PROGRAMIRANJE KOMUNIKACIONOG HARDVERA
– Prilog B –

Prilog B

U okviru ovog priloga će biti izložen Verilog programski jezik. Verilog programski jezik spada u grupu HDL jezika kao i VHDL jezik izložen u poglavlju 2. VHDL i Verilog predstavljaju trenutno najpopularnije HDL jezike. Na pitanje koji od ova dva HDL jezika je bolji nema definitivnog odgovora. Sa stanovišta jednostavnosti i efikasnosti sintaksnih konstrukcija nijedan od ova dva jezika nema značajnu prednost. U nekim slučajevima je efikasniji VHDL, a u nekim Verilog sa stanovišta pisanja koda, ali u najvećem broju slučajeva su ova dva jezika potpuno ekvivalentna. U svakom slučaju sve što može da opiše jedan od njih, može da opiše i drugi. Uglavnom, izbor između VHDL i Verilog jezika prvenstveno zavisi od politike kompanije u kojoj inženjer radi tj. u zavisnosti za šta se kompanija opredelila (Verilog ili VHDL), inženjer će morati da koristi izabrani jezik. Ako sam inženjer bira jezik koji će koristiti, onda je izbor uglavnom zasnovan na ličnim preferencama.

Verilog jezik koristi veliki broj sintaksnih konstrukcija iz C programskog jezika što omogućava lako usvajanje Verilog jezika onih inženjera koji poznaju i koriste C programski jezik. Moguće je u okviru složenih projekata koji sadrže velik broj međusobno povezanih entiteta koristiti istovremeno i VHDL i Verilog jezik (neke entitete opisati Verilog jezikom, a neke VHDL jezikom - naravno, u opisu jednog entiteta je moguće koristiti samo jedan od HDL jezika tj. ne može se mešati VHDL i Verilog sintaksa u okviru opisa entiteta), ali takav pristup treba izbegavati ukoliko je to moguće. Entitet opisan jednim jezikom se može instancirati u hijerarhijski višem entitetu opisanim drugim jezikom (jedino na taj način se mogu 'mešati' VHDL i Verilog jezik), pri čemu instanciranje zadovoljava sintaksu jezika korišćenog za opis hijerarhijski višeg entiteta. U nastavku priloga će biti izložena osnovna sintaksna pravila Verilog jezika, pri čemu sve generalne napomene (koje nisu vezane za sintaksu) navedene u poglavlju 2 za VHDL jezik važe i za Verilog jezik. Napomenimo da postoji i proširenje Verilog jezika u vidu SystemVerilog jezika koji dopunjava Verilog jezik svojstvima koja Verilog jezik ne poseduje poput podrške za enumerisane tipove.

B.1. Generalne napomene

Verilog jezik je *case-sensitive* jezik koji koristi velik broj konstrukcija iz C programskog jezika. Komentari u Verilog jeziku su sintaksno identični komentarima iz C programskog jezika. Komentar linije otpočinje sa //, a višelinijski komentar započinje sa /* i završava sa */. Signali u Verilogu mogu da imaju četiri vrednosti: 0, 1, *z* i *x*. Vrednost *z* označava visoku impedansu, a *x* nepoznatu vrednost signala. Konkretne brojne vrednosti koje se dodeljuju signalima mogu da se navedu u formatu specificirane i nespecificirane veličine. Format specificirane veličine je ***n'fv*** gde *n* označava broj bita, *'f* označava format zapisa (brojne vrednosti), a *v* označava samu brojnu vrednost. Formati mogu biti binarni (*'b*), decimalni (*'d*), heksadecimalni (*'h*) i oktalni (*'o*). Format se može pisati i velikim i malim slovom. U slučaju potrebe, vodeća mesta navedenog broja se popunjavaju nulama (na primer, ako je za petobitni broj navedena četvorobitna vrednost, smatra se da je najviši bit jednak 0). Primeri brojeva specificirane veličine:

- 4'b1101 (binarna predstava četvorobitnog broja decimalne vrednosti 13)

- 5'b1100 (binarna predstava petobitnog broja decimalne vrednosti 12, najviši peti bit nije naveden i za njega se smatra da ima vrednost 0)
- 4'd13 (decimalna predstava četvorobitnog broja decimalne vrednosti 13)
- 5'd12 (decimalna predstava petobitnog broja decimalne vrednosti 12)
- 4'hD (heksadecimalna predstava četvorobitnog broja decimalne vrednosti 13)
- 5'hC (heksadecimalna predstava petobitnog broja decimalne vrednosti 12)
- 5'h12 (heksadecimalna predstava petobitnog broja decimalne vrednosti 18)

U slučaju velikog broja cifara koje se navode, moguće je koristiti _ za razdvajanje blokova cifara radi postizanja bolje preglednosti. Na primer, 8'b11010010 se može napisati i kao 8'b1101_0010. Ako je u binarnoj predstavi najviši navedeni bit vrednosti *z* ili *x*, onda i svi preostali (nenavedeni) viši biti imaju vrednost *z* ili *x*. Ova osobina se može koristiti na sličan način kao OTHERS u VHDL jeziku za dodelu vrednosti *x* ili *z* svim bitima vektora. Na primer, 8'bz je ekvivalentno 8'bzzzzzzzz.

Nespecificirana veličina podrazumeva da se ne navodi broj bita *n*. Tada se smatra da je broj bita *n* jednak 32 (tj. vrednost koju podrazumeva samo razvojno okruženje, a koja je tipično 32). Takođe, u slučaju nespecificirane veličine ne mora da se navede ni format. Ukoliko se ne navede format, podrazumeva se decimalni format brojne vrednosti *v*.

B.2. Struktura modula

U Verilog jeziku entitet se naziva modul. Modul se sastoji od sledećih celina:

- **Naziv modula**
- **Lista portova**
- **Deklaracija parametara**
- **Deklaracija portova**
- Deklaracija internih signala
- Funkcije i procedure
- Strukturni opis
- Opis ponašanja
- Instance drugih modula

Prve četiri celine (koje su i prikazane boldovanim fontom) treba da idu na početak opisa modula. Pri tome, deklaracija parametara ne mora ići ispred deklaracije portova sem u slučaju kada se deklarisani parametri koriste u deklaraciji portova. Preostale celine mogu ići proizvoljnim redosledom, ali je najpreglednije da, ipak, idu redosledom koji je dat. Naravno, sve navedene celine nisu obavezne da se pojave u opisu modula. Na primer, ako ne postoje parametri, tada se celina deklaracija parametara neće pojaviti u opisu modula.

B.2.1. Naziv modula, lista portova

Generalna sintaksa za naziv modula i listu portova je sledeća:

```
module naziv_modula
(
    imeporta_1, imeporta_2,...,imeporta_N
);
// preostale celine
endmodule
```

Ključna reč ***module*** označava početak opisa modula i iza nje se odmah stavlja naziv modula. Potom se unutar zagrade koja sledi navode imena portova modula. Na kraju kompletnog opisa modula mora da stoji linija ***endmodule*** koja označava kraj opisa modula. Primer naziva modula i liste portova za multiplekser 4 u 1 je:

```
module mux4_u_1
(
    x1,x2,x3,x4,y,sel
);
// preostale celine
endmodule
```

B.2.2. Deklaracija parametara i portova

Uloga parametara je identična ulozi parametara u VHDL jeziku. Parametri se mogu koristiti za postizanje parametrizovanog (samim tim i fleksibilnog) dizajna, ali i za definisanje konstanti. U Verilogu se razlikuju globalni i lokalni parametri. Globalni parametri su identični parametrima iz GENERIC dela VHDL entiteta i njima se može pristupati prilikom instanciranja modula u nekom hijerarhijski višem modulu (slično GENERIC MAP principu iz VHDL jezika). Lokalni parametri su vidljivi samo unutar modula i njima se ne može pristupati spolja. Stoga su lokalni parametri pogodni za definisanje konstanti koje će se koristiti samo unutar dotičnog modula i čija se vrednost neće menjati (tj. neće biti potrebe da se pristupa ovoj vrednosti prilikom instanciranja dotičnog modula). Globalni parametri se deklarišu sa:

```
parameter ime_parametra = vrednost_parametra;
```

Lokalni parametri se deklarišu sa:

```
localparam ime_parametra = vrednost_parametra;
```

Primeri deklarisanja parametara:

```
parameter N=8;
parameter constant1=4'b0011;
localparam constant2=4'b1101;
```

Deklarisanje portova je ekvivalentno PORT delu VHDL entiteta. Sintaksa deklarisanja porta je:

mod dimenzija_vektora naziv_porta;

