UNIVERZITET U NIŠU
ELEKTRONSKI FAKULTET
KATEDRA ZA ELEKTRONIKU

JEDNOSTAVNI 16-BITNI PROCESOR SA PODRŠKOM ZA OBRADU INTERAPTA

Student: Jocić Ivan
Br.indeksa: 9659

Niš, septembar 2006.

Sadržaj:

1. Uvod 3

2. Prekidna U/I tenika 4

3. Mehanizam prekida 6
   3.1. Opsluživanje zahteva za prekidom i povratak iz prekidne rutine 7
   4. Opis strukture procesora – opšti pristup 8
   4.1. Izvršna jedinica – execute unit 9
   4.2. Upravljačka jedinica – control path 10
   4.3. *Interrupt* prekidnih rutina 12
   4.4. Struktura hardvera izvršne jedinice 12
   4.5. Skup instrukcija CPU-a 14

5. Realizacija u VHDL-u 15
   5.1. Opis funkcije Interrupt_control logike 18
   5.2. Opis kontrolnih signala bloka Interrupt_control 19
   5.3. Listing koda bloka driver 20
   5.4. Listing koda bloka PC_temp 20
   5.5. Listing koda bloka PSW 21
   5.6. Listing koda bloka ICU 22
   5.7. Listing koda bloka Interrupt_control 24

6. Rezultati simulacije 33

7. Sinteza i implementacija 35

8. Laboratorijska vežba 41
   8.1. Zadatak 44

9. Zaključak 49

10. Literatura 50

1. Uvod

Pod U/I operacijom podrazumevamo prenos podataka između U/I uređaja i *CPU-a.* Ako su U/I operacije u potpunosti kontrolisane od strane CPU-a, tj. CPU izvršava programe pomoću kojih se iniciraju, usmeravaju i završavaju U/I operacije za računar kažemo da se koristi programirani U/I prenos. Ovaj prenos se implementira ugradnjom veoma malog specijalnog U/I hardvera, ali ima za posledicu da CPU troši veliki deo svog vremena obavljajući relativno jednostavne U/I upravljačke funkcije.

Kod programiranog U/I prenosa vrši se razmena podataka između CPU-a i U/I interfejsa. CPU izvršava program koji upravlja U/I operacijom, kao što je čitanje statusa spoljnjeg uređaja, izdaje komande READ ili WRITE i vrši prenos podataka. Kada procesor izda komandu U/I uređaju on zatim čeka sve dok se U/I operacija ne završi. Kako je CPU brži od U/I interfejsa, evidentno je da procesor suviše mnogo vremena gubi na testiranje spremnosti za prenos.

U/I uređaj ili njegov kontroler (U/I interfejs) mogu imati ugrađen specijalan hardver pomoću koga se zahteva opsluživanje od strane CPU-a, tj. izvršenje specifičnog programa radi opsluživanja U/I uređaja. Ovaj tip zahteva se prekid, a tehnika prenosa je poznata kao prekidni U/I prenos. Mogućnost prekidanja oslobađa CPU-a od obavaljanja onih zadataka koji se odnose na periodično testiranje statusa U/I uređaja. Kod prekidnog U/I prenosa, CPU izda U/I komandu, zatim produžava da izvršava druge instrukcije, a prekida se od strane U/I interfejsa kada je interfejs spreman da primi ili preda podatke. Nakon opsluživanja prekida CPU može da nastavi sa izvršavanjem prekidnog programa.

Obe tehnike, programirani i prekidni U/I, se izvršavaju pod programskom kontrolom, tj. CPU čita podatke iz memorije i upisuje ih u U/I interfejs u toku operacije OUTPUT, a čita podatke iz U/I interfejsa i upisuje ih u memoriju u toku operacije INPUT.

Ako se u U/I sistem (podsistem) ugradi specijalan hardver tada se U/I uređaju može obezbediti mogućnost da vrši direktan prenos blokova informacije ka ili iz glavne memorije bez intervencije CPU-a. Ovo zahteva od U/I uređaja (ili njegovog kontrolera) da bude sposoban da generiše memorijske adrese i vrši prenos podataka ka ili iz sistemske magistrale, tj. da bude gospodar magistrale. CPU je odgovoran na nivou iniciranja prenosa blokova podataka. Kontroler U/I uređaja može obavaljati prenos bez intervencije CPU-a. Interakcija u radu između CPU-a i U/I kontrolera postoji samo kada CPU treba da preda pravo upravljanja nad sistemskom magistralom U/I kontroleru. Nakon ovoga U/I interfejs i glavna memorija direktno razmenjuju podatke bez posredstva CPU-a. Ovaj tip U/I prenosa podataka se zove direktni pristup memoriji (Direct Memory Access - DMA).

Veliki broj današnjih mikroračunara ima izvedene mogućnosti za realizaciju prekidne i DMA tehnike prenosa.

2. Prekidna U/I tehnika

Programirani *U/I* prenos ima sledeća dva glavna nedostatka:

(a) Brzina prenosa je ograničena brzinom sa kojom *CPU* može da testira i opslužuje *U/I* uređaje.

(b) Vreme koje *CPU* potroši na testiranje statusa *U/I* uređaja kao i vreme koje je potrebno da se obavi prenos podataka često je moguće efikasnije iskoristiti.

Uticaj *CPU-a* na brzinu prenosa podataka je dvostruki. Kao prvo, javlja se kašnjenje dok U/I uređaj koji zahteva opsluživanje čeka da bude testiran od strane CPU-a. Ako u sistemu postoji veći broj U/I uređaja, tada će se svaki uređaj testirati relativno retko. Kao drugo, programiranom U/I tehnikom vrši se prenos podataka preko *CPU-a,* tj. podaci se ne prenose direktno iz glavne memorije ka *U/I* uređaju i obrnuto. Alternativne tehnike koje, u odnosu na programirani *U/I,* obezbeduju brži prenos su *prekidna* i *DMA.*

Osnovni mehanizam koji se koristi kod prekidne *U/I* je isti kao i kod programirane *U/I.* To znači da *CPU* još razmenjuje podatke sa *U/I* uređajem preko "svojih registara. Drugim rečima, sastavni deo *U/I* interfejsa su i dalje portovi za podatke, upravljanje i statusni. Razlika se sastoji u tome ko je odgovoran za iniciranje prenosa. Kod programiranog *U/I* prenosa ta odgovornost je prepuštena *CPU-u.* Drugim rečima, *U/I* program mora često da analizira status *U/I* uređaja sa ciljem da odredi da li je uređaj spreman da obavi prenos. Kod prekidnog *U/I-a* prenos podataka se inicira od strane *U/I* uređaja, koji koristi prekidni mehanizam da ukaže *CPU-u* na svoju spremnost. Na ovaj način ne postoji više potreba za permanentnim testiranjem statusa *U/I* uređaja. *U/I* uređaj može takođe koristiti mehanizam prekida kada želi da *CPU* obrati njemu pažnju iz drugih razloga, kao što su: očekuje se (javila se) određena greška, ukazuje na/ završetak lokalne operacije i dr.

Dva osnovna problema se javljaju kod projektovanja prekidne *U/I* tehnike. Prvi se odnosi na način kako *CPU* određuje koji je uređaj zahtevao prekid, a drugi, ako se veći broj prekida javi istovremeno kom će zahtevu *CPU* najpre posvetiti pažnju i obraditi ga. Četiri opšte poznate kategorije izvođenja se koriste za rešavanje ovih problema, a to su:

1. *veći broj prekidnih linija* (Multiple Interrupt Lines)
2. *softversko kružno ispitivanje* (Software Poli)
3. *lančanje* (Daisy Chain - hardware poli, vectored)
4. *magistralna arbitraža* (Bus Arbitration, vectored)

1. Veći broj prekidnih linija

Najjednostavniji način da se reši problem prihvatanja zahteva za prekid je da se obezbedi veći broj prekidnih linija između CPU-a i U/I interfejsa. Svakom zahtevu za prekid može se dodeliti jedinstveni prioritet. Izvor prekida je odmah poznat *CPU-u,* a to ukazuje da ne postoji potreba za hardverskom i softverskom analizom U/I portova. Početna adresa rutine za obradu prekida može biti fiksna (na jedinstven način je određena zahtevom *INTRi)* ili se određuje softverskom proverom nakon prihvatanja zahteva za prekid.

Na žalost, ovo rešenje je nepraktično jer zahteva veliki broj veza. I u slučaju kada se koristi veći broj linija, obično se na svaku liniju povezuje veći broj *U/I* interfejsa pa se tada

način određivanja izvora prekida vrši na osnovu jedne od drugih poznatih kategorija izvođenja.

2. Softversko kružno ispitivanje

Predstavlja jednostavan način za usluživanje prekida. Na sistem je preko odgovarajućih U/I interfejsa povezan veći broj U/I uređaja. Kada U/I uređaj i želi da preda (primi) podatak on aktivira liniju INTRi. U/I interfejs i prihvata taj zahtev i kao odgovor aktivira odgovarajuću liniju INT. Svi zahtevi za prekid INTi i=l,..., n dovode se na ulaz OR kola koje na svom izlazu generiše jedinstveni zahtev INT. Jedan alternativni način organizovanja ove metode je jedinstvena *Interrupt Request* linija je deljiva za sve U/I portove. Kao odziv na prekid CPU mora da analizira sve U/I uređaje kako bi odredio izvor prekida.

3. Lančanje i vektorsko prekidanje

Nedostatak organizacije sa prozivanjem je taj što vremenski dugo traje. Efikasnija tehnika je korišćenje *lančanja,* koja u suštini obezbeđuje hardversku prozivku.
Linija *INTA* je ulančana preko interfejsa. CPU se odaziva na prekid generisanjem signala *INTA.* Signal se prostire kroz serijsku vezu *U/I interfejsa* (lanac) sve dok ne naiđe na interfejs koji je zahtevao prekid. Interfejs koji je zahtevao prekid obično se odaziva postavljanjem reči na magistrali podataka. Ova reč se zove *vektor* i obično predstavlja adresu *U/I* interfejsa ili neki drugi jedinstveni identifikator. U oba slučaja *CPU* koristi vektor kao pokazivač na odgovarajuću uslužnu rutinu za dati *U/I* interfejs. Na ovaj način se izbegava potreba da se izvrši opšta prekidno-uslužna rutina kao prvi korak u opsluživanju. Ova tehnika je takođe poznata i kao *vektorski prekid* (Vectored interrupt).

Ako dva *U/I* interfejsa istovremeno izdaju zahtev za prekid, prvo će se opslužiti onaj koji je bliži mikroprocesoru:

Prednosti lančanja su:

1. identifikacija se izvodi na jedinstveni način,
2. ne zahteva se dodatni hardver,
3. lako se proširuje i preuređuje hardver.

Nedostaci lančanja su:

1. može se proširivati i preuređivati samo hardverski,
2. prioritet opsluživanja određen je fizičkom pozicijom,
3. zahteva se dodatno vreme zbog prostiranja signala *INTA* kroz lanac, što ima za posledicu da uspori rad mikroprocesora ili da se zbog konačnog vremena
4. odziva ograniči broj U/I interfejsa koji se mogu vezivati u lanac.

CPU koristi kod koga primi od *U/I* uređaja koji je generisao zahtev za prekid kao indirektnu specifikaciju početne adrese rutine za obradu prekida. To znači da taj kod ukazuje na adresu me-morijske lokacije u kojoj se čuva željena početna adresa rutine za obradu prekida. Sadržaj ove lokacije, koji se postavlja kao nova vrednost u *PC,* poznat je kao *vektor prekida* (interrupt vector). Kod nekih mašina vektor prekida pored vrednosti za *PC* sadrži i novu vrednost za registar *PSW.*

Da bi se podržao mehanizam vektorskog prekida neophodno je izvesti određene hardverske modifikacije. Ključna modifikacija je sledeća: *CPU* se ne odaziva trenutno kada primi zahtev za prekid. Minimalno kašnjenje *CPU-ovog* odziva posledica je činjenice da se izvršenje tekuće instrukcije mora završiti. Dalje do kašnjenja dolazi ako je prekid zabranjen u trenutku kada se primi zahtev. Pošto *CPU* može da zahteva korišćenje magistrale u toku

perioda kašnjenja, uređaju koji je zahtevao prekid neće biti dozvoljeno da postavi podatke na magistralu sve dok *CPU* nije spreman da primi taj podatak. Potrebna koordinacija u radu može se ostvariti uvođenjem još jednog upravljačkog signala poznat kao *INTA* (Interrupt Acknowladge). U trenutku kada je *CPU* spreman da opsluži prekid on aktivira liniju *INTA*.

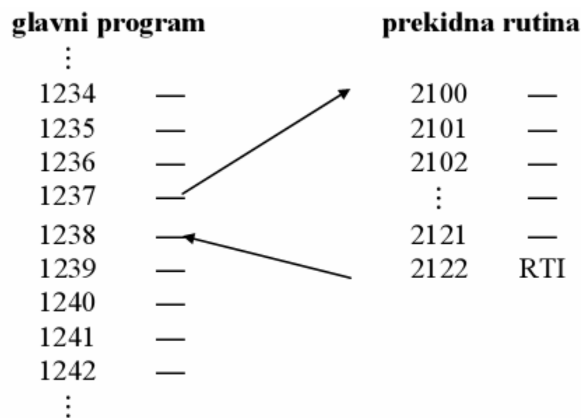
4. Magistralna arbitraža

Ovaj metod takođe koristi vektorski prekid. Naime, *U/I* interfejs mora prvo da stekne pravo upravljanja nad magistralom pre nego što aktivira liniju *INTR*. U jednom trenutku samo jedan *U/I* interfejs može da aktivira *INTR* liniju. Kada *CPU* detektuje prekid on se odaziva signalom *INTA*. *U/I* interfejs zatim postavlja svoj vektor na linijama za podatke.

3. Mehanizam prekida

Mehanizam prekida kod procesora omogućuje prekid u izvršavanju tekućeg programa, koji će se nazivati glavni program, i skok na novi program, koji će se nazivati prekidna rutina. Poslednja instrukcija u prekidnoj rutini je instrukcija RTI. Ona omogućuje povratak u glavni program. Izvršavanje glavnog programa se produžava sa onog mesta gde je bilo prekinuto. Može se uzeti da zahtev za prekid stiže u toku izvršavanja neke od instrukcija. Zahtev za prekidom generiše se od strane nekog spoljašnjeg U/I uređaja. Zbog toga se mehanizam prekida obično tako realizuje da se instrukcija u toku čijeg je izvršavanja stigao zahtev za prekid, najpre izvrši do kraja, pa se tek onda prihvata zahtev za prekid i skače na prvu instrukciju prekidne rutine. Izuzetak od ovoga predstavljaju instrukcija nad nizovima alfanumeričkih znakova. Kod ovih instrukcija se zahtev za prekid prihvata u prvom pogodnom trenutku, koji može da nastupi i pre trenutka u kom je instrukcija izvršena do kraja.

Efekti mehanizma prekida i instrukcije RTI na izvršavanje glavnog programa i prekidne rutine su prikazani na slici 1. Uzeto je da u toku izvršavanja instrukcije glavnog programa sa adrese 1237 stiže zahtev za prekid. Ova instrukcija se se najpre izvrši do kraja. Potom procesor produžava sa izvršavanjem instrukcija sa adrese 2100 na kojoj se nalazi prva instrukcija prekidne rutine umesto sa adrese 1238 na kojoj se nalazi prva sledeća instrukcija glavnog programa. Instrukcijom RTI sa adrese 2122 se obezbeđuje da procesor kao sledeću izvršava instrukciju glavnog programa sa adrese 1238. To je instrukcija glavnog programa koja bi se normalno i izvršavala posle instrukcije sa adrese 1237 da u toku njenog izvršavanja nije stigao zahtev za prekid.



Slika 1. Prekid i povratak iz prekidne rutine

Aktivnosti u procesoru kojima se prekida izvršavanje glavnog programa i skače na prekidnu rutinu nazivaju se opsluživanje zahteva za prekid, a aktivnosti kojima se obezbeđuje povratak iz prekidne rutine u glavni program i produžavanje izvršavanja glavnog programa sa mesta i pod uslovima koji su bili pre skoka na prekidnu rutinu nazivaju se povratak iz prekidne rutine.

