Elektrotehnički fakultet u Beogradu

Katedra za eletroniku

Digitalna elektronika

Laboratorijska vežba

Projektovanje hardvera na programabilnim komponentama korišćenjem myHDL programskog paketa

Beograd, 2014.

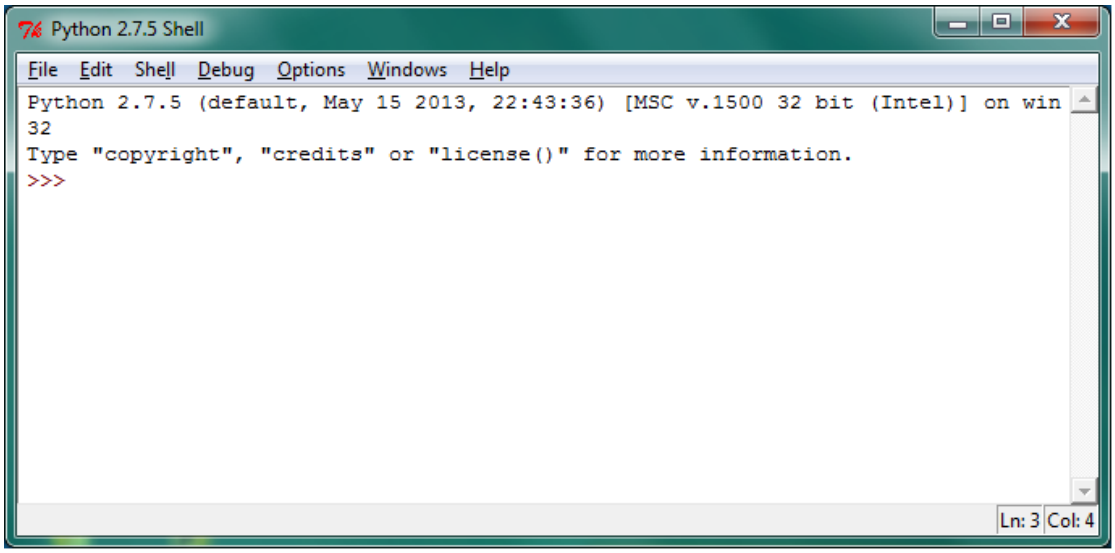
1. Cilj vežbe

Cilj ove vežbe je da se studenti upoznaju sa osnovnim konceptima projektovanja hardvera na programabilnim komponentama. U tu svrhu se koristi programski jezik visokog nivoa *Python* sa odgovarajućom *myHDL* ekstenzijom koja omogućava efikasno opisivanje i simulaciju hardverskih modula.

U okviru vežbe potrebno je da studenti kreiraju nekoliko kombinacionih i sekvencijalnih modula korišćenjem *myHDL* programskog paketa. Svaki od modula je potrebno simulirati i na taj način verifikovati ispravnost rada. Na kraju se svi moduli spajaju u jedan jedinstveni sistem koji se sintetiše i spušta na ploču sa FPGA čipom.

2. Korišćenje myHDL programskog paketa

Programski paket *myHDL* je ekstenzija *Python*-a koja omogućava jednostavno projektovanje i simulaciju hardverskih modula. Projektovanje se dakle svodi na pisanje *Python* programa uz korišćenje pogodnosti *myHDL* paketa. Za početak rada potrebno je pokrenuti IDLE (Python GUI) čime se otvara prozor prikazan na slici 1.



Slika 1. *Python GUI shell* prozor

Python se interpretira (nije potrebno kompajlirati kod pre izvršavanja) tako da se komande mogu direktno unositi u *shell* koji ih izvršava i vraća odgovarajući rezultat. Kako bi programski moduli ostali sačuvani potrebno je otvoriti novi prozor biranjem opcije *File→New Window*. Svaki modul je potrebno sačuvati pod odgovarajućim imenom sa ekstenzijom .py. Pokretanje programa opisanog u nekom modulu se obavlja biranjem opcije *Run→Run Module* iz prozora u kom se nalazi opis modula.

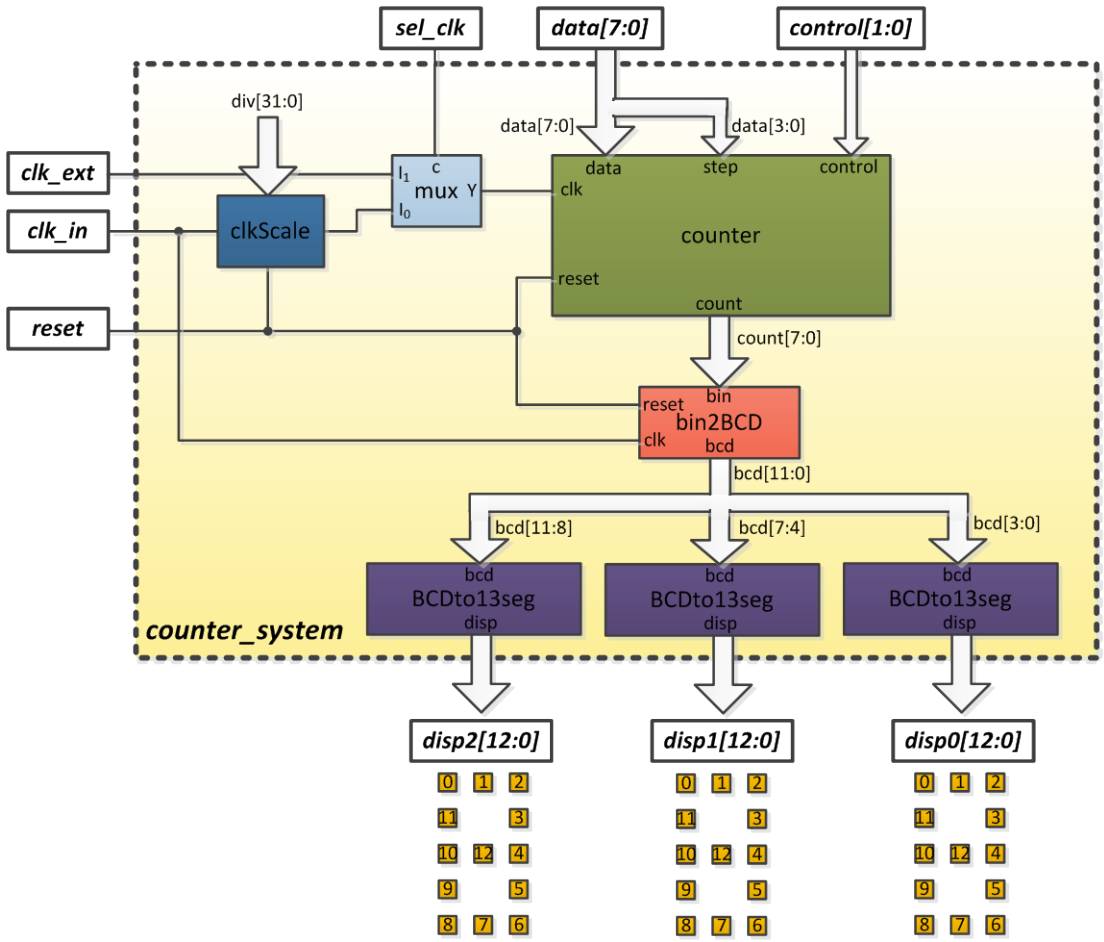
Kako bi se omogućile pogodnosi *myHDL* paketa potrebno je u zaglavlju svakog modula uključiti sledeću liniju koda:

from myhdl import *

Sve fajlove tokom ove laboratorijske vežbe čuvati u direktorijumu E:\OE3DE\Lab1.

3. Zadatak vežbe

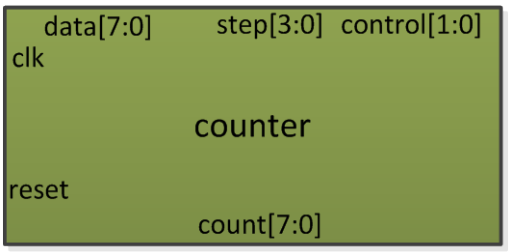
Potrebno je projektovati digitalni sistem univerzalnog obostranog brojača sa programabilnim korakom brojanja i mogućnošću sinhronog paralelnog upisa. Trenutno stanje brojača je potrebno prikazati u dekadnom formatu pri čemu se svaka cifra prikazuje na 13-segmentnom displeju. Potrebno je obezbediti mogućnost skaliranja signala takta kao i dovođenja signala takta iz eksternog izvora. Blok šema opisanog sistema data je na slici 2.



Slika 2. Blok šema sistema univerzalnog brojača

3.1. Projektovanje modula univerzalnog brojača

Blok šema modula univerzalnog brojača prikazana je na slici 3.



Slika 3. Blok šema modula univerzalnog brojača

Brojač ima mogućnost brojanja unapred ili unazad u zavisnosti od trenutne vrednosti kontrolnog signala. U slučaju da je kontrolni signal jednak 00 u brojač se sinhrono upisuje vrednost koja se nalazi na ulazu ***data*** modula brojača. Potrebno je obezbediti mogućnost promene koraka brojanja. Novi korak brojanja se čita sa ***step*** ulaza modla brojača u slučaju da je trenutna vrednost kontrolnog signala jednaka 01. Značenje kontrolnih signala je sumirano u Tabeli 1.

Tabela 1. Opis rada univerzalnog brojača

control[1:0]	action
00	**LoadData: count ← data**
01	**LoadStep: count_step ← step**
10	**CountUp: count ← count + count_step - za svaku uzlaznu ivicu signala clk**
11	**CountUp: count ← count – count_step – za svaku uzlaznu ivicu signala clk**

Projektovati brojač tako da bude 8-bitni, s tim što se korak brojanja može menjati u opsegu od 0 do 15 tako da se može smestiti u 4-bitni podatak.

Komponenta u *myHDL* se opisuje pisanjem odgovarajuće funkcije pri čemu argumenti funkcije odgovaraju portovima projektovanog modula. Za modul univerzalnog brojača čija je blok šema data na slici 3. interfejs komponente izgleda ovako:

```
def counter(reset, clk, control, data, step, count):
    opis funkcionalnosti
```

Kako je u pitanju sekvencijalna mreža (promene su sinhronizovane sa uzlaznom ivicom signala takta) potrebno je dodati sekvencijalni proces kao opis funkcionalnosti modula univerzalnog brojača. Obrazac za sekvencijalne procese koji rade na uzlaznu ivicu signala takta dat je kao:

```
@always_seq(clk.posedge, reset=reset)
def counter_logic():
    opis logike brojača iz tabele 1
```

Deskriptor *@always_seq* označava da će se funkcija opisana ispod njega izvršavati na uzlaznu ivicu signala takta (signal takta je signal *clk*) i da će se po aktiviranju signala *reset* svi signali postaviti na svoje početne vrednosti koje su im zadate pri deklaraciji. Deskriptori sa naznakom *@always* obezbeđuju i paralelno izvršavanje svih komponenti u dizajnu (kao što se to dešava u hardverskom sistemu).

Korak brojača je interni signal koji preuzima vrednost sa ulaza *step* samo u slučaju da je vrednost kontrolnog ulaza 01. Prema tome u okviru komponente *counter* treba definisati interni signal *step_int* koji se koristi pri radu brojača. Definisanje četvorobitnog signala dato je sa:

```
step_int = Signal(intbv(1)[4:])
```

Vrednost 1 unutar zagrade označava početnu vrednost koraka, odnosno vrednost koja će se postaviti pri pojavi aktivne vrednosti signala *reset*. Obratiti pažnju na opseg definisan u uglastim zagradama. *Python* standard podrazumeva poluotvorene intervale. Dakle ako se specificira interval *[max:min]* onda on uključuje brojeve od *max-1* do *min*. Izostavljanje minimalne vrednosti znači da su uključeni svi biti niže težine.