Mod označava smer porta. Postoje tri moda porta: *input* (ulazni), *output* (izlazni) i *inout* (bidirekcioni). Kao što se vidi modovi su praktično isti kao i u slučaju VHDL jezika. U slučaju Verilog jezika *output* mod je ekvivalentan BUFFER modu iz VHDL jezika, što znači da vrednost izlaznog porta može da se koristi za formiranje izraza u telu modula. U slučaju *output* porta mod se navodi kao *output* ili *output reg*. U slučaju kada se navodi mod *output*, tada je izlazni port u suštini *wire* tip signala, dok je u slučaju *output reg* moda izlazni port u suštini *reg* tip signala. Naime, sa stanovišta opisa signala Verilog razlikuje *wire* i *reg* tipove. Tip *wire* označava u suštini žicu bez memorije što i sam naziv tipa sugeriše (praktično izlaz kombinacione logike). Tip *reg* označava u suštini izlaz flip-flopa (sekvencijalne logike), tj. žicu sa memorijom što

sugeriše i sam naziv tipa (reg je u suštini skraćenje termina registar). Otuda, u zavisnosti da li je izlazni port sa memorijom ili ne, se definiše mod *output* ili *output reg*. U suštini izlaz kombinacione logike je *wire* tip, a sekvencijalne logike je *reg* tip, mada ova analogija nije u potpunosti 100% tačna jer zavisi od opisa same interne strukture modula. Na primer, ako izlaz sekvencijalne logike predstavimo internim signalom i potom taj interni signal povežemo sa izlaznim portom, tada je izlazni port u suštini *wire* tipa, iako je realno i dalje izlaz sekvencijalne logike. Postoje i obrnuti primeri kada izlaz kombinacione logike mora da se definiše kao *reg* tip. Modovi *input* i *inout* mogu biti samo *wire* tipa.

Dimenzija vektora se navodi u formatu [*dg*:*gg*] ili [*gg*:*dg*], gde je *dg* donja granica vektora (donja granica ne mora da bude 0), a *gg* gornja granica vektora. Gornje granice se navode u decimalnom zapisu. Format [*dg*:*gg*] odgovara *little endian* formatu, a format [*gg*:*dg*] odgovara *big endian* formatu. U suštini sve navedeno za dimenzije vektora i *big/little endian* format u VHDL jeziku važi i ovde, samo je sintaksa definisanja dužine vektora različita. U slučaju signala dužine 1 bit, dimenzija vektora se ne navodi sem ako ne želimo da port predstavimo kao jednobitni vektor (ako se navede tada je u pitanju jednobitni vektor - slično kao kod VHDL jezika i STD_LOGIC i STD_LOGIC_VECTOR(0 DOWNTO 0) tipova). Kada se pristupa samo jednom indeksu vektora indeks se navodi u okviru uglastih zagrada, na primer, *x*[1].

Primer deklaracije portova i parametra dužine vektora za multiplekser 4 u 1:

```
parameter N=8;
input [(N-1):0] x1,x2,x3,x4;
output [(N-1):0] y;
input [1:0] sel;
```

U ovom slučaju koristimo *output* mod za izlaz *y* jer je u pitanju kombinaciona logika. U ovom slučaju parametar mora da se deklariše pre portova, jer se parametar *N* koristi u deklaraciji portova. Redosled deklarisanja portova ne mora da se poklapa sa redosledom navođenja portova u listi portova.

B.2.3. Deklaracija internih signala

Interni signali predstavljaju signale vidljive samo u internoj strukturi modula, isto kao i interni signali u VHDL entitetima. Interni signali mogu biti *wire* ili *reg* tipa, pri čemu objašnjenja data za *output* mod važe i ovde. Interni signali se mogu deklarisati i kao vektori, pri čemu je princip deklarisanja dimenzije vektora isti kao kod portova. Interni signali se deklarišu po formatu:

tip dimenzija_vektora naziv_signala;

Tip je kao što smo naveli *wire* ili *reg*, a dimenzija vektora (dimenzija vektora je opciona kao i kod portova) se navodi identično kao i u slučaju portova. Primeri deklarisanja internih signala:

```
wire AND_output;
wire x1,x2,x3;
wire [7:0] bus_wire;
wire [5:0] a1,a2;
reg x_internal;
reg z1,z2,z3,z4;
reg [15:0] bus_reg;
reg [2:0] bus_rega, bus_regb;
```

```
reg check_ok = 1b'1;
reg [3:0] cnt = 4h'7;
```

Kao što se iz datih primera može videti, u slučaju *reg* tipa se može postaviti i inicijalna vrednost signala. Međutim, Verilog po difoltu ne podržava inicijalne vrednosti signala i samo od razvojnog okruženja zavisi da li će one zaista biti postavljene ili ne (slično kao i kod VHDL jezika). Otuda, važi napomena kao i u slučaju VHDL jezika da je najbolje inicijalne vrednosti signala sa memorijom postaviti u okviru reseta. Navedimo da postoje i drugi tipovi poput *integer* i *real* tipova, pri čemu slično kao i u VHDL jeziku *real* tip ne može hardverski da se implementira.

Verilog podržava i deklaraciju jednodimenzionih nizova dodavanjem dimenzije niza na kraju deklaracije internog signala (dimenzija niza se zadaje u istom formatu kao i dimenzija vektora). Može se pristupiti samo jednom članu niza odjednom i to kompletno, što je važno u slučaju niza vektora. U slučaju niza vektora se stoga može pristupiti samo kompletnom vektoru, ali ne i delu vektora. Takođe, nizovi portova nisu podržani u Verilog jeziku, pa se na osnovu svega navedenog može reći da je po pitanju nizova signala Verilog nešto slabiji od VHDL jezika. Primeri deklarisanja internih signala kao nizova:

```
reg bit_array [7:0];
reg [7:0] byte_array [0:31];
reg [15:0] word_array [15:0];
```

B.2.4. Strukturni opis

Strukturni opis predstavlja opis same strukture modula i u suštini predstavlja ekvivalent konkurentnom kodu iz VHDL jezika. Postoje dve varijante strukturnog opisa: *gate-level* opis i *dataflow* opis. Obe varijante se mogu uporedo koristiti u modulu. *Gate-level* opis je opis strukture pomoću primitiva, a *dataflow* opis je opis strukture pomoću dodela.

Gate-level opis koristi primitive osnovnih logičkih kola Verilog jezika. Podržane primitive su: *and*, *or*, *xor*, *nand*, *nor*, *xnor*, *buf*, *not*, *bufif1*, *bufif0*, *notif1*, *notif0*. Sama imena jasno upućuju na funkcije primitiva. Primitive *bufif* predstavljaju trostatičke bafere, a *notif* trostatičke bafere sa invertovanim izlazom, pri čemu 0 ili 1 u nazivu označavaju aktivnu vrednost kontrolnog signala koji kontroliše otvaranje trostatičkog bafera. Primitiva *buf* predstavlja bafer, a *not* invertor. Format upotrebe primitive je

naziv_primitive naziv_instance(lista izlaza, lista ulaza, lista kontrolnih signala);

Naziv primitive predstavlja ime upotrebljene primitive (na primer, *and*), dok naziv instance predstavlja naziv koji je dodeljen dotičnom instanciranom logičkom kolu. Naziv instance nije obavezan. Ulazi, izlazi i kontrolni signali su dužine jedan bit, tj. ne mogu se koristiti vektori. Ovo predstavlja veliko ograničenje u efikasnoj primeni primitiva. Signal koji se vezuje na izlaz primitive (tj. signali navedeni u listi izlaza) mora biti *wire* tipa što je logično jer sve primitive predstavljaju kombinaciona kola. U slučaju *and*, *or*, *xor*, *nand*, *nor*, *xnor* kola postoji samo jedan izlaz i više ulaza (minimalno 2). U slučaju *buf*, *not* kola postoji samo jedan ulaz i jedan ili više izlaza. U slučaju *bufif1*, *bufif0*, *notif1*, *notif0* kola postoji samo jedan ulaz i samo jedan izlaz, kao i jedan kontrolni signal. Primeri upotrebe primitiva:

```
and and_circuit(c,a,b);
xor xor_3(d,a,b,c);
nand (c,a,b);
```

```
buf buf_circuit(c,b,a);
not inv(b,a);
not (b,a);
bufif1 buf_tristate(b,a,ctrl);
notif0 inv_tristate(b,a,ctrl);
bufif0 buf0_tristate(b,a,ctrl);
```

Primer definisanja multipleksera 4 u 1 pomoću primitiva (prikazan je kompletan modul):

```
module mux4_u_1
(
        x1,x2,x3,x4,y,sel
);
        //deklaracije portova
        input x1,x2,x3,x4;
        output y;
        input [1:0] sel;
        //deklaracije internih signala
        wire a,b,c,d;
        wire sel0_inv, sel1_inv;
        //gate-level opis
        not inv1(sel0_inv,sel[0]);
        not inv2(sel1_inv,sel[1]);
        and and1(a,x1,sel0_inv,sel1_inv);
        and and2(b,x2,sel[0],sel1_inv);
        and and3(c,x3,sel0_inv,sel[1]);
        and and4(d,x4,sel[0],sel[1]);
        or or1(y,a,b,c,d);
endmodule
```

U ovom primeru nije korišćena parametrizacija korisničkih ulaza i izlaza jer bi se morale instancirati prikazane primitive za svaki bit vektora.

