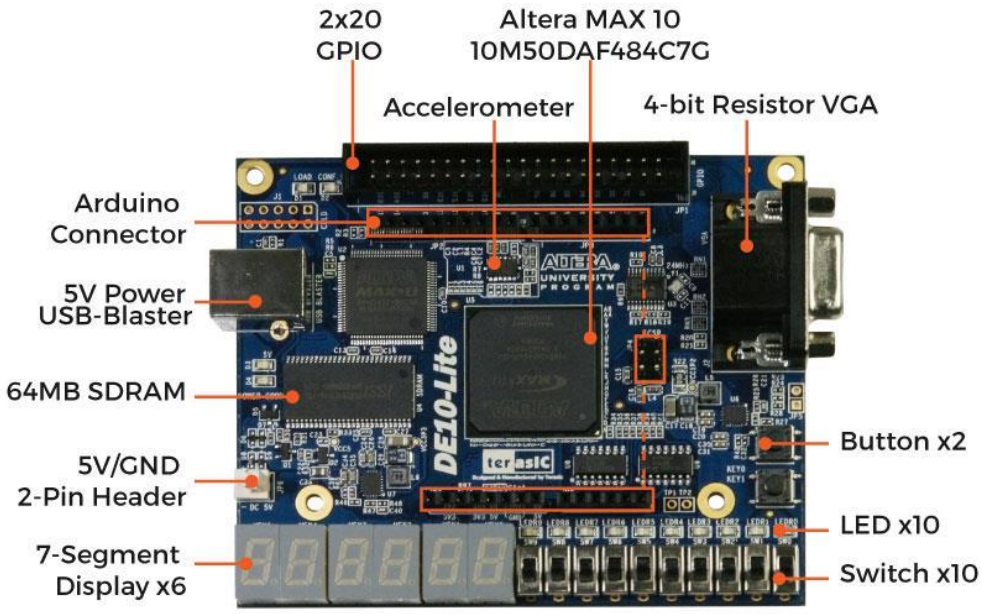
Tablica 4.2. Registar zastavica u procesoru x86 arhitekture

Bit zastavice	Opis zastavice
0	Javljanje *carry* bita na kraju operacije
1	/
2	Provjera *parity* bita (paran broj jedinica)
3	/
4	Javljanje *carry* bita nakon 3. koraka operacije
5	/
6	Nule
7	Predznak izlaza
8	Naredba za zaustavljanje operacije nakon svakog koraka
9	Naredba za zaustavljanje operacije od strane vanjskih komponenti
10	Naredba za promjenu redoslijeda obavljanja operacija
11	Javljanje *overflow* bita
12	/
13	/
14	/
15	/

5. Eksperimentalna pločica Altera DE10-Lite i programsko sučelje Quartus

5.1. Opis sučelja i pločice

Za izvedbu sklopa aritmetičko-logičke jedinice korišten je Altera DE10-Lite *FPGA* baziran na MAX10 arhitekturi. MAX 10 uređaj sadrži 50000 logičkih blokova uz 64MB SDRAM memorije za korištenje pri razvijanju. Ova razvojna pločica opremljena je mnoštvom I/O pinova od kojih su korišteni *GPIO* (*eng. General purpose I/O*) pinovi za spajanje DIP prekidača i signalnih LED dioda, ugrađeni prekidači i LED diode te 7 segmentni displeji za unos i prikaz rezultata operacija. *FPGA* se povezuje računalom pomoću USB priključka kojim se programira i putem kojeg dobiva napon od 5V za napajanje cijele periferije.



Slika 5.1. Altera DE10-Lite

5.2. Quartus prime

Razvojno sučelje u kojemu je programiran *FPGA* sklop jest Quartus Prime trenutne verzije 21.1. koji je ujedno i sučelje za pisanje *VHDL*/*Verilog* koda, kompajler samog programa te programator same pločice.

Program se sastoji od nekoliko glavnih dijelova potrebnih za izvršenje ovog zadatka. Prvi i najvažniji je *Text editor* odnosno prozor za pisanje samog koda. Uz standardne opcije obrade teksta pruža opcije uobičajenih razvojnih sučelja, poput prepoznavanja gniježđenja, prepoznavanja funkcija koje jezik koristi, itd. Sljedeći je *Pin planner* koji se koristi nakon uspješnog kompajliranja programa kako bi pridodali varijablama fizičke ulaze i izlaze, zadali njihov radni napon te provjerili raspored na pločici. Ovdje još nalazimo implementaciju analize i sinteze koja kompajlira te provjerava je li kod važeći po pravilima pisanja *VHDL*-a. Ova funkcija može nas obavijestiti o mogućim neiskorištenim varijablama, varijablama koje mijenjamo s više različitih signala te petljama koje nije moguće izvršiti.

RTL (*eng. Resgister transfer logic*) viewer jedan je od dijelova programa koji pokreće analizu sklopa te nam govori o povezanostima između komponenti. *RTL viewer* sastavlja shematski prikaz sklopa u kojemu se pojavljuju cjeline sagrađene od bazičnih logičkih vrata te komponenti zadanih sekvencijalnim kodom.

Za simulaciju ovog sklopa korišten je *University waveform program* koji pomoću *Questa* simulacijskog sustava omogućava promjenu ulaznih te prikaz izlaznih varijabli u valnom obliku. Alat je poprilično jednostavan za korištenje te omogućava prikaz ulaznih i izlaznih varijabli u decimalnom ili heksadecimalnom obliku.

6. Opisivanje rada ALU jedinice kroz komponente

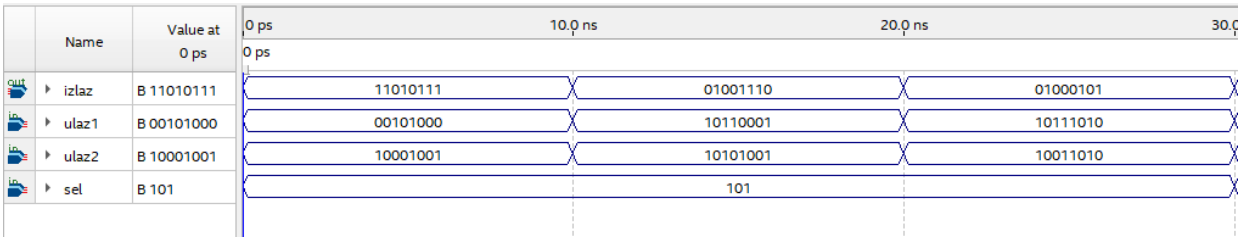
U sljedećim potpoglavljima opisati ćemo svaku komponentu ovog sklopa te razraditi njenu uporabu, shemu i realizaciju. Ovaj kod slijedi upute strukturalnog i ponašajnog modeliranja kako bi shvatili građu ovog *ALU* jedinice od najmanje do najopširnije komponente.

6.1. NE sklop

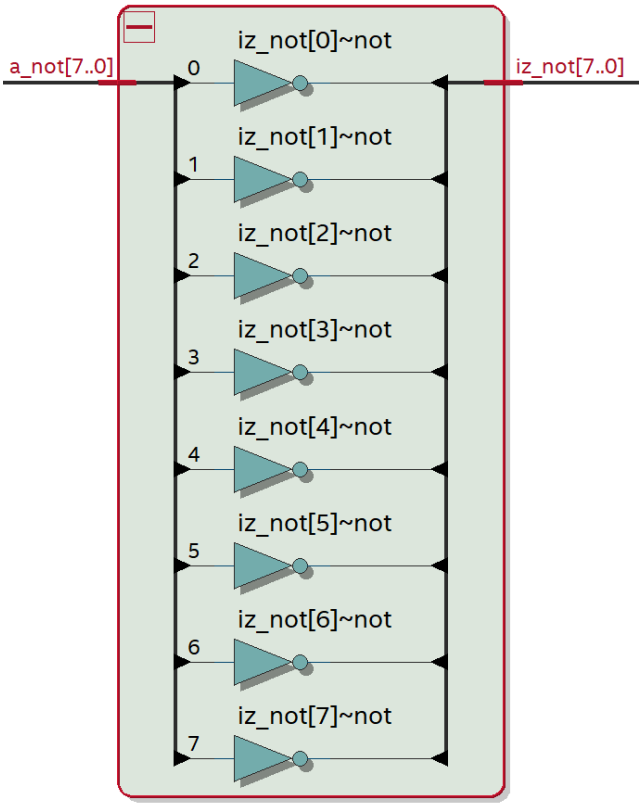
U ovom kodu deklarirana su dva porta, točnije jedan 8 bitni ulaza i 8 bitni izlaz. Sama arhitektura je vrlo jednostavna gdje vršimo *Bitwise* (*svaki bit zasebno*) negaciju nad ulazom te rezultat šaljemo na izlazni port. Ova komponenta najjednostavnija je u cijelom sklopu. Operacija se vrši nad samo jednom varijablom; *ulaz1*.

```
--ne sklop
library ieee;
use ieee.std_logic_1164.all;

entity ne_sklop is
    port(
        a_not : in std_logic_vector(7 downto 0);
        iz_not : out std_logic_vector(7 downto 0)
    );
end entity ne_sklop;


architecture ne of ne_sklop is
        begin
            iz_not <= not a_not;

end ne;
```

Na slici 6.1. prikazana je simulacija sklopa, iz koje je vidljivo da sklop radi ispravno, dok je na slici 6.2. prikazana shema sklopa.



Slika 6.1. Simulacija NE sklopa..

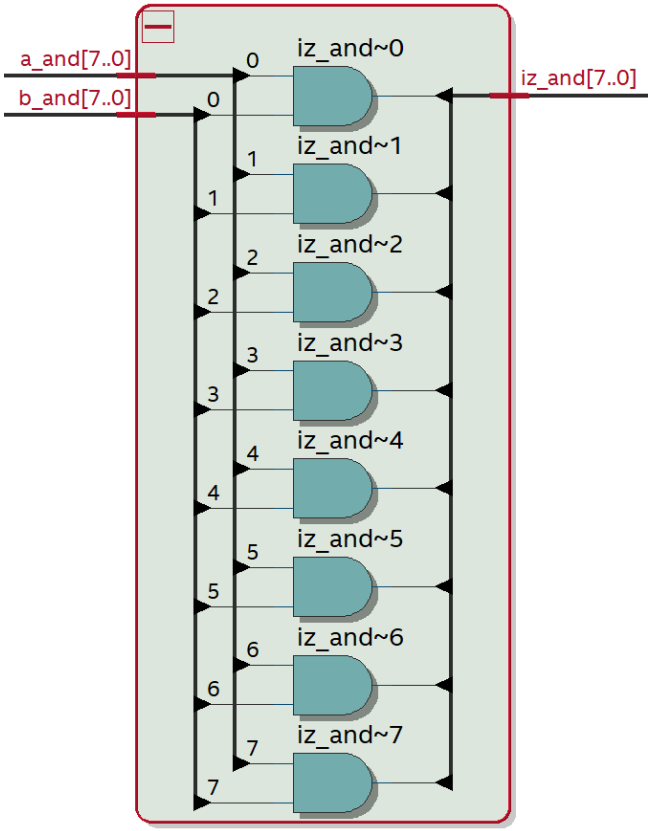


Slika 6.2. Shema NE sklopa.