Kako je brojač osmobitni potrebno je obezbediti da se u slučaju da pređe maksimalnu vrednost (255) prevrti. Ovo se može postići eksplicitnom kontrolom vrednosti prilikom inkrementiranja ili dekrementiranja brojača. Dosta elegantnije rešenje je korišćenje tipa podataka *modbv* kome se opseg zadaje prilikom definicije dok su sve kontrole prekoračenja automatski ugrađene. Primer deklaracije *modbv* signala:

```
count_int = Signal(modbv(0, min = 0, max = 2**8))
```

Kako je potrebno menjati trenutnu vrednost brojača (inkrementiranje, dekrementiranje) koja je ujedno i izlazni port modula brojača preporučuje se uvođenje internog signala ***count_int*** koji bi se koristio za interna izračunavanja. Na taj način se izbegava da se izlazni signal očitava unutar dizajna, što se smatra lošom praksom i zabranjeno je u većini jezika za opis hardvera. Da bi se interna vrednost brojača izbacila na izlazni port potrebno je uvesti kombinacioni proces kojim se ovo obezbeđuje. Kombinacioni proces predstavlja model kombinacione mreže koja menja izlazno stanje čim se promeni neki od ulaza (ne zavisi od nekog sinhronizacionog signala kao što je takt). Obrazac za kombinacione procese dat je kao:

```
@always_comb
def comb_logic():
    opis kombinacione mreže
```

Kombinacioni proces koji obezbeđuje prosleđivanje internog signala brojača na izlaz:

```
@always_comb
def out_logic():
    count.next = count_int
```

Kao što se vidi iz prethodnog primera dodeljivanje vredosti signalu se obavlja postavljanjem odgovarajuće vrednosti u polje *next* tog signala.

Na kraju opisa komponente je potrebno vratiti sve generisane module naredbom *return*. Struktura traženog modula sa sve povratnom naredbom data je kao:

```
from myhdl import *

def counter(reset, clk, control, data, step, count):

    step_int = Signal(intbv(1)[4:])
    count_int = Signal(modbv(0, min = 0, max = 2**8))

    @always_seq(clk.posedge, reset=reset)
    def counter_logic():
        opis logike brojača

    @always_comb
     def output_logic():
        count.next = count_int

    return counter_logic, output_logic
```

Popuniti logiku koja opisuje rad brojača i sačuvati modul pod imenom ***counter.py.***

Da bi testirali modul projektovanog brojača potrebno je definisati test funkciju. Test se opisuje kao i svaki drugi modul, jedino što po pravilu nema argumenata, pošto se svi ulazni signali generišu unutar test funkcije i svi izlazni signali posmatraju unutar test funkcije.

Signal *reset*-a je poseban tip signala pošto se koristi za automatsko postavljanje svih promenljivih unutar sekvencijalnih procesa na njihove početne vrednosti. Kako bi ponašanje *reset* signala bilo potpuno opisano potrebno je definisati početnu vrednost, aktivnu vrednost i tip *reset*-a (sinhroni ili asinhroni). Primer definisanja asinhronog reseta aktivnog u logičkoj nuli dat je u kodu dole:

```
reset = ResetSignal(1, active = 0, async = True)
```

Da bi se testirana komponenta uključila u test funkciju potrebno je pozvati funkciju u kojoj je opisana. Svaki poziv funkcije u kojoj je opisana komponenta stvara novu instancu te komponente. Primer instanciranja komponente brojača dat je u kodu dole:

```
cnt_inst = counter(reset, clk, control, data, step, count)
```

Nakon toga je potrebno generisati ulazne signale. Signal takta predstavlja poseban sinhronizacioni signal koji se menja u pravlinim vremenskim intervalima. Ovaj signal ćemo definisati u posebnom procesu koji se aktivira u pravilnim vremenskim intervalima. U tu svrhu koristi se funkcija sa deskriptorom *@always(uslov)* koja se izvršava svaki put kada se promeni signal definisan uslovom. Funkcija koja se generiše periodične događaje je funkcija *delay(period)* sa argumentom koji definiše

periodu generisanja događaja. Primer generisanja signala takta sa periodom 2 dat je u kodu dole:

```
half_clk_period = 1
@always(delay(half_clk_period))
def clk_gen():
    clk.next = not clk
```

Obratite pažnju da će perioda generisanog takta biti dvostruko veća od periode *delay* funkcije, pošto se po isteku periode *delay* funkcije signal takta komplementira tako da tek posle 2 periode *delay* funkcije dolazi u početno stanje, tako da to predstavlja period signala takta.

Ostatak pobudnih signala je najlakše definisati u posebnom procesu u kom se eksplicitno definišu trenuci promene signala. Dakle taj proces ne treba da se izvršava periodično već je potrebno da se izvrši samo jednom pri čemu je njegovo trajanje jednako trajanju simulacije. U slučajevima kada je potrebno eksplicitno definisati trenutke promene signala koriste se funkcije sa deskriptorom *@instance*. Unutar funkcije dodeljuju se vrednosti signalima i pozivom instrukcije *yield uslov* suspenduje se izvršavanje funkcije do generisanja uslova. Ako se instrukcija *yield* iskombinuje sa instrukcijom *delay* onda se test funkcija suspenduje za određeni vremenski interval koji je argument *delay* funkcije. Primer koda koji definiše pobudu signala *reset* tako da je na logičkoj nuli za vreme prve periode takta a potom prelazi u logičku jedinicu dat je u kodu dole:

```
@instance
def stimulus():
    reset.next = 0
    yield delay(clk_period)
    reset.next = 1
    yield delay(5*clk_period)

    raise StopSimulation
```

Kraj simulacije se označava instrukcijom *raise StopSimulation* i tada se prekida generisanje ulaznih signala.