Dataflow opis takođe opisuje strukturu, ali koristeći operatore pri čemu je najveći deo njih preuzet iz C jezika. Izrazi (dodele vrednosti) se kreiraju na identičan način kao u C jeziku, sa dodatkom ključne reči *assign* koja mora ići na početak izraza tj. dodele. Operatori se mogu grupisati u sledeće kategorije:

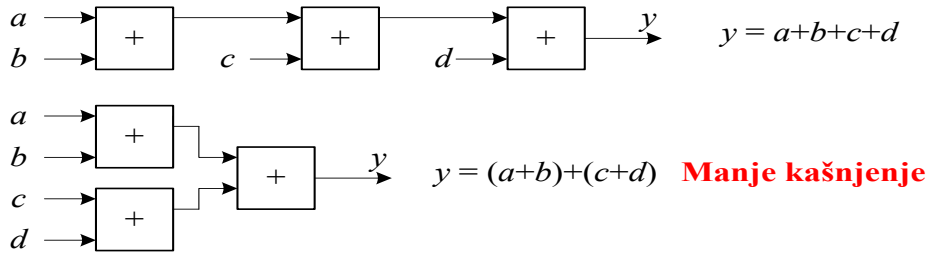
- Aritmetički (+,-,*,/,%)
- Logički (!,&&,||)
- Relacioni (<,>,<=,>=)
- Jednakosti (==,!=,===,!==)
- Bitwise (~,&,|,^, ^~,~^)
- Redukcioni (&,~&,|,~|,^, ^~,~^)
- Pomeranja (<<,>>)
- Konkatanacije ({})
- Replikacije ({{}})
- Uslovni (?:)

Kao što se vidi većina operatora je preuzeta iz C programskog jezika. Dodatni operatori su operatori konkatanacije, replikacije i redukcioni operatori. Redukcioni operatori vrše operacije

nad svim bitima vektora na konačan rezultat širine jedan bit (na primer, I operacija između svih bita vektora). Veoma su pogodni ovi operatori za slučajeve kada je potrebno izvršiti istu logičku operaciju između svih bita vektora. Na primer, ekskluzivno ILI operacija svih bita vektora bi vršila proveru parnosti vektora. Ovi operatori predstavljaju prednost u odnosu na VHDL jezik gde se za istu svrhu moraju koristiti petlje. Redukcioni operatori su inače unarni operatori pa se po tome razlikuju od bitwise operatora koji koriste iste oznake za operatore. Važno je naglasiti da operatori jednakosti === i !==, kao i aritmetički operatori / i % ne mogu da se implementiraju u hardveru. Primeri upotrebe operatora:

```
//pretpostavljene vrednosti signala
//(x1 = 8'b10010110, x2 = 8'b01111010, x3 = 4'b1010, x4 = 5'b01010)
assign y=x1+x2; //(rezultat 8'b00010000 ako je y vektor dužine 8)
assign y=x1+x2; //(rezultat 9'b100010000 ako je y vektor dužine 9)
assign y=x1+x2+x3; //(rezultat 8'b00011010 ako je y vektor dužine 8)
assign y1=x1 && x3; //(rezultat je 1'b1)
assign y1=x1 >= x2; //(rezultat je 1'b1)
assign y1=x1 < x3; //(rezultat je 1'b0, x3 se proširuje vodecim nulama)
assign y1=x1 == x2; //(rezultat je 1'b0)
assign y1=x3 == x4; //(rezultat je 1'b1, x3 se proširuje vodecom nulom)
assign y=~x1; //(rezultat 8'b01101001)
assign y=~x1 & x2; //(rezultat 8'b01101000)
assign y1=^x1; //(rezultat 1'b0)
assign y=x1 >> 2; //(rezultat 8'b00100101)
assign y=x1 >> x2[1:0]; //(rezultat 8'b00100101)
assign y={x1[3:0], x2[3:0]}; //(rezultat 8'b01101010)
assign y={x1[7:4], x2[3:0]}; //(rezultat 8'b10011010)
assign y ={ {2{x1[2:1]}}, {{x2[3:0]}} };//(rezultat 8'b11111010)
assign y ={ {2{x1[2:1]}}, {4{x2[0]}} };//(rezultat 8'b11110000)
assign y =(x3!=4'b1010)?x1:x2;//(rezultat 8'b01111010)
assign y =(x3==4'b1010)?x1:x2;//(rezultat 8'b10010110)
assign y =(x3==4'b1010)?(x1+x3):(x2+x3);//(rezultat 8'b10100000)
```

Važno je napomenuti da zagrade mogu uticati na konačnu implementaciju, što je ilustrovano na slici B.2.4.1 (na slici nije prikazana ključna reč *assign* u dodeli). Takođe, signal čija se vrednost postavlja u okviru *dataflow* opisa mora biti *wire* tipa, što je logično jer je u pitanju kombinaciona logika.



Slika B.2.4.1. – Primer značaja zagrada u *dataflow* opisu

Primer definisanja multipleksera 4 u 1 pomoću *dataflow* opisa (prikazan je kompletan modul):

```
module mux4_u_1
(
        x1,x2,x3,x4,y,sel
);
        //deklaracije portova
        input x1,x2,x3,x4;
```

```
        output y;
        input [1:0] sel;
        //dataflow opis
        assign y = (~sel[0]&~sel[1]&x1)|(sel[0]&~sel[1]&x2)|(~sel[0]&sel[1]&x3)|(sel[0]&sel[1]&x4);
endmodule
```

Kao što vidimo, opis strukture je znatno kraći nego u slučaju *gate-level* opisa. Ako želimo da parametrizujemo dužine vektora korisničkih ulaza i izlaza, tada možemo koristiti uslovne operatore:

```
module mux4_u_1
(
        x1,x2,x3,x4,y,sel
);
        //deklaracija parametra
        parameter N=8;
        //deklaracija portova
        input [(N-1):0] x1,x2,x3,x4;
        output [(N-1):0] y;
        input [1:0] sel;
        //dataflow opis
        assign y = (sel==2'b00)?x1:((sel==2'b01)?x2:((sel==2'b10)?x3:x4));
endmodule
```

B.2.5. Opis ponašanja

Opis ponašanja je praktično ekvivalent sekvencijalnom kodu VHDL jezika. Ideja je da se olakša dizajniranje složenih funkcionalnosti tako što će se opisati šta funkcija radi, a ne sama struktura njene implementacije. Kompajler će na osnovu opisa funkcionalnosti kreirati odgovarajuću hardversku strukturu koja vrši opisanu funkcionalnost.

Programski kod opisa ponašanja se smešta u *initial* i *always* blokove, koji su svojevrsni ekvivalenti procesa iz VHDL jezika (pre svega se to odnosi na *always* blok). Važno je napomenuti da se *initial* blok ne može hardverski realizovati i on se koristi u okviru simulacija za postavljanje inicijalnih vrednosti signala. S druge strane, *always* blok je uvek aktivan i on predstavlja ekvivalent procesa iz VHDL jezika, pri čemu se ovim blokom može opisati i kombinaciona i sekvencijalna logika (i dalje stoji napomena iz VHDL jezika da nije preporučljivo mešanje kombinacione i sekvencijalna logike - tj. da se ne mešaju unutar istog *always* bloka). Unutar modula se može kreirati više *initial* i *always* blokova (slično procesima u arhitekturi VHDL entiteta).

Struktura *initial* bloka je:

```
initial
begin
        //postavljanje inicijalnih vrednosti
end
```

Primer jednog *initial* bloka:

```
initial
begin
        x1=1'b1;
        x2=1'b0;
        cnt=4'h0;
end
```

U okviru *initial* bloka se mogu postavljati vrednosti samo za signale i izlazne portove *reg* tipa što je i logično jer *wire* tip nema memoriju. Otuda u prikazanom primeru signali *x1*, *x2* i *cnt* treba da se deklarišu kao *reg* tip uz pretpostavku da su u pitanju interni signali (u slučaju izlaznog porta deklaracija bi bila *output reg*).