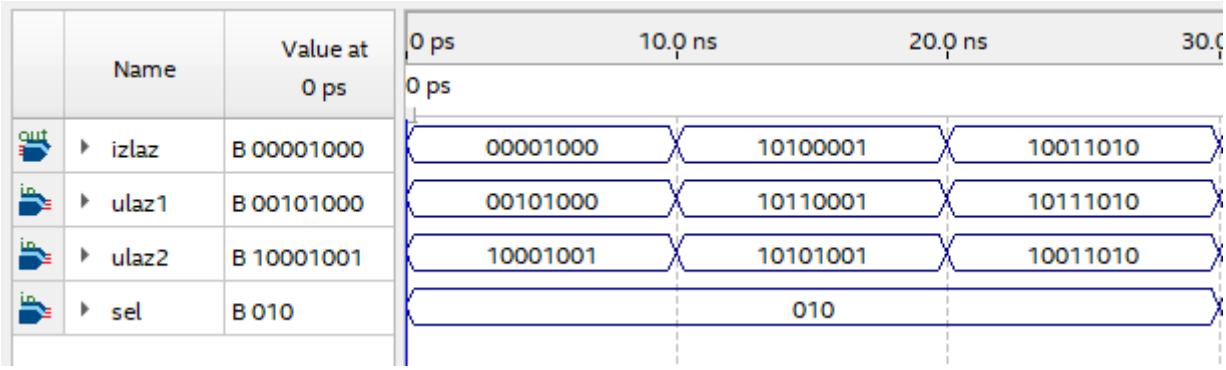
6.2. I sklop

Ova komponenta sklopa također je jedna od najjednostavnijih zajedno s ostalim bazičnim logičkim operacijama. Za razliku od NE sklopa ovdje vidimo dva 8 bitna ulaza te ponovo vršimo *bitwise* operaciju nad ulazima. Na slici 6.3. prikazana je shema sklopa, a na slici 6.4. simulacija koja potvrđuje rad sklopa.

```
--i sklop
library ieee;
use ieee.std_logic_1164.all;

entity i_sklop is
    port(
        a_and : in std_logic_vector(7 downto 0);
        b_and : in std_logic_vector(7 downto 0);
        iz_and : out std_logic_vector(7 downto 0)
    );
end entity i_sklop;

architecture iii of i_sklop is
        begin
            iz_and <= a_and and b_and;

end iii;
```



Slika 6.3. Shema I sklopa.

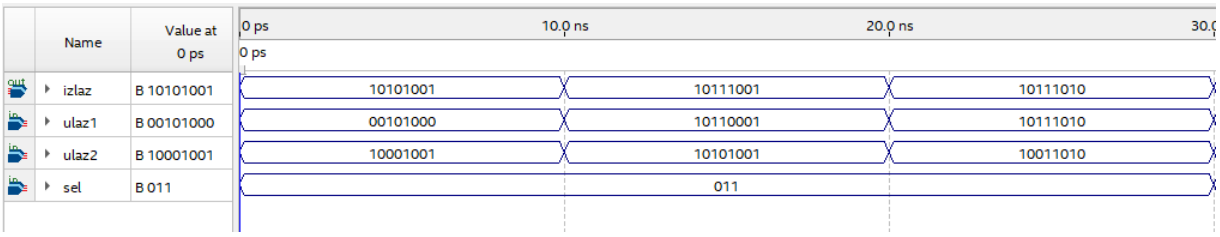


Slika 6.4. Simulacija I sklopa.

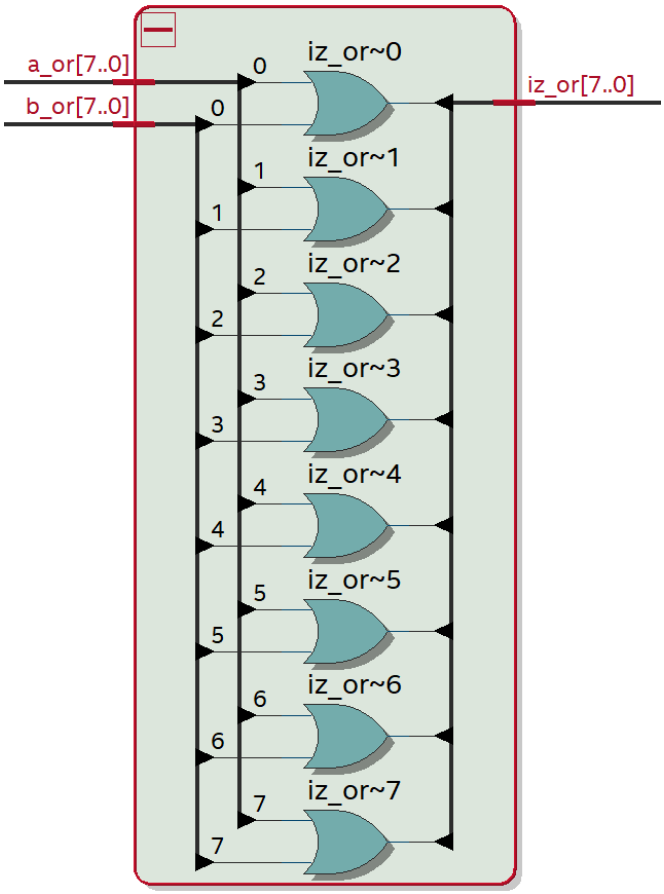
6.3. ILI sklop

Za ILI sklop, uz njegov kod, na slici 6.5. nalazi se simulacija sklopa dok je na slici 6.6. prikazana shema sklopa.

```
--ili skolp
library ieee;
use ieee.std_logic_1164.all;

entity ili_sklop is
    port(
        a_or : in std_logic_vector(7 downto 0);
        b_or : in std_logic_vector(7 downto 0);
        iz_or : out std_logic_vector(7 downto 0)
    );
end entity ili_sklop;


architecture ili of ili_sklop is
        begin
            iz_or <= a_or or b_or;

end ili;
```



Slika 6.5. Simulacija ILI sklopa.

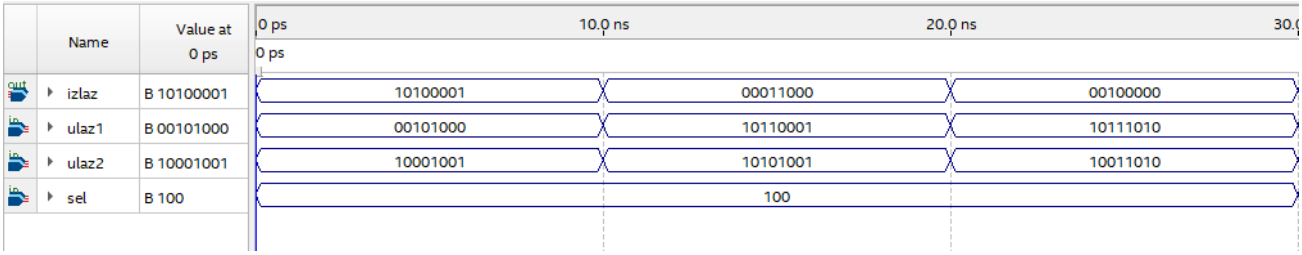


Slika 6.6. Shema ILI sklopa.

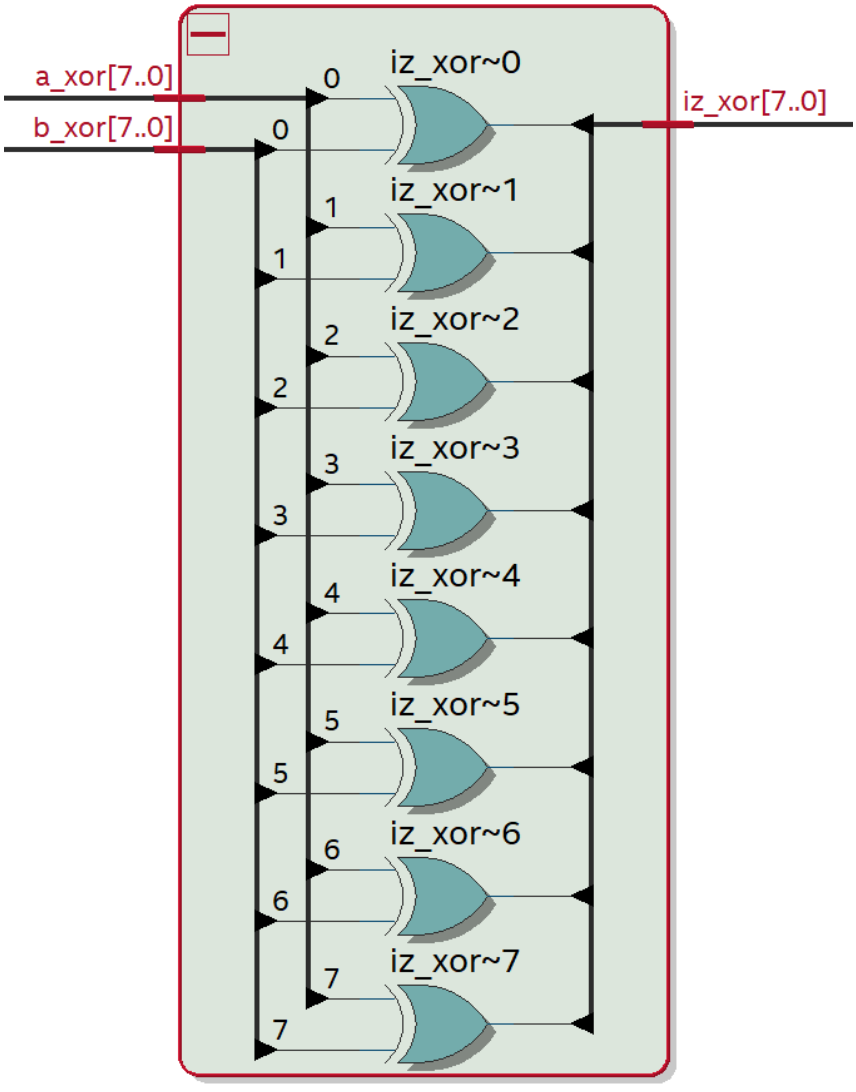
6.4. XILI sklop

Za XILI sklop na slici 6.7. nalazi se simulacija sklopa koja prikazuje ispravan rad sklopa dok je na slici 6.8. prikazana shema sklopa.

```
--xili skolp
library ieee;
use ieee.std_logic_1164.all;

entity xili_sklop is
    port(
        a_xor : in std_logic_vector(7 downto 0);
        b_xor : in std_logic_vector(7 downto 0);
        iz_xor : out std_logic_vector(7 downto 0)
    );
end entity xili_sklop;


architecture xili of xili_sklop is
        begin
            iz_xor <= a_xor xor b_xor;

end xili;
```



Slika 6.7. Simulacija XILI sklopa.



