FPGA IMPLEMENTACIJA 12-BITOVNOG PROCESORA ZASNOVANOG NA HARVARDSKOJ ARHITEKTURI

Štajnbrikner, Matej

Undergraduate thesis / Završni rad

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:911721

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-25**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

FPGA IMPLEMENTACIJA 12-BITOVNOG PROCESORA ZASNOVANOG NA HARVARDSKOJ ARHITEKTURI

Završni rad

Matej Štajnbrikner

Osijek, 2019.

Sadržaj:

1. UVOD ... 1
1.1 Zadatak završnog rada ... 2
2. ARHITEKTURA RAČUNALA ... 3
2.1 Von Neumannova arhitektura ... 3
2.2 Harvardska arhitektura ... 4
3. VHDL I ZYBO RAZVOJNA PLOČICA ... 6
3.1 VHDL ... 6
3.2 ZYBO- razvojna pločica ... 7
4. DIZAJNIRANJE PROCESORA ... 10
4.1 Vrh hijerarhije dizajna – jezgra i periferija ... 11
4.2 Jezgra procesora ... 14
4.2.1 Upravljačka jedinica i dekoder ... 14
4.2.2 Aritmetičko-logička jedinica ... 18
4.2.3 Registri unutar jezgre ... 19
4.2.4 Multiplekser ... 20
4.2.5 Programski brojač i stog adresa programskog brojača ... 22
4.2.6 Statusni registar ... 24
4.3 Periferija procesora ... 25
4.3.1 UART ... 25
4.3.2 Prekidi i prekidne rutine ... 26
4.3.3 Ulazni i izlazni sklopovi opće namjene ... 27
5. VERIFIKACIJA I VALIDACIJA DIZAJNA PROCESORA ... 29
5.1 ModelSim, simulacija sustava ... 29
5.2 Xilinx Vivado, simulacija i FPGA implementacija ... 29
5.2.1 Primjena - implementacija procesora ... 30
6. ZAKLJUČAK ... 36
LITERATURA ... 38
SAŽETAK ... 40
ABSTRACT ... 41
ŽIVOTOPIS ... 42
PRILOZI ... 43

1. UVOD

Središnja procesorska jedinica (engl. *Central Processing Unit* - CPU), procesor ili središnja jedinica za obradu je glavni i najvažniji dio računala bez kojega ono ne bi imalo svoju funkciju. Njegova je osnovna zadaća primiti programske naredbe i prema istima izvršiti osnovne operacije nad podacima kao što su: zbrajanje, oduzimanje, pomjeranje ulijevo, pomjeranje udesno, pohranjivanje u RAM (engl. *Random Access Memory*) itd.

Dvije najpoznatije arhitekture računala su: Von Neumann i Harvardska arhitektura. Glavna razlika između ovih arhitektura je ta da kod Von Neumann arhitekture postoji samo jedna memorija u kojoj su pohranjene i programske naredbe i podaci dok kod Harvardske arhitekture postoje dvije memorije. Jedna je ROM (engl. *Read Only Memory*) u kojoj su spremljene programske naredbe, a druga RAM u koju se spremaju podaci. Harvardska arhitektura u današnje vrijeme koristi se za razne primjene, a neke od najpoznatijih su: DSP (engl. *Digital Signal Processor*) i mikroupravljači (AVR ili PIC koje proizvodi Microchip Technology, Inc.).

U okviru ovog završnog rada predstavljen je dizajn procesora zasnovanog na Harvardskoj arhitekturi u jeziku za opisivanje sklopovlja. Projektirani procesor može izvršiti jednostavne aritmetičke i logičke operacije, obraditi rutine vanjskog, vremenskog i UART prekida. Također, omogućena su uvjetna grananja programa i izvođenje programskih podrutina te komunikacija s vanjskim uređajima pomoću UART protokola. Za izradu procesora korišten je VHDL (engl. *Very High Speed Integrated Circuits Hardware Description Language*) jezik za opisivanje sklopovlja i ZYBO razvojna pločica. Verifikacija rješenja napravljena je pomoću ModelSim okruženja za računalnu simulaciju i Vivado softverskog paketa za sintezu i analizu sklopovlja opisanog u jeziku za opisivanje sklopovlja. Validacija rješenja napravljena je spajanjem ZYBO razvojne pločice s računalom i ispitivanjem rada prethodno napisanog algoritma za računanje faktorijela.

Rad je strukturiran na sljedeći način. U drugom poglavlju dana je teorijska podloga Von Neumannove i Harvardske arhitekture. U trećem poglavlju opisani su VHDL i ZYBO razvojna pločica. Prijedlog dizajna procesora predstavljen je u četvrtom poglavlju. Verifikacija i validacija dizajna procesora predstavljeni su u petom poglavlju. Na kraju je dan zaključak rada.

1.1 Zadatak završnog rada

Zadatak ovog završnog rada je dizajnirati procesor zasnovan na Harvardskoj arhitekturi u jeziku za opisivanje računalne sklopovske podrške (engl. *Hardware Description Language*). U ovome primjeru koristit će se VHDL (engl. *Very High Speed Integrated Circuits Hardware Description Language*). Zatim je potrebno simulirati procesor na računalu i nakon toga programirati FPGA integrirani krug na ZYBO razvojnoj pločici te algoritmom u strojnom kodu fizički ispitati ispravnost praktičnog dijela rada.

2. ARHITEKTURA RAČUNALA

Arhitektura računala predstavlja opis funkcionalnosti, organizacije i implementacije računalnih sustava. Opisuje način na koji se podaci unose u procesor, način na koji se obrađuju, način na koji se spremaju podaci, instrukcije koje procesor može izvršiti, koliko je potrebno da se određene instrukcije izvrše itd. Dvije najčešće korištene arhitekture su Von Neumannova i Harvardska koje će biti objašnjene u daljnjem tekstu [1].

2.1 Von Neumannova arhitektura

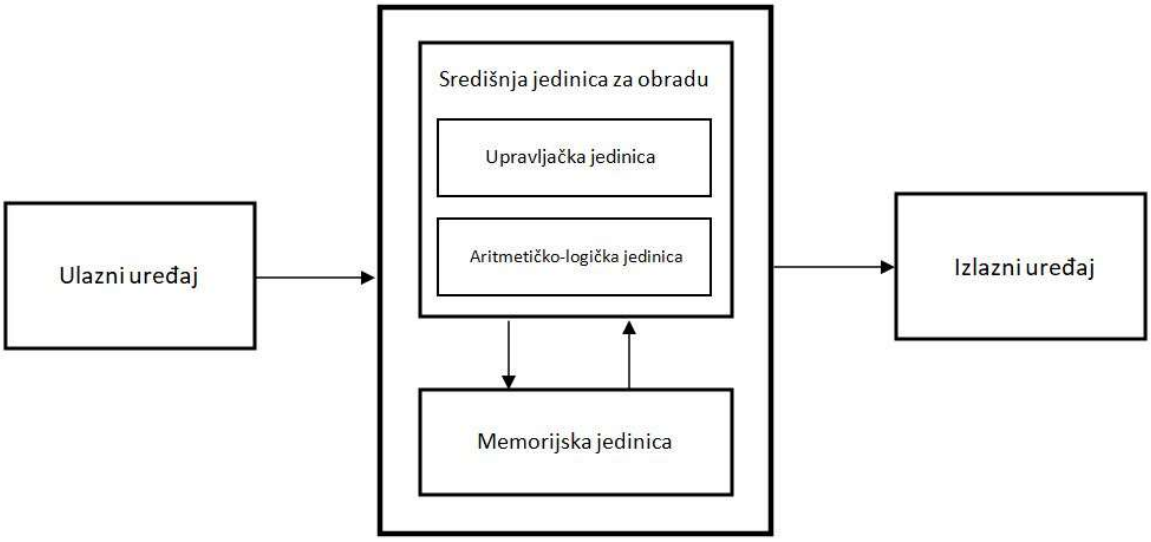
Von Neumannova arhitektura, poznata i kao Princeton arhitektura, temeljena je na opisu John von Neumann-a 1945. godine. John von Neumann bio je američko-mađarski matematičar, fizičar i računalni znalac za kojega se smatra da je bio najbolji matematičar svoga vremena.

Kada se spomene pojam von Neumann arhitekture, većinom se misli na to da se operacija s podacima i prihvaćanje instrukcije ne može obaviti u isto vrijeme jer dijele istu sabirnicu. Ovo se smatra uskim grlom (engl. *bottleneck*) arhitekture jer usporava rad cijelog računalnog sustava. Von Neumannova arhitektura opisuje računalo koje ima:

- Procesor s aritmetičko-logičkom jedinicom i registrima opće namjene
- Kontrolnu jedinicu
- Instrukcijski registar
- Programski brojač
- Memoriju koja sadrži podatke i programske naredbe
- Vanjsku memoriju velikog kapaciteta
- Mehanizme ulaza i izlaza podataka

Računala temeljena na ovoj arhitekturi prva su koja su imala ROM memoriju u kojoj su bile spremljene programske naredbe. Upisivanje programskih naredbi vršilo se na način da su se prespajale žice i sklopke, a nekada su se računala morala i redizajnirati. Taj zahtjevan i dug i postupak danas zamjenjuju razni prevoditelji (engl. *compiler*) koji omogućuju prevođenje izvornog programskog koda u strojni kod [2]. Na slici 2.1. prikazana je blok shema na kojoj se vidi da se

središnja jedinica za obradu sastoji od upravljačke jedinice i aritmetičko-logičke jedinice te da ista komunicira s memorijskom jedinicom.



Sl. 2.1. *Blok shema von Neumannove arhitekture računala.*

2.2 Harvardska arhitektura

Harvardska arhitektura je računalna arhitektura koja je nastala 1940-ih godina na Sveučilištu Harvard. Prvo računalo temeljeno na ovoj arhitekturi bilo je "Harvard Mark 1" koje je prvi programirao John von Neumann. Arhitektura je dobila ime po ovom računalu.

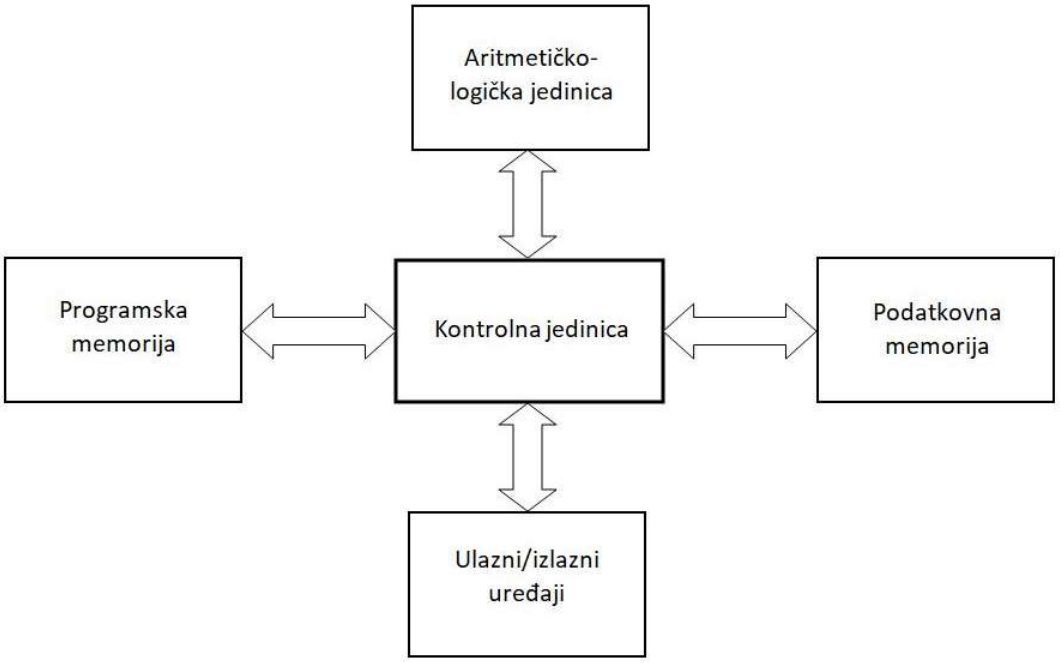
Kada se spomene pojam Harvardske arhitekture, većinom se misli na odvojenu programsku i podatkovnu memoriju koje su povezane dvjema odvojenim sabirnicama i imaju zasebne adresne prostore. Na slici 2.2. prikazana je blok shema na kojoj se vidi da upravljačka jedinica upravlja svim ostalim dijelovima procesora. Rad računala temeljenog na ovoj arhitekturi brži je od onoga temeljenog na von Neumann-ovoj jer se u isto vrijeme može pristupiti i podatkovnoj i programskoj memoriji. Prva inačica Harvardske arhitekture nije imala mogućnost učitati program iz glavne memorije u programsku već je korisnik morao ručno unositi program.

U današnje vrijeme puno procesora izmijenjenu Harvardsku arhitekturu (engl. *modified Harvard architecture*). Izmijenjena je tako da je omogućeno učitavanje programskih naredbi s vanjske memorije i izvršavanje istih.

Harvardska arhitektura opisuje računalo koje ima:

- Upravljačku jedinicu, aritmetičko-logičku jedinicu i registre opće namjene
- Programski brojač
- Instrukcijski registar
- Vanjsku memoriju velikog kapaciteta
- ROM memoriju za programske naredbe
- RAM memoriju za podatke
- Mehanizme ulaza i izlaza podataka

Brzinu rada računala temeljenog na ovoj arhitekturi teško je povećati u vidu redizajniranja iste, ali može se povećati povećavanjem priručne memorije (engl. *cache*). Priručna memorija brza je memorija koja drži zadnje adresirane podatke iz glavne memorije koja je puno sporija. Sve dok su podaci koje procesor traži u priručnoj memoriji, rad računala je brz. Isti se usporava kada traženi podatak nije u priručnoj memoriji već u glavnoj memoriji [3].