Zahteve za prekid mogu da generišu:

1. kontroleri periferija da bi procesoru signalizirali spremnost za prenos podataka (maskiraju i prekidi),
2. uređaji računara koji kontrolišu ispravnost napona napajanja, transfera na magistrali, rada memorije itd. (nemaskiraju i prekidi),
3. procesor, kao rezultat otkrivene nekorektnosti u izvršavanju tekuće instrukcije (nelegalan kod operacije, nelegalno adresiranje, greška prilikom deljenja, itd.),
4. procesor, ako je zadat takav režim rada procesora, kroz postavljanje bita T u programskoj statusnoj reči PSW, da se posle svake instrukcije skače na određenu prekidnu rutinu i
5. procesor kao rezultat izvršavanja instrukcije prekida INT.

Prekidi pod 1 i 2 se nazivaju spoljašnji, a pod 3 , 4 i 5 unutrašnji.

3.1. Opsluživanje zahteva za prekidom i povratak iz prekidne rutine

Opsluživanje zahteva za prekid se realizuje delom hardverski i delom softverski, a povratak iz prekidne rutine softverski. Hardverska realizacija dela opsluživanja zahteva za prekid zna i da se izvršavanje instrukcije u kojoj se javlja neki zahtev za prekid produžava za onoliko koraka koliko je potrebno da se taj deo realizuje. Softverska realizacija dela opsluživanja zahteva za prekid i povratka iz prekidne rutine se realizuju izvršavanjem odgovarajućih instrukcija procesora.

Opsluživanje zahteva za prekid se sastoji iz:

- čuvanja konteksta procesora i
- utvrđivanja adrese prekidne rutine

Programski brojač PC i programska statusna reč PSW se čuvaju hardverski. Preostali programski dostupni registri se čuvaju hardverski kod onih procesora kod kojih

broj ovih registara nije veliki i softverski sa nekoliko instrukcija na početku prekidne rutine kod onih procesora kod kojih je broj ovih registara veliki.

U okviru opsluživanja zahteva za prekid hardverski se još:

- brišu biti maskiranje svih maskiraju ih prekida i prekid posle svake instrukcije u programskoj statusnoj reči procesora PSW kod prekida svih vrsta i

- upisuje u bite tekući nivo prioriteta u statusnoj reči procesora PSW nivo prioriteta prekidne rutine na koju se skače u slučaju maskirajućeg prekida.

Brisanjem bita maskiranje svih maskirajućih prekida u programskoj statusnoj reči procesora PSW se obezbeđuje da procesor po ulasku u prekidnu rutinu ne reaguje na maskirajuće prekide, a brisanjem bita prekid posle svake instrukcije u programskoj statusnoj reči procesora PSW se obezbeđuje da procesor po ulasku u prekidnu rutinu ne izvršava prekidnu rutinu u režimu prekid posle svake instrukcije. Time se omogućuje obavljanje određenih aktivnosti na početku svake prekidne rutine. Posle toga moguće je u samoj prekidnoj rutini posebnim instrukcijama postaviti bit maskiranje svih maskirajućih prekida u programskoj statusnoj reči procesora PSW i time dozvoliti maskirajuće prekide i postaviti bit prekid posle svake instrukcije u programskoj statusnoj reči procesora PSW i time zadati režim rada procesora prekid posle svake instrukcije. Nakon prihvatanja zahteva za prekidom, prihvata se i adresa instrukcije na koju će procesor skočiti kako bi obradio prekid.

Povratak iz prekidne rutine se realizuje tako što se, najpre, posebnim instrukcijama pri kraju prekidne rutine restauriraju vrednostima sa steka sadržaji onih preostalih programski dostupnih registara čije su vrednosti posebnim instrukcijama sačuvane na steku na početku prekidne rutine, ukoliko se radi o procesorima kod kojih se softverski čuvaju preostali programski dostupni registri, a potom izvrši instrukcija RTI. Ovom instrukcijom se sa steka restauriraju sadržaji programske statusne reči procesora PSW i programskog broja a PC. Od tog trenutka nastavlja se izvršavanje prekinutog glavnog programa od instrukcije koja bi se izvršavala i sa kontekstom procesora koji bi bio, da nije bilo skoka na prekidnu rutinu.

Zahtev za prekid može da bude opslužen i time skok na prekidnu rutinu realizovan ili na kraju instrukcije u toku čijeg izvršavanja je generisan ili kasnije, na kraju neke od sledećih instrukcija. Kada će određeni zahtev za prekid biti opslužen zavisi od više faktora, kao što su: da li je reč o spoljašnjem ili unutrašnjem prekidu, da li su maskiraju i prekidi maskirani, i to ili selektivno ili svi, da li je stigao samo jedan ili više zahteva za prekid, itd. Stoga za svaku vrstu zahteva za prekid određeni uslovi treba da budu ispunjeni da bi se prešlo na njegovo opsluživanje. Prelazak na opsluživanje određenog zahteva za prekid naziva se prihvatanje zahteva za prekid.

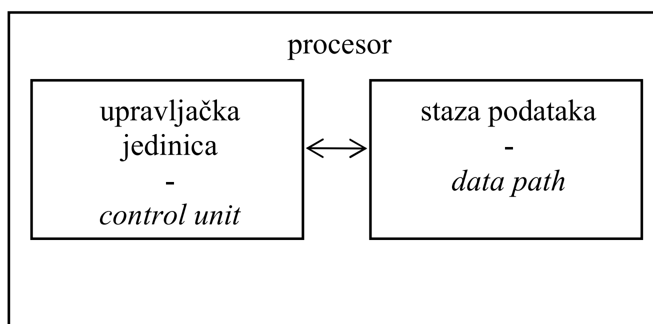
4. Opis strukture procesora – opšti pristup

Struktura procesora predstavlja skup gradivnih blokova koji: ulaze u sastav procesora, hijerarhiski su organizovani i omogućavaju izvršavanje skupa instrukcija, koji je definisan za posmatrani procesor. Obzirom na širok spektar procesora koji danas postoje zaočekivati je veliku raznolikost u pogledu strukture. Međutim, neki od primarnih blokova koji ulaze u sastav procesora su karakterisični za većinu današnjih procesora, tako da ovi blokovi trebaju biti navedeni kada je u pitanju opšti pristup.

Kao dve osnovne gradivne jedinice svakog procesora na najvišem hijerarhiskom nivou javljaju se:

-upravljačka jedinica (eng. *control unit*),
-izvršna jedinica (*execute unit*) ili staza podataka (*data path*).

Ove dve osnovne jedinice su međusobno povezane preko odgovarajuće interne magistrale. Tu možemo razlikovati deo interne magistrale koji se koristi za prenos upravljačkih signala od upravljačke jedinice ka izvršnoj, kao i deo kojim se prenose ulazni podaci iz izvršne jedinice ka upravljačkoj. Osnovni princip funkcionisanja procesora se zasniva na sekvencijanom učitavanju instrukcija iz glavne memorije u procesor, posle čega sledi niz mikrooperacija u samom procesoru u cilju obavljanja zadatka koji je specificiran instrukcijom. Broj i tip mikroopracija koje se izvršavaju za učitanu instrukciju određene su odgovarajućim bitovima u okviru same instrukcije. Prikaz strukture procesora na ovom nivou je dat na slici 2. Nadalje će biti opisana i predstavljena svaka od ovih jedinica posebno.



Slika 2 – Osnovna struktura procesora

4.1. Izvršna jedinica – *execute unit*

Izvršna jedinica je gradivni blok u kome se obavljaju sve aritmetičke operacije, logičke operacije, operacije generisanja adresa itd. Izvršavanje svake od pomenutih operacija se obavlja na osnovu specificiranih upravljačkih signala, koji se generišu u upravljačkoj jedinici. Neki od najčešćih blokova koji ulaze u sastav izvršne jedinice su:
-ALU, pomerači, sabirači, množači itd.;
- generator adresa;
-registri koji se koriste u različite svrhe npr. čuvanje podataka, čuvanje adresa, smeštaj instrukcije;
-odgovarajuće veze između gore pomenutih blokova.

U zavisnosti od hardverske opremljenosti množačima, brojačima, jedinicama za obavljanje različitih aritmetičko-logičkih operacija, procesor je u stanju da korisniku pruži brže ili sporije izvršavanje složenijih operacija. Naime, ako deo izvršne jedinice ne obiluje hardverskim potencijalom složenije operacije će se izvršavati sporo. Opet složeniji hardver zahteva i veću cenu tako da se treba tražiti neki kompromis.

Što se tiče registara koji ulaze u sastav izvršne jedinice možemo ih podeliti na:
- korisnički vidljive registre tj. one koji su vidljivi programeru na mašinskom i asemblerskom nivou;
- upravljačke i statusne registre tj. one koji nisu vidljivi programeru i imaju specificiranu funkciju u procesu izvršavanja instrukcija.

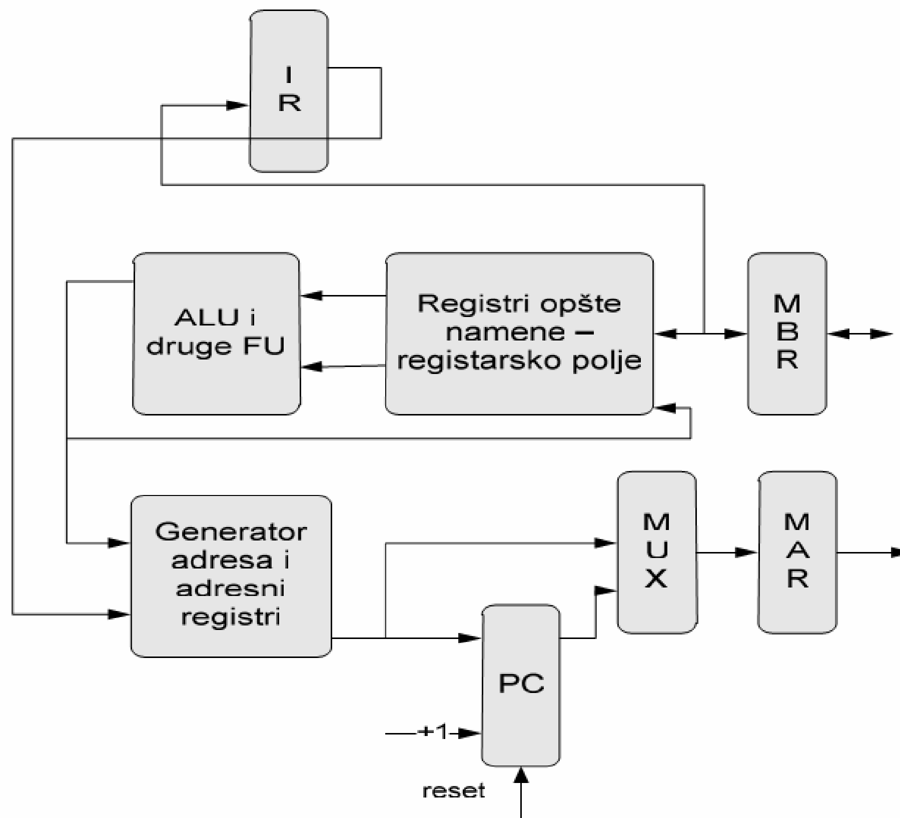
Nadalje ćemo posebno ukazati na svaku od pomenutih grupa registara i predstaviti njihovu funkciju.

U korisničko vidljive registre spadaju:
- registri opšte namene, kojima sam programer dodeljuje različite funkcije;

- registri za podatke, koji se mogu koristiti za čuvanje podataka, ali ne i adresa;
- adresni registri, koji se koriste za specificiranje određenog adresnog načina rada;
- marker registri, koji su delimično vidljivi korisniku i u kojima hardver CPU-a postavlja sam vrednosti na osnovu obavljanja različitih operacija.

Obim korisničko vidljivih registara i njihov broj predstavljaju neke od osnovnih atributa na koje projektant treba obratiti pažnju. Tako, na osnovu obima korisničkih registara razlikujmo 8 - bitne, 16 – bitne, 32 – bitne, 64 – bitne, 128 - bitne procesore. Dalje, broj dostupnih registra utiče na performanse procesora kod konkurentnog izvršavanja instrukcija.

Kao upravljačke i statusne registre nožemo navesti sledeće registre:
- programski brojač (PC – *Program Counter*), koji čuva adresu naredne instrukcije;
- registar naredbi (IR – *Instruction Register*), koji čuva kod instrukcije koja se trenutno izvršava;
- memorijsko adresni registar (MAR – *Memory Address Register*), koji čuva adresu memorijske lokacije koja se adresira;
- memorijsko baferski registar (MBR – *Memory Buffer Register*), koji sadrži podatak koji je poslednji upisan ili poslednji pročitan iz memorije;
- CCR (*Condition Code Register*) služi da čuvanje markera uslova. U markere uslova spadaju prenos – *carry*, znak – *sign*, bit parnosti – *parity bit*, nula bit – *zero*, preoračenje – *overflow*, supervizor bit, koji ukazuje da se izvršava naredbe u supervizorskom načinu rada itd. Na slici 3. su prikazani jedan od načina organizacije izvršne jedinice.



Slika 3. Organizacija izvršne jedinice

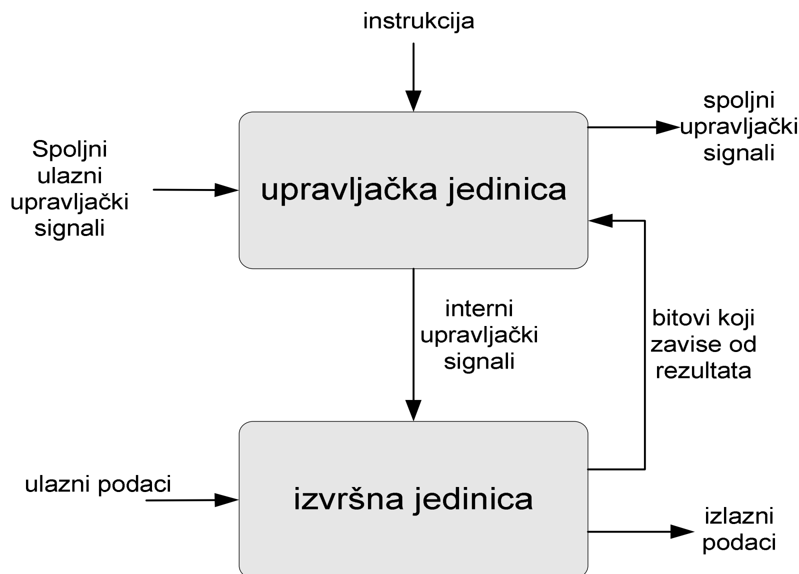
4.2. Upravljačka jedinica – control path

Upravljačka jedinica služi za generisanje upravljačkih signala. Upravljački signali koje generiše upravljačka jedinica se mogu podeliti u dve osnovne grupe:
- interni upravljački signali, koji se vode na izvršnu jedinicu;
- spoljni upravljački signali, uz pomoć kojih se upravlja radom spoljnih komponenata kao što su memorija (čitanje i upis), U/I uređaji itd.

Vrednosti upravljačkih signala na izlazu upravljačke jedinice zavise od:

- bitova na ulazu upravljačke jedinice koji specificiraju instrukciju koja se trenutno izvršava;
- spoljnih ulaznih upravljčkih signala, koji ukazuju na stanje spoljnih uređaja, npr. *Reset* signal, *Ready* signal itd;
- bitova koji specificiraju rezultat predhodne operacije (CCR).

Na slici 4. je prikazana strategija upravljanja.



Slika 4. Osnovna strategija upravljanja

Postoje dva osnovna koncepta realizacije upravljačke jedinice. To su:
- koncept upravljačke jedinice sa direktnim upravljanjem – *hardwired control*;
- koncept upravljačke jedinice sa mikroprogramskim upravljanjem – *microprogrammed control.*

Ako se opredelimo za upravljačku jedinicu sa direktnim upravljanjem, upravljačka jedinica će biti realizovana kao sekvencijalno logičko kolo. Glavna prednost ovakve upravljačke jedinice je velika brzina rada sa minimalnim hardverom, a nedostatak u tome što kada se jednom isprojektuje, ne dozvoljava dodatne izmene. Zato se ovaj tip upravljačke jedinice često sreće kod RISC procesora i neprogramabilnih kontrolera.