Slika 6.8. Shema XILI sklopa.

6.5. Poluzbrajalo

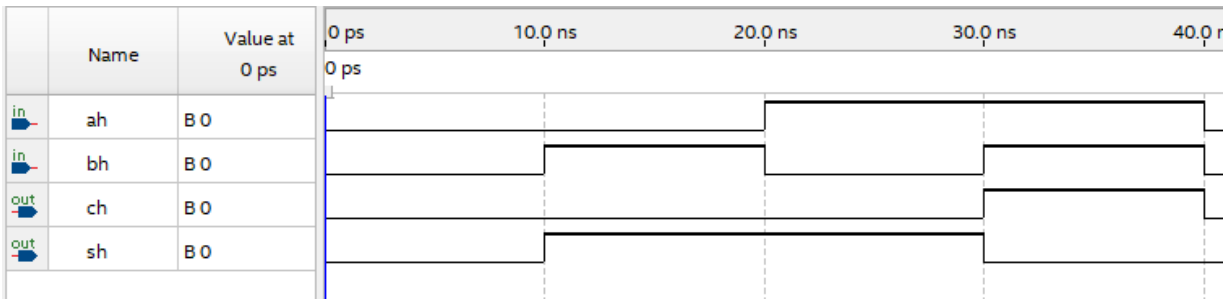
Poluzbrajalo je osnovni sklop koji je u nastavku korišten za modeliranje zbrajanja i oduzimanja. Poluzbrajalo se sastoji od samo dvaju logičkih vrata XILI te I. Ova komponenta zaslužna je za zbrajanje dva logička bita na njenim ulazima te izračunavanju rezultata zbrajanja na izlazu uz dodatni signal prijenosa (*carry*) koji koristimo za slaganje kompleksnijih zbrajala. *Carry* bit možemo shvatiti kao prijenos u znamenku veće težine.

U tablici 6.1. prikazano je ponašanje poluzbrajala, na slici 6.9. simulacija gdje je potvrđena ispravnost sklopa, dok je na slici 6.10. prikazana njegova shema.

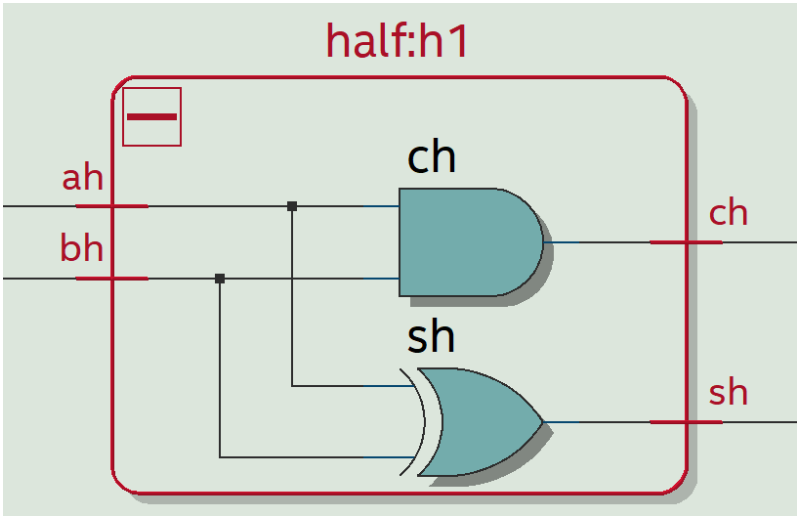
```
library ieee;
use ieee.std_logic_1164.all;

--half adder kao komponenta

entity half is
    port(
        ah : in std_logic;
        bh : in std_logic;
        sh : out std_logic;
        ch : out std_logic
        );
end entity half;

architecture half_h of half is
    begin
        sh <= ah xor bh;
        ch <= ah and bh;

end half_h;
```

Tablica 6.1. Tablica istinitosti poluzbrajala.

Ulaz 1	Ulaz 2	*Carry*	Izlaz
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Slika 6.9. Simulacija poluzbrajala.



Slika 6.10. Shema poluzbrajala.

6.6. Potpuno zbrajalo

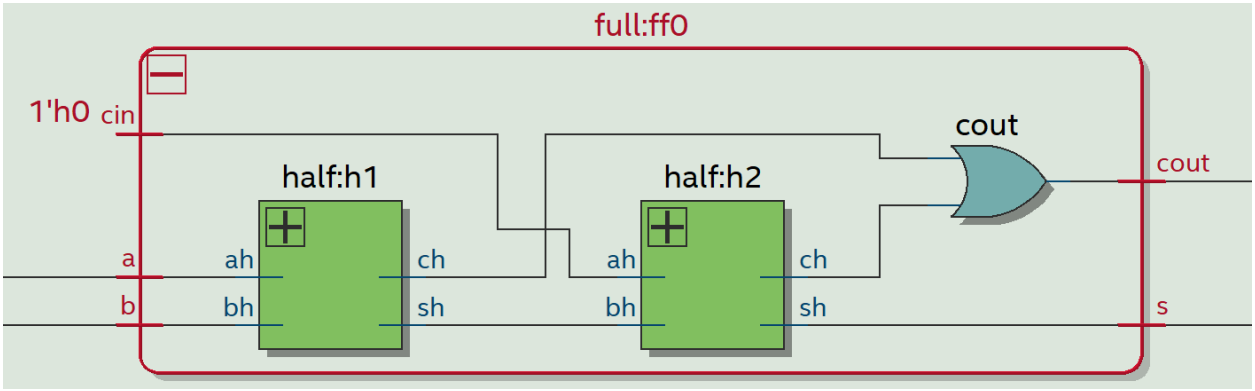
Potpuno zbrajalo komponenta je koja zbraja tri binarna jednobitna broja. Potpuno zbrajalo sačinjeno je od dva poluzbrajala te je izlaz rezultat operacije i *carry* bit. Mogućnost zbrajanja tri binarna broja za operacije unutar *ALU* jedinice bitno je zbog ranije navedenog *carry* bita koji se uvijek može pojaviti na ulazu. Implementacija *carry* bita pokazati će se važnom u složenijim komponentama.

U kodu je sada prvi puta iskorištena komponenta, u ovom slučaju to su dva poluzbrajala različitih imena, koja su spojena kroz više unutarnjih signala na ulaze i izlaze arhitekture. Tablica istinitosti ovog sklopa prikazana je u tablici 6.2. dok slika 6.11. prikazuje njegovu shemu, a slika 6.12. njegovu simulaciju.

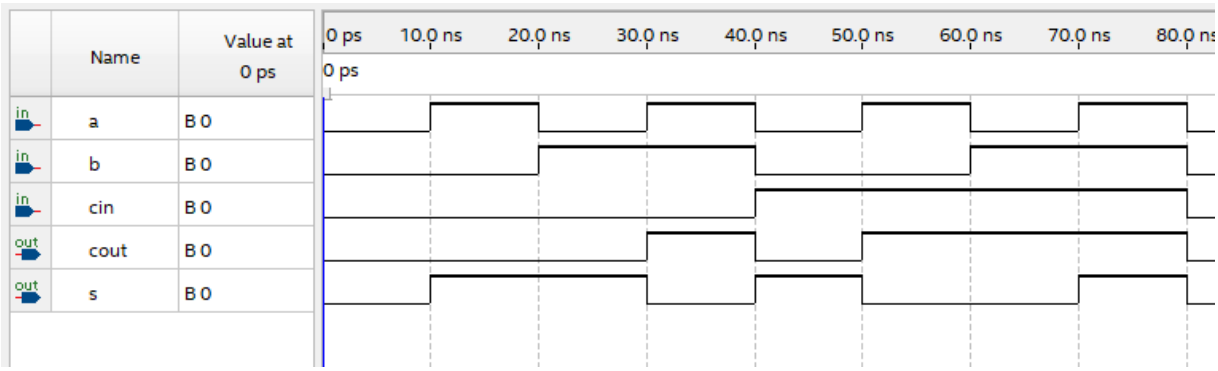
```
--FULL ADDER komponenta

library ieee;
use ieee.std_logic_1164.all;

entity full is
    port (
        a : in std_logic;
        b : in std_logic;
        cin : in std_logic;
        s : out std_logic;
        cout : out std_logic
        );
end entity full;

architecture arch1 of full is

    component half
        port(
        ah : in std_logic;
        bh : in std_logic;
        sh : out std_logic;
        ch : out std_logic
        );
        end component;

        signal z : std_logic;
        signal x : std_logic;
        signal y : std_logic;

    begin

    h1: half
    port map(
        ah => a,
        bh => b,
        sh => x,
        ch => z
        );

    h2: half
    port map(
        ah => cin,
        bh => x,
        sh => s,
        ch => y
        );

    cout <= y or z;

end arch1;
```

Tablica 6.2. Tablica istinitosti potpunog zbrajala.