Unutar tela test funkcije je potrebno definisati sve signale koji se koriste. Primer test funkcije za modul brojača data je u kodu dole:

```
from myhdl import *
from counter import counter

def test_counter():

    clk = Signal(bool(0))
    reset = ResetSignal(1, active = 0, async = True)
    control = Signal(intbv(0)[2:])
    data = Signal(intbv(0)[8:])
    step = Signal(intbv(1)[4:])

    count = Signal(modbv(0, min = 0, max = 2**8))

    cnt_inst = counter(reset, clk, control, data, step, count)

    half_clk_period = 1
    clk_period = 2*half_clk_period

    @always (delay(half_clk_period))
    def clk_gen():
        clk.next = not clk

    @instance
     def stimulus():
        reset.next = 0
        yield delay(clk_period)
        reset.next = 1
        yield delay(clk_period)

        ostali signali pobude

        raise StopSimulation

    return cnt_inst, clk_gen, stimulus
```

Po definisanju test funkcije potrebno je kreirati objekat simulacije i pokrenuti izvršavanje simulacije. Takođe bilo bi dobro da se vrednosti svih signala u svakom trenutku vremena pamte u nekom eksternom fajlu. Ovo se postiže pozivanjem funkcije *traceSignals* čiji je argument test funkcija koja je opisana ranije. Povratni argument je objekat koji se koristi dalje pri simulaciji. Za kreiranje simulacije poziva se instrukcija *Simulation* koja generiše simulacioni objekat. Simulacioni objekat implementira metod *run* čijim pozivanjem se startuje simulacija. Kod za kreiranje i pokretanje simulacije se dodaje u okviru istog fajla u kome je opisana funkcija za testiranje. Primer koda za kreiranje i pokretanje simulacije brojača dat je dole:

```
tb = traceSignals(test_counter)
Simulation(tb).run()
```

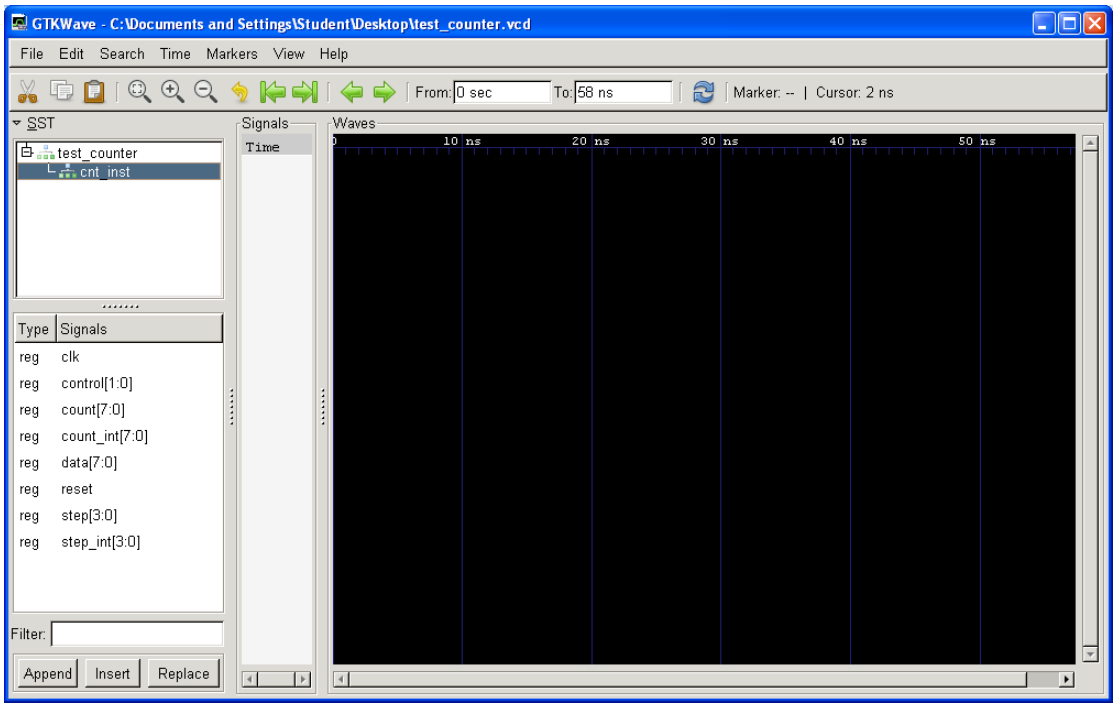
Ako se metoda *run* pozove bez argumenata kao što je prikazano na primeru onda se simulacija izvršava sve do instrukcije *raise StopSimulation*. Ovu metodu je takođe

moguće pozvati sa argumentom koji predstavlja vreme trajanja simulacije. Na primer instrukcijom ***Simulation(tb).run(10)*** se pokreće simulacija koja traje 10 jedinica vremena.

Po definisanju svih signala pobude potrebno je sačuvati test funkciju u fajl pod nazivom ***test_counter.py***. Simulacija se pokreće izvršavanjem koda biranjem opcije *Run→Run Module* iz prozora u kom se nalazi opis modula.