U slučaju *always* bloka se navodi i lista osetljivosti (slično kao kod procesa u VHDL jeziku). U slučaju kombinacione logike navode se ulazni signali kombinacione logike, a u slučaju sekvencijalne logike se navodi signal takta i asinhroni reset ako se koristi. Struktura *always* bloka je:

```
always @(lista osetljivosti)
begin
        //opis ponašanja
end
```

U slučaju sekvencijalne logike u listi osetljivosti se stavlja *posedge clk* ako je logika aktivna na uzlaznu ivicu takta *clk*, odnosno *negedge clk* ako je logika aktivna na silaznu ivicu takta *clk*. Ako se koristi i asinhroni reset onda se navodi *posedge reset* za slučaj da je aktivna vrednost reseta '1', odnosno *negedge reset* za slučaj da je aktivna vrednost reseta '0' (naravno, u ovom primeru signal reseta ima naziv *reset*). Svi uslovi navedeni u listi osetljivosti se međusobno razdvajaju sa *or* operatorom. Primeri definisanja liste osetljivosti:

```
always @(x1 or x2 or x3 or x4 or sel)//za mux 4 u 1
always @(posedge clk)//uzlazna ivica takta
always @(negedge clk)//silazna ivica takta
always @(posedge clk or posedge reset)//uzlazna ivica takta i asinhroni reset (aktivna vrednost '1')
```

Za slučaj sekvencijalne logike i asinhronog reseta bi generalna struktura opisa ponašanja tj. *always* bloka izgledala (reset aktivan na '1', i uzlazna ivica takta):

```
always @(posedge clk or posedge reset)
begin
        if(reset)
        begin
                ...
        end
        else
        begin
                ...
        end
end
```

U slučaju sinhronog reseta, reset bi se izostavio iz liste osetljivosti:

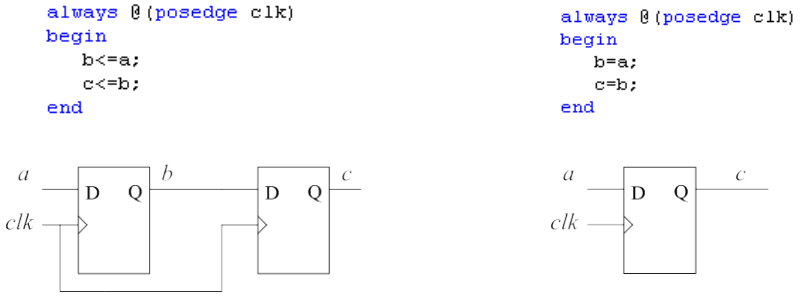
```
always @(posedge clk)
begin
        if(reset)
        begin
                ...
        end
        else
        begin
                ...
        end
end
```

Unutar *always* bloka se mogu koristiti operatori koji se koriste u *dataflow* opisu strukture, pri čemu se samo izostavlja ključna reč *assign* koja se koristi za signalizaciju da je u pitanju

dataflow izraz tj. dodela vrednosti. Naglasimo da se, kao i kod procesa u VHDL jeziku, ne može vršiti dodela vrednosti istom signalu u različitim *always* blokovima.

U VHDL kodu su se koristile varijable sa osobinom trenutnog dobijanja vrednosti što je olakšavalo pisanje koda tj. opisa ponašanja u određenim situacijama. Sličan princip omogućava i Verilog jezik. Unutar opisa ponašanja se mogu koristiti dve vrste dodele vrednosti - blokirajuća i neblokirajuća dodela. Operator blokirajuće dodele je =, a neblokirajuće je <=. Važno je napomenuti da za isti signal ne može da se koristi mešano operator dodele tj. da se na jednom mestu koristi za dodelu vrednosti neblokirajući operator, a na drugom blokirajući operator. Signali kojima se vrednost dodeljuje upotrebom blokirajuće dodele su praktično ekvivalentni varijablama iz VHDL jezika (trenutno se dobija dodeljena vrednost), a signali kojima se vrednost dodeljuje upotrebom neblokirajuće dodele su praktično ekvivalentni signalima iz VHDL jezika (dodeljena vrednost se dobija tek na kraju bloka - *always* ili *initial*). Blokirajuća dodela je dobila naziv usled toga što se blokira izvršenje programskog koda u nastavku sve dok se ne završi blokirajuća dodela. U suštini ovo svojstvo je vidljivo u slučaju simulacije gde se u okviru blokirajuće dodele može navesti posle koliko vremena se treba izvršiti dodela i sve dok se ne završi dodela (istekne zadato vreme) preostali deo koda u bloku je blokiran i ne može da se izvrši. Pošto se ne može hardverski realizovati čekanje, u opisu ponašanja implementacije koja može hardverski da se realizuje se ne mogu navoditi čekanja što znači da se blokirajuće dodele odmah izvršavaju identično kao u slučaju varijabli iz VHDL jezika.

Primer razlike blokirajuće i neblokirajuće dodele je dat na slici B.2.5.1. U slučaju neblokirajuće dodele kreiramo dva D-FF vezana na red, a u slučaju istih izraza, ali uz upotrebu blokirajuće dodele, dobijamo samo jedan D-FF.



Slika B.2.5.1. – Primer razlike u upotrebi blokirajuće i neblokirajuće dodele

Unutar *always* i *initial* blokova se pored operatora mogu koristiti i konstrukcije koje su veoma slične konstrukcijama iz C jezika. Na raspolaganju su *if* konstrukcija, *case* konstrukcija, i konstrukcije petlji. Postoje četiri vrste petlji - *for*, *while*, *repeat* i *forever*.

Varijante strukture *if* konstrukcije su prikazane na slici B.2.5.2. Ukoliko se između *begin* i *end* nalazi samo jedan izraz, onda *begin* i *end* mogu da se izostave.

Varijanta 1

```
if(uslov)
begin
    ....
end
```

Varijanta 2

```
if(uslov)
begin
    ...
end
else
begin
    ...
end
```

Varijanta 3

```
if(uslov1)
begin
   ...
end
else if (uslov2)
begin
   ...
end
...
else if (uslov_n)
begin
   ...
end
else
begin
   ...
end
```

Slika B.2.5.2. – Varijante *if* konstrukcije

Pokažimo primer multipleksera 4 u 1 kreiranog pomoću *if* konstrukcije:

```
module mux4_u_1
(
        x1,x2,x3,x4,y,sel
);
        //deklaracije parametra
        parameter N=8;
        //deklaracije portova
        input [(N-1):0] x1,x2,x3,x4;
        output reg [(N-1):0] y;
        input [1:0] sel;
        //opis ponasanja
        always @(x1 or x2 or x3 or x4 or sel)
        begin
                if(sel==2'b00)
                        y<=x1;
                else if(sel==2'b01)
                        y<=x2;
                else if(sel==2'b10)
                        y<=x3;
                else
                        y<=x4;
        end
endmodule
```

U ovom slučaju smo koristili neblokirajuću dodelu, ali je mogla da se koristi i blokirajuća dodela. Izlazni port *y* je morao da se deklariše kao *output reg* iako je *y* izlaz kombinacione logike. Razlog je veoma jednostavan. Vrednost *y* se kreira unutar *always* bloka. Kod unutar *always* bloka se 'izvršava' kada neki od signala iz liste osetljivosti promeni vrednost, pa ispada kao da *y* ima memoriju iako suštinski nema memoriju jer je u pitanju čista kombinaciona logika. Ovaj primer ilustruje da se *reg* tip može dodeliti i signalu koji je realno izlaz kombinacione logike, a razlog je upotreba koda opisa ponašanja tj. *always* bloka koji kao da dodaje svojevrsnu memoriju signalu.