Ulaz 1	Ulaz 2	Carry ulaz	Carry izlaz	Izlaz
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



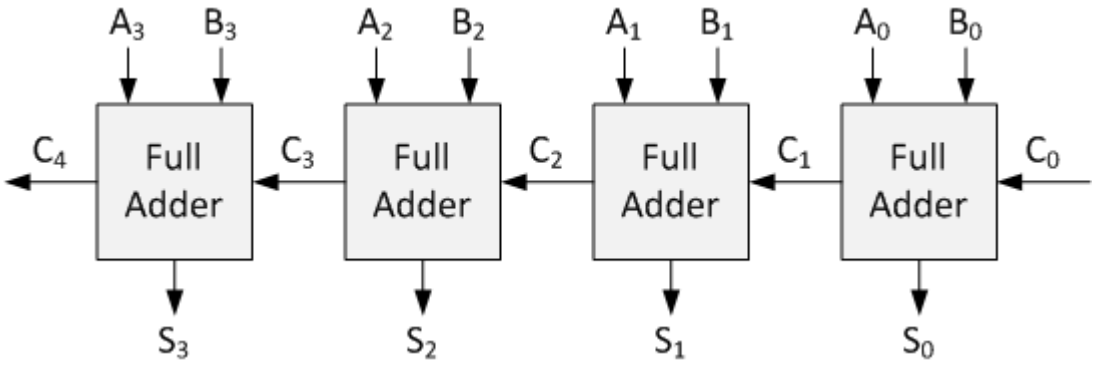
Slika 6.11. Shema potpunog zbrajala.



Slika 6.12. Simulacija potpunog zbrajala.

6.7. Paralelno zbrajalo

Paralelno zbrajalo sastoji se od istog broja potpunih zbrajala koliko bitova imaju ulazi. Pošto je opisano kako potpuno zbrajalo zbraja efektivno dva bita i ulazni prijenos, na njihove ulaze ćemo spojiti bitove iste težine prvog i drugog ulaza. Shema spajanja (Slika 6.13.) pokazuje nam da *carry* izlaz prethodnog potpunog zbrajala spajamo na *carry* ulaz sljedećeg.



Slika 6.13. Shema 4 bitnog paralelnog zbrajala.

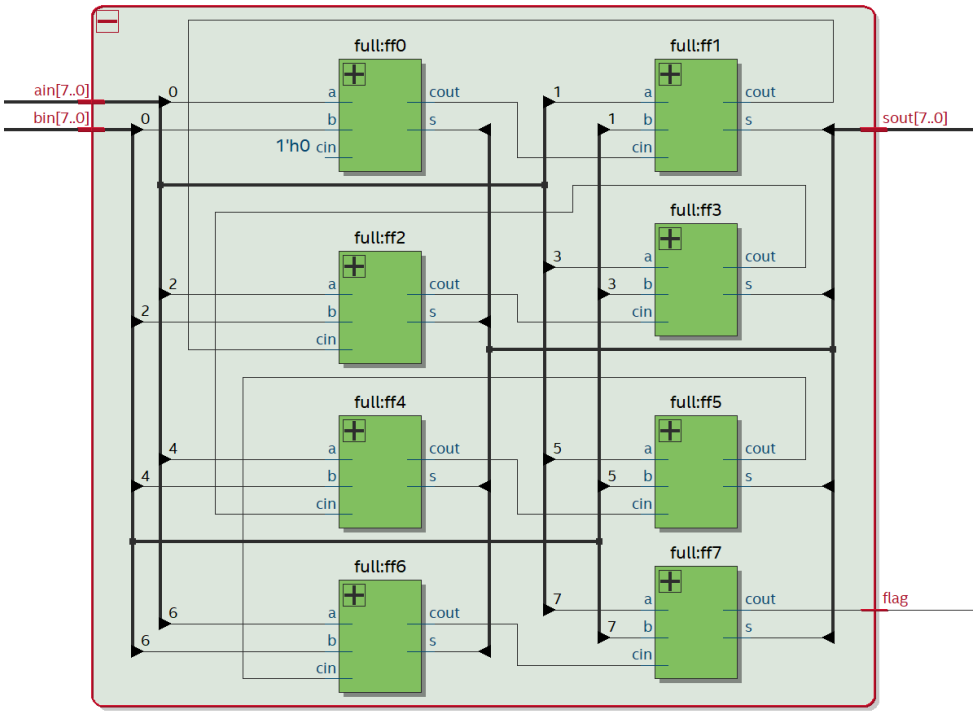
Ako pogledamo shemu (Slika 6.14.) shvaćamo da imamo pored ulaza i izlaza dodatni ulazni i izlazni *carry* bit. Ulazni *carry* postavit ćemo na logičku nulu jer operacija ne zahtjeva ulazni *carry*, a izlazni *carry* biti biti će iskorišten za zastavicu.

U simulaciji (Slika 6.15.) su testirane tri operacije zbrajanja:

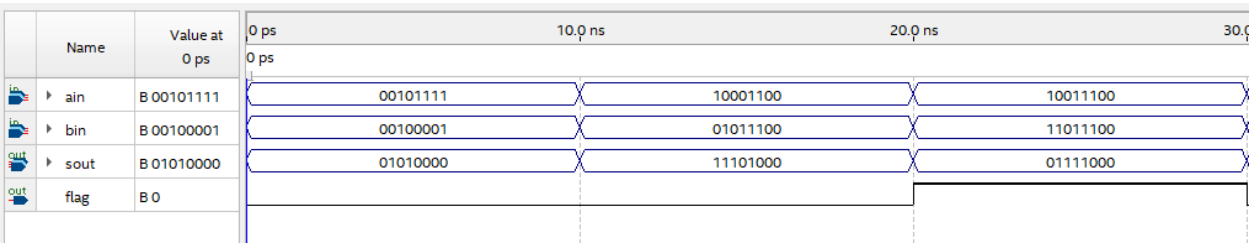
00101111 + 00100001 = 1010000 (47 + 33 = 80)
10001100 + 01011100 = 11101000 (140 + 92 = 232)
10011100 + 11011100 = **1**01101000 (156 + 220 = 376)

U trećem primjeru dolazi do preljeva u *carry* bit tako da operacija neće biti točna.

```
--RIPPLE CARRY ADDER
library ieee;
use ieee.std_logic_1164.all;

entity rcadder is
    port(
        flag : out std_logic;
        ain : in std_logic_vector(7 downto 0);
        bin : in std_logic_vector(7 downto 0);
        sout : out std_logic_vector(7 downto 0)
        );

end entity rcadder;

architecture ripple of rcadder is

    signal ccc : std_logic_vector(6 downto 0);

    component full
     port(
        a : in std_logic;
        b : in std_logic;
        cin : in std_logic;
        s : out std_logic;
        cout : out std_logic
        );
    end component;

    begin

    ff0 : full port map( ain(0), bin (0),  '0', sout(0), ccc(0));
    ff1 : full port map( ain(1), bin (1),  ccc(0), sout(1), ccc(1));
    ff2 : full port map( ain(2), bin (2),  ccc(1), sout(2), ccc(2));
    ff3 : full port map( ain(3), bin (3),  ccc(2), sout(3), ccc(3));
    ff4 : full port map( ain(4), bin (4),  ccc(3), sout(4), ccc(4));
    ff5 : full port map( ain(5), bin (5),  ccc(4), sout(5), ccc(5));
    ff6 : full port map( ain(6), bin (6),  ccc(5), sout(6), ccc(6));
    ff7 : full port map( ain(7), bin (7),  ccc(6), sout(7), flag);


end ripple;
```



Slika 6.14. Shema 8 bitnog paralelnog zbrajala.



Slika 6.15. Simulacija 8 bitnog paralelnog zbrajala.

6.8. Drugi komplement

Drugi komplement nekog binarnog broja je način njegova zapisivanja kako bi omogućili zapis N-bitnih brojeva, tj. 2^{N-1}-1 pozitivnih i 2^{N-1} negativnih vrijednosti uz vrijednost 0. U ovom zapisu MSB služi kao indikator predznaka broja odnosno 0 označava pozitivne brojeve dok 1 označava negativne.

Do drugog komplementa broja dolazimo jednostavnom matematičkom operacijom gdje prvo zamijenimo sve nule s jedinicama i obratno (invertiranje). Dobiveni rezultat naziva se prvi komplement. Nakon toga zbrojimo 1 s dobivenim rezultatom. U tablici 6.3. prikazane su sve 4-bitne kombinacije i njihova vrijednost u dekadskom sustavu.

Tablica 6.3. 4 bitni drugi komplement broja.

Drugi komplement	Dekadski zapis
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Ako sve operacije temeljimo na drugom komplementu unutar *ALU* jedinice oduzimanje brojeva je trivijalno, gdje je potrebno jedan broj pretvoriti u njegov drugi komplement te ga zbrojiti s drugim brojem.

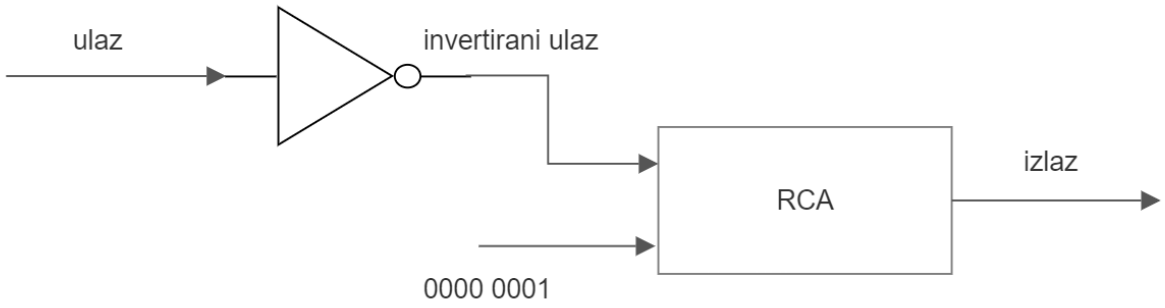
Overflow bit u sustavima s predznakom različit je od onog gdje se operacije vrše nad binarnim brojevima bez predznaka. Problem se sastoji u tome što kod zbrajanja dva broja različitih predznaka *carry out* bit možemo zanemariti te ćemo i dalje dobiti točan rezultat na izlazu. To pojašnjavamo tvrdnjom da zbrajanjem brojeva različitih predznaka uvijek ili uvećavamo najmanji mogući negativni broj (-128) ili smanjujemo najveći pozitivni broj (127) za onu vrijednost koja nam se nalazi u drugom ulazu u sklop. Prepoznavanje *overflowa* tako se svodi na uvjet; ako zbrajamo dva broja istog predznaka, u bitu predznaka ne smije se naći drugačiji predznak od onog kojeg posjeduju ulazni brojevi. Zbog minimiziranja sklopa te lakše provedbe isto tako možemo reći da će se *overflow* dogoditi ako *carry* ulaz u posljednje potpuno zbrajalo nije isti kao izlaz iz njega. Iako gore navedeni model paralelnog zbrajala radi dobro na binarnim brojevima bez predznaka, za korištenje drugog komplementa dodajemo još jedan izlazni signal koji će pokazivati da li je došlo do *overflowa* kojeg dobivamo kao XILI operaciju između ulaznog i izlaznog prijenosa iz posljednjeg potpunog zbrajala. Ovakav kod nastavit ćemo koristiti kroz cijeli sklop *ALU* jedinice.

```
signal ccc : std_logic_vector(7 downto 0);

flag <=  ccc(6) xor ccc(7);
```