Prihvatanje koncepta procesorske jedinice sa mikroprogramskim upravljanjem u konkretnom slučaju zahteva realizaciju upravljačke jedinice na bazi ROM ili RAM memorije u kojoj se smeštaju vrednosti upravljačkih signala, koji će biti aktivirani u odgovarajućem trenutku. Glavne prednosti ovakvog načina realizacije upravljačke jedinice su:
- lako uvođenje dodatnih instrukcija;

- greške se brže i lakše otklanjaju;
- ukupno projektovanje procesora je pojednostavljeno;
- korišćenjem mikroprogramske emulacije moguće je ostvarivanje kompatibilnosti sa drugom mašinom.

Glavni nedostatak ovog pristupa je manja brzina rada u odnosu na direktno upravljanje. Ipak je ovaj način realizacije upravljačke jedinice široko rasprostranjen, gotovo u svim slučajevima kada brzina rad nije od presudnog značaja. Kod upravljačkih jedinica ovog tipa, pri projektovanju se često pravi kompromis između nivoa dekodiranja i kapaciteta memorije koji je upotrebljen za pamćenje upravljačkih signala. Najekstremniji slučaj po pitanju zauzeća memorije je kada nema kodiranja, tj. kada se upravljački signali direktno pamte u memoriji. U slučaju složenih procesora broj upravljačkih signala može biti veliki tako da bi bio potreban popriličan RAM u okviru procesora. Zato se koristi kodiranje kombinacija upravljačkih signala. To dodatno usložnjava kontrolnu jedinicu, ali štedi mikroprogramsku memoriju.

4.3. *Interrupt* prekidnih rutina

U ovom podpoglavlju će biti opisan hardver jednostavnog 16 – bitnog procesora na onom nivou koji može obezbediti dovoljnu preglednost strukture hardvera radi razumevanja njegovog funkcionisanja. Najpre, naglasimo da sam procesor možemo podeliti na dva esencijalna dela tj. jedinice:
- upravljačka jedinica;
- izvršna jedinica.

Funkcije ovih jedinica smo opisali u predhodnom poglavlju tako da ih ovde nećemo ponavljati. Radi preglednosti sada ćemo navesti neke bazične osobine konkretno proučavanog procesora, sa napomenom da smo ih već u predhodnim poglavljima spomenuli i objasnili.
- Procesor je 16 – bitni.
- Procesor ima mikroprogramsko upravljalnje.
- Radimo sa 4 načina adresiranja.
- Proučavamo skup od 19 instrukcija u svim modovima adresiranja.
- Procesor predstavlja jedno – adresnu mašinu.
- Opslužuje maskirajuće i nemaskirajuće prekide

Od načina adresiranja podržani su sledeći:
- neposredno adresiranje;
- direktno adresiranje;
- indirektno adresiranje;
- registarsko adresiranje.

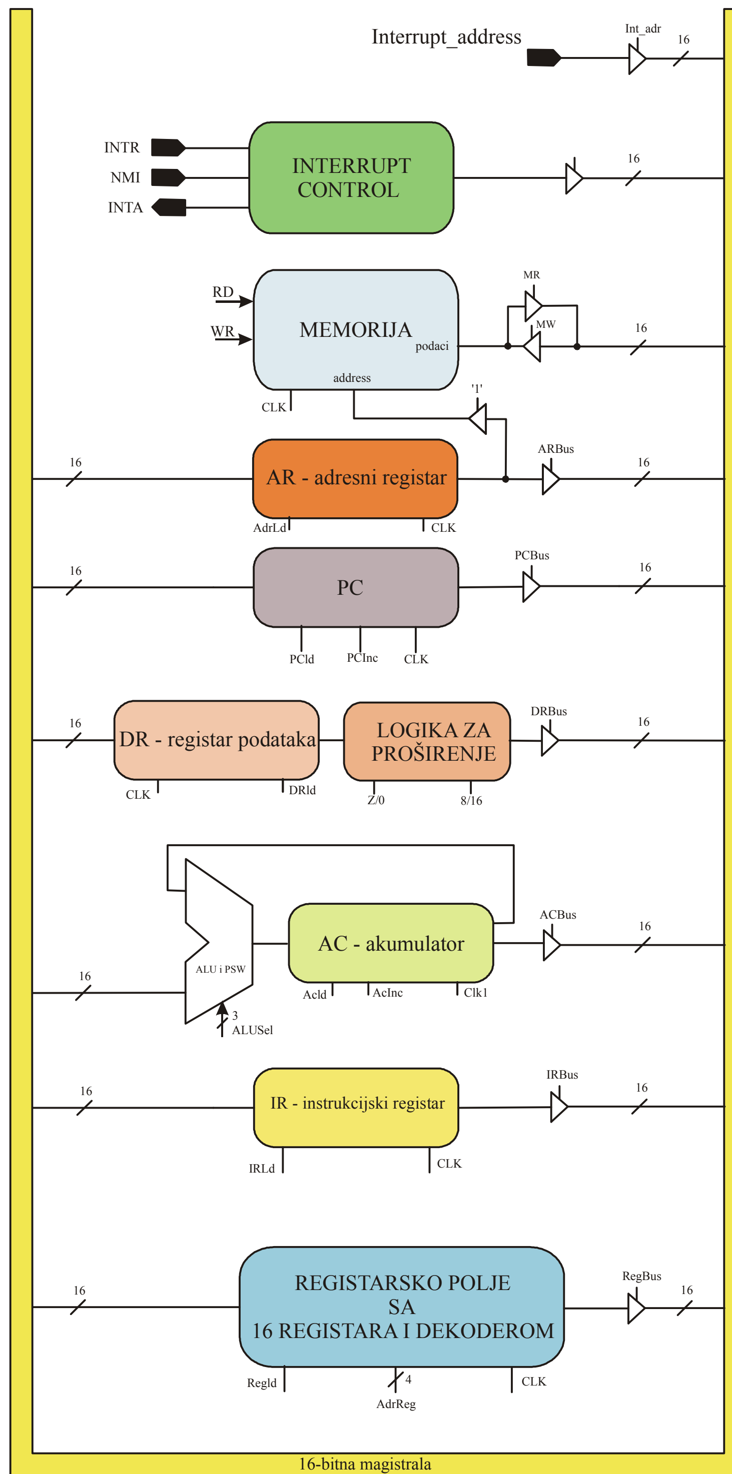
4.4. Struktura hardvera izvršne jedinice

Opisu strukture hradvera ćemo pristupiti preko slike na kojoj je prikazana struktura na nivou registara. Nadalje će biti opisana funkcija svakog od njih u procesu izvršenja instrukcija. Prikaz izvršne jedinice nalazi se na slici 5.

Svi registri i gradivni blokovi komuniciraju preko zajedničke 16 – bitne magistrale. Vidimo da ova jedinica procesora sadrži adresni registar – AR, programski brojač – PC, registar podataka – DR, logiku za proširenje, akumulatorski registar – AC, aritmetičko – logičku jedinicu koja izvršava operacije specificirane opkodom – ALU, instrukcioni registar – IR, registarsko polje sa šesnaest 16 – bitnih registara i dekoderom, PSW – registar, INTERRAPT CONTROL logiku za obradu prekida, ulazni 16-bitni port

interrupt_address i bafere koji kontrolišu pristup magistrali.

Adresni registar služi da čuva adresu memorijske lokacije koja se adresira i obima je 16 bitova. Programski brojač sadrži adresu naredne instrukcije koja će se izvršavati i poseduje port za inkrementiranje, a obima je 16 bitova. Registar podataka se koristi kao registar u kome se privremeno smeštaju željeni podaci pri izvršavanju instrukcija, kako što su adrese ili operandi (obim 16 bitova). Akumulatorski registar služi za storiranje operanda nad kojim se vrši neka operacija kao i za storiranje rezultata te operacije (obim 16 bitova). Instrukcioni registar je jedini obima 8 bitova i sadrži opkod i modifikator instrukcije čije je izvršavanje u toku. Osam ulaznih bit portova ovog registra su povezani na osam MS bitova 16 – bitne magistrale. PSW registar je 4 – bitni i sadrži bit prenosa (*carry bit*), bit znaka (*sign bit*), bit nule (*zero bit*) i bit parnosti (*parity bit*). Ovi bitovi se koriste za procenu uslova grananja pri izvršavanju instrukcije grananja , a postavljaju se od strane ALU – a. INTERRUPT CONTROL logika služi da prihvati i opslužuje spoljne zahteve za prekidom. Vrši se prihvatanje kako maskirajućih tako i nemaskirajućih prekida. Unutar nje se nalaze i privremeni registri za skladištenje tekućeg stanja PC-a i PSW-a, pri ulasku u interapt rutinu. Ulazni 16-bitni port *interrupt_address* služi za prihvat adrese interapt rutine.



Slika 5. Struktura izvršne jedinice procesora

4.5. Skup instrukcija CPU-a

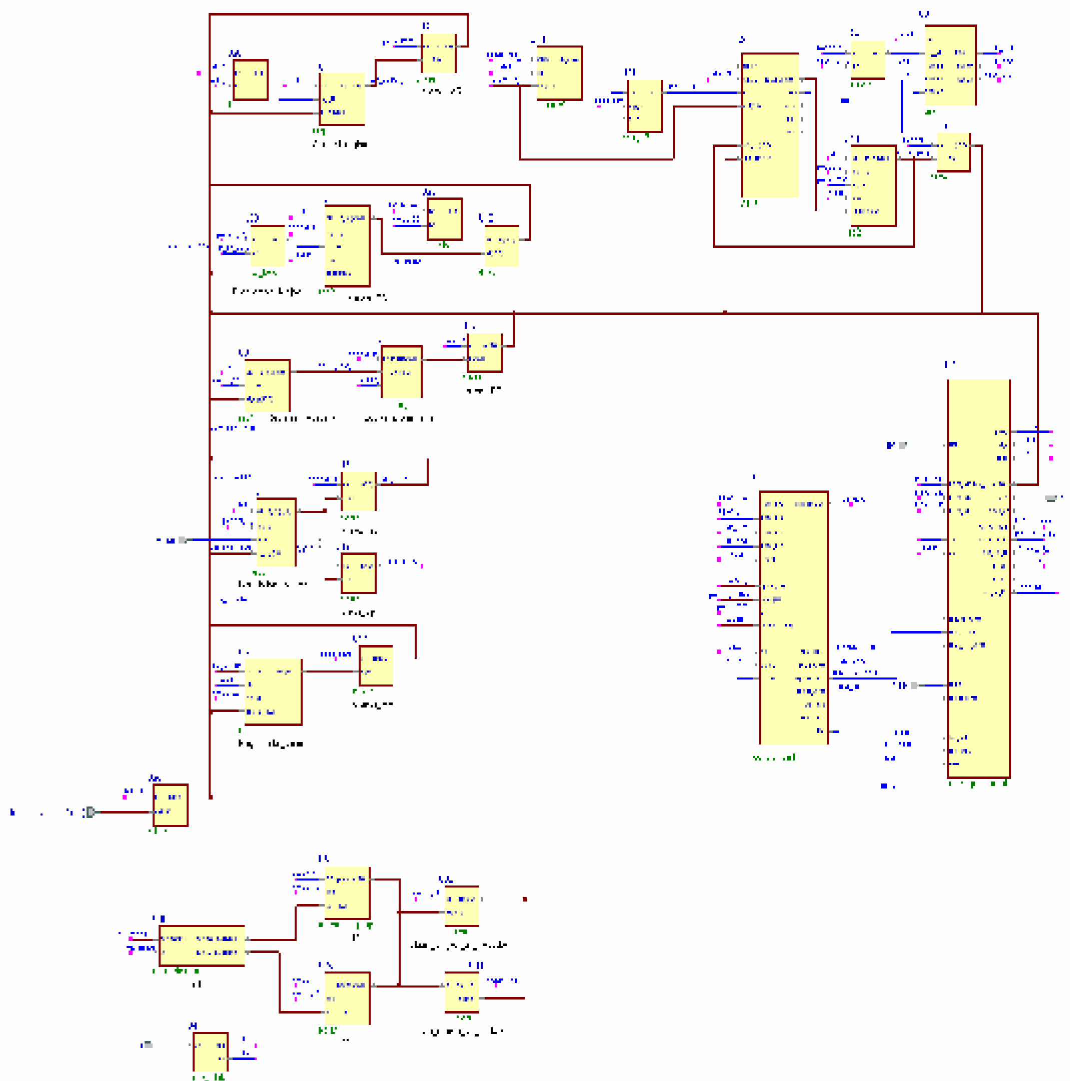
Ovaj jednostavni procesor je struktuiran tako da može da obavlja više tipova instrukcija. Shodno tome, sve instrukcije možemo podeliti u sledeće grupe:
- instrukcije za obavljanje osnovnih aritmetičkih operacija;
- instrukcije za obavljanje osnovnih logičkih operacija;
- instrukcije prenos podataka;
- instrukcije za promenu toka izvršenja programa.

Svaka od ovih instrukcija se realizuje preko četri načina adresiranja, tako da se ukupan broj instrukcija može odrediti kao četvorostruka vrednost broja instrukcija koje obavljaju različite operacije. Treba naravno naglasiti da je ovaj skup instrukcija moguće i proširiti, jer se radi o mikroprogramskom upravljanju. U tabeli na slici 8 je prikazan skup instrukcija, gde su dodate i instrukcije za rad sa *intterupt*-ima.

instrukcija	kod instrukcije	opis instrukcije	tip instrukcije - operacija
OR	000001	OR operacija	logička
AND	000010	AND operacija	logička
NOT	000011	NOT operacija	logička
XOR	000100	ExOR operacija	logička
ADD	000101	operacija sabiranja	aritmetička
SUB	000110	aritmetička operacija oduzimanja	aritmetička
NEG	000111	izračunavanje negativne vrednosti	aritmetička
LD	001000	kopiranje iz memorije u akumulator	prenos podataka
JMP	001001	bezuslovno grananje (skok)	grananje
JZ	001010	grananje ako je nula	grananje
JNZ	001011	gananje ako nije nula	grananje
MOV	001100	kopiranje iz akumulatora u registar Rx	prenos podataka
ST	001101	kopiranje iz akumulatora u memoriju	prenos podataka
INC	001110	inkrementiranje sadržaja operanda	aritmetička
SETC	001111	setovanje bita prenosa	
RESC	010000	resetovanje bita prenosa	
RET	010001	povratak iz *interrupt* rutine	
SIE	010010	dozvola *interrupt*-a	
CIE	010011	zabrana *interrupt*-a	
CIF	010100	brisanje *interrupt flag*-a	
NOP	111111	operacija bez efekta	

5. Realizacija u VHDL-u

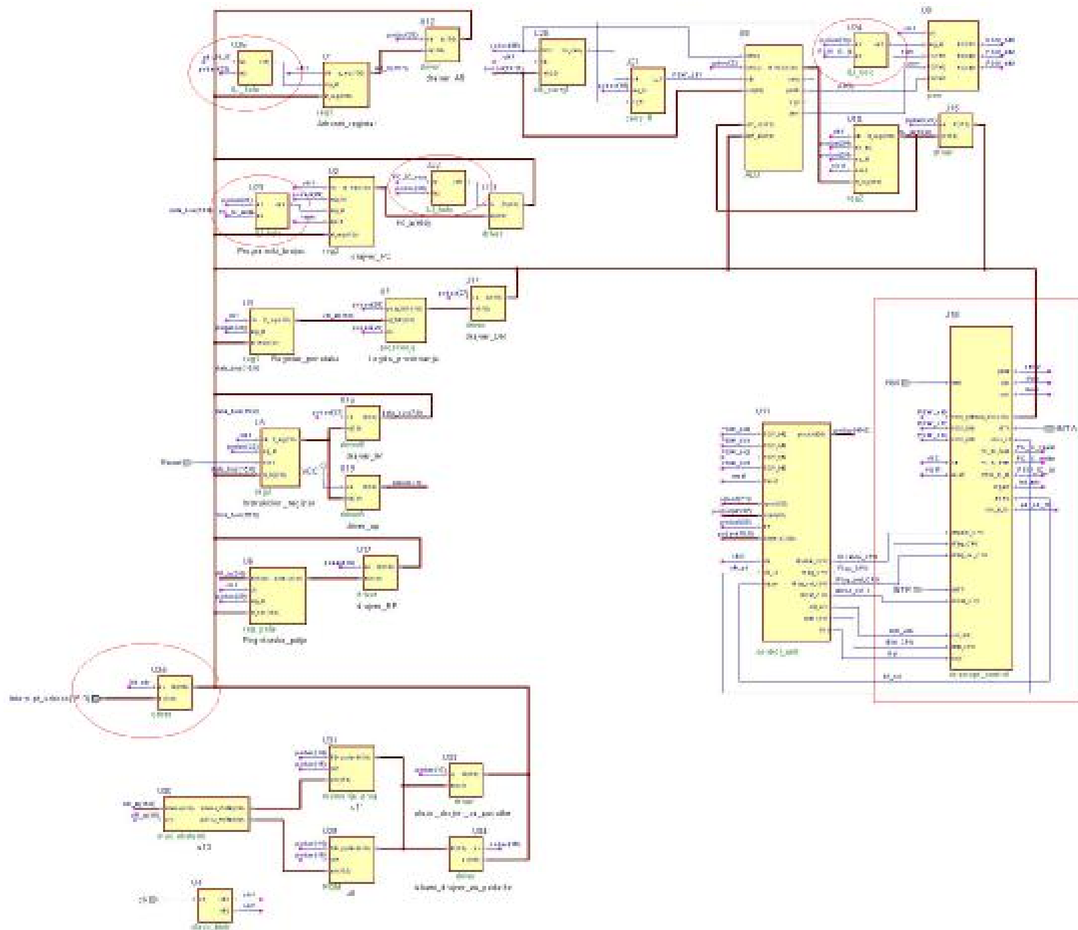
Koristeći osnovnu strukturu realizovanog mikroprocesora dodati su potrebni ulazni i izlazni pinovi, kao i odgovarajuća logika koja bi omogućila da postojeći mikroprocesor može da obradi maskirajuće i nemaskirajuće interapte. Na donjoj slici broj 6. prikazana je blok šema celokupnog mikroprocesora sa pridodatim blokovima, a u nastavku će biti opisani svi pridodati elementi.