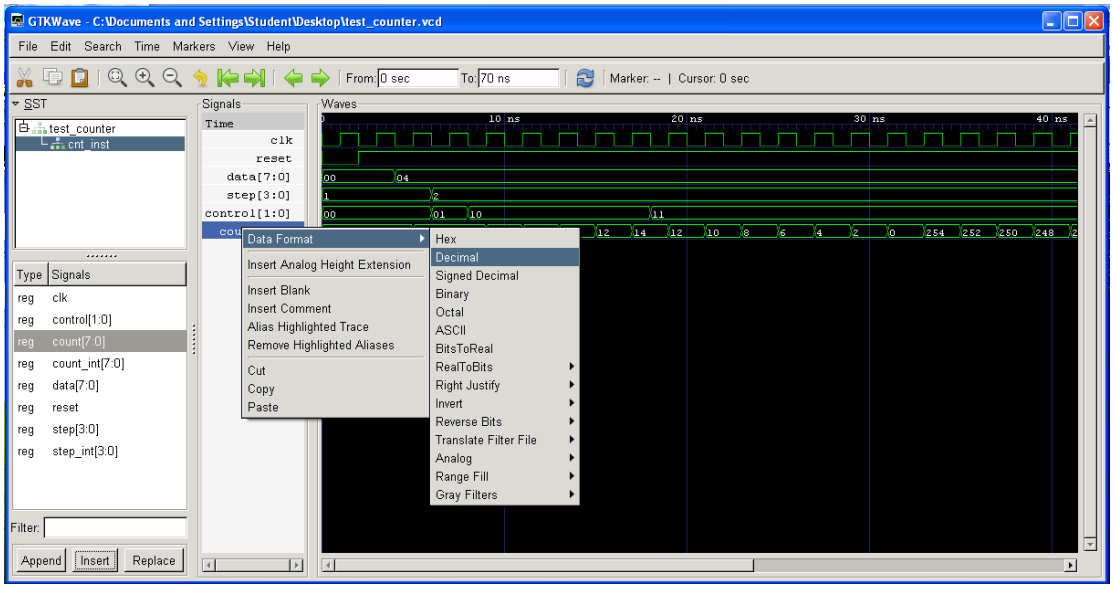
Ako nema grešaka kod će se u potpunosti izvršiti i u direktorijumu gde se nalazi kod pojaviće se fajl ***test_counter.vcd***. Ovaj fajl sadrži sve promene ulaznih i izlaznih signala tokom simulacije (*vcd – value change dump*).

Za prikaz rezultata simulacije se koristi *GTKWave* aplikacija čija prečica se nalazi na *Desktop*-u. Pokretanjem se otvara novi prozor. Prevlačenjem vcd fajla unutar ovog prozora učitavaju se signali koji su posmatrani i njihov spisak se nalazi u odeljku *Signals* kao što je prikazano na slici 4.



Slika 4. Spisak signala posmatranih tokom testiranja učitanih u GTKWave

Selektovanjem nekog signala sa liste i pritiskom na taster *Insert* prikazuje se vremenski oblik selektovanog signala u glavnom prozoru. Format prikaza može se promeniti desnim klikom na signal koji se prikazuje i izborom opcije *Data Format* kao što je prikazano na slici 5.



Slika 5. Prikaz signala i podešavanje formata

<u>Pozvati dežurnog asistenta da verifikuje uspešno kompletiranje ovog dela vežbe!</u>

3.2. Projektovanje modula za konvertovanje binarnog broja u BCD

Da bi se omogućilo prikazivanje izlaza brojača na displeju u dekadnom formatu potrebno je konvertovati izlaz brojača iz binarnog u BCD format. Algortiam konverzije se sastoji od sledećih koraka:

1) Ako je bilo koja cifra (stotine, desetice, jedinice) veća od 4 dodati 3 na tu cifru
2) Pomeriti sve bite za jedno mesto u levo
3) Ako je već obavljeno 8 pomeranja proces konverzije je završen
4) Ponoviti korak 1.

Primer konverzije broja 178 (10110010) iz binarnog u BCD korišćenjem gore opisanog algoritma je prikazan na slici 6.

shift_cnt	*shift_reg*				
7	0000	0000	0000	10110010	
6	0000	0000	0001	01100100	
5	0000	0000	0010	11001000	
4	0000	0000	0101	10010000	
4	0000	0000	1000	10010000	*korekcija!*
3	0000	0001	0001	00100000	
2	0000	0010	0010	01000000	
1	0000	0100	0100	10000000	
0	0000	1000	1001	00000000	
0	0000	1011	1100	00000000	*korekcija!*
0	0001	0111	1000	00000000	
	1	**7**	**8**		

Slika 6. Primer konverzije iz binarnog u BCD format

Opisani algoritam se može jednostavno implementirati korišćenje konačne mašine stanja. Iz algoritma se jasno izdvajaju 3 stanja: početno stanje (*Idle*), stanje pomeranja (*Shift*) i stanje korekcije, odnosno sabiranja (*Add*).

Za definisanje tipa stanja koristiti *enum* tip podataka. Primer korišćenja *enum* tipa:

state_t = enum('Idle', 'Shift', 'Add')
state = Signal(state_t.Idle)

Mašina stanja se nalazi u početnom stanju dok god je ulazni binarni broj jednak broju upisanom u interni registar koji predstavlja binarnu vrednost trenutnog *bcd* izlaza. U trenutku kada se detektuje da se binarni broj koji se nalazi u internom registru i ulaz razlikuju znači da izlaz više nije validan pa je potrebno započeti novi proces konverzije inicijalizovanjem svih internih signala i prelaskom u stanje pomeranja.

U stanju pomeranja je potrebno proveriti da li je do sada obavljeno 7 pomeranja. Ako jeste obavlja se još 1 pomeranje čime je proces konverzije završen i prelazi se u početno stanje. U suprotnom obavlja se pomeranje, brojač preostalih pomeranja se umanjuje za 1 i prelazi se u stanje korekcije. Pomeranje se najefikasnije realizuje