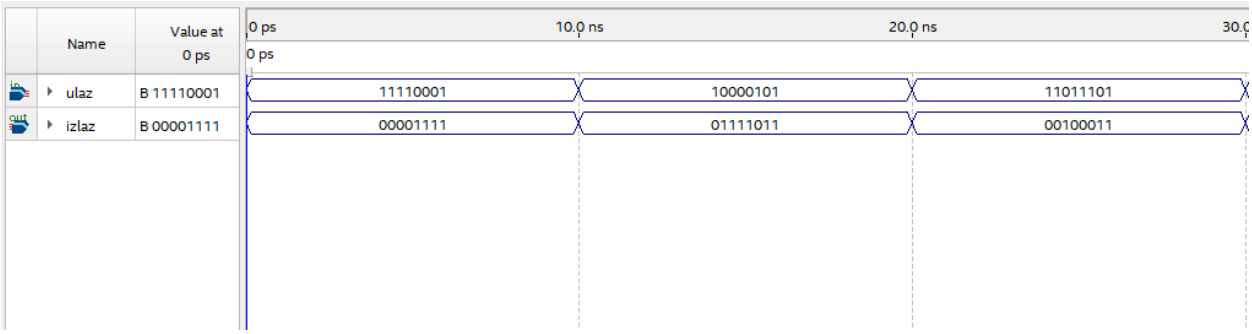
Dakle, sklop za izračun drugog komplementa vrši negiranje početnog broja te kasnije zbrajanje pomoću prethodno opisanog sklopa potpunog zbrajala kojem je jedan ulaz konstanta oblika 00000001. Kako je invertiranje odrađeno automatski prilikom kompajliranja to nećemo vidjeti u shemi koju nam pruža *Quartus* program.

Na slici 6.16. prikazana je pojednostavljena blok shema, dok je na slici 6.17. prikazana simulacija sklopa izvršena nad tri binarna broja.

```
-- drugi komplement
library ieee;
use ieee.std_logic_1164.all;

entity comp is
    port(

        ulaz : in std_logic_vector(7 downto 0);
        izlaz : out std_logic_vector(7 downto 0)

    );
end comp;

architecture komplement of comp is

        signal temp : std_logic_vector(7 downto 0);


        component rcadder
            port(
                flag : out std_logic;
                ain : in std_logic_vector(7 downto 0);
                bin : in std_logic_vector(7 downto 0);
                sout : out std_logic_vector(7 downto 0)
            );
        end component;


    begin

        temp <= ulaz nand "11111111";

        adder: rcadder
            port map(
                ain => temp,
                bin => "00000001",
                sout => izlaz
            );


end komplement;
```



Slika 6.16. Blok shema sklopa drugog komplementa



Slika 6.17. Simulacija obrade drugog komplementa

6.9. Oduzimanje

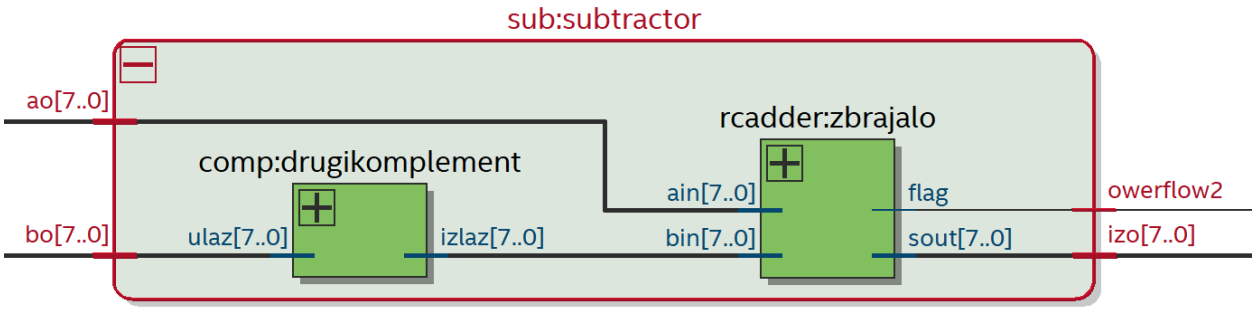
Kako smo već do sada razradili sklop koji zbraja 8 bitne brojeve te sklop koji računa drugi komplement za zapis negativnih brojeva, izrada sklopa za oduzimanje sastoji se od povezivanja ta dva bloka. U nastavku priložen je kod koji obavlja funkciju oduzimanja zajedno s jednostavnom blok shemom (Slika 6.18.) i simulacijom nad tri operacije (Slika 6.19.):

0001 1001 – 0000 0111 = 0001 0010 (25 – 7 = 18)

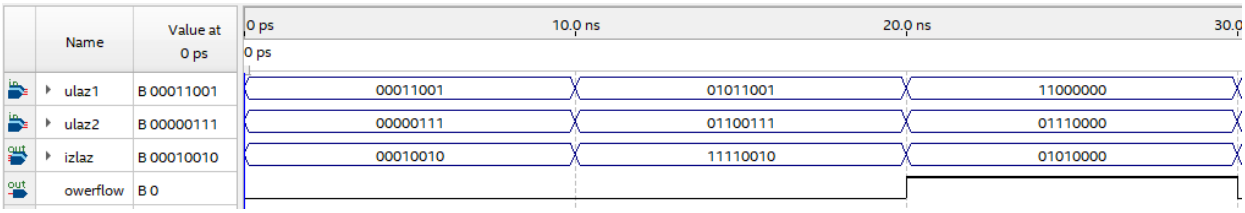
0101 1001 – 0110 0111 = 1111 0010 (89 – 103 = -14)

1100 0000 – 01110000 = **1**0101 0000(-64 – 112 = -176)

```
--oduzimanje
entity sub is
    port(
        owerflow2 : out std_logic;
        ao : in std_logic_vector(7 downto 0);
        bo : in std_logic_vector(7 downto 0);
        izo : out std_logic_vector(7 downto 0)

    );
end entity sub;


architecture oduzimac of sub is

    component rcadder
            port(
                flag : out std_logic;
                ain : in std_logic_vector(7 downto 0);
                bin : in std_logic_vector(7 downto 0);
                sout : out std_logic_vector(7 downto 0)
            );
        end component;


    component comp
            port(

                ulaz : in std_logic_vector(7 downto 0);
                izlaz : out std_logic_vector(7 downto 0)

                );
        end component;

    signal temp : std_logic_vector(7 downto 0);







begin

        zbrajalo: rcadder
            port map(
                flag => owerflow2,
                ain => ao,
                bin => temp,
                sout => izo
            );

        drugikomplement: comp
            port map(
                ulaz => bo,
                izlaz => temp
                );

end oduzimac;
```



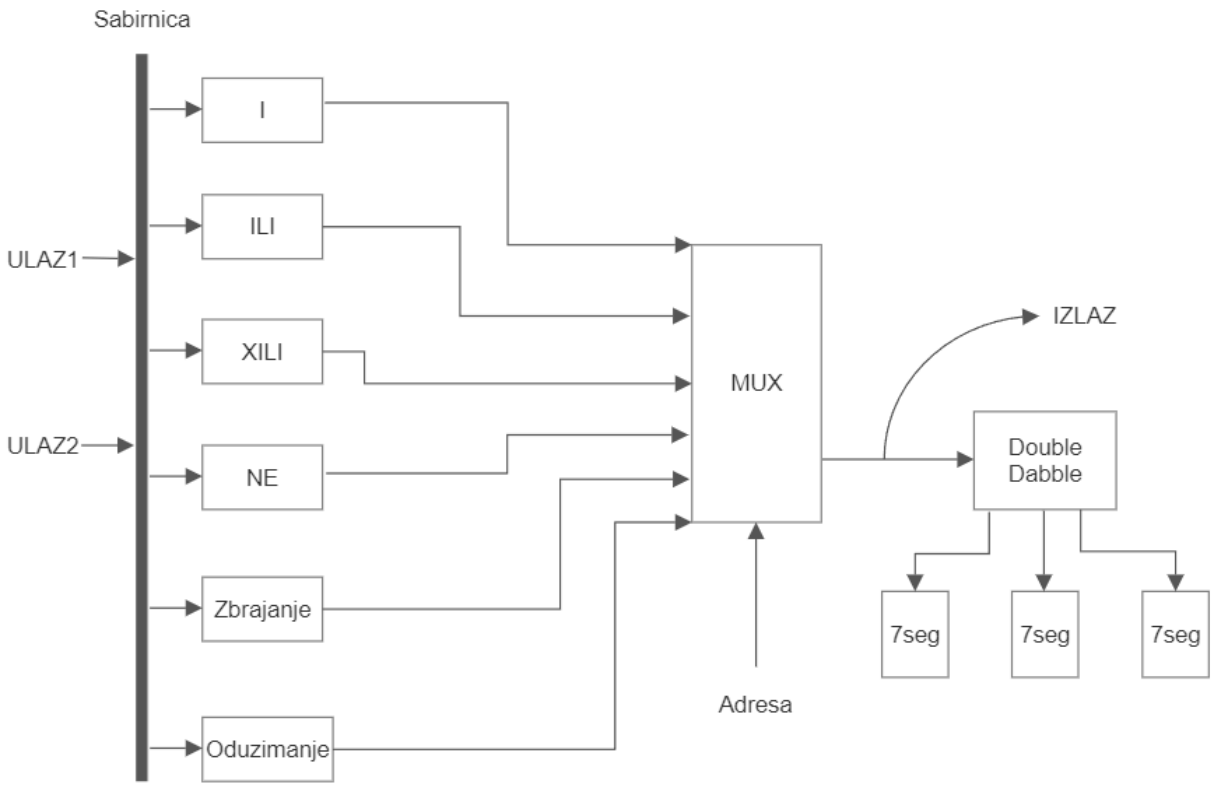
Slika 6.18. Shema sklopa oduzimanja.