Slika 6. Blok šema mikroprocesora sa podrškom za interapte

Najpre od ulaznih i izlaznih pinova dodati su ulazni pinovi INTR, NMI i ulazni port Interrupt_adress(15:0) širine 16 bitova, kao i izlazni pin INTA. Na ulazni pin INTR dovodi se zahtev za maskirajući interapt i on je osetljiv na nivo, aktivan visokim stanjem (stanje logičke jedinice). Pin NMI služi za prihvatanje nemaskirajućih interapta, aktivan rastućom ivicom signala. Izlazni pin INTA služi za potvrdu prijema interapta, generiše se najpre impuls logičke jedinice kad se prihvati interapt, a nakon toga još jedan kad se završi izvršenje tekuće instrukcije. Nakon pojave ovog drugog impulsa potvrde na

izlaznom pinu INTA sa ulaznog porta Interrupt_adress(15:0) prihvata se adresa na kojoj se nalazi interapt rutina koju treba obraditi u okviru nastalog interapta. Pored pomenutih dodatih pinova i porta u samoj strukturi procesora dodati su odgovarajući sklopovi koji su naznačeni na sledećoj slici 7. (zaokruženi).



Slika 7. Blok šema sa označenim pridodatim elementima

Osnovna logika koja je ovde realizovana treba da omogući da nakon što se prihvati zahtev za interapt, generiše jedan impult na izlazu INTA, a nakon toga kada se završi izvršavanje tekuće instrukcije u mikroprocesoru, generiše još jedan impuls. Nakon ovoga impulsa sa ulaza Interrupt_adress(15:0) se prihvata adresa interapt rutine na koju će da se skoči da bi se obradio nastali interapt. Naravno pre toga se u privremene registre u okviru interrupt_control bloka privremeno pamte vrednosti PC-a (Program Counter - programski brojač) i PSW-a (Program Status Word - statusni registar), čije je stanje bitno da bi po povratku mogli da se vratimo tamo gde je program prekinut u trenutku nastajanja interapta.

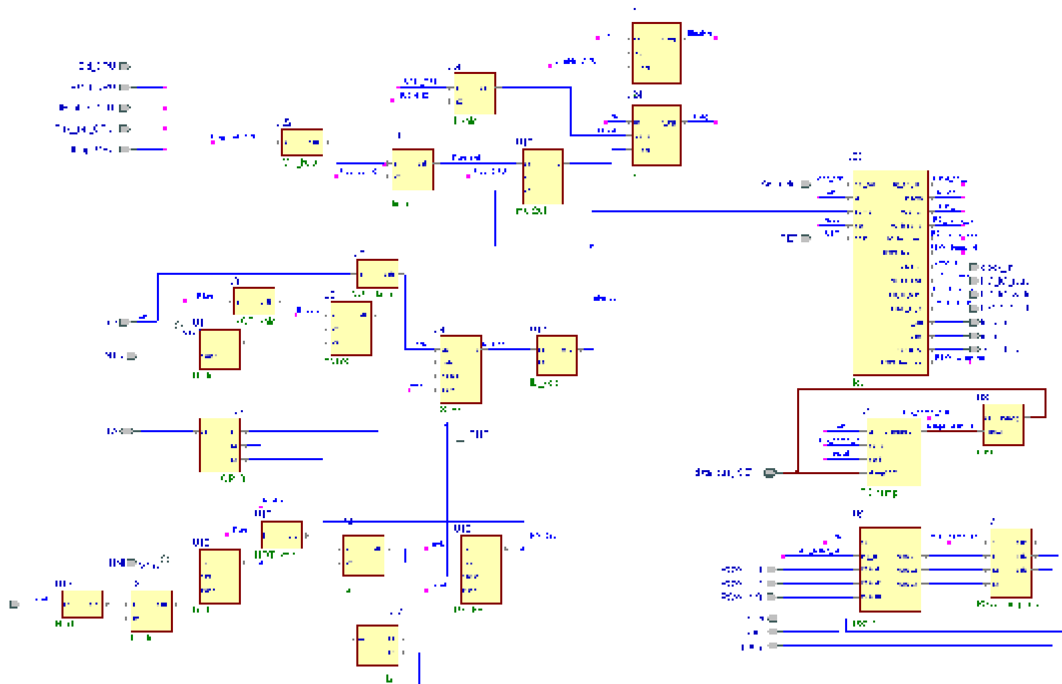
Opis elemenata:
- U25 je 16-bitni trostatički drajver, koji služi da se preko njega upiše adresa interapt rutine na internu magistralu mikroprocesora data_bus(15:0). Signal dozvole za ovaj trostatički bafer je Int_adr koji je aktivan visokim stanjem (logička jedinica).
- U23 je ILI logičko kolo koje omogućava da se kao signal "load" za upis u programski brojač koriste i signal podaci(31) iz postojeće upravljačke logike i signal PC_IC_write iz

bloka interrupt_control. Zapravo u standardnom radu kao signal za upis u PC (programski brojač) se koristi signal podaci(31), dok pri vraćanju zapamćenog stanja PC-a pri izlasku iz interapt rutine koristi se signal PC_IC_write da se zapamćeno stanje upiše u PC.

- U22 je ILI logičko kolo koje omogućava dozvolu trostatičkom baferu na izlazu PC-a, i od strane postojeće upravljačke logike i od strane bloka interrupt_control. U standardnom radu kao signal dozvole koristi se signal podaci(30) da bi se sadržaj PC-a učinio dostupnim na internu magistralu data_bus. Dok pri ulasku u rutinu za obradu prekida kada je potrebno da najpre zapamtimo postojeće stanje PC-a da bi smo nakon povratka iz interapt rutine znali gde smo stali, signalom PC_IC_read stanje PC-a puštamo na internu magistralu data_bus(15:0) nakon čega se ono upisuje u privremeni registar u okviru bloka interrupt_control.
- U26 je ILI logičko kolo koje omogućava da se kao signal "load" za upis u adresni registar koriste i signal podaci(33) iz postojeće upravljačke logike i signal adr_ld_IC iz bloka interrupt_control. Nakon povratka iz interapt rutine i vraćanja zapamćenog stanja PC-a, potrebno je to stanje upisati u adresni registar kako bi mikroprocesor dobio narednu instrukciju koju treba da obavi.
- U24 je ILI logičko kolo koje omogućava da se kao signal "load" za upis u PSW (program status word - statusni registar) koriste i signal podaci(18) iz postojeće upravljačke logike i signal PSW_IC_ld iz bloka interrupt_control. U standardnom radu kao signal za upis u PSW se koristi signal podaci(18), dok pri vraćanju zapamćenog stanja PSW-a pri izlasku iz interapt rutine koristi se signal PSW_IC_ld da se zapamćeno stanje upiše nazad u PSW.
- U18 je blok interrupt_control koji predstavlja glavnu kontrolnu logiku za obradu interapta. Ona naravno i generiše sve potrebne upravljačke signale. Pored postojećih ulaznih pinova INTR,NMI i izlaznog pina INTA, kao ulaze ima i neophodne signale koje smo dodali na glavnu upravljačku logiku, a koji su nam bili neophodni za realizaciju interapta.

5.1. Opis funkcije *Interrupt_control* logike

Na sledećoj slici 8. prikazana je detaljna šema bloka interrupt_control:



Slika 8. Blok interrupt_control

U ulaznom delu se nalazi logika koja omogućava prihvat nemaskirajućih i maskirajućih zahteva za interaptima u zavisnosti od stanja na ulaznim pinovima INTR i NMI, kao i pinovima IEld_CPU, IRCld_CPU, IEnable_CPU, IFlag_sel_CPU i IFlag_CPU, koji se dovode iz osnovne upravljačke logike mikroprocesora i koji zavise od instrukcija SIE, CIE, CIF i tekućeg stanja u izvršenju instrukcije. Blok PC_temp sa izlaznim drajverom predstavlja privremeni registar za pamćenje stanja PC-a. Isto tako blokovi PSW i PSW_temp_drv služe za privremeno pamćenje stanja PSW-a. I na kraju blok ICU predstavlja glavni upravljačku jedinicu za kontrolu interapta i predstavlja konačan automat.

Pri pojavi recimo zahteva za maskirajućim interaptom na ulazu INTR, koji je aktivan kako je ranije naglašeno rastućom ivicom signala, stanje logičke jedinice na *d* ulazu *Dlatch*-a U1 preneće se na njegov izlaz. Izlaz se dalje vodi na U2 AND3 kolo, gde na ostala dva ulaza kao uslove imamo da su dozvoljeni interapti *IEnable ='1'* i da je *interrupt flag IFlag ='0'* . Stanje *IFlag='0'* pri prolazu kroz invertor U3 daje na ulazu U2 logičku jedinicu, tako da i pri pojavi logičke jedinice na izlazu U1, svi ulazi U2 su u stanju logičke jedinice i na izlazu kola imamo takođe stanje logičke jedinice, pošto se radi o AND kolu. To stanje će se na rastuću ivicu novoformiranog *clock* signal *nclk (pomeren za 90 stepeni)* upisati u flip-flop U4 i izlazni signal INTReq postaće logička jedinica. To se prenosi preko ILI kola U12 i signal *interrupt* postaje takođe logička jedinica i vodi se kao takav na ulaz *icu* bloka, kolo U23. Na taj način izvršeno je prihvatanje zahteva za maskirajućim interaptom. Logika *icu* aktiviraće signal *IFLag_sel_IC*, koji će dalje aktivirati *IFLag_sel,* a koji će aktivirati *ul2* na multiplekseru U10. Time će se signal *interrupt* upisati na rastuću ivicu *clk* u *irc* registar U24, pri signalu *IRCld* koji je rezultat aktiviranja signala *IRCld_IC.* Na taj način na izlazu U24 imamo aktivan signal *IFlag* koji

nam signalizira da je *interrupt* prihvaćen i onemogućava prihvat novog *interrupt*-a sve dok se *IFlag* ne izbriše (postavi na logičku nulu) od stane CPU-a. Signal *interrupt* se takođe vodi na ulaz OR_3 kola U6 i time na izlazu INTA generiše impuls potvrde prijema *interrupt*-a.

Logika *icu* pristupa redom, prvo znači čeka da se završi tekuća instrukcija i nakon toga zaustavlja *Interrupt_control logiku* i pamti u svoje privremene registre tekuće stanje registara *PC* i *PSW*. Na kraju generiše interni signal *INTAck* koji preko OR_3 kola U6 generiše još jedan *INTA* signal koji signalizira periferiji koja je izazvala *interrupt* da će adresa sa ulaza *interrupt_address* biti prihvaćena i upisuje se u *PC*, a zatim i u *Adresni_registar*.

Nakon izvršenja instrukcija u okviru prekidne rutine koja se obavezno završava instrukcijom RET, vraća se stanje registara *PC* i *PSW* i program nastavlja dalje sa izvršavanjem tamo gde je stao.

Pri nailasku zahteva za nemaskirajućim *interrapt-om* na ulazu *NMI* koji je kako je ranije naglašeno aktivan na visoko stanje, na izlazu D flip-flopa U13 postaviće se stanje logičke jedinice koje se vodi na jedan ulaz logičkog AND kola U14. Na drugi ulaz dovodi se signal *IFlag* koji se prethodno propušta kroz invertor U15. Tako da ukoliko je *IFlag='0'* izlaz U14 prelazi u vikoso stanje i to se rastućom ivicom signala nclk upisuje u RS flip-flop U16. Na izlazu U16 se dalje pojavljuje stanje logičke jedinice i to je zapravo signal *NMIReq* koji dalje preko U12 postavlja signal *Interrupt* u visoko stanje i dalje je priča slična kao i u pretnodnom slučaju. Suština je da kod nemaskirajućeg *interrupt-a*, zahtev se prihvata bez obriza da li su *interrupt-i* dozvoljeni ili ne (signal IEnable).

Nakon što je izgenerisan signal *NMIReq* preko povratne grane, kola U17 i U18 resetuje se ulazni flip-flop za prihvat nemaskirajućih prekida U13. Time je omogućeno prihvatanje novih prekida. Naravno kolo poseduje i glavni *reset*.