Sl. 2.2. *Blok shema Harvardske arhitekture računala.*

3. VHDL I ZYBO RAZVOJNA PLOČICA

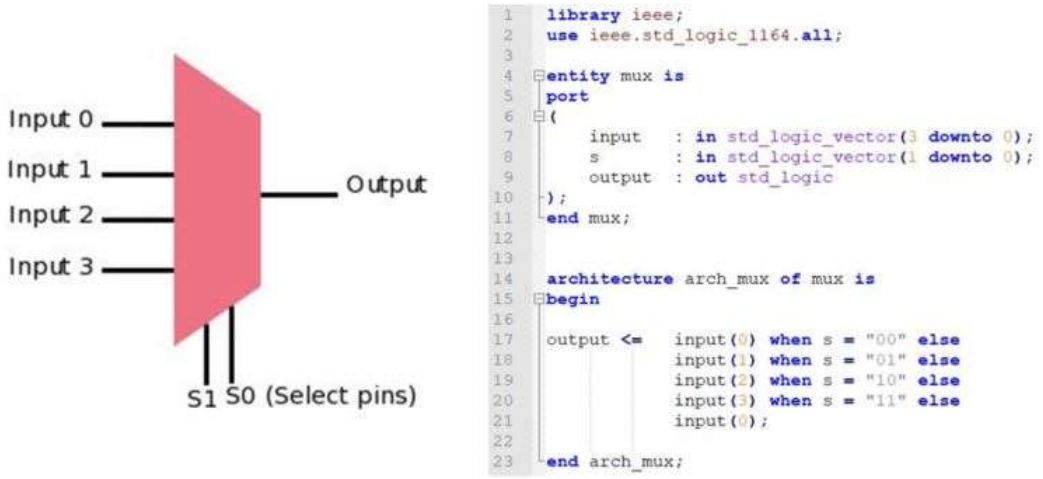
Za dizajn procesora potrebno je isti opisati u jeziku za opisivanje sklopovlja. U ovome primjeru korišten je VHDL. Također, za validaciju rješenja potrebno je imati razvojnu pločicu s FPGA integriranim krugom na istoj. U ovome primjeru korištena je ZYBO razvojna pločica.

3.1 VHDL

VHDL, jezik je za opis sklopovlja. Prvi puta se pojavio 1983. godine za potrebe američke vojske, točnije za dokumentiranje ponašanja integriranih krugova specifične primjene (engl. *Application-Specific Integrated Circuit* - ASIC). Nastao je pod utjecajem programskog jezika ADA pa su koncepti pisanja i sintaksa veoma slični istome. VHDL je zatim postao zanimljiv inženjerima pa se počeo primjenjivati za opisivanje sklopovlja, simulaciju, a nakon toga su osmišljeni alati koji su omogućili logičku sintezu. Ti alati čitaju VHDL datoteku i kao rezultat daju fizičku implementaciju digitalne logike.

VHDL se danas koristi u automatizaciji elektroničkog dizajna za opisivanje digitalnih sustava i sustava s analognim i digitalnim signalima kao što su FPGA i integrirani krugovi. VHDL raspoznaje tipove podataka iz programskog jezika ADA kako bi se korisnicima olakšao rad u istome, ali izlazne i ulazne vrijednosti iz cijelog sustava uvijek su logički vektori.

Glavni dijelovi svake VHDL datoteke su entitet (engl. *entity*) i arhitektura (engl. *architecture*). Entitet ima svoje ime (npr. controlUnit), popis ulaznih i izlaznih priključaka te popis općih (engl. *generic*) vrijednosti. Unutar bloka “arhitektura” koji također ima svoj naziv (npr. arch_controlUnit) opisuje se ponašanje odabranog entiteta s time da se prije ključne riječi *begin* navode signali koje želimo koristiti u dizajnu, a nakon iste opisuje se ponašanje odabranog entiteta [4]. Slika 3.1. prikazuje primjer opisa multipleksera u VHDL-u i njegov blokovski prikaz.



Sl. 3.1. *Opis multipleksera u VHDL-u.*

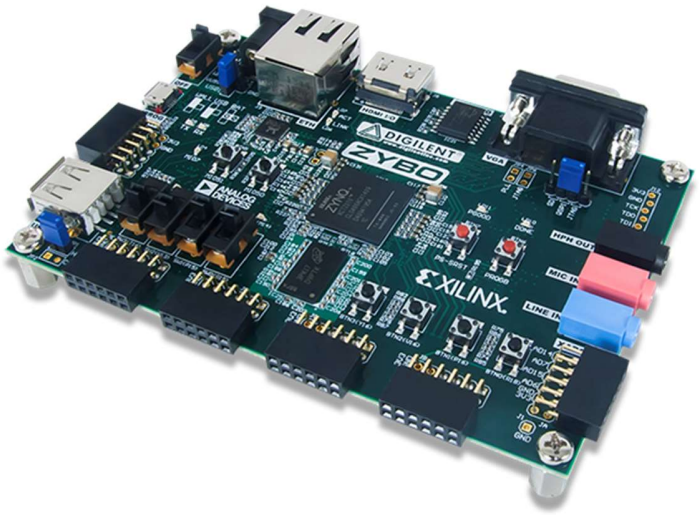
3.2 ZYBO- razvojna pločica

ZYBO (engl. *Zynq Board*), pločica je bogata značajkama koja služi za ispitivanje sklopova opisanih u nekom od jezika za opis sklopovlja i prikazana je na slici 3.2. Opremljena je najmanjim članom Xilinx Zynq-7000 porodice, Z-7010. Ovaj integrirani krug temeljen je na Xilinx-ovom sve-programabilnom sustavu na integriranom krugu (engl. *System-on-Chip* - SoC). Njega čini spoj ARM Cortex-A9 dvojezgrenog procesora i Xilinx-ove FPGA programabilne logike.

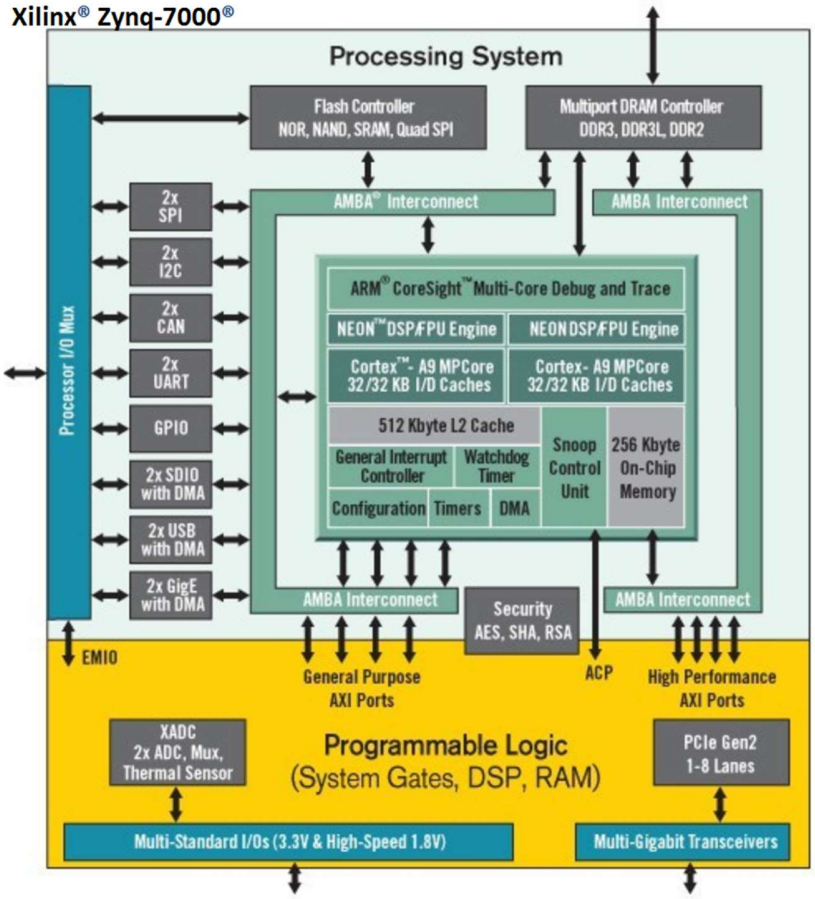
Ova razvojna pločica također je bogata sučeljima za komunikaciju s periferijom kao što je vidljivo na slici 3.3. Ona posjeduje VGA (engl. *Video Graphics Array*), HDMI (engl. *High-Definition Multimedia Interface*), audio ulaze i izlaze, USB 2.0 (engl. *Universal Serial Bus*), utor za MicroSD karticu, Gigabit Ethernet, LE diode (engl. *Light-Emitting Diode*), sklopke i tipkala. Ima 512 MB DDR3 radne memorije i 240 KB blokovske RAM memorije. Još je važno napomenuti da na pločici postoji i A-D pretvornik (engl. *Analog to Digital converter*) i pretvornik s UART-a (engl. *Universal Asynchronous Receiver Transmitter*) na USB.

Uz sve navedeno na pločici postoji 6 PMOD priključaka pomoću kojih se lako može proširiti komunikacija s periferijom. Isti mogu imati 3 stanja, a to su: 0 – nisko logičko stanje, 1 – visoko logičko stanje i Z – stanje visoke impedancije.

Na slici 3.3 se može vidjeti arhitektura ZYNQ 7000 integriranog kruga. Također se može vidjeti i podjela integriranog kruga na PS (engl. *Processing System*) i PL (engl. *Programmable Logic*). PS sadržava ARM Cortex A9 procesor i sučelja za komunikaciju s periferijom i PL-om. PL je FPGA dio integriranog kruga, tj. onaj dio koji se opisuje u jeziku VHDL dok se ARM procesor programira u programskom jeziku C [5].



Sl. 3.2. *Fizički izgled ZYBO razvojne pločice* [5].



Sl. 3.3. *Blokovska shema Zynq 7000 integriranog kruga* [5].

4. DIZAJNIRANJE PROCESORA

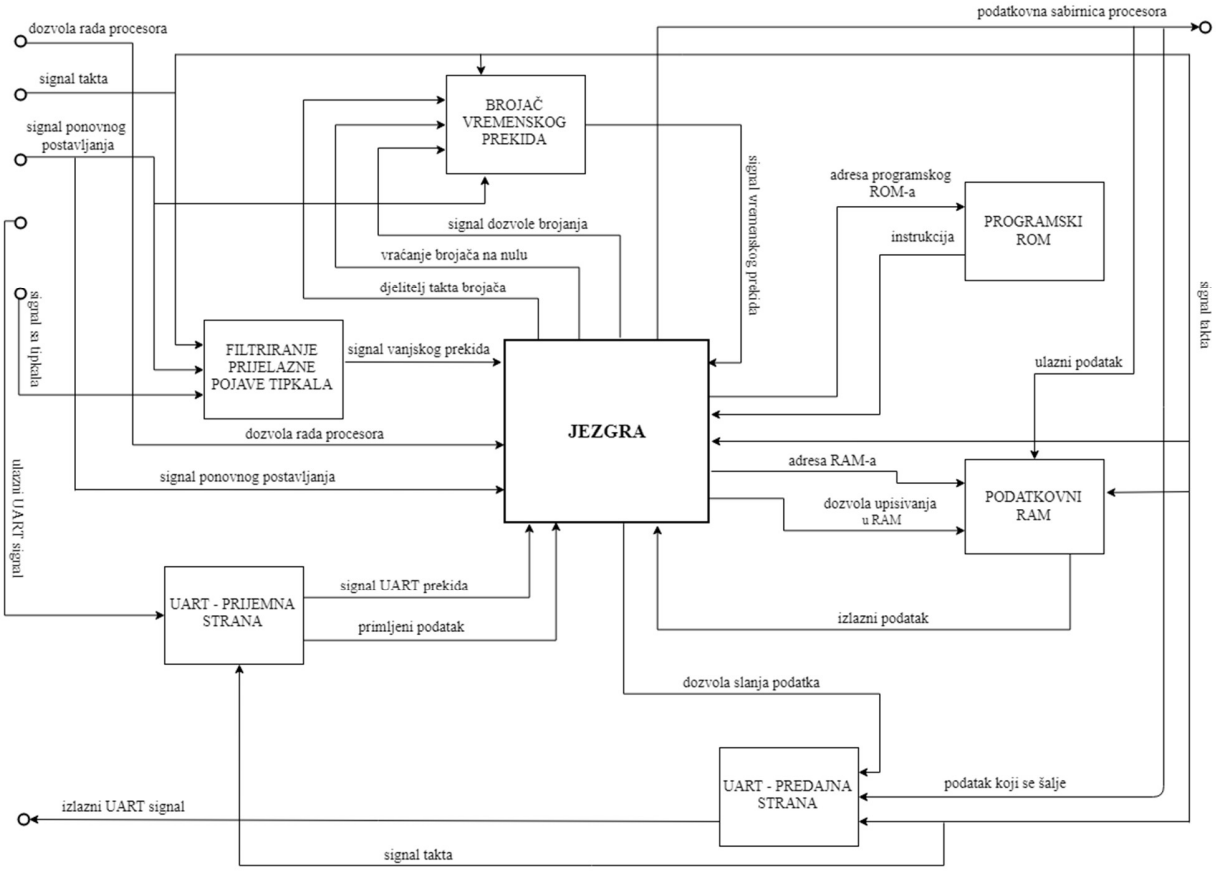
U ovome poglavlju opisan je svaki dio procesora u vidu njegove funkcije, načina rada i blokovskog prikaza. Na samom početku prikazana je blokovska shema vrha hijerarhije dizajna s ciljem lakšeg razumijevanja implementirane Harvardske arhitekture, zatim je opisana blokovska shema vrha hijerarhije dizajna, a zatim svaki entitet pojedinačno.