Slika 6.19. Simulacija sklopa oduzimanja.

6.10. Spajanje sklopa u cjelinu i multipleksor

Za spajanje sklopa u cjelinu prvo je potrebno navesti sve prijašnje komponente i sklopove (I, ILI, XILI, NE, potpuno zbrajalo, sklop za oduzimanje) te dodati adresni 3-bitni ulaz kako bi mogli odabrati željenu operaciju koju ćemo vršiti nad varijablama. To ćemo izvršiti moduliranjem multipleksora s 3 adresna i 6 podatkovnih ulaza. Ovaj sklop vršimo pomoću naredbe *WHEN ELSE* kojoj pridodajemo za svaki izlazni signal komponente adresni ulaz te tako prenosimo podatak na izlaznu sabirnicu. Na slici 6.20. je prikazana pojednostavljena blok shema cijelog sklopa.



Slika 6.20. Blok shema multipleksora i izlaznog sklopa.

Na jednak način kako smo napravili sabirnicu za sam rezultat operacije isto tako je potrebno napraviti i sabirnicu za dva *overflow* bita koja se javljaju kod zbrajanja i oduzimanja. Ponovo adresni ulaz kontrolira protok te ako zbrajamo brojeve šaljemo na izlaz *overflow* zbrajanja, a ako oduzimamo šaljemo na izlaz *overflow* oduzimanja. U ostalim operacijama ne može se pojaviti *overflow* te taj izlaz postavljamo na 0.

Od jednostavnijih sklopova ovdje još možemo naći procesnu naredbu koja pri svakoj promjeni prijenosne varijable na izlazu iz multipleksora provjera dali su se na izlazu pojavile sve nule te ako jesu, izlaz porta „nule“ postaje '1' što predstavlja našu drugu zastavicu u sklopu.

Nakon što smo dobili izlaz zapisan u drugom komplementu na 8 bitnom signalu, potrebno ga je pretvoriti u oblik broja baze 10 kako bismo ga mogli prikazati na displeju. Korišten je *Double Dabble* algoritam koji pretvara izlazni signal (bez bita predznaka) u 3 skupa od 4 bita pisana u BCD kodu. BCD kod jednostavan je kod koji svaku decimalnu znamenku kodira kao 4 bitni binarni broj. Svaka od 10 znamenki pripada kodu koji odgovara numeričkoj vrijednosti binarnog broja pa npr. 7 postaje 0111 i 9 postaje 1001.

Algoritam kreće postavljanjem binarne vrijednosti desno od onoliko skupova 4 bitnih binarnih brojeva koliko znamenki najviše možemo očekivati (127 zauzima najviše 3 znamenke). Nakon toga vršimo dva koraka u onoliko ponavljanja koliko prvotni broj ima bitova. Svakom iteracijom ulazni broj pomičemo jedno mjesto ulijevo prema BCD znamenkama te provjeravamo dali smo u bilo kojem skupu od 4 bita došli do broja većeg ili jednakog 5 odnosno „0101“. Ukoliko se to dogodilo, tom 4-bitnom broju dodajemo 3, tj. „0011“ čime osiguravamo da nakon sljedećeg pomaka ulijevo vrijednost niti jednog skupa ne prijeđe znamenku 9 koja nije kodna riječ u BCD kodu. Brojevi 3 i 5 dobivaju smisao tek kada shvatimo da pomakom ulijevo poduplavamo brojeve zbog baze 2 isto kao što pomakom ulijevo u bazi 10 možemo reći da brojeve množimo s 10. Ovaj algoritam prikazan je na kratkom 5 bitnom primjeru na slici 6.21.

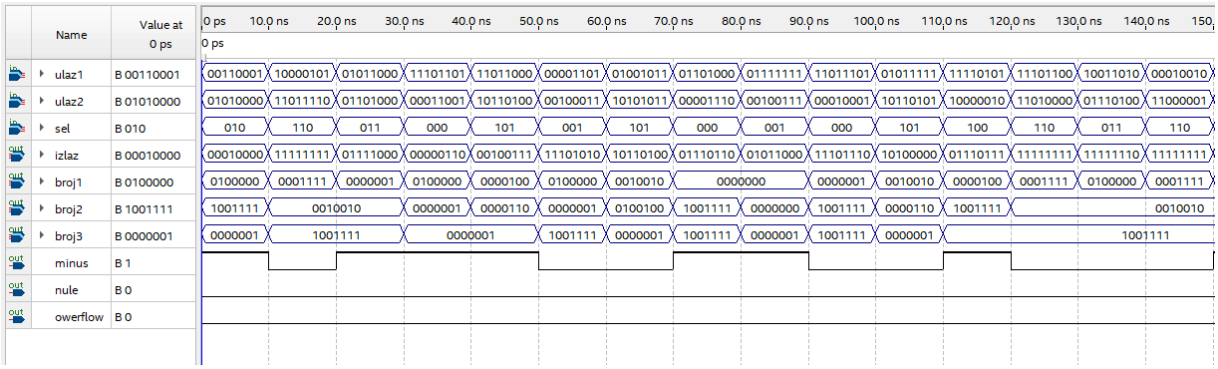
```
 0   0     22

0000 0000 10110 sll
0000 0001 0110x sll
0000 0010 110xx sll
0000 0101 10xxx dodaj 3
0000 1000 10xxx sll
0001 0001 0xxxx sll
0010 0010 xxxxx kraj algoritma

 2    2    xx
```

Slika 6.21. Primjer Double dabble algoritma.

Navedeni algoritam u ovom završnom radu implementiran je tako da umjesto jednog 12 bitnog registra u koji pomičemo naš binarni broj korištena su tri 4-bitna signala u kojima su spremljene tri BCD znamenke. Za pomak je korišten logički operand *sll* (*eng. shift left logical*) koji ispunjava varijablu nulama nakon pomicanja. Nove vrijednosti binarnog broja koje ulaze u petlju dohvaćamo korištenjem iteracijske varijable *i* za odabir određene znamenke (*binarno(i)*).

Dobivene četiri BCD znamenke sada je samo potrebno pretvoriti u skup 7 bitnih binarnih brojeva koji će aktivirati pojedine segmente 7-segmentnog displeja. Ta veza biti će tablica istinitosti aktivacije određenih segmenata 7-segmentnog displeja koji će prikazivati decimalne brojeve. To je napravljeno ručnim pisanjem svake znamenke posebno povezujući ih s BCD kodom. Jednom ovako napisan skup naredbi možemo iskoristiti za ostale dvije znamenke te se kod ne mijenja. Zato ćemo ovdje koristiti naredbu *case when* u 10 slučajeva za svaku znamenku. Proces u kojem se ove tvrdnje izvode imaju listu osjetljivosti na svaku od znamenki posebno. Minus bit za posljednji 7-segmentni displej je indikator predznaka.

U prilogu 1. dan je potpuni kod napisan u sklopu ovog završnog rada, a na slici 6.22. simulacija sklopa kroz više ulaznih i adresnih vrijednosti.



Slika 6.22. Nasumična simulacija većeg broja kombinacija našeg finalnog sklopa.

7. Izrada fizičkog sklopa

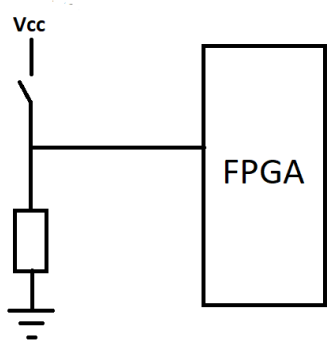
Kod izrade fizičkog sklopa korištena su dva DIP prekidača s 10 ulaza za odabir adrese multipleksora i druge ulazne varijable, dvije LED diode za prikaz izlaznih zastavica, 8 prekidača na razvojnoj pločici za jednu ulaznu varijablu, 8 LED dioda isto s pločice te četiri 7-segmentna displeja za prikaz rezultata operacije. Cijeli rad izrađen je kompaktno s ciljem testiranja i jednostavnosti čitanja rezultata u čemu najviše pomažu displeji uz overflow zastavicu za otkrivanje pogrešaka.

Kako bi na pločicu mogli povezati komponente potrebno je uz pregledavanje dokumentacije u *Quartus* programu sastaviti „plan pinova“ (Slika 7.2.). To je potrebno napraviti za interne i eksterne komponente.

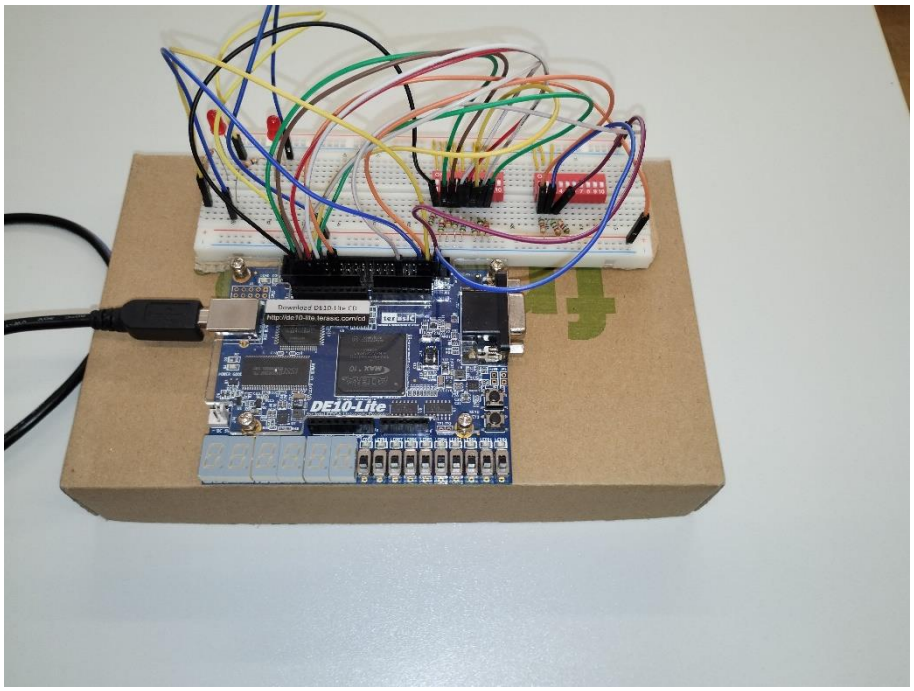
Nakon toga, razvojna pločica je spojena s prekidačima u *pull-down* konfiguraciji (Slika 7.1.) i LED diodama spojenih na prototipnoj pločici. *Pull-down* konfiguracija omoguće jasno definiranu logičku nulu kada su prekidači u neaktivnom stanju. Na slici 7.3. prikazan je završni izgled sklopa na prototipnoj pločici.