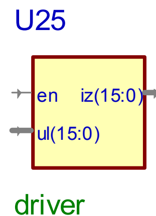
5.2. Opis kontrolnih signala bloka *Interrupt_control*

Adr_ack	signal iz Control_unit-a koji označava da je adresa tekuće mikroinstrukcije 0, to nam ustvari ukazuje da je završena prethodna instrukcija i da treba sa izvršavanjem da krene nova.Tako da tad omogućavamo skok na interapt rutinu
IEld_CPU	signal *load* za učitavanje flega dozvole interapta od strane CPU-a
IEnable_CPU	fleg *Interrapt Enable* (dozvola interapta) od strane CPU-a, koji je rezultat instrukcija SIE i CIE
IFlag_CPU	fleg *Interrapt Flag* (flag za nastanak interapta) od strane CPU-a, koji je rezultat instrukcija CIF
IFlag_sel_CPU	signal koji označava da se upis Interrapt Flag-a vrši od strane CPU-a
INTR	ulaz na koji se dovodi zahtev za maskirajućim interaptom, aktivan rastućom ivicom ulaznog signala
INTA	signal potvrde prijema *interrapt*-a, *interrupt acknowladge*
NMI	ulaz na koji se dovodi zahtev za nemaskirajućim interaptom, aktivan viskokim stanjem
IRCld_CPU	signal *load* za učitavanje flega nastanka interapta (interrupt flag) od strane CPU-a, koristi se kod instrukcije CIF, kad CPU treba da upiše logičku nulu u registar za IF
PSW_bit0	signal *sign flag* iz PSW registra koji se upisuje u PSW temp regista
PSW_bit2	signal *zero flag* iz PSW registra koji se upisuje u PSW temp regista
PSW_bit3	signal *parity flag* iz PSW registra koji se upisuje u PSW temp regista
parity	*parity flag* iz PSW temp registra, koji se nazad upisuje u PSW registar
sign	*sign flag* iz PSW temp registra, koji se nazad upisuje u PSW registar
zero	*zero flag* iz PSW temp registra, koji se nazad upisuje u PSW registar
RET	signal RET koji je rezultat izvršenja instrukcije RET za povratka iz interupt rutine

Adr_ld_IC	signal *load* za učitavanje sačuvanog sadržaja adresnog registra pri povratku iz interrupt rutine
Int_adr	Signal dozvole kojim se učitava interupt adresa sa *interrupt_adress* ulaza, na koju skače program pri zahtevu za interapt
PC_IC_read	signal kojim se iščitava tekuće stanje programskog brajača da bi se privremeno zapamtilo u interrupt control logici
PC_IC_write	signal kojim se zapamćeno stanje programskog brojača nazad upisuje u programski brojač, da bi znao da nastavi sa izvršavanjem programa tamo gde je prekinut
PSW_IC_ld	signal load kojim se zapamćeno stanje PSW registra, vraća nazad pri povratku iz prekidne rutine
clock_st	signal kojim se deaktiviraju svi signali iz mikroprogramske memorije dok se ne usposavi novo stabilno stanje
int_res	signal za resetovanje *control_unit*-a pri povratku iz interapt rutine

5.3. Listing koda bloka *driver*

Blok *driver* predstavlja 16-bitni tri-state buffer. Od ulaznih priključaka ima ulazni 16-bitni port **ul(15:0)**, koji predstavlja ulazni port za podatke i ulazni signal **en** koji predstavlja signal dozvole. Tu je i jedan izlazni 16-bitni port **iz(15:0).**

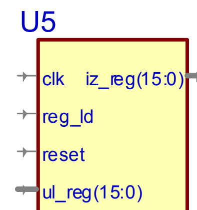



```
library IEEE;
use IEEE.std_logic_1164.all;

entity driver is
        port(ul:in std_logic_vector(15 downto 0);
        en:in std_logic;
        iz:out std_logic_vector(15 downto 0));
end entity;

architecture driver_arch of driver is
begin
pr1:process(ul,en)
    begin
        if en='0' then iz<="ZZZZZZZZZZZZZZZZ";
        elsif en='1' then iz<=ul;
        end if;
    end process pr1;
end driver_arch;
```

5.4. Listing koda bloka *PC_temp*

Blok *PC_temp* predstavlja 16-bitni registar koji se koristi za privremeno smeštanje sadržaja PC-ja. Ima ulazne priključke: signal **clk**, gde se dovodi signal clock-a, signal **reset** signal kojim se resetuje sadržaj registra, signal **reg_ld** kojim se učitava podatak sa ulaza u registar, 16-bitni port **ul_reg(16:0)** koji predstavlja ulaz za registar. Od izlaznih priključaka tu je 16-bitni port **iz_reg(16:0)** koji predstavlja izlaz za podatke. Upis podataka u registar vrši se rastućom ivicom signala **clk** pri aktivnom signalu **reg_ld**.

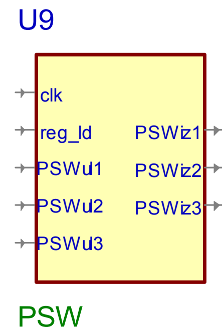
```
library IEEE;
use IEEE.std_logic_1164.all;
```



```
entity PC_temp is
        port( ul_reg:in std_logic_vector(15 downto 0);
        iz_reg: out std_logic_vector(15 downto 0);
        reset,clk,reg_ld : in std_logic);
end entity PC_temp;

architecture PC_temp_arch of PC_temp is
begin
proc:process(ul_reg,clk,reg_ld,reset)
  begin
        if reset='0' then iz_reg<="0000000000000000";
        else
        if (rising_edge(clk) and reg_ld = '1') then
        iz_reg <= ul_reg;
        end if;
        end if;
   end process proc;
end PC_temp_arch;
```

5.5. Listing koda bloka *PSW*

Bok *PSW* predstavlja registar gde se privremeno smešta sadržaj registra PSW tokom obrade interapta. Od ulaznih priključaka tu su: signal **clk** na koji se dovodi clock, signal **reg_ld** koji dozvoljava upis podataka u registar, i signali **PSW ul1**, **PSW ul2** i **PSW ul3** koji predstavljaju ulaze za podatke u registar.Od izlaznih priključaka tu su signali **PSW iz1**, **PSW iz2** i **PSW iz3** koji predstavljaju izlaze za podatke iz registra. Upis u registar se vrši rastućom ivicom signala **clk** pri aktivnom signalu **reg_ld**.




```
library IEEE;
use IEEE.std_logic_1164.all;

entity PSW is
        port( PSWul1,PSWul2,PSWul3:in std_logic;
                PSWiz1,PSWiz2,PSWiz3: out std_logic;
       clk,reg_ld : in std_logic);
end entity PSW;

architecture PSW_arch of PSW is
begin
        process(PSWul1,PSWul2,PSWul3,clk,reg_ld)
        begin
                if reg_ld='1' then
                if rising_edge(clk) then
                PSWiz1 <= PSWul1;
                        PSWiz2 <= PSWul2;
                        PSWiz3 <= PSWul3;
        end if;
                end if;
end process;
```

```
end PSW_arch;
```

5.6. Listing koda bloka *ICU*

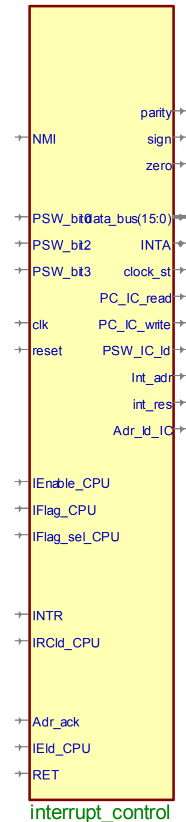
U nastavku će biti samo dat listing koda za blok ICU, koji zapravo predstavlja **I**nterrupt **C**ontrol **U**nit. ICU zapravo predstavlja jedan automat konačnog stanja koji upravlja radom kompletne logike za opsluživanje prekida.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity ICU is
	port(Adr_ack:in std_logic;
			clk:in std_logic;
	  interrupt:in std_logic;
	  reset:in std_logic;
	  RET:in std_logic;
	  clock_st:out std_logic;
	  INTAck:out std_logic;
	  IFlag_sel_IC:out std_logic;
	  IRCld_IC:out std_logic;
	  Int_adr:out std_logic;
	  Int_res:out std_logic;
	  Adr_ld_IC:out std_logic;
	  PC_temp_ld:out std_logic;
	  PC_temp_out:out std_logic;
	  PC_IC_read:out std_logic;
	  PC_IC_write:out std_logic;
	  PSW_IC_ld:out std_logic;
	  PSW_temp_en:out std_logic;
	  PSW_temp_ld:out std_logic);
end ICU;

--}} End of automatically maintained section

architecture ICU of ICU is
type state_type is (reset_state,ispitaj_adr,ispitaj_int,S0,S1,S2,S3,R0,R1);
signal state,next_state: state_type;

begin
	proc1:process(clk,reset) is
	begin
		if reset ='0' then state <= reset_state;
		elsif rising_edge(CLK) then state <= next_state;
		end if;
end process proc1;

proc2:process (state,Adr_ack,interrupt,RET)is
begin
```

```
        INTAck <='0';
        IFlag_sel_IC <='0';
        IRCld_IC <='0';
        PC_temp_ld <='0';
        PC_temp_out <='0';
        PSW_IC_ld<='0';
        PSW_temp_ld <='0';
        PSW_temp_en <='0';
        clock_st<='1';
        PC_IC_read<='0';
        PC_IC_write<='0';
        Int_adr<='0';
        Int_res<='0';
        Adr_ld_IC<='0';
        case state is
                when reset_state => next_state<=ispitaj_int;
                when ispitaj_int =>
                if interrupt='1' then next_state <= S0;
                        IFlag_sel_IC<='1';
                elsif RET='1' then next_state <= R0;
                        clock_st<='0';
                        PC_temp_out<='1';
                        adr_ld_IC<='1';
                        PC_IC_write<='1';
                        PSW_temp_en<='1';
                        PSW_IC_ld<='1';
                else
                        next_state <=ispitaj_int;
                end if;

                when R0 => next_state <=ispitaj_int;
                when S0 => next_state <=S1;
                if Adr_ack='1' then next_state <=S1;
                        INTAck<='1';
                        PC_IC_read<='1';
                        PC_temp_ld<='1';
                        PSW_temp_ld<='1';
                        clock_st<='0';
                        else next_state<=S0;
                end if;
                when S1 => next_state <=S2;
                   clock_st<='0';
                        Int_adr<='1';
                        PC_IC_write<='1';
                        Adr_ld_IC<='1';
                when S2=> next_state <=S3;
                        int_res<='0';
                when S3=> next_state <=ispitaj_int;
        when others  => next_state <= reset_state;
        end case;
end process proc2;
```

end ICU;

5.7. Listing koda bloka *Interrupt_control*




```
library IEEE;
use IEEE.std_logic_1164.all;


entity Interrupt_control is
  port(
       Adr_ack : in STD_LOGIC;
       IEld_CPU : in STD_LOGIC;
       IEnable_CPU : in STD_LOGIC;
       IFlag_CPU : in STD_LOGIC;
       IFlag_sel_CPU : in STD_LOGIC;
       INTR : in STD_LOGIC;
       IRCld_CPU : in STD_LOGIC;
       NMI : in STD_LOGIC;
       PSW_bit0 : in STD_LOGIC;
       PSW_bit2 : in STD_LOGIC;
       PSW_bit3 : in STD_LOGIC;
       RET : in STD_LOGIC;
       clk : in STD_LOGIC;
       reset : in STD_LOGIC;
       Adr_ld_IC : out STD_LOGIC;
       Int_adr : out STD_LOGIC;
       PC_IC_read : out STD_LOGIC;
       PC_IC_write : out STD_LOGIC;
       PSW_IC_ld : out STD_LOGIC;
       clock_st : out STD_LOGIC;
       int_res : out STD_LOGIC;
       parity : out STD_LOGIC;
       sign : out STD_LOGIC;
       zero : out STD_LOGIC;
       INTA : inout STD_LOGIC;
       data_bus : inout STD_LOGIC_VECTOR(15 downto 0)
  );
end Interrupt_control;

architecture Interrupt_control of Interrupt_control is

---- Component declarations -----

component AND3
  port (
       in1 : in STD_LOGIC;
       in2 : in STD_LOGIC;
       in3 : in STD_LOGIC;
       out1 : out STD_LOGIC
  );
```

```
end component;
component Dlatch
  port (
      d : in STD_LOGIC;
      enable : in STD_LOGIC;
      q : out STD_LOGIC
  );
end component;
component driver
  port (
      en : in STD_LOGIC;
      ul : in STD_LOGIC_VECTOR(15 downto 0);
      iz : out STD_LOGIC_VECTOR(15 downto 0)
  );
end component;
component D_ff
  port (
      clk : in STD_LOGIC;
      data : in STD_LOGIC;
      reset : in STD_LOGIC;
      q : out STD_LOGIC
  );
end component;
component icu
  port (
      Adr_ack : in STD_LOGIC;
      RET : in STD_LOGIC;
      clk : in STD_LOGIC;
      interrupt : in STD_LOGIC;
      reset : in STD_LOGIC;
      Adr_ld_IC : out STD_LOGIC;
      IFlag_sel_IC : out STD_LOGIC;
      INTAck : out STD_LOGIC;
      IRCld_IC : out STD_LOGIC;
      Int_adr : out STD_LOGIC;
      Int_res : out STD_LOGIC;
      PC_IC_read : out STD_LOGIC;
      PC_IC_write : out STD_LOGIC;
      PC_temp_ld : out STD_LOGIC;
      PC_temp_out : out STD_LOGIC;
      PSW_IC_ld : out STD_LOGIC;
      PSW_temp_en : out STD_LOGIC;
      PSW_temp_ld : out STD_LOGIC;
      clock_st : out STD_LOGIC
  );
end component;
component ILI_kolo
  port (
      in1 : in STD_LOGIC;
      in2 : in STD_LOGIC;
      out1 : out STD_LOGIC
```

```
  );
end component;
component IRC
  port (
      clk : in STD_LOGIC;
      reg_ld : in STD_LOGIC;
      ul_reg : in STD_LOGIC;
      iz_reg : out STD_LOGIC
  );
end component;
component I_kolo
  port (
      in1 : in STD_LOGIC;
      in2 : in STD_LOGIC;
      out1 : out STD_LOGIC
  );
end component;
component mux2u1
  port (
      sel : in STD_LOGIC;
      ul1 : in STD_LOGIC;
      ul2 : in STD_LOGIC;
      iz : out STD_LOGIC
  );
end component;
component NOT_kolo
  port (
      in1 : in STD_LOGIC;
      out1 : out STD_LOGIC
  );
end component;
component OR_3
  port (
      in1 : in STD_LOGIC;
      in2 : in STD_LOGIC;
      in3 : in STD_LOGIC;
      out1 : out STD_LOGIC
  );
end component;
component PC_temp
  port (
      clk : in STD_LOGIC;
      reg_ld : in STD_LOGIC;
      reset : in STD_LOGIC;
      ul_reg : in STD_LOGIC_VECTOR(15 downto 0);
      iz_reg : out STD_LOGIC_VECTOR(15 downto 0)
  );
end component;
component PSW
  port (
      PSWul1 : in STD_LOGIC;
```

```
        PSWul2 : in STD_LOGIC;
        PSWul3 : in STD_LOGIC;
        clk : in STD_LOGIC;
        reg_ld : in STD_LOGIC;
        PSWiz1 : out STD_LOGIC;
        PSWiz2 : out STD_LOGIC;
        PSWiz3 : out STD_LOGIC
  );
end component;
component PSW_temp_drv
  port (
        en : in STD_LOGIC;
        in1 : in STD_LOGIC;
        in2 : in STD_LOGIC;
        in3 : in STD_LOGIC;
        out1 : out STD_LOGIC;
        out2 : out STD_LOGIC;
        out3 : out STD_LOGIC
  );
end component;
component RS_FF
  port (
        clk : in STD_LOGIC;
        data : in STD_LOGIC;
        preset : in STD_LOGIC;
        reset : in STD_LOGIC;
        q : out STD_LOGIC
  );
end component;

----     Constants     -----
constant VCC_CONSTANT   : STD_LOGIC := '1';
constant GND_CONSTANT   : STD_LOGIC := '0';

---- Signal declarations used on the diagram ----

signal GND : STD_LOGIC;
signal IEnable : STD_LOGIC;
signal IFlag : STD_LOGIC;
signal IFlag_sel : STD_LOGIC;
signal IFlag_sel_IC : STD_LOGIC;
signal INTAck : STD_LOGIC;
signal interrupt : STD_LOGIC;
signal INTReq : STD_LOGIC;
signal IRCld : STD_LOGIC;
signal IRCld_IC : STD_LOGIC;
signal nclk : STD_LOGIC;
signal NET10739 : STD_LOGIC;
signal NET10761 : STD_LOGIC;
signal NET11244 : STD_LOGIC;
signal NET12745 : STD_LOGIC;
```

```
signal NET21250 : STD_LOGIC;
signal NET30273 : STD_LOGIC;
signal NET30277 : STD_LOGIC;
signal NET30281 : STD_LOGIC;
signal NET5886 : STD_LOGIC;
signal NET5970 : STD_LOGIC;
signal NET7809 : STD_LOGIC;
signal NET8097 : STD_LOGIC;
signal NET8142 : STD_LOGIC;
signal NET8223 : STD_LOGIC;
signal NMIReq : STD_LOGIC;
signal PC_temp_ld : STD_LOGIC;
signal PC_temp_out : STD_LOGIC;
signal PSW_temp_en : STD_LOGIC;
signal PSW_temp_ld : STD_LOGIC;
signal VCC : STD_LOGIC;
signal tempPC : STD_LOGIC_VECTOR (15 downto 0);

begin

----  Component instantiations  ----

U1 : Dlatch
  port map(
       d => VCC,
       enable => INTR,
       q => NET21250
  );

U10 : mux2u1
  port map(
       iz => NET7809,
       sel => IFlag_sel,
       ul1 => IFlag_CPU,
       ul2 => interrupt
  );

U11 : IRC
  port map(
       clk => clk,
       iz_reg => IEnable,
       reg_ld => IEld_CPU,
       ul_reg => IEnable_CPU
  );