Struktura *case* konstrukcije je:

```
case (izraz selekcije)
        vrednost_1:begin ... end
        vrednost_2:begin ... end
        ...
```

```
        vrednost_n:begin ... end
        default:begin ... end
endcase
```

Ključna reč *default* se koristi za označavanje svih preostalih vrednosti koje prethodno nisu navedene. U slučaju da se unutar *begin end* dela nalazi samo jedan izraz, *begin* i *end* se mogu izostaviti. Kreiranje multipleksera 4 u 1 pomoću *case* konstrukcije:

```
module mux4_u_1
(
        x1,x2,x3,x4,y,sel
);
        //deklaracije parametra
        parameter N=8;
        //deklaracije portova
        input [(N-1):0] x1,x2,x3,x4;
        output reg [(N-1):0] y;
        input [1:0] sel;
        //opis ponasanja
        always @(x1 or x2 or x3 or x4 or sel)
        begin
                case(sel)
                        2'b00: y<=x1;
                        2'b01: y<=x2;
                        2'b10: y<=x3;
                        default: y<=x4;
                endcase
        end
endmodule
```

I u ovom slučaju se mogao koristiti blokirajući operator dodele. Napomene date u poglavlju 2 za izbor između *case* i *if* strukture važe i ovde. Takođe, kao i u VHDL jeziku, *case* struktura se može koristiti za kreiranje konačnih automata, ali uz razliku da se stanja automata moraju direktno kodirati jer Verilog ne podržava enumerisane tipove. Na primer, ako automat ima četiri stanja, ona bi se mogla kodirati, na primer, kao 2'b00, 2'b01, 2'b10 i 2'b11 (ovakav direktan način kodiranja se može koristiti i u VHDL jeziku, ali očigledno je da je upotreba enumerisanih tipova bolja sa stanovišta razumljivosti funkcije stanja konačnog automata).

Struktura *for* petlje je identična strukturi iz C jezika sa izuzetkom da se početak i kraj tela petlje označava sa *begin*/*end* umesto sa {} zagradom (isto kao u C jeziku *begin*/*end* se mogu izostaviti ako se petlja sastoji samo od jedne linije koda). Primer upotrebe *for* petlje:

```
module brojac_jedinica
(
        x,y
);
        //deklaracije parametara
        parameter N=8;
        parameter M=4;
        //deklaracije portova
        input [(N-1):0] x;
        output reg [(M-1):0] y;
        //interni signal
        integer i;
        //opis ponasanja
        always @(x)
        begin
                y=0;
```

```
                for(i=0;i<N;i=i+1)
                begin
                    if(x[i])
                        y=y+1'b1;
                end
        end
endmodule
```

U datom primeru kreiran je modul koji vrši brojanje jedinica unutar ulaznog vektora. Kao što vidimo, struktura *for* petlje je identična onoj iz C programskog jezika (napomena je da se ne može koristiti operator ++ koji je veoma čest u *for* petljama C jezika jer nije podržan u Verilog jeziku). Takođe, treba primetiti da se promenjiva *i* mora deklarisati i to kao *integer* tip. U VHDL jeziku se promenjiva koja se koristi za iteraciju u *for* petlji nije morala deklarisati. Bez obzira što je modul iz primera kombinaciona logika, izlazni port *y* se morao deklarisati kao *reg* tip jer se njegova vrednost postavlja unutar *always* strukture. Takođe, na ovom primeru se jasno vidi da u Verilog jeziku izlazni mod porta ima svojstva BUFFER moda iz VHDL jezika.

Struktura *while* petlje je takođe identična ekvivalentnoj strukturi iz C jezika sa izuzetkom da se početak i kraj tela petlje označava sa *begin*/*end* umesto sa {} zagradom (isto kao u C jeziku *begin*/*end* se mogu izostaviti ako se petlja sastoji samo od jedne linije koda). Primer realizacije modula za brojanje jedinica u vektoru primenom *while* petlje:

```
module brojac_jedinica
(
        x,y
);
        //deklaracije parametara
        parameter N=8;
        parameter M=4;
        //deklaracije portova
        input [(N-1):0] x;
        output reg [(M-1):0] y;
        //interni signal
        integer i;
        //opis ponasanja
        always @(x)
        begin
                y=0;
                i=0;
                while (i<N)
                begin
                    if(x[i])
                        y=y+1'b1;
                    i=i+1;
                end
        end
endmodule
```

Kao što vidimo, struktura zahteva nešto više linija koda jer se inicijalizacija promenjive za brojanje iteracija mora inicijalizovati pre *while* petlje, a takođe unutar petlje se mora izvršavati inkrementiranje promenjive za brojanje iteracija (kod *for* petlje je sve smešteno u jednoj liniji).

Struktura *repeat* petlje je veoma jednostavna i sastoji se samo od ključne reči *repeat* i broja iteracija navedenog u zagradi. Kod sadržan u petlji se smešta između *begin* i *end* (oni se mogu izostaviti ako je u petlji samo jedna linija koda). Važna napomena za *repeat* petlju je da se

ona može koristiti samo za petlje sa fiksnim brojem iteracija da bi bila hardverski ostvariva za razliku od prethodne dve. Primer realizacije modula za brojanje jedinica u vektoru primenom *repeat* petlje:

```
module brojac_jedinica
(
        x,y
);
        //deklaracije parametara
        parameter N=8;
        parameter M=4;
        //deklaracije portova
        input [(N-1):0] x;
        output reg [(M-1):0] y;
        //interni signal
        integer i;
        //opis ponasanja
        always @(x)
        begin
                y=0;
                i=0;
                repeat (N)
                begin
                    if(x[i])
                        y=y+1'b1;
                    i=i+1;
                end
        end
endmodule
```

I u ovom slučaju postoje dodatne dve linije kao i kod *while* petlje. Treba primetiti da se u sva tri primera modula za brojanje jedinica u vektoru morao koristiti blokirajući operator dodele jer u suprotnom modul ne bi radio ispravno svoju funkciju. Kao što je već napomenuto u poglavlju 2, petlje sa promenjivim brojem iteracija treba izbegavati ukoliko je moguće jer one značajno povećavaju kompleksnost dizajna u najvećem broju slučajeva. U slučaju petlji sa fiksnim brojem iteracija svejedno je koja će se od tri prethodno navedene petlje koristiti, i izbor zavisi prvenstveno od preferenci dizajnera modula.

Petlja *forever* je bezuslovna petlja koja se izvršava beskonačno puta. Ona se stoga ne može hardverski realizovati i koristi se u okviru simulacije pre svega za kreiranje signala takta ili drugih periodičnih signala. Struktura ove petlje se sastoji samo od ključne reči *forever*, a kod unutar petlje se smešta između *begin* i *end* (oni se mogu izostaviti ako je u petlji samo jedna linija koda). Primer kreiranja takta za potrebe simulacije:

```
initial
begin
        clk=1'b0;
        forever
            #5 clk = ~clk;
end
```

U datom primeru, početna vrednost simuliranog takta je ’0’, a potom se svakih 5ns vrši invertovanje vrednosti takta. U ovom slučaju blokirajuća dodela zaista vrši blokiranje na 5ns dok se instrukcija ne izvrši.

B.2.6. Funkcije i procedure

Funkcije i procedure imaju istu ulogu kao u slučaju VHDL jezika. Struktura funkcije je:

```
function tip/opseg ime_funkcije;
        //deklaracije ulaznih argumenata funkcije
        //deklaracije parametara
        //deklaracije lokalnih promenjivih
        begin
                //telo funkcije
        end
endfunction
```

Tip/opseg navode tip ili opseg rezultata funkcije. Ako se opseg ne navede podrazumeva se da je u pitanju jednobitni signal, tj. opseg se navodi samo u slučaju ako je izlazni rezultat funkcije vektor. Tip se navodi u slučaju da je rezultat funkcije *integer*, *real* i sl. Unutar deklarativnog dela funkcije se ulazni argumenti deklarišu u istom formatu kao ulazni portovi modula. Važno je napomenuti da se unutar funkcije mogu koristiti samo blokirajuće dodele, što je i logično jer funkcija u suštini predstavlja kombinacionu logiku. Izlazni rezultat se vraća tako što se konačna vrednost dodeli promenjivoj čiji je naziv jednak nazivu funkcije. Ova promenjiva se ne deklariše, jer je naziv funkcije istovremeno i deklaracija izlazne promenjive funkcije. Primer primene funkcije za brojanje jedinica u vektoru u modulu *brojac_jedinica* čije su varijante prikazane u primerima u prethodnoj sekciji:

```
module brojac_jedinica
(
        x,y
);
        //deklaracije parametara
        parameter N=8;
        parameter M=4;
        //deklaracije portova
        input [(N-1):0] x;
        output [(M-1):0] y;
        //funkcija
        function [(M-1):0] broji_jedinice;
                input [(N-1):0] x;
                integer i;
                reg [(M-1):0] brojac;
         begin
                brojac = 1'b0;
                for(i=0;i<N;i=i+1)
                begin
                    if(x[i])
                        brojac=brojac+1'b1;
                end
                broji_jedinice=brojac;
         end
        endfunction
        //strukturni opis
        assign y=broji_jedinice(x);
endmodule
```

Izlazni port *y* je sada deklarisan kao *wire* tip jer mu se dodeljuje vrednost koristeći *dataflow* opis. Rezultat funkcije se vraća dodelom vrednosti unutar tela funkcije promenjivoj čiji naziv se poklapa sa nazivom funkcije kao što smo već napomenuli. Važno je napomenuti da se ulazni argumenti u pozivu funkcije moraju navoditi istim redosledom kao što su navedeni u

deklarativnom delu funkcije. Na primer, ako su u funkciji *f1* deklarisani ulazni argumenti sledećim redosledom *x1*, *x2*, *x3*, tada poziv funkcije *f1*(*a*,*b*,*c*) podrazumeva da *a* odgovara ulaznom argumentu *x1*, *b* odgovara *x2*, i *c* odgovara *x3*.