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	Strict Preservation
broj2[2]	Output	PIN_E18	6	B6_N0	PIN_E18	2.5 V		12mA (default)	2 (default)		
broj2[3]	Output	PIN_B16	7	B7_N0	PIN_B16	2.5 V		12mA (default)	2 (default)		
broj2[4]	Output	PIN_A17	7	B7_N0	PIN_A17	2.5 V		12mA (default)	2 (default)		
broj2[5]	Output	PIN_A18	7	B7_N0	PIN_A18	2.5 V		12mA (default)	2 (default)		
broj2[6]	Output	PIN_B17	7	B7_N0	PIN_B17	2.5 V		12mA (default)	2 (default)		
broj3[0]	Output	PIN_B20	6	B6_N0	PIN_B20	2.5 V		12mA (default)	2 (default)		
broj3[1]	Output	PIN_A20	7	B7_N0	PIN_A20	2.5 V		12mA (default)	2 (default)		
broj3[2]	Output	PIN_B19	7	B7_N0	PIN_B19	2.5 V		12mA (default)	2 (default)		
broj3[3]	Output	PIN_A21	6	B6_N0	PIN_A21	2.5 V		12mA (default)	2 (default)		
broj3[4]	Output	PIN_B21	6	B6_N0	PIN_B21	2.5 V		12mA (default)	2 (default)		
broj3[5]	Output	PIN_C22	6	B6_N0	PIN_C22	2.5 V		12mA (default)	2 (default)		
broj3[6]	Output	PIN_B22	6	B6_N0	PIN_B22	2.5 V		12mA (default)	2 (default)		
izlaz[7]	Output	PIN_D14	7	B7_N0	PIN_D14	2.5 V		12mA (default)	2 (default)		
izlaz[6]	Output	PIN_E14	7	B7_N0	PIN_E14	2.5 V		12mA (default)	2 (default)		
izlaz[5]	Output	PIN_C13	7	B7_N0	PIN_C13	2.5 V		12mA (default)	2 (default)		
izlaz[4]	Output	PIN_D13	7	B7_N0	PIN_D13	2.5 V		12mA (default)	2 (default)		
izlaz[3]	Output	PIN_B10	7	B7_N0	PIN_B10	2.5 V		12mA (default)	2 (default)		
izlaz[2]	Output	PIN_A10	7	B7_N0	PIN_A10	2.5 V		12mA (default)	2 (default)		
izlaz[1]	Output	PIN_A9	7	B7_N0	PIN_A9	2.5 V		12mA (default)	2 (default)		
izlaz[0]	Output	PIN_A8	7	B7_N0	PIN_A8	2.5 V		12mA (default)	2 (default)		
minus	Output	PIN_E17	6	B6_N0	PIN_E17	2.5 V		12mA (default)	2 (default)		
nule	Output	PIN_AA6	3	B3_N0	PIN_AA6	2.5 V		12mA (default)	2 (default)		
owerflow	Output	PIN_AB3	3	B3_N0	PIN_AB3	2.5 V		12mA (default)	2 (default)		
sel[2]	Input	PIN_Y3	3	B3_N0	PIN_Y3	2.5 V		12mA (default)			
sel[1]	Input	PIN_AB2	3	B3_N0	PIN_AB2	2.5 V		12mA (default)			
sel[0]	Input	PIN_AA2	3	B3_N0	PIN_AA2	2.5 V		12mA (default)			
ulaz1[7]	Input	PIN_A14	7	B7_N0	PIN_A14	2.5 V		12mA (default)			

Slika 7.2. Primjer deklaracije pinova u Pin planneru



Slika 7.1. Pull-down konfiguracija



Slika 7.3. Završen sklop na prototipnoj pločici

8. Zaključak

U ovom radu upoznao sam se s novim programskim jezikom *VHDL*-om i novim razvojnim uređajem *FPGA*-om. Obije komponente ovog rada vrlo su interesantne te vrlo korisne u stručnim primjenama. Možemo zaključiti da *FPGA* uređaje možemo smatrati vrlo moćnim alatima u dizajnu digitalnih sklopova. Kroz izradu sklopa shvatio sam kompleksnost nekadašnjih, a još više današnjih računala i procesora, s time da je u ovom radu modeliran samo njihov malo dio. Dolazim do zaključka kako će uporaba *FPGA* uređaja postati još učestalija i potreba za razvojem u programskim jezicima za opis hardwarea sve češća.

9. Literatura

[1] Farooq U. i dr.: „Tree-based Heterogeneous *FPGA* Architectures“, Springer Science & Business Media, 2012, Berlin

[2] Perry, D.L.: „*VHDL*: Programing by example“, McGraw-Hill, 2002, New York

[3] Kleitz W.: „Digital electronics A Practical Approach with *VHDL*“, Pearson , 2003., Boston

[4] Literatura s interneta: https://surf-*VHDL*.com/*VHDL*-syntax-web-course-surf-*VHDL*/*VHDL*-structural-modeling-style/

[5] Tehnnička dokumentacija: SN74LS181, 1988, Texas Instruments Incorporated

[6] Literatura s interneta: https://www.techopedia.com/definition/7509/half-adder, Margaret Rouse, 13. studenog 2020.

[7] Literatura s interneta: https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html, Thomas Finley, Travanj 2000.

[8] Literatura s interneta: http://www.c-jump.com/CIS77/CPU/Overflow/lecture.html#O01_0120_signed_overflow_cont

[9] Literatura s interneta: https://www.embeddedrelated.com/showthread/comp.arch.embedded/18060-1.php

[10] Literatura s interneta: https://faculty.kfupm.edu.sa/COE/aimane/assembly/pagegen-27.aspx.htm

10. Prilog

1. Cjelokupni *VHDL* kod za *ALU* logiku

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;

entity project1 is

    port(
        owerflow : out std_logic;
        ulaz1 : in std_logic_vector(7 downto 0);
        ulaz2 : in std_logic_vector(7 downto 0);
        sel : in std_logic_vector(2 downto 0);
        minus : out std_logic;
        nule : out std_logic;
        broj1 : out std_logic_vector(0 to 6);
        broj2 : out std_logic_vector(0 to 6);
        broj3 : out std_logic_vector(0 to 6);
        izlaz : out std_logic_vector(7 downto 0)
        );

end entity project1;



architecture mux of project1 is

        signal x : std_logic_vector(7 downto 0);
        signal y : std_logic_vector(7 downto 0);
        signal z : std_logic_vector(7 downto 0);
        signal w : std_logic_vector(7 downto 0);
        signal u : std_logic_vector(7 downto 0);
        signal v : std_logic_vector(7 downto 0);
        signal prijenos : std_logic_vector(7 downto 0);
        signal binarni : std_logic_vector(6 downto 0);
        signal pretvorba1 : std_logic_vector(3 downto 0);
        signal pretvorba2 : std_logic_vector(3 downto 0);
        signal pretvorba3 : std_logic_vector(3 downto 0);
        signal o1 : std_logic;
        signal o2 : std_logic;

        component rcadder
            port(
                flag : out std_logic;
                ain : in std_logic_vector(7 downto 0);
                bin : in std_logic_vector(7 downto 0);
                sout : out std_logic_vector(7 downto 0)
            );
        end component;

        component sub
            port(
                owerflow2: out std_logic;
                ao : in std_logic_vector(7 downto 0);
                bo : in std_logic_vector(7 downto 0);
                izo : out std_logic_vector(7 downto 0)

            );
        end component;
```

```
component i_sklop
            port(
                a_and : in std_logic_vector(7 downto 0);
                b_and : in std_logic_vector(7 downto 0);
                iz_and : out std_logic_vector(7 downto 0)
            );
        end component;

        component ili_sklop
            port(
                a_or : in std_logic_vector(7 downto 0);
                b_or : in std_logic_vector(7 downto 0);
                iz_or : out std_logic_vector(7 downto 0)
            );
        end component;

        component xili_sklop
            port(
                a_xor : in std_logic_vector(7 downto 0);
                b_xor : in std_logic_vector(7 downto 0);
                iz_xor : out std_logic_vector(7 downto 0)
            );
        end component;

        component ne_sklop
            port(
                a_not : in std_logic_vector(7 downto 0);
                iz_not : out std_logic_vector(7 downto 0)
            );
        end component;



        begin


        adder : rcadder
            port map(
                flag => o1,
                ain => ulaz1,
                bin => ulaz2,
                sout => x
            );

        subtractor : sub
            port map(
                owerflow2 => o2,
                ao => ulaz1,
                bo => ulaz2,
                izo => y
            );


        i_comp : i_sklop
            port map(
                a_and => ulaz1,
                b_and => ulaz2,
                iz_and => z
            );
```

```
  ili_comp : ili_sklop
        port map(
            a_or => ulaz1,
            b_or => ulaz2,
            iz_or => w
        );

    xili_comp : xili_sklop
        port map(
            a_xor => ulaz1,
            b_xor => ulaz2,
            iz_xor => u
        );

    ne_comp : ne_sklop
        port map(
            a_not => ulaz1,
            iz_not => v
        );

    owerflow <= o1 when sel="000" else
                    o2 when sel="001" else
                    '0';




    prijenos <= x when sel = "000" else
                y when sel = "001" else
                z when sel = "010" else
                w when sel = "011" else
                u when sel = "100" else
                v when sel = "101" else
                "11111111";

    binarni <= prijenos(6 downto 0);