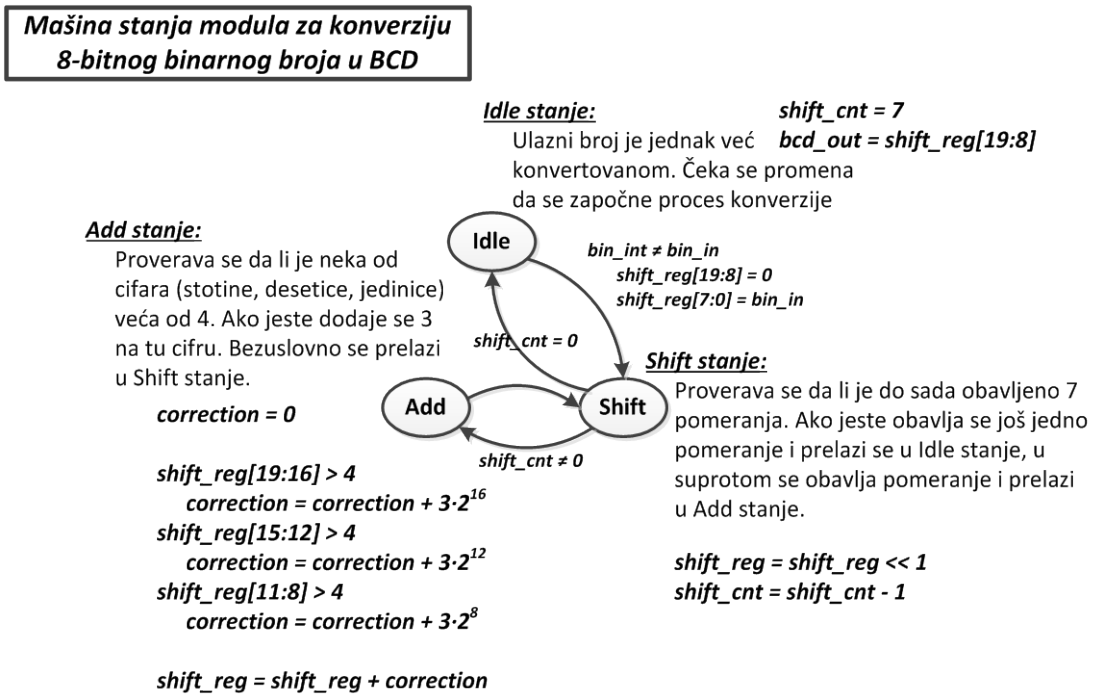
pomoću operatora za konkatenaciju. Primer pomeranja za 1 mesto u levo dat je u kodu dole:

shift_reg.next = concat(shift_reg[19:0], intbv(0)[1:])

U stanju korekcije proverava se vrednost svake od cifara i dodaje odgovaraju i dodaje odgovarajući faktor na korekcioni član. Korekcioni član je potrebno definisati unutar samog stanja korekcije pošto on predstavlja promenljivu lokalnog karaktera.
Stepenovanje se u Pythonu opisuje operatorom **. Primer korišćenja ovog operatora dat je u kodu dole:

correction = correction + 3*(2**16)

Arhitektura gore opisane mašine stanja prikazana je na slici 7.



Slika 7. Mašina stanja kojom se realizuje konverzija 8-bitnog binarnog broja u BCD

Realizovati opisanu mašinu stanja unutar posebnog modula i sačuvati je kao ***bin2bcd.py***.

Napisati test za realizovanu mašinu stanja i sačuvati ga kao ***test_bin2bcd.py***.

Prikazati vremenske dijagrame signala projektovanog modula pomoću *GTKWave*-a i verifikovati ispravno funkcionisanje.

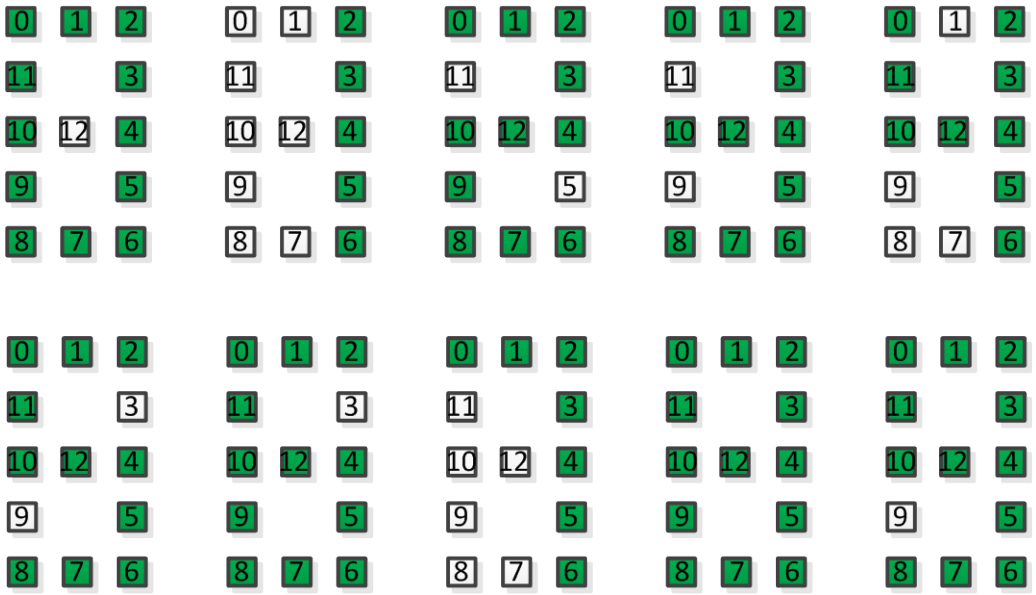
Pozvati dežurnog asistenta da verifikuje uspešno kompletiranje ovog dela vežbe!

3.3. Projektovanje modula za prikaz BCD cifre na 13-segmentnom displeju

Trenutno stanje brojača se prikazuje na 13-segmentnom LED displeju (za prikaz svake cifre se koristi 13 dioda). Potrebno je projektovati komponentu koja ulaznu BCD cifru konvertuje u signale za pobudu jedne cifre LED displeja. Blok šema tražene komponente data je na slici 8.

Slika 8. Blok šema komponente za konvertovanje iz BCD-a u 13 segmenata

Način mapiranja signala *disp* na diode prikazan je na slici 9. Sa slike se vidi da je za prikaz cifre 1 na displeju potrebno postaviti podatak 0000001111100 na izlaz *disp* projektovane komponente.