Procedure mogu da imaju i više ulaznih argumenata i više izlaznih rezultata. Struktura procedure je:

```
task ime_procedure;
        //deklaracije ulaznih, izlaznih i bidirekcionih argumenata
        //deklaracije parametara
        //deklaracije lokalnih promenjivih
        begin
            //telo procedure
        end
endtask
```

Važno je napomenuti da, kao i kod funkcija, redosled deklarisanja argumenata u deklarativnom delu procedure, definiše kojim redosledom se navode argumenti prilikom pozivanja procedure. Primer upotrebe procedure u modulu koji vrši komparaciju dva vektora i kao rezultat daje tri jednobitna signala *g* (slučaj *x1*>*x2*), *e* (slučaj *x1*=*x2*) i *l* (slučaj *x1*<*x2*):

```
module komparator
(
        x1,x2,g,e,l
);
        //deklaracije parametara
        parameter N=8;
        //deklaracije portova
        input [(N-1):0] x1,x2;
        output reg g,e,l;
        //procedure
        task komparacija;
                input [(N-1):0] x1,x2;
                output g,e,l;
                begin
                        if(x1>x2)
                                g=1'b1;
                        else
                                g=1'b0;
                        if(x1==x2)
                                e=1'b1;
                        else
                                e=1'b0;
                        if(x1<x2)
                                l=1'b1;
                        else
                                l=1'b0;
                end
        endtask
        // opis ponasanja
        always @(x1,x2)
        begin
            komparacija(x1,x2,g,e,l); // redosled navodjenja argumenata je vazan
        end
endmodule
```

Pošto procedura mora da se poziva u opisu ponašanja, izlazni portovi modula su morali da se deklarišu kao *reg* tip.

B.2.7. Instanciranje modula

Kao i VHDL, i Verilog podržava hijerhijski dizajn koji podrazumeva da se u telu modula može instancirati primerak nekog drugog modula (princip komponenti iz VHDL jezika). Postoje dva načina za instanciranje modula - direktan i preko naziva.

Direktno instanciranje podrazumeva da se signali koji se vezuju na portove instanciranog modula navode u istom redosledu kako su portovi navedeni u listi portova originalnog modula čiji se primerak instancira. Ukoliko instancirani modul sadrži parametre, onda se oni moraju navoditi u istom redosledu u kom su definisani (deklarisani) u modulu koji se instancira. Format direktnog instanciranja je:

Naziv_modula #(lista mapiranih parametara) naziv_instance(lista mapiranih signala)

Parametri ne moraju da se instanciraju (deo *#(lista mapiranih parametara)* može da se izostavi) i tada se koriste vrednosti definisane u instanciranom modulu. Napomenimo još jednom da se pod parametrima podrazumevaju parametri deklarisani sa ključnom reči *parameter*, dok lokalni parametri (deklarisani sa ključnom reči *localparam*) ne mogu da se mapiraju. Nazivi instanci moraju biti jedinstveni u telu modula (svaka instanca mora imati naziv koji se ne poklapa sa nazivima drugih instanci).

Primer direktnog instanciranja prilikom kreiranja 4x4 sviča od multipleksera 4 u 1 (bilo koji parametrizovan modul multipleksera iz prethodnih sekcija se može koristiti za ovaj primer):

```
module switch4x4
(
        ulaz1,ulaz2,ulaz3,ulaz4,sel1,sel2,sel3,sel4,izlaz1,izlaz2,izlaz3,izlaz4
);
        //deklaracija parametra
        parameter N = 8;
        //deklaracija portova
        input [(N-1):0] ulaz1,ulaz2,ulaz3,ulaz4;
        input [1:0] sel1,sel2,sel3,sel4;
        output [(N-1):0] izlaz1,izlaz2,izlaz3,izlaz4;
        //direktno instanciranje multipleksera
        mux4_u_1 #(N) mux1(ulaz1,ulaz2,ulaz3,ulaz4,izlaz1,sel1);
        mux4_u_1 #(N) mux2(ulaz1,ulaz2,ulaz3,ulaz4,izlaz2,sel2);
        mux4_u_1 #(N) mux3(ulaz1,ulaz2,ulaz3,ulaz4,izlaz3,sel3);
        mux4_u_1 #(N) mux4(ulaz1,ulaz2,ulaz3,ulaz4,izlaz4,sel4);
endmodule
```

Slično kao i u slučaju VHDL jezika, preporuka je ne koristiti direktno instanciranje jer je takav način pisanja koda manje pregledan, skloniji greškama i teži za ažuriranje.

Bolji i pregledniji način instanciranja je instanciranje preko naziva. Razlika u odnosu na direktno instanciranje je što se sada navode i nazivi portova i parametara pa redosled mapiranja nije bitan. Format instanciranja je isti kao u slučaju direktnog instanciranja, uz razliku samog navođenja mapiranog porta (parametra) koje se radi u sledećem formatu:

.naziv_porta(naziv_mapiranog_signala)

.naziv_parametra(naziv_mapiranog_parametra)

Primer instanciranja preko naziva prilikom kreiranja 4x4 sviča od multipleksera 4 u 1 (bilo koji parametrizovan modul multipleksera iz prethodnih sekcija se može koristiti za ovaj primer):

```
module switch4x4
(
        ulaz1,ulaz2,ulaz3,ulaz4,
        sel1,sel2,sel3,sel4,
        izlaz1,izlaz2,izlaz3,izlaz4
);
        //deklaracija parametra
        parameter N = 8;
        //deklaracija portova
        input [(N-1):0] ulaz1,ulaz2,ulaz3,ulaz4;
        input [1:0] sel1,sel2,sel3,sel4;
        output [(N-1):0] izlaz1,izlaz2,izlaz3,izlaz4;
        //direktno instanciranje multipleksera
        mux4_u_1 #(.N(N)) mux1(.x1(ulaz1),.x2(ulaz2),.x3(ulaz3),.x4(ulaz4),.y(izlaz1),.sel(sel1));
        mux4_u_1 #(.N(N)) mux2(.x1(ulaz1),.x2(ulaz2),.x3(ulaz3),.x4(ulaz4),.y(izlaz2),.sel(sel2));
        mux4_u_1 #(.N(N)) mux3(.x1(ulaz1),.x2(ulaz2),.x3(ulaz3),.x4(ulaz4),.y(izlaz3),.sel(sel3));
        mux4_u_1 #(.N(N)) mux4(.y(izlaz4),.sel(sel4),.x1(ulaz1),.x2(ulaz2),.x3(ulaz3),.x4(ulaz4));
endmodule
```

U poslednjoj instanci (instanca *mux4*) je namerno naveden drugačiji redosled mapiranja da bi se naglasilo da redosled mapiranja nije bitan kod mapiranja preko naziva. I kod mapiranja preko naziva, parametri ne moraju da se instanciraju (deo *#(lista mapiranih parametara)* može da se izostavi) i tada se koriste vrednosti definisane u instanciranom modulu.