    --double dabble proces koji daje bcd kod na svom izlazu


process(binarni)

          variable zn0 : unsigned(3 downto 0);
          variable zn1 : unsigned(3 downto 0);
          variable zn2 : unsigned(3 downto 0);


    begin
              zn2 := "0000";
              zn1 := "0000";
              zn0 := "0000";

for i in 6 downto 0 loop

  if (zn2 >= 5) then zn2 := zn2 + 3; end if;
  if (zn1 >= 5) then zn1 := zn1 + 3; end if;
  if (zn0 >= 5) then zn0 := zn0 + 3; end if;
```

```
    zn2 := zn2 sll 1;
      zn2(0) := zn1(3);
    zn1 := zn1 sll 1;
      zn1(0) := zn0(3);
    zn0 := zn0 sll 1;
      zn0(0) := binarni(i);
  end loop;

      pretvorba1 <=  std_logic_vector(zn0);
      pretvorba2 <=  std_logic_vector(zn1);
      pretvorba3 <=  std_logic_vector(zn2);


end process;

      izlaz <= prijenos;
      minus <= not(prijenos(7));


process(pretvorba1, pretvorba2, pretvorba3)
  begin

  case  pretvorba1 is
      when "0000"=> broj1 <="0000001";  -- '0'
      when "0001"=> broj1 <="1001111";  -- '1'
      when "0010"=> broj1 <="0010010";  -- '2'
      when "0011"=> broj1 <="0000110";  -- '3'
      when "0100"=> broj1 <="1001100";  -- '4'
      when "0101"=> broj1 <="0100100";  -- '5'
      when "0110"=> broj1 <="0100000";  -- '6'
      when "0111"=> broj1 <="0001111";  -- '7'
      when "1000"=> broj1 <="0000000";  -- '8'
      when "1001"=> broj1 <="0000100";  -- '9'
      when others=> broj1 <="1111111";
  end case;

  case  pretvorba2 is
      when "0000"=> broj2 <="0000001";  -- '0'
      when "0001"=> broj2 <="1001111";  -- '1'
      when "0010"=> broj2 <="0010010";  -- '2'
      when "0011"=> broj2 <="0000110";  -- '3'
      when "0100"=> broj2 <="1001100";  -- '4'
      when "0101"=> broj2 <="0100100";  -- '5'
      when "0110"=> broj2 <="0100000";  -- '6'
      when "0111"=> broj2 <="0001111";  -- '7'
      when "1000"=> broj2 <="0000000";  -- '8'
      when "1001"=> broj2 <="0000100";  -- '9'
      when others=> broj2 <="1111111";
  end case;


  case  pretvorba3 is
      when "0000"=> broj3 <="0000001";  -- '0'
      when "0001"=> broj3 <="1001111";  -- '1'
      when "0010"=> broj3 <="0010010";  -- '2'
      when "0011"=> broj3 <="0000110";  -- '3'
      when "0100"=> broj3 <="1001100";  -- '4'
      when "0101"=> broj3 <="0100100";  -- '5'
      when "0110"=> broj3 <="0100000";  -- '6'
      when "0111"=> broj3 <="0001111";  -- '7'
      when "1000"=> broj3 <="0000000";  -- '8'
      when "1001"=> broj3 <="0000100";  -- '9'
      when others=> broj3  <="1111111";
```

```
    end case;

    end process;

    process (prijenos)
        begin

            if prijenos = "00000000" then
                nule <= '1';
            else
                nule <= '0';
            end if;

    end process;


end mux;


library ieee;
use ieee.std_logic_1164.all;

--oduzimanje
entity sub is
    port(
        owerflow2 : out std_logic;
        ao : in std_logic_vector(7 downto 0);
        bo : in std_logic_vector(7 downto 0);
        izo : out std_logic_vector(7 downto 0)

    );
end entity sub;


architecture oduzimac of sub is

    component rcadder
            port(
                flag : out std_logic;
                ain : in std_logic_vector(7 downto 0);
                bin : in std_logic_vector(7 downto 0);
                sout : out std_logic_vector(7 downto 0)
            );
        end component;


    component comp
            port(

                ulaz : in std_logic_vector(7 downto 0);
                izlaz : out std_logic_vector(7 downto 0)

                );
        end component;

    signal temp : std_logic_vector(7 downto 0);


    begin

        zbrajalo: rcadder
            port map(
                flag => owerflow2,
```

```
                ain => ao,
                bin => temp,
                sout => izo
            );

        drugikomplement: comp
            port map(
                ulaz => bo,
                izlaz => temp
                );

end oduzimac;


-- drugi komlement
library ieee;
use ieee.std_logic_1164.all;

entity comp is
    port(

        ulaz : in std_logic_vector(7 downto 0);
        izlaz : out std_logic_vector(7 downto 0)

    );
end comp;

architecture komplement of comp is

        signal temp : std_logic_vector(7 downto 0);


        component rcadder
            port(
                flag : out std_logic;
                ain : in std_logic_vector(7 downto 0);
                bin : in std_logic_vector(7 downto 0);
                sout : out std_logic_vector(7 downto 0)
            );
        end component;


    begin

        temp <= ulaz nand "11111111";

        adder: rcadder
            port map(
                ain => temp,
                bin => "00000001",
                sout => izlaz
            );


end komplement;

--RIPPLE CARRY ADDER
library ieee;
use ieee.std_logic_1164.all;


entity rcadder is
    port(
```

```
        flag : out std_logic;
        ain : in std_logic_vector(7 downto 0);
        bin : in std_logic_vector(7 downto 0);
        sout : out std_logic_vector(7 downto 0)
        );

end entity rcadder;

architecture ripple of rcadder is

    signal ccc : std_logic_vector(7 downto 0);


    component full
     port(
        a : in std_logic;
        b : in std_logic;
        cin : in std_logic;
        s : out std_logic;
        cout : out std_logic
        );
    end component;

    begin

    ff0 : full port map( ain(0), bin (0),  '0', sout(0), ccc(0));
    ff1 : full port map( ain(1), bin (1),  ccc(0), sout(1), ccc(1));
    ff2 : full port map( ain(2), bin (2),  ccc(1), sout(2), ccc(2));
    ff3 : full port map( ain(3), bin (3),  ccc(2), sout(3), ccc(3));
    ff4 : full port map( ain(4), bin (4),  ccc(3), sout(4), ccc(4));
    ff5 : full port map( ain(5), bin (5),  ccc(4), sout(5), ccc(5));
    ff6 : full port map( ain(6), bin (6),  ccc(5), sout(6), ccc(6));
    ff7 : full port map( ain(7), bin (7),  ccc(6), sout(7), ccc(7));

    flag <=  ccc(6) xor ccc(7);

end ripple;

--FULL ADDER komponenta

library ieee;
use ieee.std_logic_1164.all;

entity full is
    port (
        a : in std_logic;
        b : in std_logic;
        cin : in std_logic;
        s : out std_logic;
        cout : out std_logic
        );
end entity full;


architecture arch1 of full is


    component half
        port(
        ah : in std_logic;
        bh : in std_logic;
        sh : out std_logic;
        ch : out std_logic
```

```
        );
        end component;

        signal z : std_logic;
        signal x : std_logic;
        signal y : std_logic;

    begin

    h1: half
    port map(
        ah => a,
        bh => b,
        sh => x,
        ch => z
        );

    h2: half
    port map(
        ah => cin,
        bh => x,
        sh => s,
        ch => y
        );

    cout <= y or z;

end arch1;

library ieee;
use ieee.std_logic_1164.all;


--half adder kao komponenta

entity half is
    port(
        ah : in std_logic;
        bh : in std_logic;
        sh : out std_logic;
        ch : out std_logic
        );
end entity half;

architecture half_h of half is
    begin
        sh <= ah xor bh;
        ch <= ah and bh;

end half_h;

--i sklop
library ieee;
use ieee.std_logic_1164.all;

entity i_sklop is
    port(
        a_and : in std_logic_vector(7 downto 0);
        b_and : in std_logic_vector(7 downto 0);
        iz_and : out std_logic_vector(7 downto 0)
    );
end entity i_sklop;
```

```
architecture iii of i_sklop is
        begin
            iz_and <= a_and and b_and;

end iii;


--ili skolp
library ieee;
use ieee.std_logic_1164.all;

entity ili_sklop is
    port(
        a_or : in std_logic_vector(7 downto 0);
        b_or : in std_logic_vector(7 downto 0);
        iz_or : out std_logic_vector(7 downto 0)
    );
end entity ili_sklop;


architecture ili of ili_sklop is
        begin
            iz_or <= a_or or b_or;

end ili;

--xili skolp
library ieee;
use ieee.std_logic_1164.all;

entity xili_sklop is
    port(
        a_xor : in std_logic_vector(7 downto 0);
        b_xor : in std_logic_vector(7 downto 0);
        iz_xor : out std_logic_vector(7 downto 0)
    );
end entity xili_sklop;


architecture xili of xili_sklop is
        begin
            iz_xor <= a_xor xor b_xor;

end xili;


--ne sklop
library ieee;
use ieee.std_logic_1164.all;

entity ne_sklop is
    port(
        a_not : in std_logic_vector(7 downto 0);
        iz_not : out std_logic_vector(7 downto 0)
    );
end entity ne_sklop;

architecture ne of ne_sklop is
        begin
            iz_not <= not a_not;

end ne;
```

11. Sažetak

Ovaj završni rad opisuje programski kod napisan u *VHDL* programskom jeziku koji simulira računalnu aritmetičko-logičku jedinicu, te njenu implementaciju na razvojnoj pločici *FPGA* uređaja. U radu su opisane komponente od kojih je sklop napravljen te su pobliže opisane sve cjeline s kojima se susrećemo. Posebna pažnja posvećena je konstrukciji aritmetičko-logičke jedinice od osnovnih logičkih komponenti te njihovoj međusobnoj povezanosti. U radu je izvršena računalna simulacija te praktična simulacija pomoću elektroničkih komponenti.

Ključne riječi: *VHDL*, *FPGA*, *ALU*, Digitalna logika

Abstract

This bachelor's thesis describes a program code written in the *VHDL* programming language which simulates the computer’s arithmetic-logic unit along with its implementation on the *FPGA* device. In this thesis each component used to build this circuit is described in details. Special attention was brought into design of the arithmetic-logic unit using only basic logic units and their interconnections. This work also demonstrates computer simulation of this circuit as well as physical simulation done with electronic components.

Keywords: *VHDL*, *FPGA*, *ALU*, Digital logic