Slika 9. Pobuđivanje 13-segmentnog displeja

Realizovati opisanu komponentu unutar posebnog modula i sačuvati je kao ***BCDto13seg.py***.

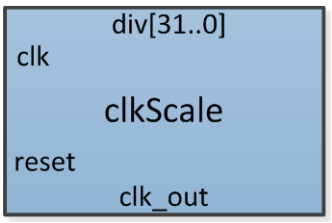
Napisati test za realizovanu komponentu i sačuvati ga kao ***test_ BCDto13seg.py***.

Prikazati vremenske dijagrame signala projektovanog modula pomoću *GTKWave*-a i verifikovati ispravno funkcionisanje.

Pozvati dežurnog asistenta da verifikuje uspešno kompletiranje ovog dela vežbe!

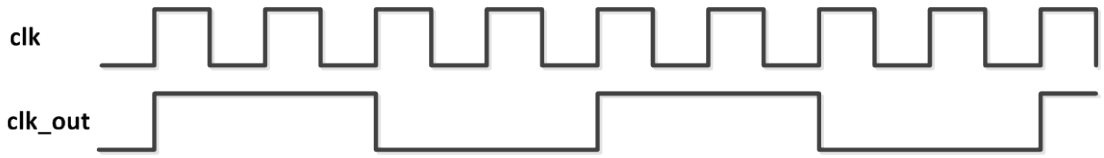
3.4. Projektovanje modula za skaliranje signala takta

Potrebno je realizovati komponentu kojom se omogućava skaliranje ulaznog signala takta. Na izlazu se generiše signal koji ima *2·div* puta manju učestanost od ulaznog signala takta. Blok šema tražene komponente je prikazana na slici 10.



Slika 10. Blok šema komponente za skaliranje signal takta

Na slici 11. je prikazan očekivani rad ove kompononte za slučaj da je *div* ulaz postavljen na 2.



Slika 11. Primer rada modula za skaliranje takta u slučaju da je *div* = 2

Realizovati opisanu komponentu unutar posebnog modula i sačuvati je kao ***clkScale.py***.

Napisati test za realizovanu komponentu i sačuvati ga kao ***test_clkScale.py***.

Prikazati vremenske dijagrame signala projektovanog modula pomoću *GTKWave*-a i verifikovati ispravno funkcionisanje.

Pozvati dežurnog asistenta da verifikuje uspešno kompletiranje ovog dela vežbe!

3.5. Projektovanje sistema univerzalnog brojača

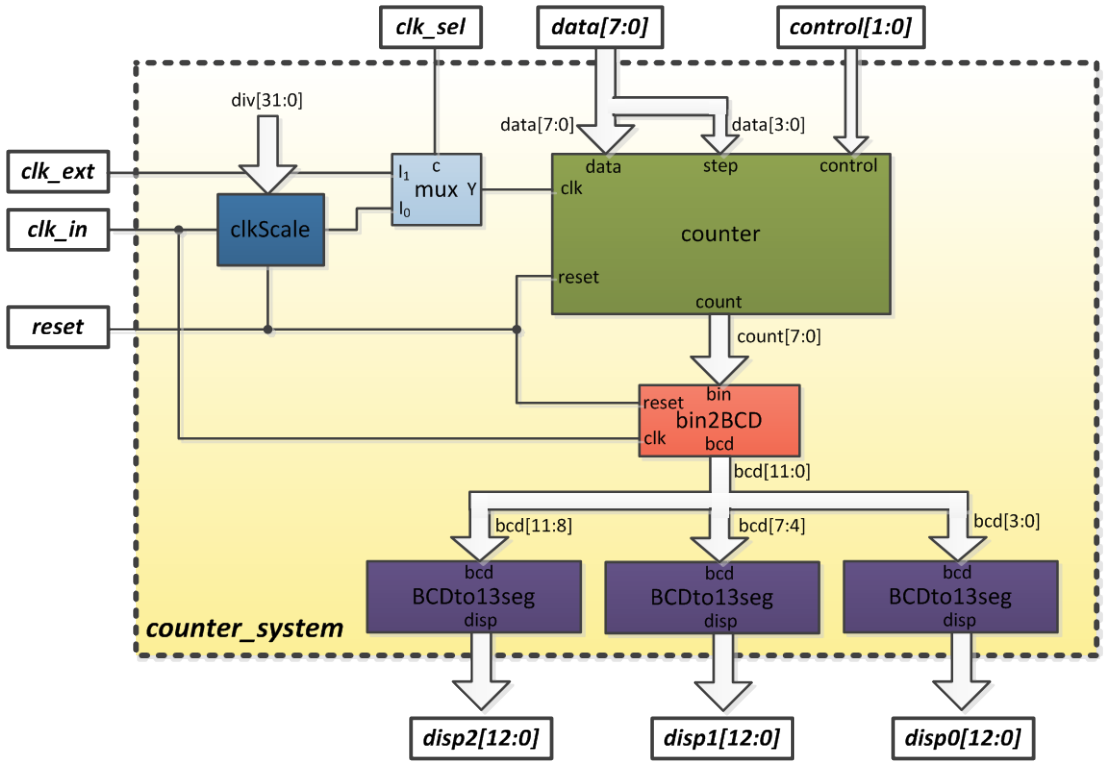
Na kraju je potrebno spojiti sve prethodno projektovane module u sistem univerzalnog brojača čija je blok šema prikazana na slici 12. Potrebno je pobezbediti da brojač ima mogućnost korišćenja eksternog signala takta *clk_ext* kao što je prikazano na slici. Ulazi brojača *data* i *step* se povezuju na istu magistralu podataka s tim što se nižih 4 bita koristi za *step* ulaz brojača dok se svih 8 bita koristi za *data* ulaz. Kako bi se obezbedilo deljenje magistrale podataka potrebno je kreirati novi signal koji će se sastojati samo od 4 niža bita *data* ulaza i njih povezati na *step* ulaz brojača. Ovo se najjednostavnije može postići korišćenjem *_SliceSignal* funkcionalosti. Na taj način se kreira signal koji automatski prati promene odredjenog

dela ulaznog signala. Primer za kreiranje novog signala koji prati promene niža 4 bita ulaznog signala dat je u kodu dole:

step = data(4,0)

Obratite pažnju da su korišćene obične umesto uglastih zagrada što označava da se koristi *_SliceSignal* funkcionalnost.