Parametri mogu da se mapiraju i upotrebom *defparam* instrukcije. Ukoliko se koristi ovaj metod tada se izostavlja mapiranje parametara u instanci (deo *#(lista mapiranih parametara)* se izostavlja). Koristi se sledeći format za *defparam* mapiranje parametara:

naziv_instance.nazivparametra = mapirana_vrednost_parametra

Ovaj metod je pogodan ako instancirani modul sadrži veliki broj parametara, a potrebno je mapirati nove vrednosti na veoma mali broj parametara, dok se za ostale parametre želi zadržati vrednost deklarisana unutar instanciranog modula (difolt vrednost). U suprotnom, ovaj metod zahteva znatno veći broj linija koda od mapiranja parametara na neki od dva prethodno navedena načina i stoga nije praktičan u najvećem broju slučajeva. Primer *defparam* mapiranja parametara za slučaj kreiranja sviča 4x4:

```
module switch4x4
(
        ulaz1,ulaz2,ulaz3,ulaz4,
        sel1,sel2,sel3,sel4,
        izlaz1,izlaz2,izlaz3,izlaz4
);
        //deklaracija parametra
        parameter N = 8;
        //deklaracija portova
        input [(N-1):0] ulaz1,ulaz2,ulaz3,ulaz4;
        input [1:0] sel1,sel2,sel3,sel4;
        output [(N-1):0] izlaz1,izlaz2,izlaz3,izlaz4;
        //direktno instanciranje multipleksera
        defparam mux1.N=N;
        defparam mux2.N=N;
        defparam mux3.N=N;
        defparam mux4.N=N;
        mux4_u_1 mux1(.x1(ulaz1),.x2(ulaz2),.x3(ulaz3),.x4(ulaz4),.y(izlaz1),.sel(sel1));
        mux4_u_1 mux2(.x1(ulaz1),.x2(ulaz2),.x3(ulaz3),.x4(ulaz4),.y(izlaz2),.sel(sel2));
        mux4_u_1 mux3(.x1(ulaz1),.x2(ulaz2),.x3(ulaz3),.x4(ulaz4),.y(izlaz3),.sel(sel3));
        mux4_u_1 mux4(.y(izlaz4),.sel(sel4),.x1(ulaz1),.x2(ulaz2),.x3(ulaz3),.x4(ulaz4));
```

```
endmodule
```

B.2.8. Direktive

Verilog omogućava korišćenje direktiva, slično kao i C jezik, pri čemu je velik broj direktiva istovetan onima iz C jezika. U okviru ove sekcije će biti dat pregled najčešće korišćenih direktiva. Direktive se navode u formatu

'naziv_direktive

Direktive koje se koriste češće su *define*, *include*, *ifdef*, *timescale*, ali postoje i druge. Direktive se navode pre početka definisanja modula (koje počinje ključnom reči *module*). Direktiva *define* ima istu ulogu kao i *define* u C jeziku. Jedna njena česta primena je kreiranje konstanti (u istu svrhu je ipak bolje koristiti parametre), na primer:

'define word_size 16

Direktiva *include* ima istu ulogu kao *include* direktiva iz C jezika, na primer:

'include drugi_fajl.v

Direktiva *ifdef* ima ulogu koja je slična ulozi *if generate* konstrukcije iz VHDL jezika. U suštini ova direktiva je analogna *ifdef* direktivi iz C jezika. Pomoću ove direktive se može izabrati koji deo koda u telu modula će biti kompajliran. Uz *ifdef* se može koristiti i *else*, a sama *ifdef* direktiva se mora zatvoriti sa *endif*. Pri tome, ova direktiva se, s obzirom na njenu ulogu, uglavnom navodi u telu modula. Primer upotrebe direktive:

```
'define TIP 0
module switch4x4
(
        ulaz1,ulaz2,ulaz3,ulaz4,sel1,sel2,sel3,sel4,izlaz1,izlaz2,izlaz3,izlaz4
);
        //deklaracija parametra
        parameter N = 8;
        //deklaracija portova
        input [(N-1):0] ulaz1,ulaz2,ulaz3,ulaz4;
        input [1:0] sel1,sel2,sel3,sel4;
        output [(N-1):0] izlaz1,izlaz2,izlaz3,izlaz4;
        //direktno instanciranje multipleksera
        'ifdef TIP
                mux4_u_1 #(.N(N)) mux1(.x1(ulaz1),.x2(ulaz2),.x3(ulaz3),.x4(ulaz4),.y(izlaz1),.sel(sel1));
                mux4_u_1 #(.N(N)) mux2(.x1(ulaz1),.x2(ulaz2),.x3(ulaz3),.x4(ulaz4),.y(izlaz2),.sel(sel2));
                mux4_u_1 #(.N(N)) mux3(.x1(ulaz1),.x2(ulaz2),.x3(ulaz3),.x4(ulaz4),.y(izlaz3),.sel(sel3));
                mux4_u_1 #(.N(N)) mux4(.x1(ulaz1),.x2(ulaz2),.x3(ulaz3),.x4(ulaz4),.y(izlaz4),.sel(sel4));
        'else
                mux4_u_1 #(.N(N)) mux1(.x1(ulaz1),.x2(ulaz2),.x3(ulaz3),.x4(ulaz4),.y(izlaz1),.sel(sel4));
                mux4_u_1 #(.N(N)) mux2(.x1(ulaz1),.x2(ulaz2),.x3(ulaz3),.x4(ulaz4),.y(izlaz2),.sel(sel3));
                mux4_u_1 #(.N(N)) mux3(.x1(ulaz1),.x2(ulaz2),.x3(ulaz3),.x4(ulaz4),.y(izlaz3),.sel(sel2));
                mux4_u_1 #(.N(N)) mux4(.x1(ulaz1),.x2(ulaz2),.x3(ulaz3),.x4(ulaz4),.y(izlaz4),.sel(sel1));
        'endif
endmodule
```

U datom primeru, ukoliko se definiše TIP upotrebom *define* direktive, tada će biti kompajliran (implementiran) deo koda koji se nalazi pod *ifdef*, u suprotnom će biti generisan deo koda koji se nalazi pod *else* delom.

Direktiva *timescale* se koristi za definisanje vremenske skale i preciznosti vremenske skale. Ova direktiva se tipično koristi za potrebe simulacije. Primer definisanja vremenske skale:

'timescale 1ns / 1ps

U datom primeru, vremenska skala je 1ns, a preciznost skale je 1ps.

B.2.9. Kreiranje testbenča

U okviru ove sekcije će biti navedene osnovne tehnike koje se mogu koristiti u kreiranju testbenča sa stanovišta generisanja stimulusa ulaznih portova modula za potrebe simulacije. Stimulusi mogu biti generisani asinhrono ili sinhrono. Asinhrono generisanje podrazumeva da se trenuci generisanja željenih vrednosti stimulusa definišu u kodu testbenča, dok sinhrono podrazumeva da se stimulusi generišu na signal takta. Naravno, unutar testbenča se mogu koristiti istovremeno oba principa generisanja. Međutim, sa stanovišta jednog stimulusa se mora koristiti ili asinhroni ili sinhroni pristup, tj. nije poželjno mešati pristupe za isti stimulus.

U slučaju asinhronog generisanja stimulusa se koristi definisanje kašnjenja. Ispred izraza se dodaje izraz za kašnjenje u formatu #*T*, gde *T* predstavlja brojnu vrednost kašnjenja. Dimenzija ove brojne vrednosti se definiše upotrebom *timescale* direktive. Da bi kašnjenje imalo efekat koriste se blokirajuće dodele signalima stimulusa. Primer kreiranja stimulusa za 4x4 svič (prikazan je testbenč modul za 4x4 svič):

```
`timescale 1ns / 1ps
//testbench za 4x4 svic
module testbench();
	parameter duzina_magistrale = 16;
	//simulirani ulazi u svic
	//simulirani data ulazi
	reg [(duzina_magistrale -1):0] stimulus_ulaz1;
	reg [(duzina_magistrale -1):0] stimulus_ulaz2;
	reg [(duzina_magistrale -1):0] stimulus_ulaz3;
	reg [(duzina_magistrale -1):0] stimulus_ulaz4;
	//simulirani kontrolni ulazi
	reg [1:0] stimulus_sel1;
	reg [1:0] stimulus_sel2;
	reg [1:0] stimulus_sel3;
	reg [1:0] stimulus_sel4;
	//za posmatranje izlaza svica
	wire [(duzina_magistrale -1):0] test_izlaz1;
	wire [(duzina_magistrale -1):0] test_ izlaz2;
	wire [(duzina_magistrale -1):0] test_ izlaz3;
	wire [(duzina_magistrale -1):0] test_ izlaz4;
	//interna struktura testbencha
	initial //initial blok za generisanje vrednosti stimulusa
	begin
		stimulus_ulaz1 = 16'h1234;
		stimulus_ulaz2 = 16'h5678;
		stimulus_ulaz3 = 16'h9abc;
		stimulus_ulaz4 = 16'hdef0;
		stimulus_sel1 = 2'b01;
		stimulus_sel2 = 2'b10;
		stimulus_sel3 = 2'b11;
		stimulus_sel4 = 2'b00;
		#10  stimulus_ulaz1 = 16'h4321;
		stimulus_ulaz2 = 16'h8765;
		stimulus_ulaz3 = 16'hcba9;
		stimulus_ulaz4 = 16'h0fed;
		#20  stimulus_ulaz1 = 16'h1356;
		stimulus_ulaz2 = 16'h2879;
		stimulus_ulaz3 = 16'h3443;
```

```
                stimulus_ulaz4 = 16'h0055;
                stimulus_sel1 = 2'b11;
                stimulus_sel2 = 2'b00;
                stimulus_sel3 = 2'b01;
                stimulus_sel4 = 2'b10;
                #10 stimulus_sel1 = 2'b01;
                stimulus_sel2 = 2'b11;
                stimulus_sel3 = 2'b10;
                stimulus_sel4 = 2'b00;
                #10  stimulus_ulaz1 = 16'h59ac;
                stimulus_ulaz2 = 16'h1ff4;
                stimulus_ulaz3 = 16'h0809;
                stimulus_ulaz4 = 16'h22aa;
                stimulus_sel1 = 2'b11;
                stimulus_sel2 = 2'b00;
                stimulus_sel3 = 2'b11;
                stimulus_sel4 = 2'b00;
        end

        //instanca svica koji se testira tj. simulira
        switch4x4
        #(.N (duzina_magistrale))
        switch_instanca
        (
                .ulaz1(stimulus_ulaz1),
                .ulaz2(stimulus_ulaz2),
                .ulaz3(stimulus_ulaz3),
                .ulaz4(stimulus_ulaz4),
                .sel1(stimulus_sel1),
                .sel2(stimulus_sel2),
                .sel3(stimulus_sel3),
                .sel4(stimulus_sel4),
                .izlaz1(test_izlaz1),
                .izlaz2(test_izlaz2),
                .izlaz3(test_izlaz3),
                .izlaz4(test_izlaz4)
        );

endmodule
```