Zahtjevi koje ispunjava procesor su sljedeći:

- Sadržava:
    - 12-bitovnu podatkovnu sabirnicu
    - 8 registara opće namjene
    - Akumulatorski registar
    - Registar za spremanje rezultata s ALU
    - 22-bitovni registar za instrukcije
    - Multiplekser za selekciju sadržaja s registara opće namjene
    - Aritmetičko-logičku jedinicu
    - 7-bitovni programski brojač
    - Upravljačku jedinicu
    - Dekoder za generiranje dozvole upisa podataka u registre opće namjene
    - RAM i ROM memoriju
    - Statusni registar sa zastavicama nule, predznaka i prijenosa
- Brzina procesiranja:
    - Svaka instrukcija se izvršava 4 perioda takta
- Sadržava periferiju:
    - Podrška za UART protokol
    - 12-bitovna ulazna i izlazna sabirnica
    - Zahtjevi za vanjskim, vremenskim i UART prekidom
    - 16-bitovni vremenski brojač
- Sadržava instrukcije:
    - Za upis i ispis podataka iz registara opće namjene te RAM i ROM memorije
    - Za zbrajanje i oduzimanje cjelobrojnih vrijednosti

- Booleova logička algebra: I, ILI, NE, NI, NILI, EXILI, EXNILI
- Za logički posmak ulijevo i udesno

4.1 Vrh hijerarhije dizajna – jezgra i periferija



Sl. 4.1. *Blokovski prikaz vrha hijerarhije dizajna - jezgra procesora i periferija.*

Slika 4.1. prikazuje blokovski prikaz vrha hijerarhije dizajna procesora. Središte cijelog sustava je jezgra koja izvršava zadane programske naredbe, a ostali blokovi koji se smatraju njezinom periferijom omogućuju komunikaciju iste s vanjskim uređajima kao što je npr. računalo ili mikroupravljač. Jezgra će biti opisana kasnije jer je prije potrebno znati kakvi signali ulaze i izlaze iz iste.

Signali koji ulaze u procesor su: dozvola rada procesora, signal takta, signal ponovnog postavljanja, signal s tipkala i ulazni UART signal. Dozvola rada procesora omogućava rad automata konačnog stanja unutar entiteta UPRAVLJAČKA JEDINICA, što će kasnije biti detaljnije objašnjeno. Signal takta, frekvencije 125 MHz dolazi s oscilatora i omogućuje sinkronizaciju cijelog sustava. Istim se može manipulirati pomoću specifičnog entiteta u Vivado softverskom paketu. Signal ponovnog postavljanja dolazi s jednog od tipkala. Kada je isto pritisnuto, svi dijelovi sustava se inicijaliziraju na unaprijed postavljene vrijednosti. Signal s tipkala vezan je uz vanjski prekid. Isti se dovodi na entitet FILTRIRANJE PRIJELAZNE POJAVE TIPKALA. Ulazni UART signal dolazi preko USB-UART mosta s računala koje šalje podatke. Isti je povezan s UART prekidom (UART prekid se ostvaruje primanjem podatka s UART-a).

Signali koji izlaze iz procesora su: podatkovna sabirnica i izlazni UART signal. Podatkovna sabirnica je ona koja izlazi iz jezgre, tj. sabirnica na kojoj se pojavljuju izlazni podaci iz procesora, a izlazni UART signal je onaj kojim se šalju podaci na neki periferni uređaj kao što je računalo poštivajući pravila UART protokola.

Entiteti UART – PRIJEMNA STRANA i UART – PREDAJNA STRANA prema slici 4.1. omogućuju komunikaciju s perifernim uređajima, tj. omogućuju asinkrono i serijsko slanje i primanje podataka. Ovi entiteti opisani su prema pravilima UART protokola. Kada podatak stigne, signal UART prekida iz entiteta UART – PRIJEMNA STRANA ostane u logičkom stanju '1' jedan period takta. Ako korisnik omogući prekid kroz programske naredbe zbog prethodno navedenog dogodit će se prekid te će se normalan tijek izvođenja programa prekinuti, a isti će otići u UART prekidnu rutinu. JEZGRA također može zatražiti slanje podatka prema perifernom uređaju tako da se podatak koji se želi poslati postavi na izlaz podatak koji se šalje, a izlaz dozvola slanja podatka postavi u logičko stanje '1' tijekom jednog perioda takta.

Entitet FILTRIRANJE PRIJELAZNE POJAVE TIPKALA (engl. *debouncer*) prema slici 4.1. vezan je uz vanjski prekid. Prva od njegovih zadaća je da otkloni utjecaj pojave istitravanja kontakta tipkala kod promjene stanja tipkala (pritisnuto/ne pritisnuto), a druga da održi signal vanjskog prekida koji se šalje na JEZGRA u logičkom stanju '1' tijekom jednog perioda takta. Ovo je riješeno na način da se detektira prva promjena tipkala iz logičkog stanja '0' u '1' i tijekom jednog perioda takta postavi signal vanjskog prekida u logičko stanje '1'. Zatim se 200 ms zanemaruje bilo kakva promjena tipkala, a nakon toga, ako je signal sa tipkala u logičkom stanju '1' ponovno se šalje signal

vanjskog prekida prema JEZGRA. Ovako je urađeno da bi se osiguralo izvođenje jedne prekidne rutine po jednom pritisku tipkala.

Entitet BROJAČ VREMENSKOG PREKIDA prema slici 4.1. vezan je uz vremenski prekid. Omogućavanje brojanja izvodi se postavljanjem signala signal omogućavanja brojanja u logičko stanje '1'. Vraćanje brojača na nulu vrši se postavljanjem signala vraćanje brojača na nulu u logičko stanje '1'. Interval brojanja može se povećati odabiranjem određenog djelitelja takta brojača. Djelitelj takta brojača dijeli ulaznu frekvenciju takta brojača i time smanjuje brzinu kojom se brojač inkrementira. Signal djelitelj takta brojača, 2 – bitni je vektor koji omogućuje četiri različita djelitelja takta, a to su: 1, 256, 512 i 1024. Ovaj signal funkcionira na isti način kao i ostali signali prekida, tj. postavlja se u logičko stanje '1' u trajanju od jednog perioda takta svaki puta kada brojač izbroji do kraja. Kao i u prošlim slučajevima prekida i u ovome slučaju korisnik mora kroz programske naredbe omogućiti vremenski prekid.

Entitet PROGRAMSKI ROM prema slici 4.1. pohranjuje programske naredbe. U ovu memoriju nije moguće upisivati podatke tijekom rada već je jedino moguće zadati adresu adresa programskog ROM-a prema kojoj će ovaj entitet na izlazu instrukcija dati tražene podatke. Adresna sabirnica ROM memorije je 7 – bitovna pa iz toga zaključujemo da najveća dubina memorije može biti 128, tj. u nju se može smjestiti 128 programskih naredbi (instrukcija). Veličina riječi je 22 bita, tj. programska naredba je 22 – bitovna.

Entitet PODATKOVNI RAM prema slici 4.1., intelektualno je vlasništvo (engl. *Intelectual Property* - IP) razvojnog okruženja Vivado izvornog naziva "Block Memory Generator". Parametre istog namješta korisnik u razvojnom okruženju. U ovome slučaju veličina adresne sabirnice je 10 – bitovna. Za upis podataka u memoriju potrebno je ulazni podatak postaviti na signal ulazni podatak, a signal dozvola upisivanja postaviti u logičko stanje '1'. Ako ipak želimo čitati podatke, potrebno je postaviti adresu s koje želimo čitati na signal adresa RAM-a, a signal dozvola upisivanja postaviti u logičko stanje '0'. Tada će se na signalu izlazni podatak pojaviti željeni podatak.

4.2 Jezgra procesora

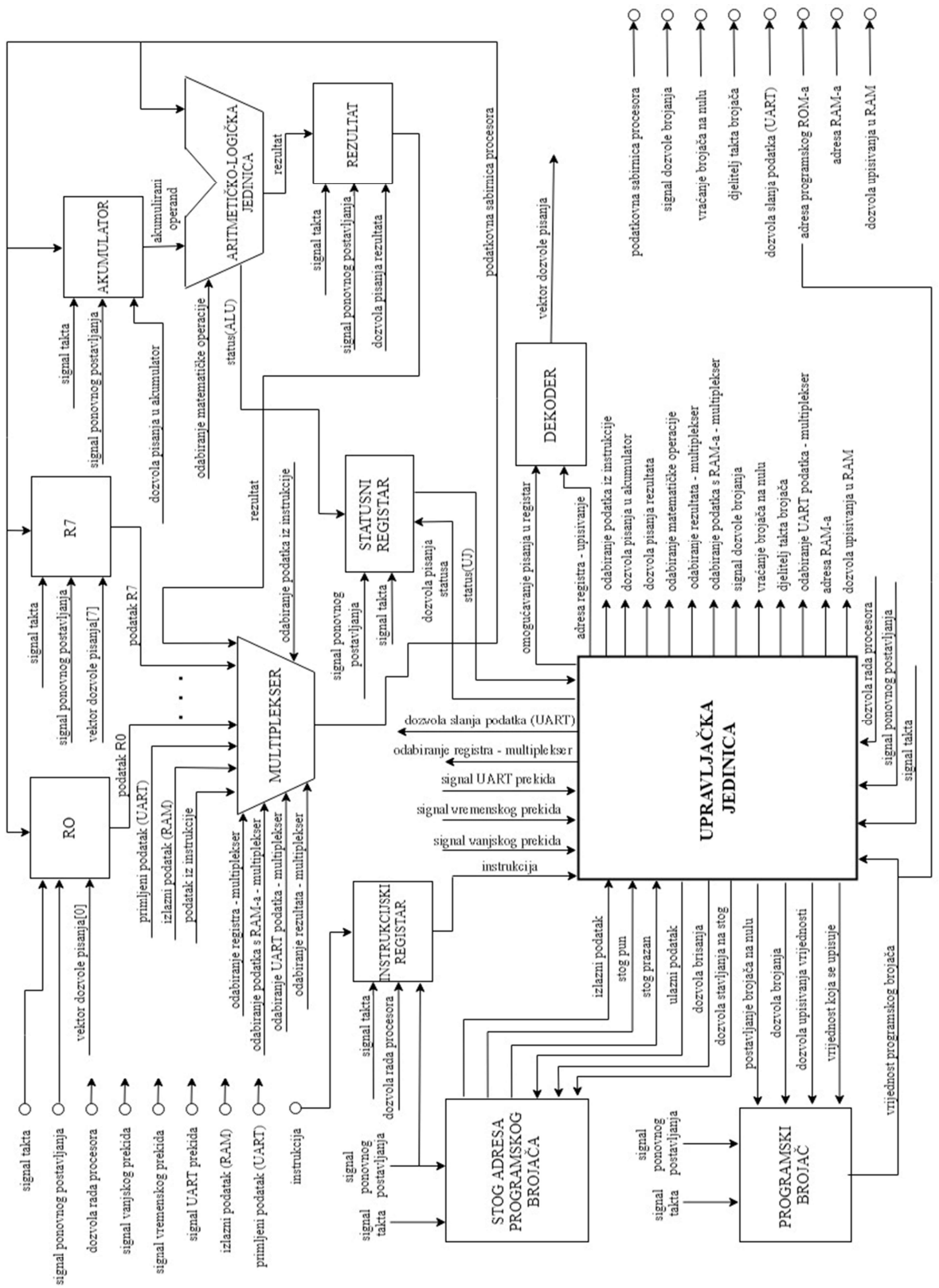
Jezgra procesora sadrži blokove i module na silicijskom čipu uključujući upravljačku jedinicu i aritmetičko logičku jedinicu te je odgovorna za izvršavanje raznih strojnih uputa. Za svaku zadanu instrukciju, u jezgri je točno određeno kojim će se redoslijedom izvoditi promjene određenih signala da bi se kao rezultat podatak spremio u registar ili u RAM ili se poslao preko UART-a itd. Kako bi se lakše shvatio rad same jezgre, svaki entitet unutar iste bit će pojedinačno opisan uz blok shemu spajanja svih entiteta unutar jezgre koja se može vidjeti na slici 4.2.

4.2.1 Upravljačka jedinica i dekoder

Upravljačka jedinica dio je jezgre procesora koji upravlja svim ostalim dijelovima jezgre. Ona prema primljenoj instrukciji na svom ulazu daje upravljačke signale koji omogućuju protok podataka kroz procesor kao npr. odabir aritmetičko-logičke operacije ili odabir podatka koji će se preko multipleksera pojaviti na podatkovnoj sabirnici. Također, ona omogućuje sekvencijalno izvođenje programskih naredbi jer je točno određeno u kojemu stanju koji upravljački signal mijenja svoje logičko stanje i kada se počinje izvršavati iduća instrukcija. Ovakav način rada moguće je jednostavno postići automatom konačnih stanja. Broj stanja automata konačnih stanja ovisi o složenosti operacije koja se izvodi [6].

Slikom 4.2. prikazan je entitet `UPRAVLJAČKA JEDINICA` (u daljnjem tekstu UJ) povezan s ostalim entitetima unutar jezgre. U sljedećim odjeljcima bit će opisana veza `UPRAVLJAČKA JEDINICA` - svi ostali entitetI, a shematski prikaz potrebno je pratiti prema slici 4.2.