U12 : ILI_kolo
  port map(
       in1 => INTReq,
       in2 => NMIReq,
       out1 => interrupt
  );
```

```
U13 : D_ff
  port map(
       clk => NMI,
       data => VCC,
       q => NET8223,
       reset => NET10739
  );

U14 : I_kolo
  port map(
       in1 => NET8097,
       in2 => NET8223,
       out1 => NET8142
  );

U15 : NOT_kolo
  port map(
       in1 => IFlag,
       out1 => NET8097
  );

U16 : RS_FF
  port map(
       clk => nclk,
       data => NET8142,
       preset => GND,
       q => NMIReq,
       reset => reset
  );

U17 : I_kolo
  port map(
       in1 => NMIReq,
       in2 => NET8223,
       out1 => NET11244
  );

U18 : ILI_kolo
  port map(
       in1 => NET10761,
       in2 => NET11244,
       out1 => NET10739
  );

U19 : NOT_kolo
  port map(
       in1 => reset,
       out1 => NET10761
  );
```

```
U2 : AND3
  port map(
      in1 => IEnable,
      in2 => NET5886,
      in3 => NET21250,
      out1 => NET5970
  );

U20 : ILI_kolo
  port map(
      in1 => IRCld_CPU,
      in2 => IRCld_IC,
      out1 => IRCld
  );

U21 : I_kolo
  port map(
      in1 => NET12745,
      in2 => IFlag_sel_IC,
      out1 => IFlag_sel
  );

U22 : NOT_kolo
  port map(
      in1 => IFlag_sel_CPU,
      out1 => NET12745
  );

U23 : icu
  port map(
      Adr_ack => Adr_ack,
      Adr_ld_IC => Adr_ld_IC,
      IFlag_sel_IC => IFlag_sel_IC,
      INTAck => INTAck,
      IRCld_IC => IRCld_IC,
      Int_adr => Int_adr,
      Int_res => int_res,
      PC_IC_read => PC_IC_read,
      PC_IC_write => PC_IC_write,
      PC_temp_ld => PC_temp_ld,
      PC_temp_out => PC_temp_out,
      PSW_IC_ld => PSW_IC_ld,
      PSW_temp_en => PSW_temp_en,
      PSW_temp_ld => PSW_temp_ld,
      RET => RET,
      clk => clk,
      clock_st => clock_st,
      interrupt => interrupt,
      reset => reset
  );
```

```
U24 : IRC
  port map(
       clk => clk,
       iz_reg => IFlag,
       reg_ld => IRCld,
       ul_reg => NET7809
  );

U25 : NOT_kolo
  port map(
       in1 => clk,
       out1 => nclk
  );

U3 : NOT_kolo
  port map(
       in1 => IFlag,
       out1 => NET5886
  );

U4 : RS_FF
  port map(
       clk => nclk,
       data => NET5970,
       preset => GND,
       q => INTReq,
       reset => reset
  );

U5 : PC_temp
  port map(
       clk => clk,
       iz_reg => tempPC,
       reg_ld => PC_temp_ld,
       reset => reset,
       ul_reg => data_bus
  );

U6 : OR_3
  port map(
       in1 => interrupt,
       in2 => INTAck,
       in3 => NMIReq,
       out1 => INTA
  );

U7 : PSW_temp_drv
  port map(
       en => PSW_temp_en,
       in1 => NET30281,
       in2 => NET30277,
```

```
      in3 => NET30273,
      out1 => sign,
      out2 => zero,
      out3 => parity
  );

U8 : driver
  port map(
      en => PC_temp_out,
      iz => data_bus,
      ul => tempPC
  );

U9 : PSW
  port map(
      PSWiz1 => NET30281,
      PSWiz2 => NET30277,
      PSWiz3 => NET30273,
      PSWul1 => PSW_bit0,
      PSWul2 => PSW_bit2,
      PSWul3 => PSW_bit3,
      clk => clk,
      reg_ld => PSW_temp_ld
  );


---- Power , ground assignment ----

VCC <= VCC_CONSTANT;
GND <= GND_CONSTANT;

end Interrupt_control;
```

Pored pobrojanih blokova tu se još nalaze i opisi nekih osnovnih elemenata poput NOT kola, AND kola , OR kola, RS flip-flopa, D flip-flopa čije opise možete pogledati u VHDL opisu samog procesora.

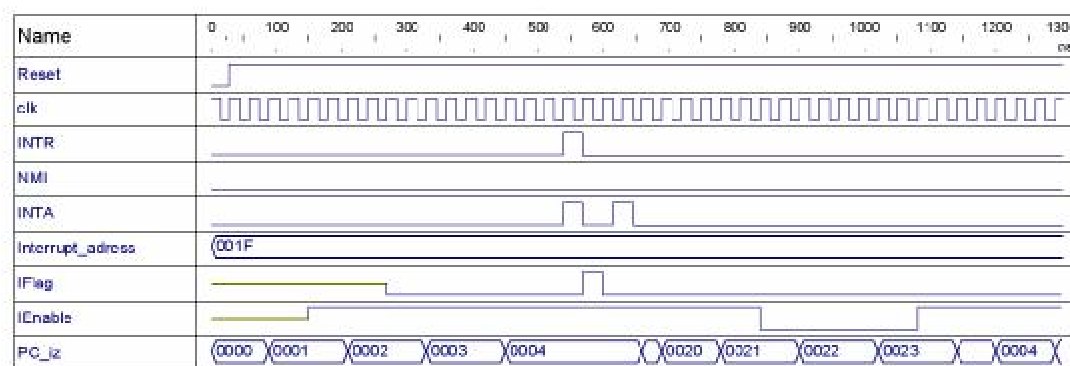
6. Rezultati simulacije

Da bi smo pokazali ispravnost rada interapt logike, simulacija je izvršena tako što smo na ulaze kola dovodili zahteva za interaptima preko ulaza **INTR** i **NMI**, a na ulaz **Interrupt_adress** doveli adresu 0x1F na koju program skače kada se prihvati zahtev za interaptom. Unutar interapt rutine najpre smo zabranili interapte, potom izbrisali interrupt flag, dozvolili prekide i na kraju naravno naredbom RET omogućili povratak iz interapt rutine. Naravno između linija koda gde brišemo interrupt flag i dozvolimo prekide, može da se nađe set naredbi koje će se obaviti pri pojavi interapta, ali forma ostalih naredbi se uvek koristi u interapt rutini.

"0100110000000000",--adr=32 --CIE - zabrana prekida
"0101000000000000",--adr=33 --CIF - brisanje interrupt flaga
"0100100000000000",--adr=34 --SIE - dozvola prekida
"0100010000000000");--adr=35--RET - povratak iz interrupt rutine

U nastavku biće prikazani rezultati simulacije za četiri slučaja kako bi prikazali sve situacije koje se mogu javiti.

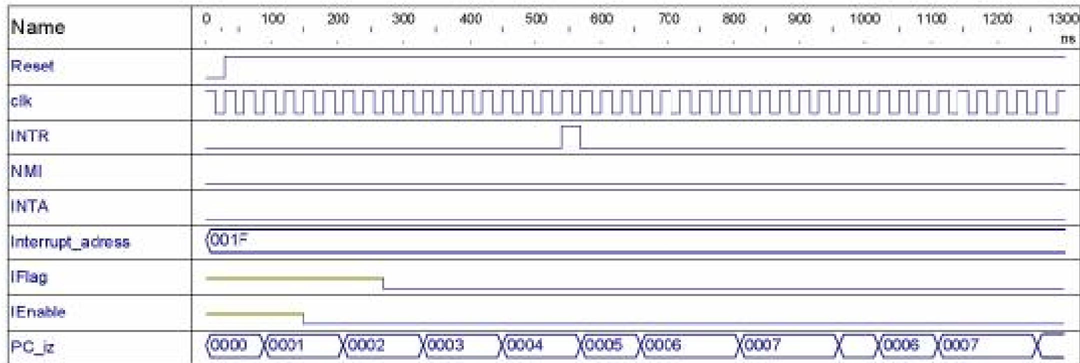
Najpre na grafiku 1. posmatramo situaciju kada se javlja zahtev za maskirajućim interaptom (*INTR=1*), a pri tome su prekidi dozvoljeni (*SIE=1*).



Grafik 1.

Sa grafika vidimo da je došao zahtev za interapt na ulazu *INTR* u trenutku 540ns, pri čemu je signal *IEnable* aktivan, što ukazuje da su interapti dozvoljeni. Interapt je prihvaćen i generisan je prvi impuls *INTA*, a nakon toga i drugi, kao i *IFlag* bit koji nam ukazuje da je došlo do interapta. U trenutku dolaska zahteva za interapt vidimo da je stanje programskog brojača (*PC_iz*) bilo 4. Po prijemu interapta program skače na adresu 0x1F i izvršava sekvencijalno sledeće instrukcije sve dok ne dođe do *RET* instrukcije koja označava povratak iz interapt rutine i nastavak sa obradom tamo gde je program prekinut.Vidimo da signal *IEnable* koji je pre pojave zahteva za interaptom bio aktivan (logička "1") u interapt rutini se postavlja na neaktivno stanje (logička "0") što je rezultat izvršavanja instrukcije *CIE* (zabrana prekida) i na kraju interapt rutine opet se vraća u aktivno stanje jer je instrukcijom *SIE* ponovno dozvoljen prihvat interapta. Nakon obrade interapta u trenutku 1200ns, vidimo da je stanje vraćeno na 4 i dalje se nastavlja izvršavanje programa.

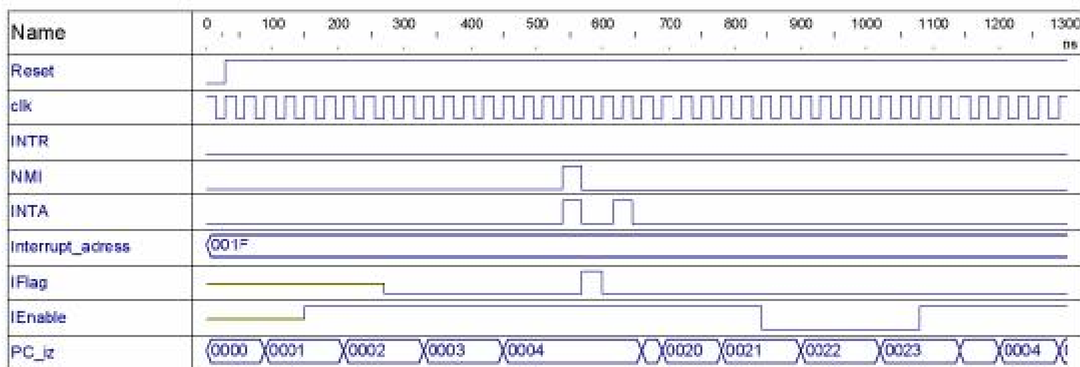
Na grafiku 2. prikazana je situacija kada se javlja zahtev za maskirajućim interaptom, ali je pri tome prekid zabranjen (*CIE=0*).



Grafik 2.

Na grafiku se vidi da i pored zahteva za interaptom koji se javio u trenutku 540ns, nije došlo do prihvatanja interapta (*INTA=0*), jer je dozvola interapta zabranjena (*IEnable='0'*). Vidimo da se i tok programa nastavlja bez prekida.

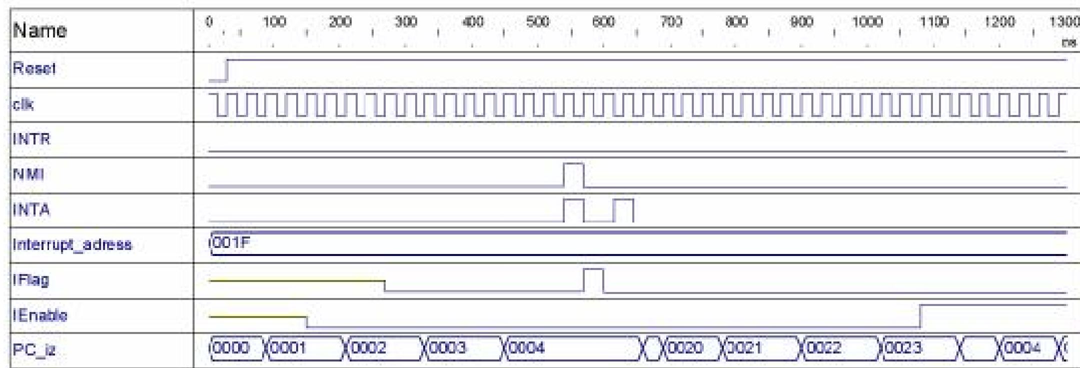
Sledeća situacija ilustruje slučaj kada se javlja nemaskirajući prekid *NMI*, a uz to prekidi su dozvoljeni (*SIE=1*). Dobijeni rezultati su prikazani na grafiku 3.



Grafik 3.

U trenutku 540ns javlja se signal *NMI* i vidimo da se javljaju signali *INTA* dva puta, nakon čega je procesor spreman da prihvati adresu interapt rutine i procesor će pristupiti obradi instrukcija u okviru interapt rutine skokom na adresu 0x1F. Nakon čega vidimo da je sve kao i u prvom slučaju, *IEnable* postaje neaktivno, čime su zabranjeni prekidi, a na kraju postaje opet aktivan kad se završi izvršavanje instrukcija u okviru prekidne rutine.

I na kraju razmatramo slučaj kada se pojavljuje nemaskirajući interapt *NMI*, a pri tome su interapti zabranjeni, grafik 4.

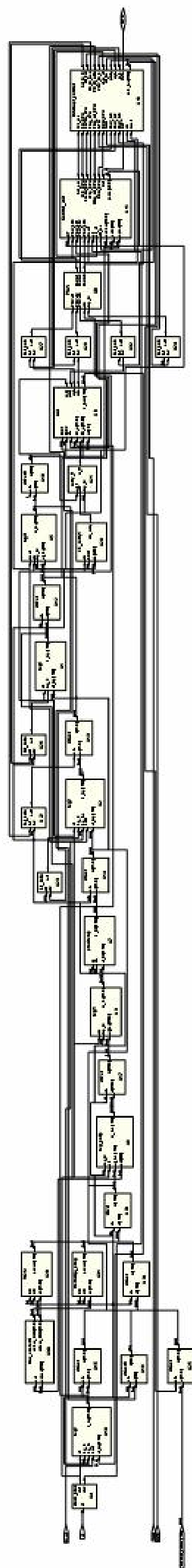


Grafik 4.

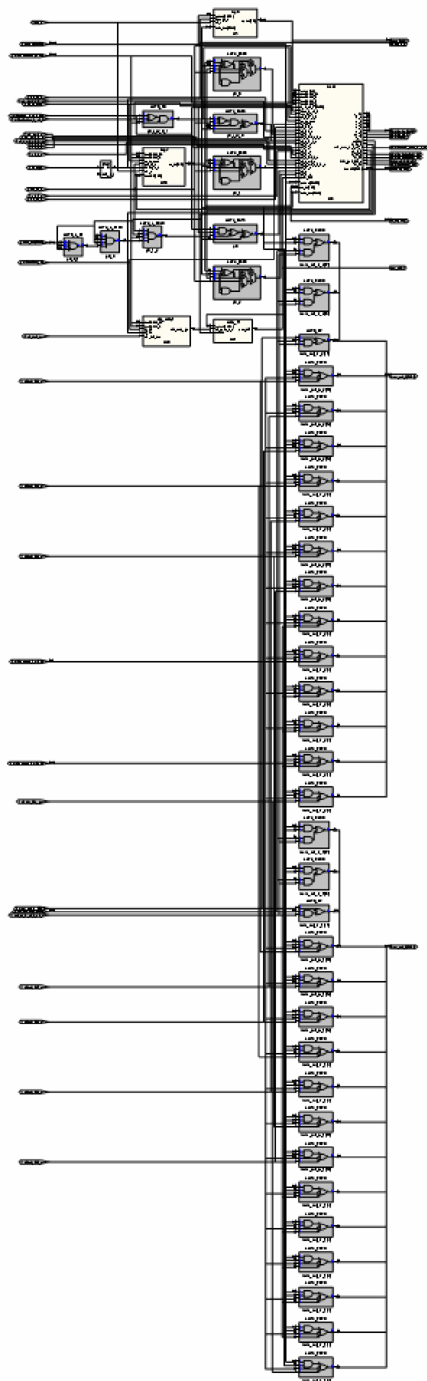
U trenutku 540ns javlja se signal *NMI*, a pri tome vidimo da su zabranjeni interapri, signal *IEnable* neaktivan. Vidimo da je interapt prihvaćen i pored toga, što se naravno i očekivalo za nemaskirajuće interapte. Na kraju je instrukcijom *SIE* signal *IEnable* postavljen u aktivno stanje.

7. Sinteza i implementacija

Sinteza opisanog procesora izvršena je u programskom paketu Symplify Pro v7.70 firme Synplicity. Kao rezultat dobili smo sintetizovanu šemu na RTL nivou i nivou gejtova. U nastavku slede dobijene šeme.



Slika 9. Sintetizovana šema na gate nivou



Slika 10. Sintetizovana šema na RTL nivou

Implementacija je potom izvršena na XILINX-ovom FPGA čipu serije SPARTAN3 XS3S1000FT256, koje dolazi u kućištu BGA256. Naravno nije nam bio potreban veliki broj IO linija, ali nam je trebao veći broj logičkih elemenata tako da smo se opredelili za ovaj čip. U nastavku su priloženi osnovni podaci dobiveni nakon sinteze, izgled veza, raspored pinova, zauzeće čipa, vremenski parametri (kašnjenja kroz veze i maksimalna frekvencija rada), potrošnja, itd.

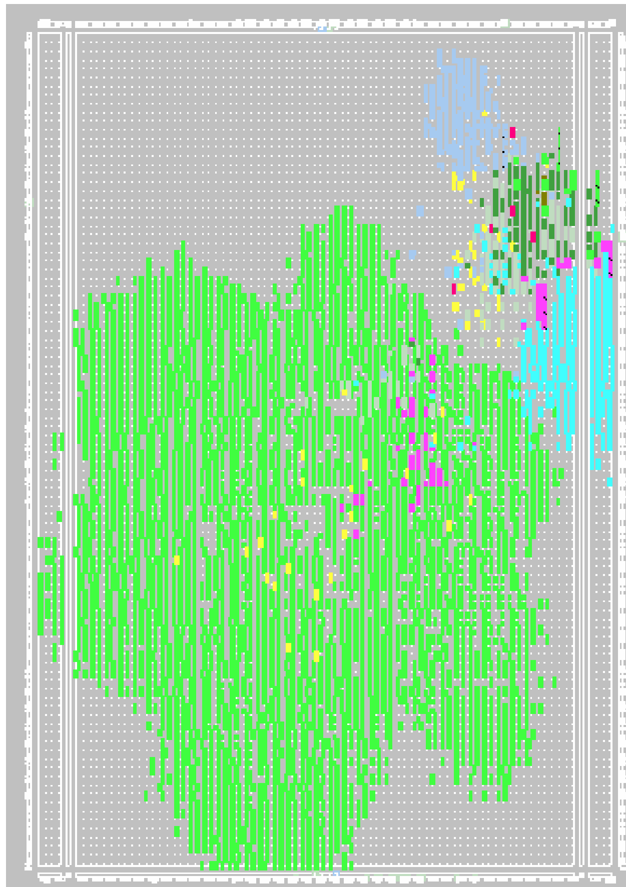
```
Started process "Map".