Instanciranja komponenti se obavljaju pozivom funkcija u kojima su opisani odgovarajući moduli na identičan način kao prilikom testiranja.



Slika 12. Blok šema sistema univerzalnog brojača

Realizovati sistem univerzalnog brojača prema zadatoj blok šemi unutar posebnog modula i sačuvati ga kao ***counter_system.py***.

Napisati test za realizovanu komponentu i sačuvati ga kao ***teset_counter_system.py***.

Prikazati vremenske dijagrame signala projektovanog modula pomoću *GTKWave*-a i verifikovati ispravno funkcionisanje.

Pozvati dežurnog asistenta da verifikuje uspešno kompletiranje ovog dela vežbe!

3.6. Testiranje na razvojnom sistemu

Za testiranje projektovanog sistema se koristi *FPGA4U* razvojni sistem koji na sebi sadrži FPGA čip iz *Cyclone II* familije kompanije *Altera*. Mapiranje odgovarajućih signala univerzalnog brojača na komponente razvojnog sistema je prikazano na slici

13. Da bi se omogućilo testiranje sistema na razvojoj ploči potrebno je najpre iz projektovanog *myHDL* modula sintetisati *VHDL* kod projektovane komonente. *VHDL* je jezik za opis hardvera, nižeg nivoa od *myHDL*. Alati za sintezu mogu iz *VHDL* opisa da sintetišu odgovarajući fajl koji se koristi za programiranje FPGA čipa.

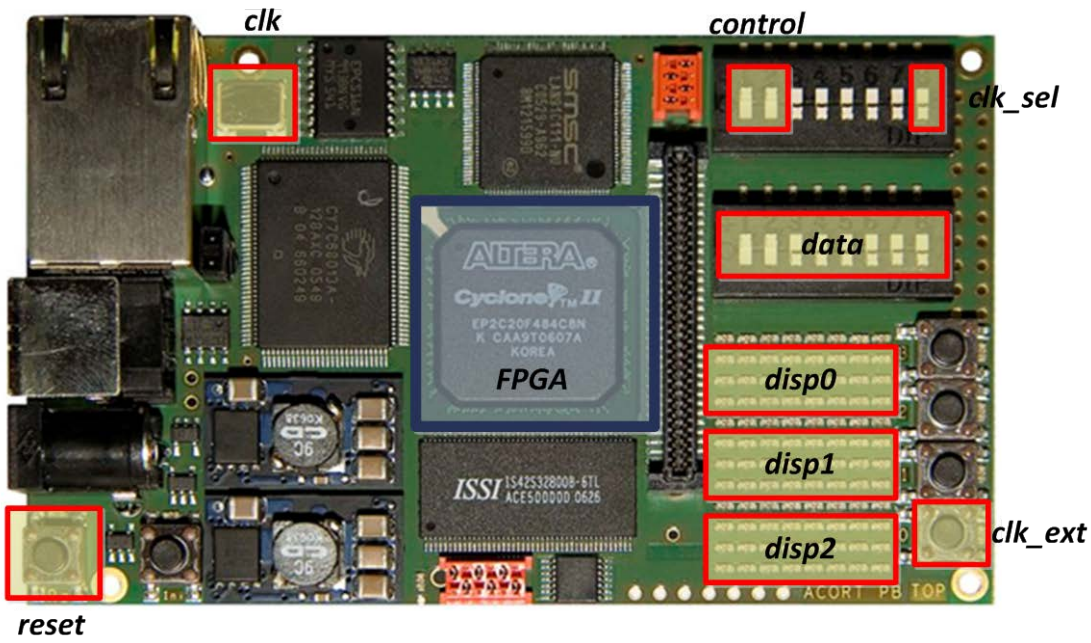
Da bi se sintetisao *VHDL* opis sistema univerzalnog brojača potrebno je u okviru fajla *counter_system.py* na samom kraju (izvan opisa glavne funkcije) definisati sve signale koji se koriste. Vodiiti računa da imena signala i njihova bitska širina odgovaraju opisu sa slike 12.

Na razvojnom sistemu je na raspolaganju takt učestanosti 24 MHz. Upisati odgovarajuću vrednost u *div* signal koja će obezbediti da brojač u slučaju da je *clk_sel = 0* broji sa periodom 1 s.

Generisanje *VHDL* opisa se obavlja pozivom funkcije *toVHDL*. Prvi argument ove funkcije je ime funkcije u kojoj je opisan modul koji se sintetiše dok su ostali argumenti signali

toVHDL(counter_system, reset, clk, clk_ext, clk_sel, control, data, disp2, disp1, disp0)

Pokretanjem modula generiše se *VHDL* opis u fajlu ***counter_system.vhd***.



Slika 12. Mapiranje signala sistema univerzalnog brojača na razvojni sistem

<u>Pozvati dežurnog asistenta da donese razvojni sistem za testiranje.</u>

Po priključenju pločice pokrenuti fajl ***program_fpga.bat*** kojim se pokreće prevođenje VHDL opisa u fajl za programiranje i spušta realizovana konfiguracija na razvojni sistem. Verifikovati ispravan rad sistema univerzalnog brojača na razvojnom sistemu.

<u>Pozvati dežurnog asistenta da verifikuje uspešno kompletiranje ovog dela vežbe!</u>