Veza `UJ` i entiteta `PROGRAMSKI BROJAČ` ostvarena je izlaznim signalima: `postavljanje brojača na nulu`, `dozvola brojanja`, `dozvola upisivanja vrijednosti`, `vrijednost koja se upisuje` i ulaznim signalom `vrijednost programskog brojača` koji obavještava `UJ` s koje adrese stiže programska naredba iz `PROGRAMSKI ROM` ili koja je iduća adresa s koje stiže programska naredba. Način na koji ovi izlazni signali upravljaju entitetom `PROGRAMSKI BROJAČ` moguće je pročitati u potpoglavlju 4.2.5.



Sl. 4.2. *Blokovski prikaz jezgre procesora.*

Veza UJ i entiteta STOG ADRESA PROGRAMSKOG BROJAČA ostvarena je izlaznim signalima: ulazni podatak, dozvola brisanja i dozvola stavljanja na stog te ulaznim signalima: izlazni podatak, stog pun, stog prazan. Način na koji ovi izlazni signali upravljaju entitetom STOG ADRESA PROGRAMSKOG BROJAČA moguće je pročitati u potpoglavlju 4.2.5.

Veza UJ i entiteta INSTRUKCIJSKI REGISTAR ostvarena je ulaznim signalom instrukcija što znači da UJ ne upravlja ovim entitetom već od istoga dobiva instrukciju koju treba obraditi.

Veza UJ i entiteta MULTIPLEKSER ostvarena je izlaznim signalima: odabiranje registra - multiplekser, odabiranje podatka s RAM-a – multiplekser, odabiranje UART podatka - multiplekser, odabiranje podatka iz instrukcije i odabiranje rezultata – multiplekser. UJ ne zahtijeva nikakve podatke od entiteta MULTIPLEKSER već prema datoj instrukciji tim entitetom odabire koje podatke će propustiti na podatkovnu sabirnicu. Način upravljanja entitetom MULTIPLEKSER moguće je pročitati u potpoglavlju 4.2.4.

Veza UJ i skupa entiteta R0…R7 ostvarena je izlaznim signalima: omogućavanje pisanja u registar i adresa registra – upisivanje. UJ ne zahtijeva nikakve podatke od registara opće namjene već odabire kada će i u koji registar biti upisan podatak s podatkovne sabirnice. Ovi izlazni signali ne idu direktno do registara već preko entiteta posrednika DEKODER koji omogućava da se pomoću 3-bitne adrese registra može upravljati s 8 registara. Na izlazu iz entiteta DEKODER zato se nalazi 8-bitovni vektor dozvole pisanja. Način upravljanja registrima opće namjene moguće je pročitati u potpoglavlju 4.2.3.

Veza UJ i entiteta: AKUMULATOR, REZULTAT i STATUSNI REGISTAR ostvarena je izlaznim signalima: dozvola pisanja u akumulator, dozvola pisanja rezultata i dozvola pisanja statusa te ulaznim signalom: status(UJ). Po imenu signala i shemi na slici 4.2. može se zaključiti koji signal pripada kojoj pojedinoj vezi. Veze između navedenih entiteta su slične pa će biti objašnjene u istom odlomku. Svi navedeni izlazni signali služe kao dozvola pisanja u registar. Jedino je veza između statusnog registra dvosmjerna, tj. ulazni signal status(UJ) potreban je UJ da bi znala obraditi uvjetne skokove ili uvjetne izlaske iz podrutina.

Veza UJ i entiteta ARITMETIČKO-LOGIČKA JEDINICA (u daljnjem tekstu ALU) ostvarena je izlaznim signalom odabiranje matematičke operacije. Navedeni signal kazuje o kojoj je aritmetičkoj ili logičkoj operaciji riječ.

Kao što je već rečeno, automat konačnih stanja omogućuje sekvencijalno izvođenje programskih naredbi i lakše opisivanje ponašanja UJ u VHDL-u. U ovome primjeru koristi se Mealyev automat konačnih stanja (u daljnjem tekstu AKS).

Automat korišten za dizajniranje ovog procesora je Mealyev s 5 stanja, a to su redom: `stanje ponovnog postavljanja` (engl. *reset*), `stanje prihvaćanja instrukcije`, `stanje dekodiranja`, `stanje izvršenja` (engl. *execute*) i `stanje prelaska na iduću instrukciju`. Kada se pritisne vanjsko tipkalo, namijenjeno za ponovno postavljanje procesora, AKS asinkrono odlazi u `stanje ponovnog postavljanja`, a kada se isto pusti AKS postavlja iduće stanje. Tablica 4.1. prikazuje slijed mijenjanja stanja AKS-a.

Tab. 4.1. *Slijed mijenjanja stanja AKS-a.*

TRENUTNO STANJE	**IDUĆE STANJE**	**STANJE TIPKALA PONOVNOG POSTAVLJANJA**
stanje ponovnog postavljanja	stanje ponovnog postavljanja	‘1’
stanje ponovnog postavljanja	stanje prihvaćanja instrukcije	‘0’
stanje prihvaćanja instrukcije	stanje dekodiranja	‘0’
stanje dekodiranja	stanje izvršenja	‘0’
stanje izvršenja	stanje prelaska na iduću instrukciju	‘0’
stanje prelaska na iduću instrukciju	stanje prihvaćanja instrukcije	‘0’

Ovaj primjer procesora raspoznaje 48 različitih operacija (instrukcija). Operacije `UJ` su radi lakšeg razumijevanja grupirane u tablicama koje se nalaze u Prilogu 3.

Širina instrukcijske sabirnice je 22 bita kako bi svaka instrukcija imala dovoljno bitova da obuhvati sve što mora stići do `UJ`. Podaci koji se mogu nalaziti unutar koda instrukcije su: adrese registara opće namjene, adresa u RAM memoriji, stanje programskog brojača i 12-bitovni podatak.

4.2.2 Aritmetičko-logička jedinica

Aritmetičko-logička jedinica (engl. *Arithmetic Logic Unit* - ALU) digitalni je elektronički sklop unutar jezgre procesora. Njena zadaća je da vrši osnovne aritmetičke operacije kao što su zbrajanje, oduzimanje, množenje i logičke operacije kao što su I, ILI, NE, XOR, XNOR itd. Budući da je sadržana od isključivo kombinacijske logike, rad je asinkron što znači da ne postoji signal takta koji sinkronizira rad iste. Zbog prethodno navedenog, potreban je određeni vremenski period da se stanje na izlazu stabilizira prema postavljenim ulazima, tj. da se izvrši zadana operacija [7].

Na slici 4.2 može se vidjeti kako je entitet `ARITMETIČKO-LOGIČKA JEDINICA` povezan s ostalim entitetima u jezgri. Zbog same primjene ovog entiteta, potrebno je na ulazu imati dvije sabirnice, tj. dva operanda. Jedan je akumulirani operand, a drugi dolazi izravno s `podatkovne sabirnice` procesora. O ovome se više može pročitati u potpoglavlju 4.2.1. Popis aritmetičkih i logičkih operacija i njihov kod, vidljiv je u tablici 4.2.

Razlika operacije „zbrajanje“ i operacije „zbrajanje s podatkom iz instrukcije“, „oduzimanje“ i „oduzimanje s podatkom iz instrukcije“ itd. je u tome da se npr. kod operacije „zbrajanje“ podaci uzimaju iz registara čije se adrese nalaze u kodu instrukcije, a kod operacije „zbrajanje s podatkom iz instrukcije“ u kodu instrukcije postoji adresa registra iz koje se uzima prvi podatak, a drugi podatak s kojim će se vrijednost prethodno spomenutog registra zbrojiti se nalazi direktno u kodu instrukcije. Kod operacija ovakvog tipa, rezultat se uvijek sprema u prvi navedeni registar u kodu instrukcije.

Izlazni signal na koji treba obratiti pažnju je `status(ALU)`. Isti je spojen na ulaz statusnog registra, a funkcija mu je postavljanje zastavica nule, prijenosa i predznaka. Zastavica nule postavlja se u logičko stanje ‘1’ ako je rezultat bilo koje matematičke operacije 0. Zastavica prijenosa postavlja se u logičko stanje ‘1’ ako je prilikom zbrajanja, rezultat prešao maksimalnu vrijednost određenu brojem bitova podatka (u ovome slučaju radi se o broju 4095) ili se prilikom oduzimanja prešla vrijednost 0. Prilikom izvođenja operacije “pomjeranje ulijevo”, zastavica prijenosa postavlja se na vrijednost MSB-a dok se kod prilikom izvođenja operacije “pomjeranje udesno”, zastavica prijenosa postavlja na vrijednost LSB-a (engl. *Least Significant Bit*). Zastavica predznaka postavlja se u logičko stanje ‘1’ ako je pri operaciji uspoređivanja, oduzimanja ili oduzimanja s podatkom iz instrukcije rezultat bio negativan.

Tab. 4.2. *Tablica aritmetičkih i logičkih operacija te pripadajućih instrukcijskih kodova.*

OPERACIJA	INSTRUKCIJSKI KOD
zbrajanje	000010
oduzimanje	000011
pomjeranje ulijevo	000100
pomjeranje udesno	000101
I	000110
ILI	000111
NE	001000
NI	001001
NILI	001010
EXILI	001011
EXNILI	001100
uspoređivanje	011111
zbrajanje s podatkom iz instrukcije	100000
oduzimanje s podatkom iz instrukcije	100001
uspoređivanje s podatkom iz instrukcije	100010

4.2.3 Registri unutar jezgre

Registri su komponenta sklopovlja računala. Njihova glavna zadaća je spremanje podataka, a u jezgri su od velike važnosti jer su u vidu pristupa podacima veoma brzi. Iako im je glavna primjena spremanje podataka, također se koriste u kombinacijskoj logici ako signal pri nekoj frekvenciji takta ne uspije doći na odredište (engl. *pipelining*). Isti se registrira i u idućem periodu takta stiže na odredište. U sinkroniziranom sustavu važno je svakom registru dovesti signal takta i signal ponovnog postavljanja. Također se u RTL (engl. *Register-Transfer Level*) inženjerstvu podrazumijeva svakom registru dovesti signal dozvole upisivanja podatka i ulazni podatak, a izlaz iz registra je izlazni podatak, tj. podatak spremljen u registru. Način na koji registri funkcioniraju je vrlo jednostavan. Na ulaz registra dovodi se podatak koji se želi spremiti, a zatim se signal dozvole pisanja postavlja u logičko stanje '1' [8].

Entiteti RO … R7 sa slike 4.2. prikazuju registre opće namjene i ima ih 8. Njihova glavna zadaća jest privremeno spremanje podataka nad kojima se obavlja neka matematička operacija ili koji će kasnije biti spremljeni u PODATKOVNI RAM, poslani preko UART-a itd. Veličina istih može biti različita i mjeri se u broju bitova koje može pohraniti. U ovome slučaju su 12 – bitovni. Registri opće namjene su najbrži registri. Upravljanjem signala dozvole pisanja bavi se UPRAVLJAČKA JEDINICA preko entiteta DEKODER. Ovim registrima može se pristupati direktno iz programskih naredbi i to preko adrese registra. Na ovaj način, korisnik može ručno zadati registre nad čijim podacima želi da se obavi operacija.

Slika 4.2. također prikazuje entitete AKUMULATOR i REZULTAT. Usko su vezani za entitet ARITMETIČKO-LOGIČKA JEDINICA s time da AKUMULATOR stoji na ulazu, a REZULTAT na izlazu iste. Upravljanjem signalima dozvole pisanja također se bavi UPRAVLJAČKA JEDINICA Tijekom jednog perioda takta izvršavanja instrukcije, podatak s podatkovne sabirnice sprema se u AKUMULATOR, a tijekom drugog perioda takta nad istim podatkom i drugim podatkom koji je stigao na podatkovnu sabirnicu izvršava se operacija i dozvoljava pisanje u REZULTAT. Podatak iz entiteta REZULTAT zatim se može spremiti u neki od registara opće namjene i koristiti u druge svrhe.

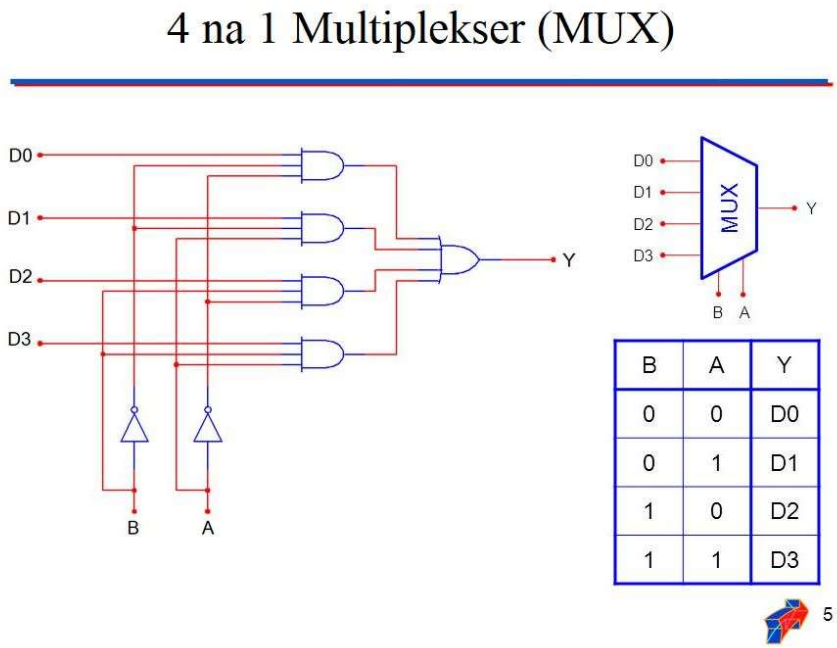
Prema slici 4.2., entitet INSTRUKCIJSKI REGISTAR registar je u koji se sprema instrukcija. Ulaz registra spojen je na entitet PROGRAMSKI ROM što se može vidjeti na slici 4.1., a na izlazu se instrukcija šalje prema entitetu UPRAVLJAČKA JEDINICA. Može se vidjeti kako je ulaz dozvole pisanja spojen na signal dozvole rada procesora zato što instrukcija ne treba biti spremljena u registar kada isti ne radi, tj. programske naredbe se ne izvršavaju pa se instrukcija ne treba mijenjati.