Using target part "3s1000ft256-4".
Removing unused or disabled logic...
Running cover...
Running directed packing...
Running delay-based LUT packing...
Running related packing...

Design Summary:
Number of errors:      0
Number of warnings:  258
Logic Utilization:
  Total Number Slice Registers:     4,552 out of  15,360   29%
    Number used as Flip Flops:                 425
    Number used as Latches:                  4,127
  Number of 4 input LUTs:           3,743 out of  15,360   24%
Logic Distribution:
  Number of occupied Slices:                       4,209 out of   7,680   54%
    Number of Slices containing only related logic:   4,209 out of   4,209  100%
    Number of Slices containing unrelated logic:          0 out of   4,209    0%
      *See NOTES below for an explanation of the effects of unrelated logic
Total Number 4 input LUTs:          3,786 out of  15,360   24%
  Number used as logic:              3,743
  Number used as a route-thru:          43
  Number of bonded IOBs:               21 out of     173   12%
  Number of GCLKs:                      4 out of       8   50%

Total equivalent gate count for design:  51,764
Additional JTAG gate count for IOBs:  1,008
Peak Memory Usage:  126 MB
```

Vidimo da je ukupno zauzeće 51764 ekvivalentnih gejtova.

```
   The AVERAGE CONNECTION DELAY for this design is:        2.281
   The MAXIMUM PIN DELAY IS:                              10.980
   The AVERAGE CONNECTION DELAY on the 10 WORST NETS is:   9.672