4.2.4 Multiplekser

Multiplekser (engl. *Multiplexer*) je sklop kojim se bira jedan od više analognih ili digitalnih signala spojenih na njegov ulaz. Multiplekser koji ima 2^n ulaza, mora imati n linija za odabir ulaza (engl. *select lines*). Postavljanjem ovih linija u unaprijed određeno logičko stanje na izlazu se dobiva željeni podatak s ulaza. Na slici 4.3. može se vidjeti jedna od izvedbi multipleksera korištenjem NE, ILI i I logičkih vrata. Također može se vidjeti blokovska shema i zaključiti kako multiplekser funkcionira prema tablici istinitosti [9].

Na slici 4.2. može se vidjeti na koji je način multiplekser spojen s ostalim entitetima unutar jezgre. Budući da procesor radi s 12-bitnim podacima, svi ulazi i izlaz multipleksera su 12-bitni. Izuzetak je ulazni signal `primljeni podatak (UART)` koji je 8-bitni, te se u tom slučaju ostalih 4 MSB *(*engl. *Most Significant Bit*) bita postavlja u logičko stanje '0'.

Ulazi multipleksera su redom `izlazni podatak (RAM)`, `primljeni podatak (UART)`, `podaci iz registara "R0…R7"`, `rezultat` te `podatak iz instrukcije`, a izlaz istoga je spojen direktno na `podatkovnu sabirnicu`. Ovaj multiplekser nešto je drugačiji od onoga na slici 4.3. u vidu toga da ima 5 sabirnica za odabiranje ulaza, a jedna od njih je 3-bitna dok su sve ostale 1-bitne. Problem zbog ovoga ne nastaje jer je procesor dizajniran tako da nikada ne postavlja više ulaza za odabiranje u logičko stanje '1'. Npr. kada su svi 1-bitni ulazi za odabiranje u logičkom stanju '0', prema izlazu se odabire ulaz s registara određen kombinacijom 3-bitnog ulaznog vektora za odabiranje. Način odabiranja ulaza možemo detaljno vidjeti u tablici 4.3.



B	A	Y
0	0	D0
0	1	D1
1	0	D2
1	1	D3

Sl. 4.3. *Jedna od izvedbi 4 na 1 multipleksera pomoću osnovnih logičkih sklopova (lijevo), blokovska shema (desno gore) i tablica istinitosti(desno dolje)* [10].

Tab. 4.3. *Tablica istinitosti, način odabiranja ulaza multipleksera (X – nije važno u kojemu je stanju).*

ODABIR PODATKA S UART-a	ODABIR IZ RAM-a	ODABIR REZULTATA	ODABIR PODATKA IZ INSTRUKCIJE	ODABIR PODATKA S REGISTRA	IZLAZ
0	0	0	0	000	R0
0	0	0	0	001	R1
0	0	0	0	010	R2
0	0	0	0	011	R3
0	0	0	0	100	R4
0	0	0	0	101	R5
0	0	0	0	110	R6
0	0	0	0	111	R7
0	0	1	0	X	rezultat
0	1	0	0	X	podatak iz RAM-a
1	0	0	0	X	podatak s UART-a
0	0	0	1	X	podatak iz instrukcije

4.2.5 Programski brojač i stog adresa programskog brojača

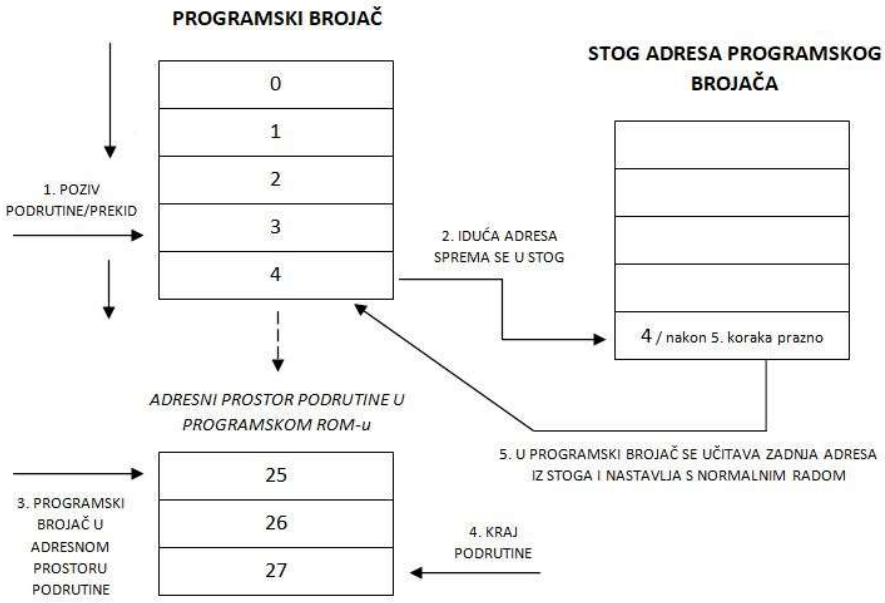
Programski brojač, registar je unutar jezgre procesora koji pohranjuje adresu instrukcije koja se trenutno izvodi ili adresu instrukcije koja će se iduća izvoditi. U normalnom načinu rada procesora, programske naredbe izvode se jedna za drugom (sekvencijalno), a to se postiže inkrementiranjem stanja brojača za jedan nakon svake izvršene instrukcije, a u ovome slučaju inkrementiranje se događa svakih 4 perioda takta. Ako su za procesor definirane instrukcije koje omogućavaju grananja kao što su uvjetni i bezuvjetni skokovi, pozivanje podrutina i vraćanje iz podrutina, prekidi i vraćanje iz prekida onda se vrijednost programskog brojača postavlja na njegov ulaz za postavljanje

vrijednosti, a ulaz omogućavanja upisa vanjske vrijednosti, postavlja se u logičko stanje ‘1’ u trajanju jednog perioda takta [11].

Stog adresa programskog brojača zapravo je LIFO (engl. *Last In First Out*) memorija. Rad ovakve memorije jednostavno je objasniti tako da podatak koji zadnji ulazi u istu, prvi izlazi s time da se kod svakog pisanja i čitanja iz memorije ažurira pokazivač stoga (engl. *stack pointer)* koji uvijek pokazuje na sljedeću adresu na koju će se upisivati podatak. Ovaj dio procesora nema svoju primjenu ako se ne koriste operacije vezane uz grananje (mijenjanje toka programa). Ako dođe do prekida ili poziva podrutine, iduća adresa programskog brojača sprema se na stog. Zatim se izvodi podrutina ili prekidna rutina te se programskom naredbom povratka zadnja adresa upisana na stog upisuje u programski brojač. Ovim pristupom spremanja adresa, omogućuje se pozivanje više podrutina unutar jedne podrutine. Slikovito objašnjenje funkcije stoga adresa programskog brojača može se vidjeti na slici 4.4.

Na slici 4.2. može se vidjeti kako je entitet `PROGRAMSKI BROJAČ` spojen s ostalim entitetima unutar jezgre. Ulazi u entitet su: `signal takta`, `signal ponovnog postavljanja`, `postavljanje brojača na nulu`, `dozvola brojanja`, `dozvola upisivanja vrijednosti`, `vrijednost koja se upisuje` dok je izlazni signal samo jedan: `vrijednost programskog brojača`. Također prema slici 4.2. može se zaključiti da programskim brojačem u potpunosti upravlja `UPRAVLJAČKA JEDINICA`.

Na slici 4.2. može se vidjeti kako je entitet `STOG ADRESA PROGRAMSKOG BROJAČA` povezan s ostalim entitetima unutar jezgre. Ulazi u entitet su: `signal takta`, `signal ponovnog postavljanja`, `ulazni podatak`, `dozvola brisanja` i `dozvola stavljanja na stog`. Izlazi iz entiteta su: `izlazni podatak`, `stog pun` i `stog prazan`. Ovim entitetom kao i entitetom `PROGRAMSKI BROJAČ` u potpunosti upravlja `UPRAVLJAČKA JEDINICA`. Ako se izvršava programska naredba poziva podrutine ili dođe do prekida, signal `dozvola stavljanja` postavlja se u logičko stanje ‘1’ u trajanju od jednog perioda takta, a na signal `ulazni podatak` postavlja se vrijednost programskog brojača uvećana za 1. Nakon što se adresa upiše, pokazivač stoga povećava se za 1 ako stog nije pun. Ako se izvršava programska naredba povratka iz podrutine ili prekidne rutine, signal `dozvola brisanja` postavlja se u logičko stanje ‘1’ u trajanju od jednog perioda takta. Pokazivač stoga automatski se smanjuje za 1, a na izlazu `izlazni podatak` nalazi se posljednja spremljena adresa.



Sl. 4.4. *Funkcija stoga adresa programskog brojača.*

4.2.6 Statusni registar

Statusnim registrom obično se smatra skup registara-zastavica (engl. *Flag registers*) koje pokazuju u kojemu se stanju nalazi procesor. Najčešće se radi o zastavicama koje se postavljaju prema rezultatu aritmetičkih ili logičkih operacija (ove zastavice postavlja ALU), ali također može sadržavati i zastavice drugih primjena kao što je zastavica globalne dozvole prekida koja ako je u logičkom stanju '0' ne dopušta odlazak niti u jednu prekidnu rutinu. Stanje zastavica mijenja se svakom novom aritmetičkom ili logičkom operacijom. Ove zastavice najčešće služe kao uvjet kod grananja programa kao što je npr. skok na određenu adresu ako je rezultat prethodne operacije bio 0 ili je bilo prijenosa pri zbrajanju [12].

Na slici 4.2. može se vidjeti kako je entitet "`STATUSNI REGISTAR`" spojen s ostalim entitetima unutar jezgre. Ulazi u entitet su: "`signal takta`", "`signal ponovnog postavljanja`", "`status(ALU)`" i "`dozvola pisanja statusa`", a izlaz je: "`status(UJ)`". U ovome primjeru statusni registar sastoji se od zastavica nule, prijenosa i predznaka. `Signal ponovnog postavljanja` postavlja sve zastavice u logičko stanje '0'. U poglavlju 4.2.2 može se pročitati više o uvjetima postavljanja zastavica u određeno logičko stanje. Ako se radi o aritmetičko-logičkoj

operaciji, upravljačka jedinica postavlja signal "`dozvola pisanja statusa`" u logičko stanje '1' tijekom jednog perioda takta, a zastavice se postavljaju na vrijednost signala "`status(ALU)`" te se iste, registrirane vrijednosti mogu očitati na izlazu "`status(UJ)`". Zastavice nule i prijenosa su od značaja u ovom primjeru jer omogućuju uvjetna grananja programa.

4.3 Periferija procesora

4.3.1 UART

UART (engl. *Universal Asynchronous Receiver-Transmitter*), komunikacijski je protokol koji omogućuje asinkronu serijsku komunikaciju preko serijskog priključka računala ili serijskog priključka nekog drugog perifernog sklopa. Svaki sklop koji podržava UART uvijek ima „RX" i „TX" priključke. Slika 4.5. prikazuje strukturu podatkovnog paketa UART protokola. Moguće je definirati duljinu podatka koji se šalje i brzinu kojom se podaci šalju. Također, moguće je definirati hoće li paket sadržavati bit pariteta koji služi za otkrivanje pogreški u prijenosu te koliko će "stop" bitova sadržavati paket. Brzina slanja podataka izražava se u baud-ima što je jednako bit/s. U ovome primjeru, brzina prijenosa je 115200 baud-a, duljina podatka je 8 bita, nema bitova pariteta i postoji jedan stop bit.

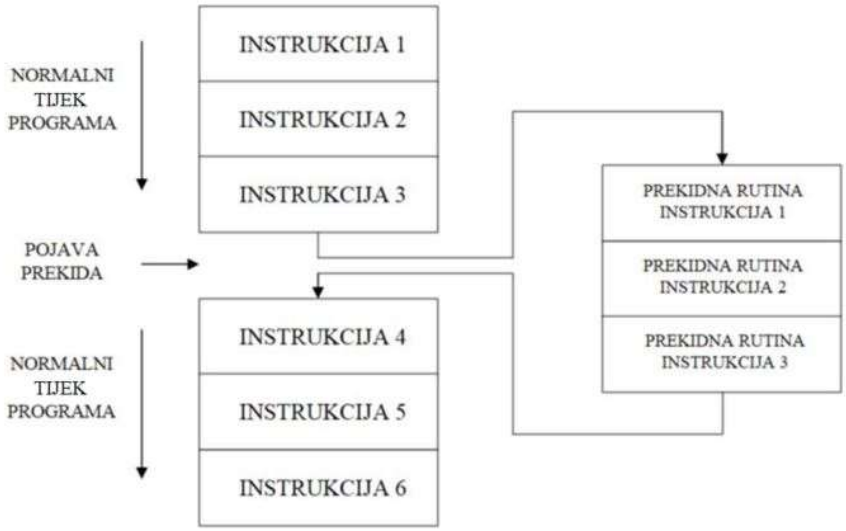
Slanje podatka pomoću UART protokola odvija se preko „TX" linije. Započinje se slanjem start bita (logičko stanje '0'), zatim se šalju podatkovni bitovi od LSB-a do MSB-a, zatim se računa i šalje paritetni bit ukoliko je korisnički postavljen i na kraju se šalju stop bitovi (logičko stanje '1'). Signal kojim se šalju podaci ostaje u logičkom stanju '1' do slanja idućeg podatka.

Primanje podatka pomoću UART protokola odvija se preko „RX" linije. Kod primanja podataka, podatke je potrebno uzorkovati frekvencijom većom od frekvencije kojom stižu podaci. U ovome slučaju ta frekvencija jest 50 MHz. Primanje podatka pomoću UART protokola započinje detekcijom start bita (promjena logičkog stanja iz '1' u '0'). Zatim se čeka 1.5 trajanje slanja jednog bita i detektira LSB bit podatka. Svi ostali bitovi podatka, paritet i stop bitovi uzorkuju se svako 1 trajanje slanja jednog bita. Na ovaj način, svaki primljeni bit se uzorkuje na polovici trajanja njegovog slanja što povećava vjerojatnost da je poslani podatak jednak primljenom.

Sl. 4.5. *Struktura podatkovnog paketa UART protokola.*

4.3.2 Prekidi i prekidne rutine

U normalnom načinu rada, procesor izvršava instrukcije jednu za drugom sve dok se ne dogodi neki od ovih događaja: brojač je izbrojao do kraja, došao je podatak preko UART protokola ili je korisnik pritisnuo tipkalo vanjskog prekida. Ovi događaji zovu se zahtjevi za prekidom i generira ih periferija. Kako bi se ostvario neki od prekida, npr. UART prekid, u logičko stanje '1' je potrebno postaviti zastavicu globalne dozvole prekida i zastavicu dozvole UART prekida. Isto vrijedi za sve ostale vrste prekida (vanjski, vremenski). Prekidna rutina je kod koji započinje na unaprijed određenoj adresi i uvijek završava s instrukcijom za povratak u glavni program, a izvršava se ako je prihvaćen zahtjev za prekidom, tj. ispunjeni su svi uvjeti. Ako su ostvareni svi prethodno spomenuti uvjeti i dođe do zahtjeva za prekidom, procesor do kraja izvrši instrukciju koju je započeo izvoditi, zatim učitava adresu prekidne rutine u programski brojač, izvršava instrukcije u prekidnoj rutini te izvršavanjem instrukcije za povratak iz prekidne rutine upisuje u programski brojač adresu instrukcije na kojoj je rad procesora bio prekinut, povećanu za 1. U ovome primjeru svi prekidi imaju jednak prioritet što znači da prilikom izvršavanja neke prekidne rutine, istu ne može prekinuti ni jedan drugi prekid. Primjer prekida prikazan je slikom 4.6.

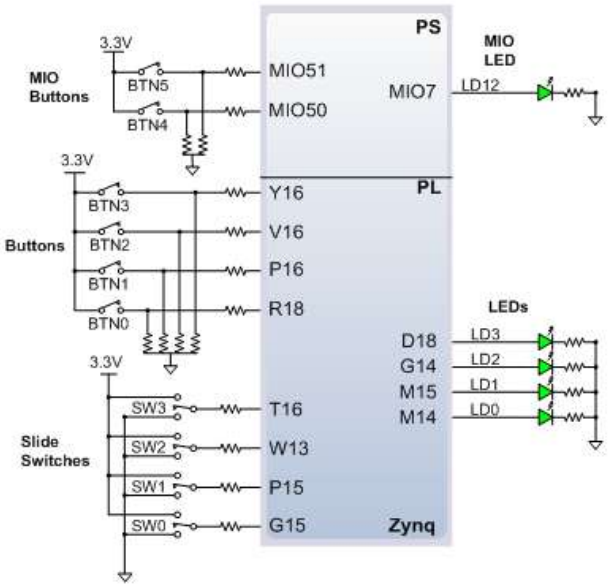


Sl. 4.6. *Prekid; skok u prekidnu rutinu.*

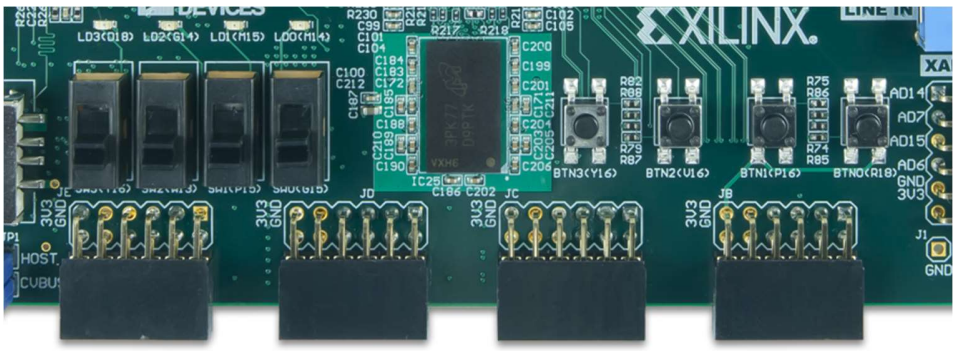
4.3.3 Ulazni i izlazni sklopovi opće namjene

Ulazni i izlazni sklopovi opće namjene (engl. *General Purpose Input-Output* - GPIO) korišteni u ovom primjeru su oni koji pripadaju ZYBO razvojnoj pločici. ZYBO pločica posjeduje 4 klizne sklopke i 4 tipkala u vidu ulaznih sklopova te 4 LED-ice u vidu izlaznih sklopova opće namjene. Tipkala i klizne sklopke su spojene na ZYNQ preko serijski spojenih otpornika radi zaštite od kratkog spoja koji može nastati slučajnim definiranjem sklopke ili tipkala kao izlaza. Ne pritisnuta tipkala generiraju logičko stanje '0', a pritisnuta logičko stanje '1'. Način na koji su sklopke, tipkala i LED-ice spojeni na ZYNQ može se vidjeti na slici 4.7., a fizički smještaj na ZYBO pločici može se vidjeti na slici 4.8. [13].

U ovom primjeru procesora, jedna od sklopki korištena je za dozvolu rada procesora, jedno tipkalo za signal ponovnog postavljanja i još jedno tipkalo za signal vanjskog prekida.



Sl. 4.7. *Shema spoja kliznih sklopki, tipkala, LED-ica i ZYNQ-a;gornji dio slike (PS) odnosi se na procesorski dio ZYNQ-a pa ga ne treba promatrati [13].*



Sl. 4.8. *Fizički smještaj kliznih sklopki (sredina-lijevo), tipkala (sredina-desno) i LED-ica (gore-lijevo) na ZYBO pločici [13].*

5. VERIFIKACIJA I VALIDACIJA DIZAJNA PROCESORA

Verifikacija dizajna procesora odrađena je u više koraka. Nakon svakog opisanog entiteta, za isti je kreirana datoteka za ispitivanje funkcionalnosti dizajna (engl. *Test Bench*) te se u ModelSim okruženju za računalnu simulaciju ispitala točnost rada tog entiteta. Tako je urađeno i nakon spajanja svih entiteta u cjelinu. Idući korak verifikacije odrađen je u softverskom paketu Vivado. U Vivado-u je sustav nadograđen RAM memorijom i sklopom za manipulaciju frekvencijom takta, zatim je odrađena simulaciju nakon sinteze procesora i konačno je programiran ZYNQ integrirani krug. Validacija dizajna procesora odrađena je spajanjem ZYBO razvojne pločice na računalo preko UART sučelja gdje je korisnik zadao vrijednost od koje je dizajnirani procesor izračunao faktorijel te ga poslao na računalo.

5.1 ModelSim, simulacija sustava

ModelSim, Mentor Graphics je okruženje za simulaciju sklopovlja opisanog u jeziku za opisivanje sklopovlja kao što je npr. VHDL, Verilog ili SystemC. Simulacija se provodi pomoću grafičkog korisničkog sučelja ili pomoću TCL (engl. *Tool Command Language*) skripti. Za simulaciju sklopa opisanog u VHDL-u, potrebno je kreirati datoteku za ispitivanje funkcionalnosti opisanog sklopa za *.vhd datoteku koja je vrh hijerarhije opisanog sklopa. ModelSim se najčešće koristi kao prvi korak u ispitivanju RTL dizajna, a za implementaciju dizajna na FPGA integrirani krug potrebno je koristiti složenije programske pakete kao što je Xilinx Vivado koji će biti objašnjen u idućem potpoglavlju [14].

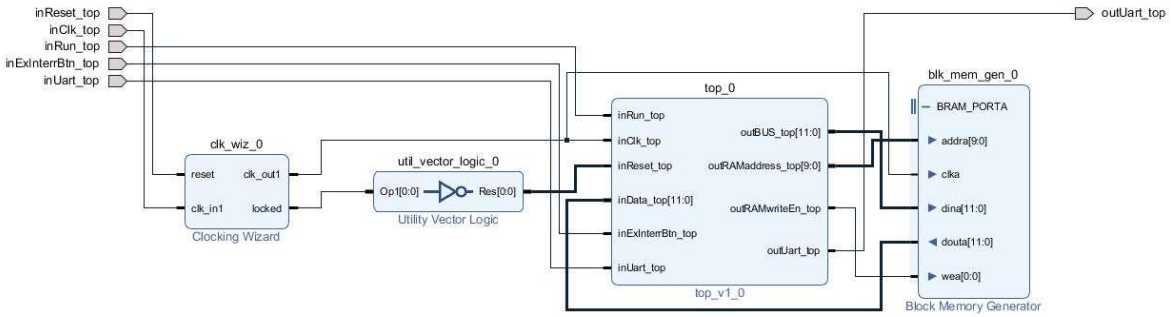
5.2 Xilinx Vivado, simulacija i FPGA implementacija

Vivado je softverski paket za sintezu i analizu sklopovlja dizajniranog u jeziku za opisivanje sklopovlja koji je izradio Xilinx 2012. godine po uzoru na ISE, ali s dodatnim značajkama kao što je razvoj sustava na integriranom krugu i sinteza visoke razine. Korisnicima omogućuje sintetiziranje, vremensku analizu, kreiranje blokovskih dijagrama te spremanje vlastitog projekta u obliku blokovskog elementa, simulaciju nakon izvršene sinteze i implementacije te programiranje

isključivo Xilinx-ovog FPGA integriranog kruga. Jedna od komponenti ovog softverskog paketa je "IP integrator" koji omogućuje kreiranje IP-a. Osnovna ideja IP-a je mogućnost kriptiranja koda kako bi isti bio nevidljiv korisnicima, uz zadržavanje funkcionalnosti dizajna. IP je moguće zapakirati i koristiti unutar blokovskog dijagrama. Na ovaj način olakšano je povezivanje više entiteta u jednu cjelinu [15].

5.2.1 Primjena - implementacija procesora

Nakon opisivanja procesora (pisanja *.vhd datoteka) i uspješne simulacije u ModelSim-u, za implementaciju procesora je potrebno koristiti Vivado softverski paket. Prvo je potrebno kreirati projekt, odabrati razvojnu pločicu na koju želimo implementirati naš dizajn te u projekt dodati *.vhd datoteke izvornih kodova i u simulacijske datoteke dodati "test bench" datoteku. Vivado će automatski otkriti koji entitet je vrh hijerarhije. Zatim je poželjno pokrenuti sintezu kako bi se otkrile moguće pogreške u dizajnu. Nakon uspješne sinteze važno je pokrenuti simulaciju koja se zbog vremenske propagacije signala u kombinacijskim mrežama može razlikovati od one u ModelSim-u koji isto ne uzima u obzir. Vivado će nas upozoriti na loše načine opisivanja sklopovlja pa se uz ostala upozorenja trebamo osloniti i na ova pri otklanjanju pogrešaka. Budući da želimo iskoristiti dio intelektualnog vlasništva Vivado alata kao što je npr. "Clocking Wizard", potrebno je kreirati novi IP i u njega zapakirati naš projekt. Sada se može stvoriti blokovski dizajn i u njega kao blok dodati procesor i ostale potrebne IP-eve te ih međusobno spojiti. Slikom 5.1. prikazan je blokovski dizajn procesora, a dizajnirani procesor predstavljen je blokom naziva "top_0". Svi ostali blokovi su intelektualno vlasništvo Vivado alata. Krajnje lijevo možemo vidjeti ulazne priključnice, a krajnje desno izlazne priključnice. Signali koji dolaze na ulazne priključnice ili odlaze s izlaznih priključnica dolaze izvan integriranog kruga u vidu sklopki, tipkala i PMOD priključaka.



Sl. 5.1. *Blokovski dizajn procesora.*

Kako bi Vivado znao s kojih sklopki ili tipkala signali dolaze ili na koji PMOD priključak ili npr. LED odlazi, potrebno je na odgovarajući način preurediti *.xdc datoteku za odabranu razvojnu pločicu. *.xdc datoteka je datoteka u kojoj korisnik definira koje će fizičke priključke FPGA integriranog kruga koristiti u odnosu s *.vhd opisom sklopa. Slikom 5.2. prikazan je segment koda*.xdc datoteke za signale "inUart_top" i "outUart_top". Isti su spojeni na priključke "V12" i "W16" na PMOD priključku "JE".