Design statistics:
   Minimum period:  97.781ns (Maximum frequency:  10.227MHz)
   Minimum input required time before clock:  23.484ns
   Minimum output required time after clock:  29.619ns
```

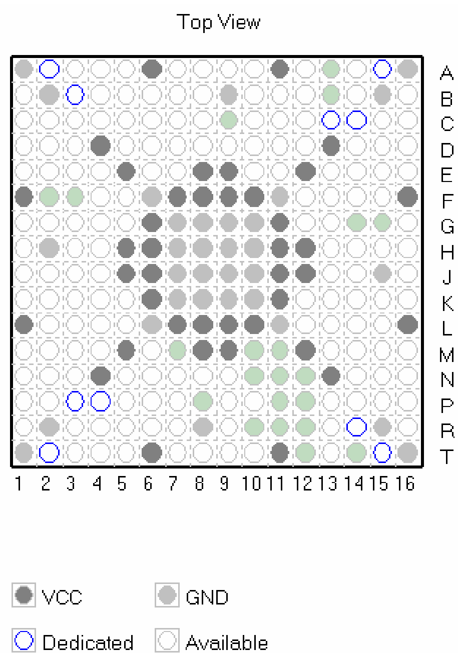
Prosečna potrošnja kola je 308mW pri naponu napajanja od 2.5V.
Pri temperaturi ambijenta od 25°C, temperatura kućišta je isto 25°C. Što znači da ne dolazi do zagrevanja kola pri radu.

Na slici 11. prikazan je izgled unutrašnjeg zauzeća čipa dobiven programom XILINX Floorplanner, koji je sastavni deo razvojnog okruženja XILINX ISE.

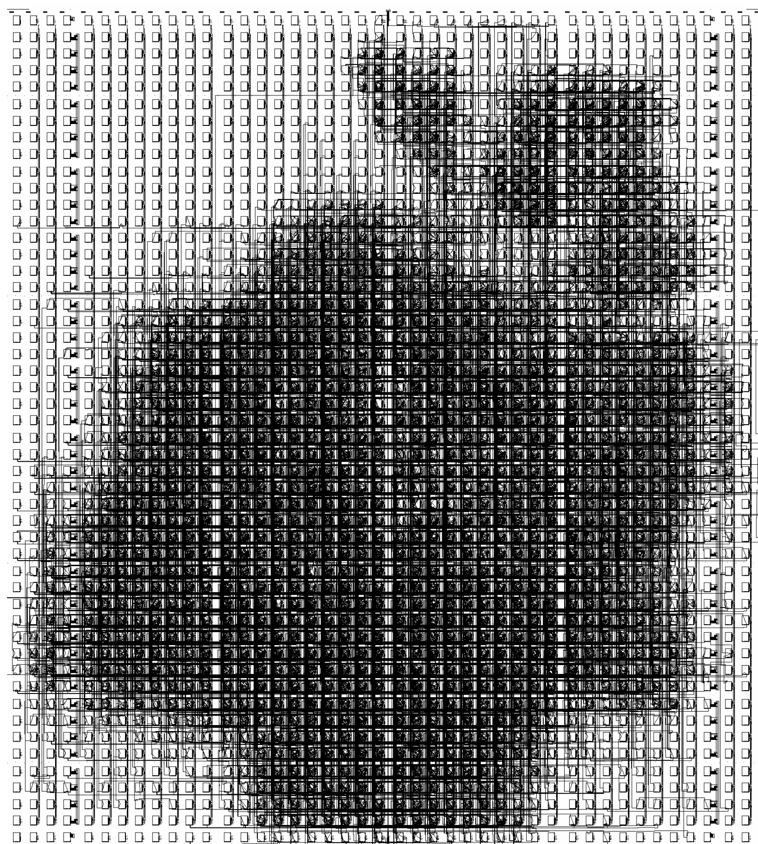


Slika 11. Raspored zauzeća čipa

Na slici 12. prikazan je raspored pinova na kućištu, a na slici 13 je prikazana unutrašnja mreža netova između logičkh ćelija.



Slika 12. Raspored pinova

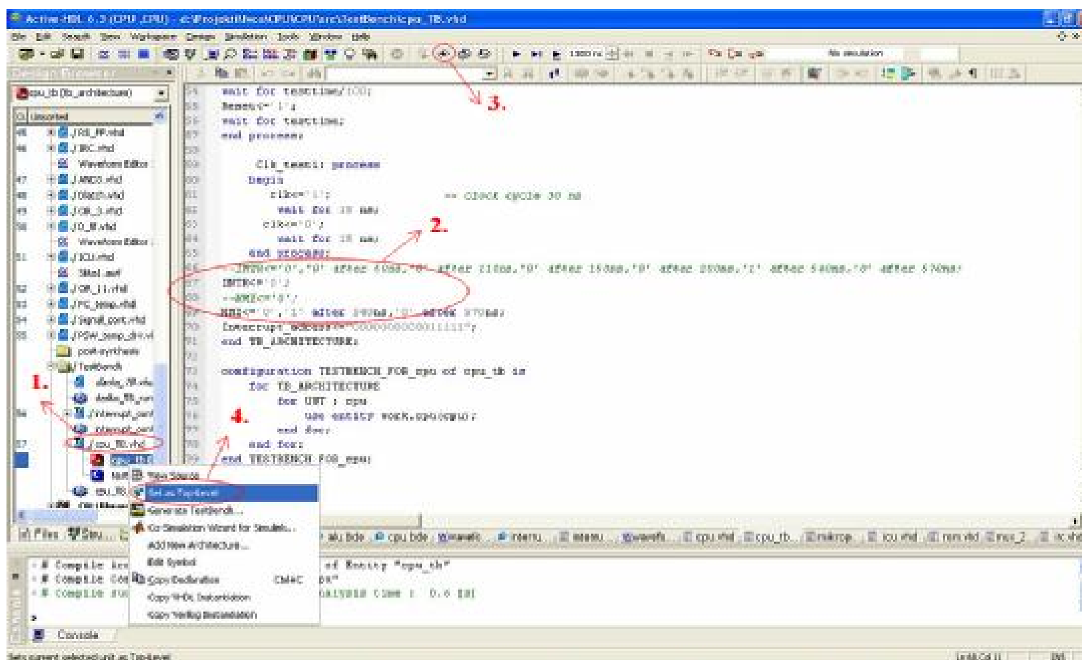


Slika 13. Raspored net-ova

8. Laboratorijska vežba

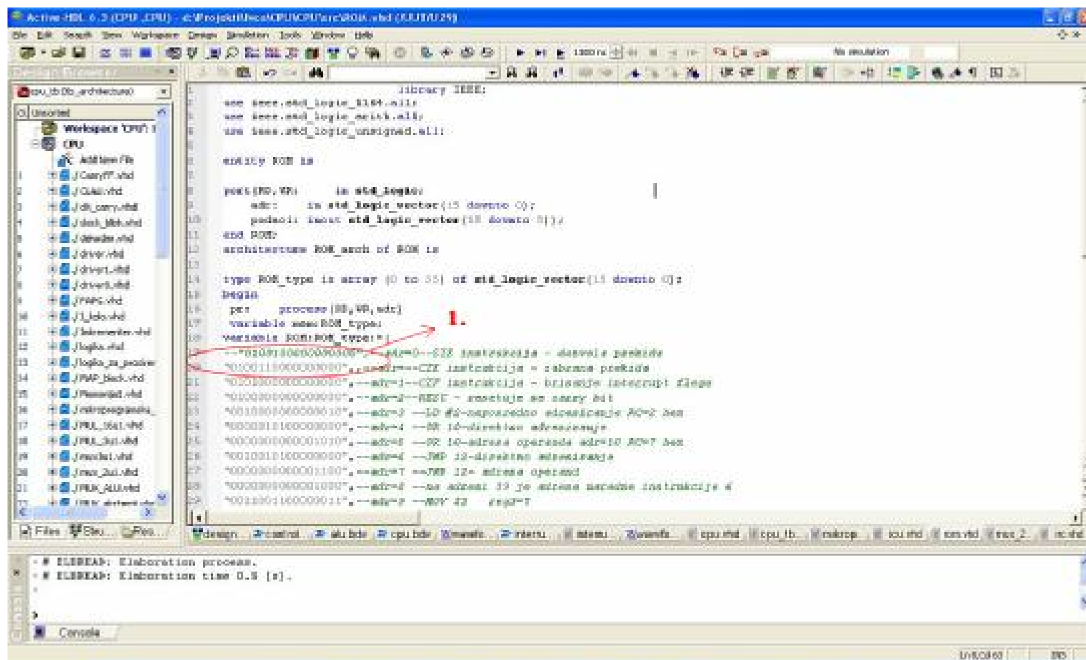
Proučiti režime prihvatanja interapta od strane CPU-a.

1. Pokrenuti program *Active-HDL* čija se ikonica nalazi na *Desktop*-u;
2. Otvoriti dizajn **CPU;**
3. Otvoriti listing *TestBench*-a, **cpu_tb.vhd**, koji se nalazi u *folder*-u *TestBench* **(**segment 1. na slici 14);
4. Promeniti vrednosti za vrednosti signala NMI i INR u zavisnosti od situacije koju posmatramo, da li testiramo maskirajuće ili nemaskirajuće interapte. Ako posmatramo nemaskirajuće interapte potrebno je ostaviti liniju u kojoj je postavljena vrednost INR=0, a za NMI konkretne vrednosti (segment 2. na slici 14). U primeru na slici INTR=0, dok se NMI postavlja na logičku jedinicu u trenutku 540ns. Znači analiziramo nemaskirajući mrekid ;
5. Izvršiti kompajliranje pritiskom na uokvirenu ikonicu (segment 3. na slici 14);
6. Podesiti *Top level* entitet na **cpu_tb.vhd** (segment 4. na slici 14);



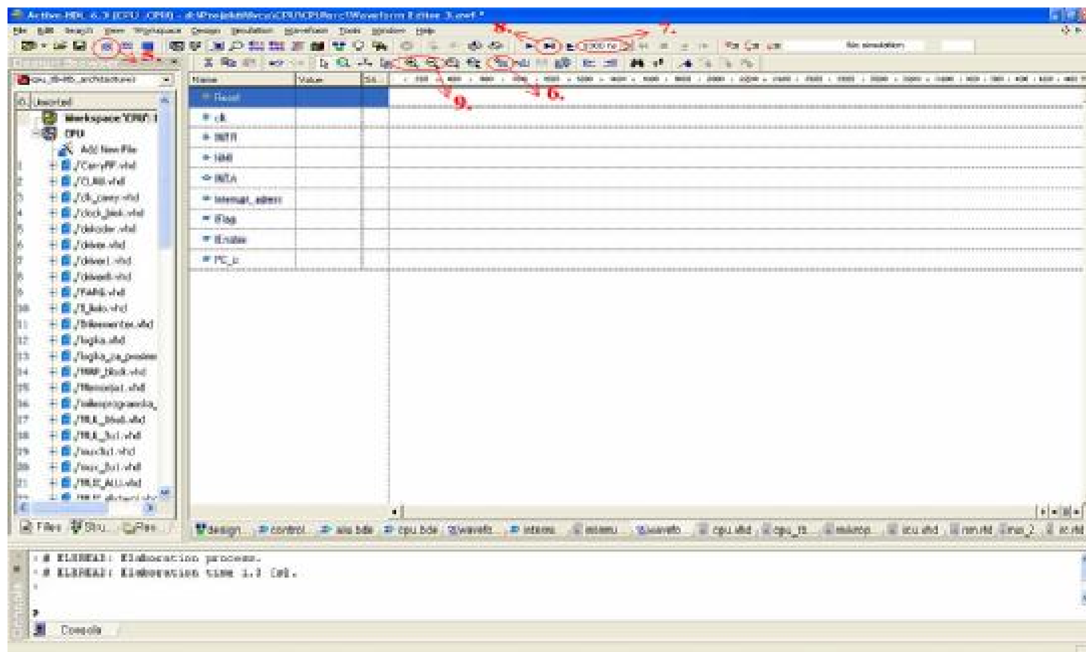
Slika 14. Otvaranje dizajna i upisivanje vrednosti

7. Takođe potrebno je u bloku ROM, gde su smeštene instrukcije, ostaviti nekomentarisanu liniju za adresu 0 u zavisnosti da li želimo da dozvolimo ili zabranimo prekide (segment 1. na slici 15). U našem primeru nekomentarisana je ostavljena druga solucija gde je upisana CIE="0100110000000000" instrukcija, znači zabranjeni su prekidi;



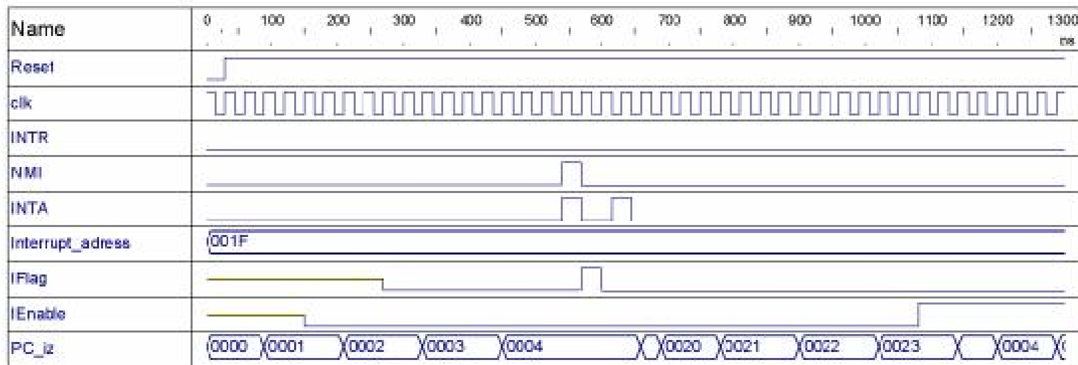
Slika 15. Listing bloka ROM

8. Otvoriti *New Waveform* (segment 5. na slici 16);
9. Dodati signale (segment 6. na slici 16);
10. Podesiti vreme simulacije na 1300ns, (segment 7. na slici 16);
11. Pokrenuti simulaciju pritiskom na uokvirenu ikonicu (segment 8. na slici 16);
12. Po završetku simulacije može biti potrebno da se vremenski dijagram uveća ili umanji radi bolje preglednosti. Komande za uveličavanje i umanjivanje su uokvirene (segment 9. na slici 16);



Slika 16. Waveform editor

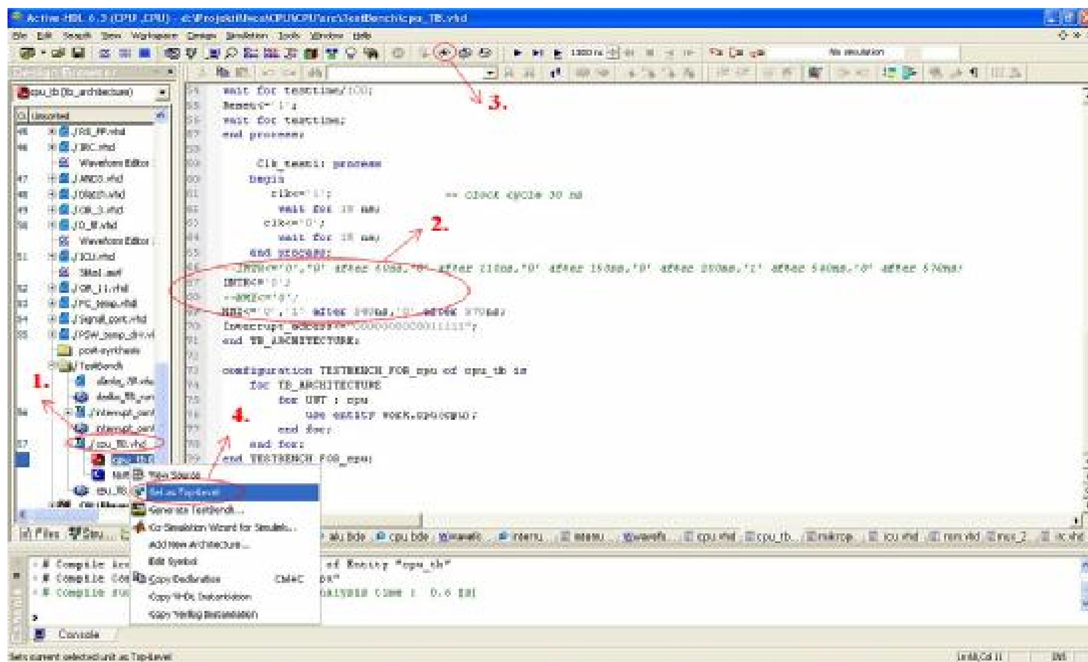
13.Kao rezultat simulacije dobijeni su sledeći vremenski dijagrami (vremenski dijagram na slici 17);



Slika 17. Vremenski dijagram rada kola

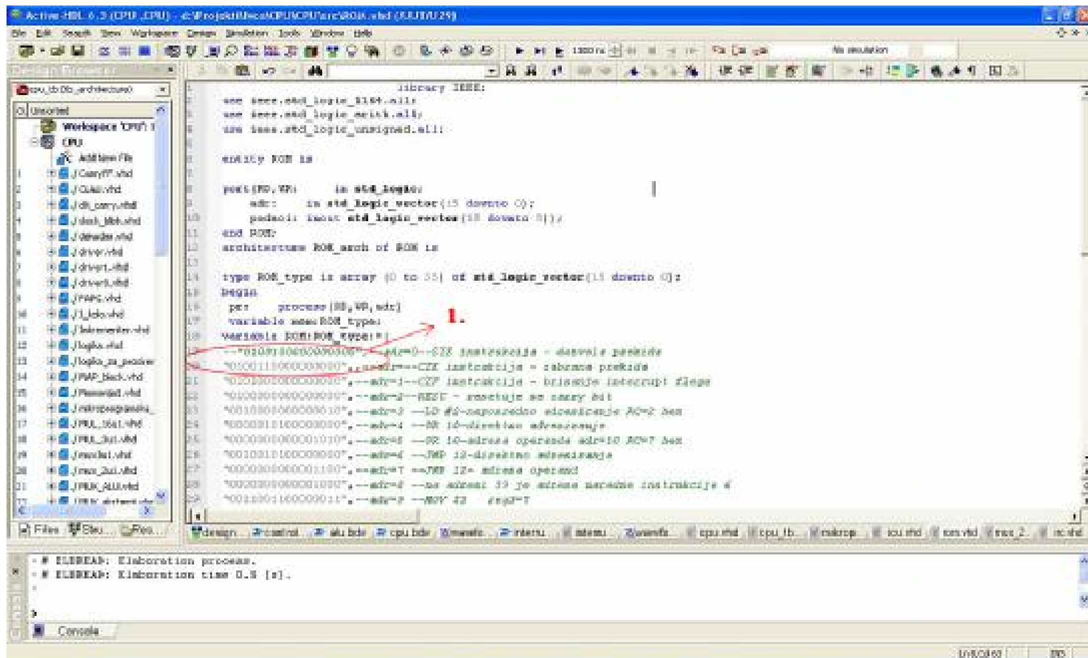
8.1. Zadatak

1. Pokrenuti program *Active-HDL* čija se ikonica nalazi na *Desktop*-u;
2. Otvoriti dizajn **CPU;**
3. Otvoriti listing *TestBench*-a, **cpu_tb.vhd**, koji se nalazi u *folder*-u *TestBench* **(**segment 1. na slici 18);
4. Promeniti vrednosti za vrednosti signala NMI i INR u zavisnosti od situacije koju posmatramo, da li testiramo maskirajuće ili nemaskirajuće interapte (segment 2. na slici 14).
5. Izvršiti kompajliranje pritiskom na uokvirenu ikonicu (segment 3. na slici 18);
6. Podesiti *Top level* entitet na **cpu_tb.vhd** (segment 4. na slici 18);



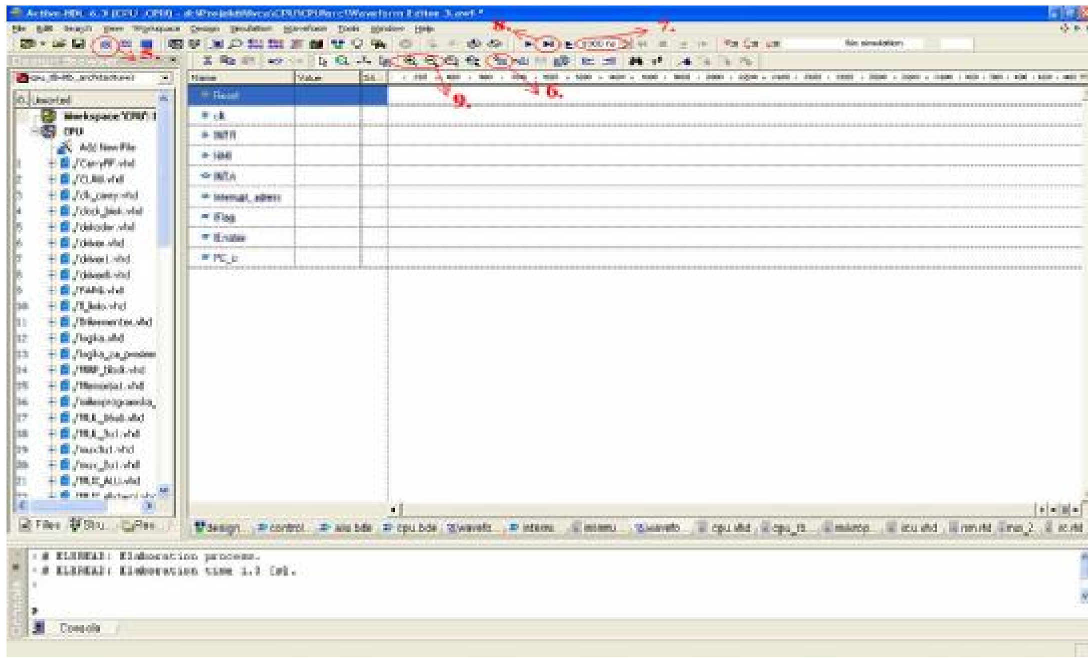
Slika 18. Otvaranje dizajna i upisivanje vrednosti

7. Takođe potrebno je u bloku ROM, gde su smeštene instrukcije, ostaviti nekomentarisanu liniju za adresu 0 (komentari se obeležavaju sa "--") u zavisnosti da li želimo da dozvolimo ili zabranimo prekide (segment 1. na slici 19).



Slika 19. Listing bloka ROM

8. Otvoriti *New Waveform* (segment 5. na slici 20);
9. Dodati signale (segment 6. na slici 20);
10. Podesiti vreme simulacije na 1300ns, (segment 7. na slici 20);
11. Pokrenuti simulaciju pritiskom na uokvirenu ikonicu (segment 8. na slici 20);
12. Po završetku simulacije može biti potrebno da se vremenski dijagram uveća ili umanji radi bolje preglednosti. Komande za uveličavanje i umanjivanje su uokvirene (segment 9. na slici 20);



Slika 20. Waveform editor

13. Skicirati dobijeni dijagram

<u>Grupa 1:</u>

Instrukcija: CIE;
INR = ’0’;
NMI = ’0’, ’1’ u 220nS, ’0’ u 250nS;

<u>Grupa 2:</u>

Instrukcija: CIE;
INR = ’0’;
NMI = ’0’, ’1’ u 150nS, ’0’ u 180nS;

<u>Grupa 3:</u>

Instrukcija: CIE;
INR = ’0’;
NMI = ’0’, ’1’ u 300nS, ’0’ u 310nS;

<u>Grupa 4:</u>

Instrukcija: CIE;
INR = ’0’;
NMI = ’0’, ’1’ u 300nS, ’0’ u 330nS;

<u>Grupa 5:</u>

Instrukcija: SIE;
INR = ’0’;
NMI = ’0’, ’1’ u 220nS, ’0’ u 250nS;

Grupa 6:

Instrukcija: SIE;
INR = ’0’;
NMI = ’0’, ’1’ u 320nS, ’0’ u 350nS;

Grupa 7:

Instrukcija: SIE;
INR = ’0’;
NMI = ’0’, ’1’ u 420nS, ’0’ u 450nS;

Grupa 8:

Instrukcija: SIE;
INR = ’0’;
NMI = ’0’, ’1’ u 400nS, ’0’ u 410nS;

Grupa 9:

Instrukcija: CIE;
INR = ’0’, ’1’ u 150nS, ’0’ u 170nS;
NMI = ’0’;

Grupa 10:

Instrukcija: CIE;
INR = ’0’, ’1’ u 180nS, ’0’ u 200nS;
NMI = ’0’;

Grupa 11:

Instrukcija: CIE;
INR = ’0’, ’1’ u 420nS, ’0’ u 450nS;
NMI = ’0’;

Grupa 12:

Instrukcija: CIE;
INR = ’0’, ’1’ u 500nS, ’0’ u 520nS;
NMI = ’0’;

Grupa 13:

Instrukcija: SIE;
INR = ’0’, ’1’ u 600nS, ’0’ u 620nS;
NMI = ’0’;

Grupa 14:

Instrukcija: SIE;
INR = ’0’, ’1’ u 200nS, ’0’ u 250nS;
NMI = ’0’;

Grupa 15:

Instrukcija: SIE;
INR = ’0’, ’1’ u 320nS, ’0’ u 330nS;
NMI = ’0’;

Grupa 16:

Instrukcija: SIE;
INR = ’0’, ’1’ u 420nS, ’0’ u 450nS;
NMI = ’0’;

9. Zaključak

Kao što smo mogli videti iz predočenog, mehanizam prekida nam omogućava jednostavnije opsluživanje spoljnjih uređaja i događaja. Ovo naročito dolazi do izražaja kod situacija gde procesor veći deo svog vremena izvršava glavni program, tako da svaka potreba za neprestanim nadgledanjem "pooling" ulaznih linija bi predstavljala guljenje dragocenog vremena. Mehanizam prekida se javlja kao jedno vrlo elegantno rešenje za kontakt procesora sa spoljašnjim svetom. U pojedinim situacijama kada se u glavnom programu izvršava neka bitna operacija moguće je zabraniti prihvat spoljašnjih prekida, isto tako preko ulaza za nemaskirajuće prekide prihvata se prekid čije neopsluživanje nije poželjno. Tu se obično radi o prekidima koji se javljaju kao rezultat nekog otkaza u sistemu, prekida komunikacije neke jedinice, nestanka napajanja, ... itd. Glavna osobina prekida je da nakon povratka iz prekidne rutine stanja ključnih registara u procesoru (programski brojač i PSW – statusni registar) dobijaju vrednosti koje se su imali pre ulaska u prekidnu rutinu. Time je omogućeno da po povratku iz prekidne rutine procesor nastavi sa daljim izvršavanjem glavnog programa od tačke gde je stao pre pojave zahteva za prekidom.

10. Literatura

1. Mile Stojčev, *RISC,CISC i DSP procesori*, Niš, 1997.

2. Jovan Đorđević, *Arhitektura i organizacija računara*, Beograd, 2006.

3. John D. Carpinelli, *Computer Systems Organization and Architecture,* Addison Wesley Longman, 1999.

4. Actel Corporation, *Actel HDL Coding Style Guide,* 2002.

5. Douglas L. Perry, *VHDL Programming by Example*, McGraw-Hill, 2000.

6. Peter J. Ashenden, *The VHDL Cookbook*, University of Adelaide, 1990.

7. Enoch O. Hwang, *Digital Logic and Microprocessor Design With VHDL*, La Sierra University, 2005.

8. Ben Cohen, *VHDL Coding Styles and Methodologies,* Kluwer Academic Publishers, 1995.