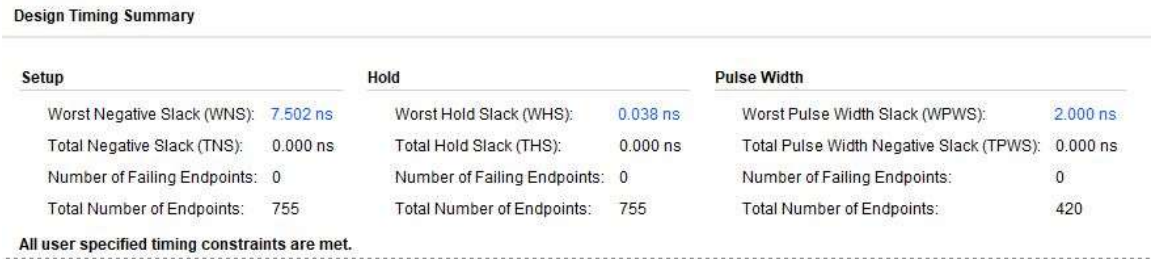
```
294
295 ##Pmod Header JE
296 ##IO_L4P_T0_34
297 set_property PACKAGE_PIN V12 [get_ports {inUart_top}]
298 set_property IOSTANDARD LVCMOS33 [get_ports {inUart_top}]
299
300 ##IO_L18N_T2_34
301 set_property PACKAGE_PIN W16 [get_ports {outUart_top}]
302 set_property IOSTANDARD LVCMOS33 [get_ports {outUart_top}]
303
```

Sl. 5.2. *Pridruživanje signala vanjskom priključku razvojne pločice.*

Sljedeći korak jest stvaranje HDL omotača (entitet – vrh hijerarhije) za blokovski dizajn sa slike 5.1. te pokretanje generiranja datoteke s binarnim zapisom (engl. *Generate Bitstream*) kojime će se FPGA integrirani krug programirati. Automatski će se izvršiti sinteza i implementacija.

Otvorimo li implementirani dizajn, vidjet ćemo rezultate vremenske analize, a Vivado će prijaviti pogrešku ako sustav ne može raditi na zadanoj frekvenciji takta. Ovaj problem, rješava se ubacivanjem registara u složene kombinacijske mreže (engl. *pipelining*) ili smanjivanjem izlazne

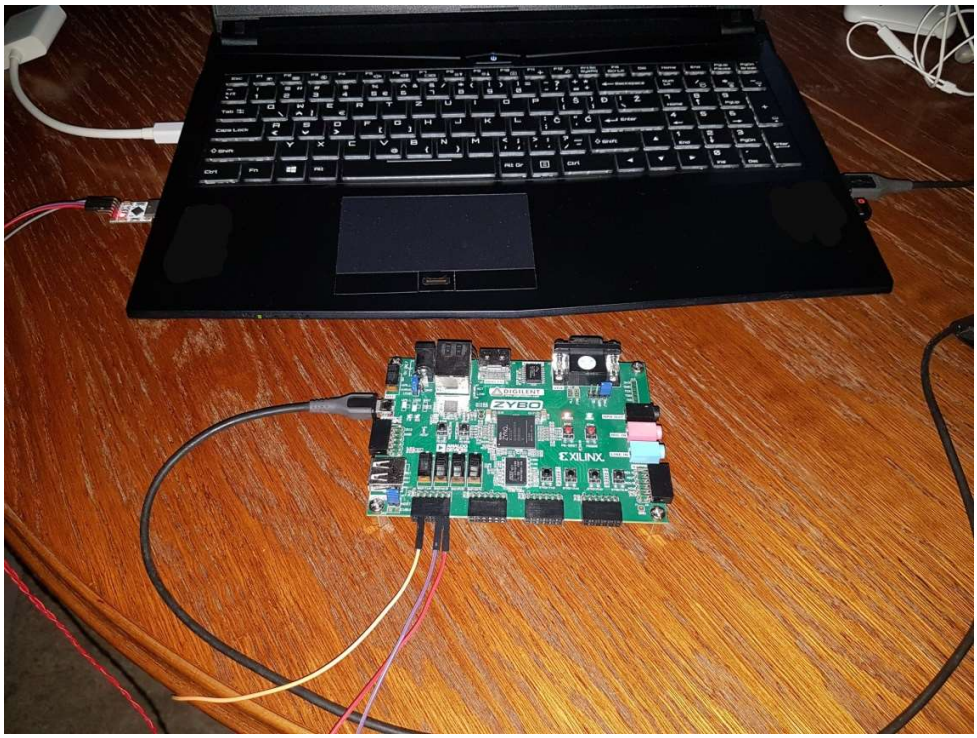
frekvencije u entitetu “Clocking Wizard”. Slika 5.3. prikazuje rezultate vremenske analize. Promotrimo parametar “Worst Negative Slack”. On nam govori koliko još možemo smanjiti period takta (povećati frekvenciju), a da sustav i dalje radi bez greške. Budući da je prije sinteze i implementacije frekvencija takta bila postavljena na 50 MHz (period je 1/50 000 000 = 20 ns), maksimalna frekvencija takta iznosi (20 – 7.502 = 12.498 ns; 1/0.012498 = 80.01 MHz) 80.01 MHz.



Sl. 5.3. *Rezultati vremenske analize.*

Ako postoji zahtjev da procesor radi na frekvenciji većoj od 50 MHz, moguće je tu promjenu učiniti unutar „ClockingWizard“ entiteta pritom pazeći da frekvencija maksimalno može iznositi 80.01 MHz.

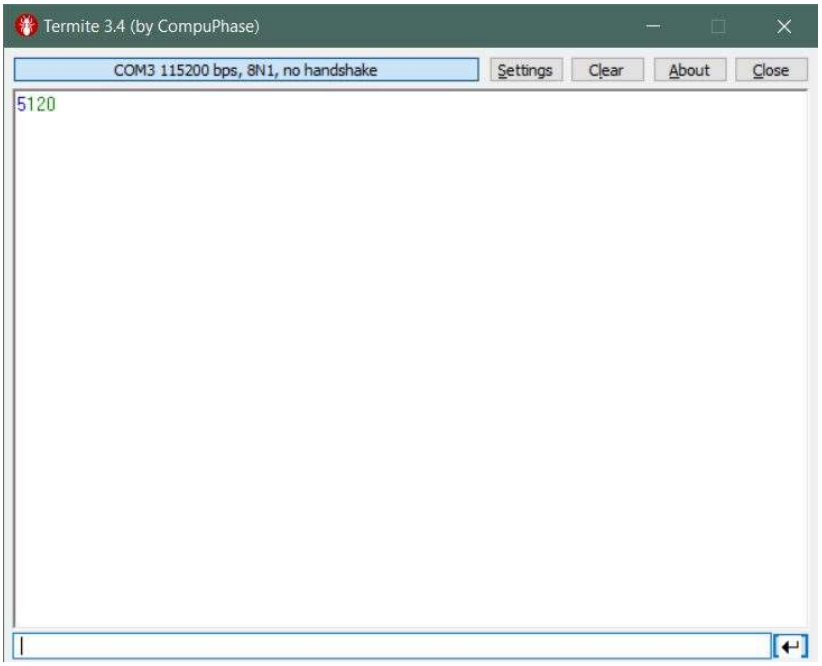
Povezivanjem računala sa ZYBO pločom pomoću USB kabela (omogućuje programiranje ZYNQ-a) i USB-UART adaptera (omogućuje UART komunikaciju između ZYBO-a i računala) moguće je programirati ZYNQ i konačno ispitati točnost sustava. Ovako spojen sustav prikazan je na slici 5.4. Kao primjer u strojnom kodu je izrađen algoritam za računanje faktorijela. Algoritam je osmišljen na način ispita velik broj mogućnosti procesora kao što su: upisivanje u RAM, čitanje iz RAM-a, slanje podataka preko UART-a, primanje podataka preko UART-a, programske podrutine, vremenski prekid, UART prekid, uvjetni skokovi itd.



Sl. 5.4. *Sustav za ispitivanje dizajniranog procesora.*

Algoritam u ovom primjeru računa faktorijel na sljedeći način. Iz nekog od popularnih „terminal“ aplikacija kao što je „Termite“ korišten u ovom primjeru, šalje se jednoznamenkasti broj preko UART-a do procesora implementiranog u ZYBO razvojnoj pločici. Zatim se računa faktorijel pomoću niza instrukcija definiranih unutar procesora te se rezultat dijeli na znamenke i šalje UART-om natrag na računalo. Pri dijeljenju rezultata na znamenke i slanja svake znamenke posebno, potrebno je između slanja znamenki ostvariti vremensku odgodu pomoću vremenskog prekida kako bi se znamenka stigla poslati. Ovako je urađeno jer se instrukcije puno brže izvode nego slanje pojedinih bitova podatka preko UART-a. Zbog 12-bitovne podatkovne sabirnice i jednostavnosti algoritma, moguće je izračunati jedino faktorijel od brojeva čije su vrijednosti manje od 7 zato što je rezultat faktorijela od 7 veći od 4095 ($2^{12} = 4096$). Složenijim algoritmom, tj. preljevom vrijednosti u idući registar, moguće je pohraniti 24-bitovni podatak. Slika 5.5. prikazuje primjer rješavanja faktorijela od 5, tj. rezultat ispitivanja prikazan u računalnoj aplikaciji „Termite“. Na istoj slici, plavom bojom označen je broj koji je poslan s računala, tj. broj od kojeg želimo izračunati faktorijel, a zelenom bojom prikazan je rezultat koji je vratio procesor. Slika 5.6. prikazuje prethodno opisani algoritam napisan u „assembly“ jeziku. Algoritam je prikazan u ovom jeziku zbog lakšeg

razumijevanja te je u isti preveden ručno (bez pisanja prevoditelja iz „assembly“ jezika u strojni kod). Strojni kod algoritma je u ROM zapisan u samom dizajnu u VHDL-u te se isti fizički upisuje u ROM prilikom programiranja FPGA, tj. kada unutar FPGA nastaje procesor.



Sl. 5.5. *Rezultat računanja faktorijela u aplikaciji „Termite“.*

```
1                   SETGIE      1
2                   SETUIE      1
3                   SETTPRESC   00
4                   SETTEN      1
5   start:          LOAD        R6, 0
6   label1:         CMPI        R6, 1
7                   JUMPNZ      label1
8                   CMPI        R7, 0
9                   JUMPZ       nulaIliJedan
10                  CMPI        R7, 1
11                  JUMPZ       nulaIliJedan
12                  MV          R4, R7
13                  MV          R5, R7
14                  SUBI        R5, 1
15                  MV          R0, R5
16                  CALL        MNOZENJE
17                  MV          R4, R3
18                  SUBI        R0, 1
19                  MV          R5, R0
20                  JUMPZ       kraj_slanje
21                  CALL        MNOZENJE
22                  JUMP        MV R4, R3
23  nulaIliJedan:   LOAD        R3, 1
24                  SENDU       R3
25                  JUMP        start
26  kraj_slanje:    LOAD        R7, 0
27                  LOAD        R6, 10
28                  MV          R4, R3
29                  LOAD        R5, 10
30  label2:         CALL        DIJELJENJE
31                  ADDI        R7, 1
32                  STORER      R4, R6
33                  ADDI        R6, 1
34                  MV          R4, R3
35                  CMPI        R3, 0
36                  JUMPNZ      label2
37                  SUBI        R6, 1
38                  LOAD        R1, 0
39  slanje:         MVIR        R0, R6
40                  SENDU       R0
41                  SETTCL      1
42                  SETTIE      1
43  label3:         CMPI        R1, 1
44                  JUMPNZ      label3
45                  SETTIE      0
46                  LOAD        R1, 0
47                  SUBI        R6, 1
48                  CMPI        R6, 9
49                  JUMPNZ      slanje
50                  JUMP        start
51  MNOZENJE:       LOAD        R3, 0
52  label4:         ADD         R3, R4
53                  SUBI        R5, 1
54                  JUMPNZ      label4
55                  RET
56  DIJELJENJE:     LOAD        R3, 0
57  label5:         CMP         R4, R5
58                  RETC
59                  ADDI        R3, 1
60                  SUB         R4, R5
61                  JUMP        label5
62
63  uartISR:        MVU
64                  LOAD        R6, 1
65                  RETI
66
67  timeISR:        LOAD        R1, 1
68                  RETI
```

Sl. 5.6. *Algoritam za računanje faktorijela napisan u „assembly“ jeziku.*

6. ZAKLJUČAK

Cilj ovog rada je opisati procesor temeljen na Harvardskoj arhitekturi u VHDL jeziku za opisivanje sklopovlja te isti implementirati na FPGA integrirani krug. Postojale su dvije moguće arhitekture procesora koje su se mogle iskoristiti, a to su Von Neumannova i Harvardska. Iako je inače zastupljenija Von Neumannova, ovdje je korištena Harvardska jer se korištenjem odvojenih RAM i ROM memorija i pripadajućih sabirnica omogućuje brži rad procesora.

Jezik za opisivanje sklopovlja korišten u ovome radu je VHDL uz koji još postoje i Verilog, SystemC itd. U ovome radu korištena je i ZYBO razvojna pločica temeljena na ZYNQ 7000 integriranom krugu. Štoviše, ovaj integrirani krug posjeduje i dvojezgreni ARM procesor i široki spektar periferije. Uz sve ove specifikacije, ZYBO je jedan od trenutno najboljih dostupnih odabira.

Jedan od najvažnijih i najsloženijih riješenih problema u ovome radu opisivanje je procesora u VHDL-u. Radi složenosti samog procesora, svaki od pojedinih dijelova kao što su aritmetičko-logička jedinica, upravljačka jedinica ili instrukcijski registar opisani su u pojedinim entitetima, tj. pojedinim *.vhd datotekama. Iste su povezane u entitetima više hijerarhije kao što je jezgra ili konačno najviši entitet koji predstavlja cijeli procesor. Ovim pristupom RTL dizajniranja omogućeno je lakše pronalaženje i ispravljanje grešaka te snalaženje u kodu.

Procesor dizajniran u VHDL-u bilo je potrebno simulirati u okruženju za računalnu HDL simulaciju kao što je ModelSim. Nakon uspješne simulacije potrebno je na trenutnu razinu sustava dodati RAM memoriju i sklop za manipulaciju frekvencijom signala takta u vidu intelektualnog vlasništva softverskog paketa Vivado. Pomoću ovog softverskog paketa, odrađena je sinteza, implementacija te generiranje datoteke s binarnim zapisom i programiranje ZYNQ-a istom. Za zadnje ispitivanje točnosti rada procesora pomoću UART protokola poslan je s računala jednoznamenkasti broj od kojega je procesor pomoću algoritma u ROM memoriji izračunao faktorijel te isti poslao na računalo kao rezultat.

Procesor realiziran u radu moguće je proširiti te koristiti kao i neki drugi iz npr. AVR tvrtke. Rezultati postignuti ovim procesorom su zadovoljavajući te je maksimalna frekvencija signala takta u usporedbi s ostalima ove vrste približno jednaka. Proširenjem periferije procesora moglo bi biti omogućeno njegovo korištenje u stvarnom svijetu kao npr. upravljanje CNC strojevima primanjem

uputa preko UART sučelja. Također bi se moglo omogućiti programiranje istog u C programskom jeziku, pisanjem prevoditelja (engl. *compiler*). Nedostatci ovog procesora su manjak periferije, nemogućnost programiranja iz višeg programskog jezika, nedostatak instrukcija važnih za izvođenje složenih operacija itd. Uz sve nedostatke koje ovaj procesor ima, može se reći da ima dovoljan broj instrukcija da riješi velik broj problema i time zadovolji velik broj korisnika.

LITERATURA

[1] J.L.Hennessy; D. A. Patterson, Computer Architecture: A Quantitative Approach, Elsevier Inc., Cambridge, 2017.

[2] C. G. Bell; R. Caddy; H. McFarland; J. O'Laughlin; R. Noonan; W. Wulf, A new Architecture for Mini-Computers-The DEC PDP-11, Spring Joint Computer Conference, stranice 657-675, 1970

[3] Harvardska arhitektura, https://tdck.weebly.com/uploads/7/7/0/5/77052163/03_-_harvard_architecture_comparison.pdf, pristupljeno 15.08.2019.

[4] M. Čupić, Kratki uvod u jezik VHDL, 2015.

[5] ZYBO, https://reference.digilentinc.com/reference/programmable-logic/zybo/start, pristupljeno 18.08.2019.

[6] Upravljačka jedinica, https://www.geeksforgeeks.org/introduction-of-control-unit-and-its-design/, pristupljeno 18.08.2019.

[7] Aritmetičko-logička jedinica, http://www.righto.com/2013/09/the-z-80-has-4-bit-alu-heres-how-it.html, pristupljeno 19.08.2019.

[8] Registri, https://www.javatpoint.com/computer-registers, pristupljeno 19.08.2019.

[9] D. M. Harris; S. L. Harris, Digital Design and Computer Architecture, Elsevier Inc., San Francisco, 2007.

[10] Slika multipleksera, https://www.slideshare.net/na491/1-multiplexer, pristupljeno 23.08.2019.

[11] Programski brojač,https://avr-tutorials.com/general/avr-program-counter, pristupljeno 23.08.2019.

[12] Statusni registar, https://microchipdeveloper.com/8avr:status, pristupljeno 25.08.2019.

[13] Ulazni i izlazni sklopovi opće namjene - ZYBO, https://reference.digilentinc.com/reference/programmable-logic/zybo/reference-manual, pristupljeno 25.08.2019.

[14] ModelSim, https://www.mentor.com/products/fv/modelsim/, pristupljeno 26.08.2019.

[15] Xilinx Vivado, https://www.xilinx.com/products/design-tools/vivado.html#overview, pristupljeno 28.08.2019.

SAŽETAK

Cilj ovog završnog rada bio je implementirati 12-bitovni procesor temeljen na Harvardskoj arhitekturi na FPGA integrirani krug koristeći VHDL jezik za opisivanje sklopovlja, ModelSim okruženje za računalnu simulaciju, Vivado okruženje za sintezu i analizu opisanog sklopa i ZYBO razvojnu pločicu za fizičko ispitivanje valjanosti rješenja. U radu je prvo predstavljen pojam arhitekture računala kao temelj za dizajniranje procesora. Zatim je opisan VHDL i način na koji se sklopovlje opisuje u istome te ZYBO razvojna pločica i mogućnosti koje ista pruža kod fizičkog ispitivanja valjanosti rješenja. Nakon toga je opisano dizajniranje svakog dijela procesora te njegova funkcija i komunikacija s ostalim dijelovima procesora. U istom poglavlju opisana je i periferija procesora. Na kraju rada predstavljeni su alati ModelSim i Vivado koji su korišteni za verifikaciju rješenja. Dizajnirani procesor može raditi na maksimalnoj frekvenciji takta koja iznosi 80.01 MHz. Komunikacija s vanjskim uređajima odvija se pomoću UART protokola. Procesor nema mogućnost programiranja iz nekog od viših programskih jezika kao što je „C“ već se programira u strojnom kodu. Ispravnost rješenja dokazana je spajanjem ZYBO pločice na računalo preko UART sučelja i slanjem jednoznamenkastog broja s računala od kojega je procesor izračunao faktorijel i isti poslao na računalo.

Ključne riječi: VHDL, FPGA, procesor, Harvard, ZYNQ, ZYBO

ABSTRACT

FPGA implementation of 12-bit processor based on Harvard architecture

The goal of this paper was to implement a 12-bit processor based on Harvard architecture on an FPGA integrated circuit using VHDL hardware description language, ModelSim environment for computer simulation, Vivado environment for synthesis and analysis of the described hardware and ZYBO development board for testing validity of the solution. Firstly, term computer architecture was introduced as a foundation for designing a processor. Secondly, the term VHDL and the way hardware is described in it were mentioned. After that, ZYBO development board and features it provides when testing validity of the solution were described. After describing ZYBO, the process of designing every part of processor, its function and communication with other parts of processor were described. In the same chapter, periphery of the processor was also described. By the end of the paper, ModelSim and Vivado were introduced for verification of the solution. The designed processor can work on a maximum frequency of 80.01 MHz. Communication with external devices is realized with UART protocol. The processor does not have the possibility to be programmed with high programming languages like “C”, but only with machine code. The correctness of the solution was proven by connecting the ZYBO board to the computer by UART interface and by sending a single-digit number from a computer to the processor. The processor calculated factorial of the number and sent it back to the computer.

Keywords: VHDL, FPGA, processor, Harvard, ZYNQ, ZYBO

ŽIVOTOPIS

Matej Štajnbrikner rođen je 08.05.1997. godine u Osijeku. Osnovnu školu završio je u Petrijevcima, a srednju u Elektrotehničkoj i prometnoj školi Osijek, smjer elektrotehničar. Tijekom srednjoškolskog obrazovanja sudjelovao je na natjecanjima iz fizike i matematike te završavao razrede s odličnim uspjehom. Trenutno je redoviti student 3.godine preddiplomskog studija računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek. Govornik je engleskog jezika. Od 01.12.2018. pohađao je FPGA školu u Institutu RT-RK Osijek te je istu uspješno završio.

Matej Štajnbrikner

_____

PRILOZI

P1 Izvorni kod u VHDL-u, nalazi se na CD-u

P2 Video snimak ispitivanja rada procesora, nalazi se na CD-u

P3 Operacije procesora

Tab. P3.1 Operacije manipulacije podacima između registara, RAM-a i periferije

NAZIV U DIZAJNU	INSTRUKCIJSKI KOD	ZADAĆA OPERACIJE
mv	000000	kopiranje podatka iz jednog registra u drugi
mvi	000001	spremanje podatka iz RAM-a s zadane adrese u zadani registar
mvu	010111	spremanje podatka koji je stigao preko UART-a u R7
store	011000	spremanje podatka iz zadanog registra na zadanu adresu u RAM-u
load	011110	spremanje podatka zadanog u instrukciji u zadani registar
sendu	100011	slanje podatka iz zadanog registra preko UART-a
storer	101110	spremanje podatka iz zadanog registra na adresu u RAM-u spremljenu u drugom registru
mvir	101111	spremanje podatka iz RAM-a s adrese spremljene u jednom registru u neki drugi registar

Tab. P3.2 Matematičke operacije

NAZIV U DIZAJNU	INSTRUKCIJSKI KOD	ZADAĆA OPERACIJE
add	000010	Spremanje rezultata zbrajanja podataka iz dva registra u prvi navedeni u instrukciji
sub	000011	Spremanje rezultata oduzimanja podataka iz dva registra u prvi navedeni u instrukciji
cmp	011111	Oduzimanje podataka iz dva registra; rezultat se ne sprema, ali utječe na zastavice u statusnom registru kao i ostale operacije; uspoređivanje podataka
addi	100000	Spremanje rezultata zbrajanja podatka iz registra i podatka iz instrukcije u isti registar
subi	100001	Spremanje rezultata oduzimanja podatka iz registra i podatka iz instrukcije u isti registar
cmpi	100010	Oduzimanje podatka iz jednog registra i podatka iz instrukcije; rezultat se ne sprema, ali utječe na zastavice u statusnom registru kao i ostale operacije; uspoređivanje podataka

Tab. P3.3 Logičke operacije

NAZIV U DIZAJNU	INSTRUKCIJSKI KOD	ZADAĆA OPERACIJE
shl	000100	pomjeranje podatka iz zadanog registra u lijevo za jedno mjesto
shr	000101	pomjeranje podatka iz zadanog registra u desno za jedno mjesto
myand	000110	spremanje rezultata I logičke operacije nad podacima iz dva registra u prvi
myor	000111	spremanje rezultata ILI logičke operacije nad podacima iz dva registra u prvi
mynot	001000	spremanje rezultata NE logičke operacije nad podatkom iz jednog registra u isti
Mynand	001001	spremanje rezultata NI logičke operacije nad podacima iz dva registra u prvi
Mynor	001010	spremanje rezultata NILI logičke operacije nad podacima iz dva registra u prvi
Myxor	001011	spremanje rezultata EXILI logičke operacije nad podacima iz dva registra u prvi
myxnor	001100	spremanje rezultata EXNILI logičke operacije nad podacima iz dva registra u prvi

Tab. P3.4 Operacije vezane uz prekide

NAZIV U DIZAJNU	INSTRUKCIJSKI KOD	ZADAĆA OPERACIJE
reti	001111	Vraća program iz prekidne rutine u glavnu nit programa uzimajući zadnju adresu sa stoga adresa programskog brojača
setgie	010000	postavlja zastavicu globalne dozvole prekida u zadano logičko stanje
setexie	010001	Postavlja zastavicu dozvole vanjskog prekida u zadano logičko stanje
settie	010010	Postavlja zastavicu vremenske dozvole prekida u zadano logičko stanje
setuie	010011	Postavlja zastavicu dozvole UART prekida u zadano logičko stanje
setten	010100	Postavlja signal dozvole brojanja vanjskog brojača u zadano logičko stanje
settcl	010101	Postavlja stanje vanjskog brojača na nulu
settpresc	010110	Postavlja djelitelj takta brojača na zadanu vrijednost; “00” - 0, “01” – 255, “10” – 511, “11” - 1023

Tab. P3.5 Bezuvjetni i uvjetni skokovi

NAZIV U DIZAJNU	INSTRUKCIJSKI KOD	ZADAĆA OPERACIJE
jump	011001	bezuvjetni skok na zadanu adresu programskog ROM-a
jumpz	011010	skok na zadanu adresu programskog ROM-a ako je zastavica nule u logičkom stanju ‘1’
jumpnz	011011	skok na zadanu adresu programskog ROM-a ako je zastavica nule u logičkom stanju ‘0’
jumpc	011100	skok na zadanu adresu programskog ROM-a ako je zastavica prijenosa u logičkom stanju ‘1’
jumpnc	011101	skok na zadanu adresu programskog ROM-a ako je zastavica prijenosa u logičkom stanju ‘0’
jumpn	100100	skok na zadanu adresu programskog ROM-a ako je zastavica predznaka u logičkom stanju ‘1’; ako je rezultat negativan
jumpnn	100101	skok na zadanu adresu programskog ROM-a ako je zastavica predznaka u logičkom stanju ‘0’; ako je rezultat pozitivan

Tab. P3.6 Operacije vezane uz podrutine

NAZIV U DIZAJNU	INSTRUKCIJSKI KOD	ZADAĆA OPERACIJE
call	100110	poziv podrutine sa zadane adrese programskog ROM-a
ret	100111	Bezuvjetno vraćanje iz podrutine u glavnu nit programa uzimajući zadnju adresu sa stoga adresa programskog brojača
retz	101000	Vraćanje iz podrutine u glavnu nit programa na isti način kao *ret*, ali ako je zastavica nule u logičkom stanju '1'
retnz	101001	Vraćanje iz podrutine u glavnu nit programa na isti način kao *ret*, ali ako je zastavica nule u logičkom stanju '0'
retc	101010	Vraćanje iz podrutine u glavnu nit programa na isti način kao *ret*, ali ako je zastavica prijenosa u logičkom stanju '1'
retnc	101011	Vraćanje iz podrutine u glavnu nit programa na isti način kao *ret*, ali ako je zastavica prijenosa u logičkom stanju '0'
retn	101100	Vraćanje iz podrutine u glavnu nit programa na isti način kao *ret*, ali ako je zastavica predznaka u logičkom stanju '1'
retnn	101101	Vraćanje iz podrutine u glavnu nit programa na isti način kao *ret*, ali ako je zastavica predznaka u logičkom stanju '0'

Tab. P3.7 Operacije raznih primjena

NAZIV U DIZAJNU	INSTRUKCIJSKI KOD	ZADAĆA OPERACIJE
sleep	001101	Procesor odlazi u “spavanje”; stanja AKS-a se ne mijenjaju; iz ovog načina rada ponovni rad procesora može pokrenuti jedino prekid
nop	001110	(engl. *no operation*) ova operacija doslovno ne radi ništa, ali se ponekad koristi za rješavanje vremenskih problema u radu procesora kao što je npr. upisivanje podataka u memoriju