Ulazni stimulusi su deklarisani kao *reg* tip jer se njihove vrednosti definišu unutar *initial* bloka. Kašnjenje #10 iznosi 10ns u skladu sa definicijom vremenske skale date *timescale* direktivom. Ovde se vidi uloga kašnjenja u blokirajućoj dodeli. Naime u trenutku 0ns stimulusi dobijaju svoje inicijalne vrednosti (prvih 8 dodela u *initial* bloku). Potom se navodi dodela:

```
#10  stimulus_ulaz1 = 16'h4321;
```

koja treba da se izvrši sa kašnjenjem od 10ns. Pošto se koristi blokirajuća dodela sve preostale linije su takođe blokirane. Nakon isteka 10ns tj. u trenutku 10ns stimulusi data ulaza dobijaju nove vrednosti. Sledeća dodela novih vrednosti će biti u trenutku 30ns (10ns+20ns). Kao što se vidi, na ovaj način se asinhrono mogu dodeljivati vrednosti stimulusima. Naravno, svi stimulusi, kojima se na ovaj način dodeljuju vrednosti, ne moraju da budu u istom *initial* bloku, već mogu da se rasporede u različite *initial* blokove, pri čemu je bitno da istom stimulus signalu vrednost ne može biti dodeljivana u različitim *initial* blokovima.

Sinhrono generisanje stimulusa podrazumeva da se vrednosti stimulusa generišu sinhrono sa taktom. Generisanje signala takta za simulaciju se postiže sledećom strukturom:

```
initial
begin
        clk=1'b0;
        forever
            #5 clk = ~clk;
end
```

Veličina kašnjenja definisana u izrazu koji se nalazi pod *forever* petljom predstavlja jednu polovinu periode kašnjenja, pa je u datom primeru vrednost periode takta 10ns uz pretpostavku da je *timescale* direktivom vremenska skala postavljena na 1ns. Potom se upotrebom brojača i *if* ili *case* strukture lako mogu generisati stimulusi. Primer upotrebe sinhronog generisanja stimulusa za potrebe simuliranja 4x4 sviča:

```
`timescale 1ns / 1ps
//testbench za 4x4 svic
module testbench();
        parameter duzina_magistrale = 16;
        //simulirani ulazi u svic
        //simulirani data ulazi
        reg [(duzina_magistrale -1):0] stimulus_ulaz1;
        reg [(duzina_magistrale -1):0] stimulus_ulaz2;
        reg [(duzina_magistrale -1):0] stimulus_ulaz3;
        reg [(duzina_magistrale -1):0] stimulus_ulaz4;
        //simulirani kontrolni ulazi
        reg [1:0] stimulus_sel1;
        reg [1:0] stimulus_sel2;
        reg [1:0] stimulus_sel3;
        reg [1:0] stimulus_sel4;
        //za posmatranje izlaza svica
        wire [(duzina_magistrale -1):0] test_izlaz1;
        wire [(duzina_magistrale -1):0] test_ izlaz2;
        wire [(duzina_magistrale -1):0] test_ izlaz3;
        wire [(duzina_magistrale -1):0] test_ izlaz4;
        //takt i brojac signali
        reg clk;
        reg [2:0] brojac=3'b000;
        //interna struktura testbencha
        initial //initial blok za generisanje signala takta
        begin
                clk=1'b0;
                forever
                        #5 clk = ~clk;
        end

        always @(posedge clk) //always blok za generisanje stimulusa
        begin
                case(brojac)
                3'b000:
                begin
                        stimulus_ulaz1 = 16'h1234;
                        stimulus_ulaz2 = 16'h5678;
                        stimulus_ulaz3 = 16'h9abc;
                        stimulus_ulaz4 = 16'hdef0;
                        stimulus_sel1 = 2'b01;
                        stimulus_sel2 = 2'b10;
                        stimulus_sel3 = 2'b11;
                        stimulus_sel4 = 2'b00;
                end
                3'b001:
```

```
		begin
			stimulus_ulaz1 = 16'h4321;
			stimulus_ulaz2 = 16'h8765;
			stimulus_ulaz3 = 16'hcba9;
			stimulus_ulaz4 = 16'h0fed;
		end
		3'b011:
		begin
			stimulus_ulaz1 = 16'h1356;
			stimulus_ulaz2 = 16'h2879;
			stimulus_ulaz3 = 16'h3443;
			stimulus_ulaz4 = 16'h0055;
			stimulus_sel1 = 2'b11;
			stimulus_sel2 = 2'b00;
			stimulus_sel3 = 2'b01;
			stimulus_sel4 = 2'b10;
		end
		3'b100:
		begin
			stimulus_sel1 = 2'b01;
			stimulus_sel2 = 2'b11;
			stimulus_sel3 = 2'b10;
			stimulus_sel4 = 2'b00;
		end
		3'b101:
		begin
			stimulus_ulaz1 = 16'h59ac;
			stimulus_ulaz2 = 16'h1ff4;
			stimulus_ulaz3 = 16'h0809;
			stimulus_ulaz4 = 16'h22aa;
			stimulus_sel1 = 2'b11;
			stimulus_sel2 = 2'b00;
			stimulus_sel3 = 2'b11;
			stimulus_sel4 = 2'b00;
		end
		default:
		begin
		end
		endcase
		brojac=brojac+1'b1;
end

//instanca svica koji se testira tj. simulira
switch4x4
#(.N (duzina_magistrale))
switch_instanca
(
		.ulaz1(stimulus_ulaz1),
		.ulaz2(stimulus_ulaz2),
		.ulaz3(stimulus_ulaz3),
		.ulaz4(stimulus_ulaz4),
		.sel1(stimulus_sel1),
		.sel2(stimulus_sel2),
		.sel3(stimulus_sel3),
		.sel4(stimulus_sel4),
		.izlaz1(test_izlaz1),
		.izlaz2(test_izlaz2),
		.izlaz3(test_izlaz3),
		.izlaz4(test_izlaz4)
);
```

```
endmodule
```

Sinhrona varijanta je pogodnija kada se želi vršiti poređenje izlaza testiranog modula sa očekivanim vrednostima jer je lakše vremenski pratiti događaje u simulaciji.

Verilog podržava i rad sa tekstualnim fajlovima. Moguće je čitati stimulus vrednosti iz fajla, a isto tako i zapisivati vrednosti generisane unutar simulacije u tekstualni fajl. Zapis vrednosti dobijenih simulacijom je pogodan za naknadnu softversku analizu ispravnosti rada dizajna. Instrukcija *$fopen* otvara fajlove, pri čemu se fajlovi mogu otvoriti u jednom od tri moda, mod ’*w*’ za pisanje u fajl pri čemu se stari sadržaj fajla briše, ’*a*’ za pisanje u fajla pri čemu se novi sadržaj dopisuje na kraj postojećeg sadržaja i ’*r*’ za čitanje iz fajla (*’w’-write, ’a’-append,’r’-read*). Instrukcija *$fscanf* vrši čitanje iz fajla, a *$fwrite* vrši upis u fajl. Strukture navedenih instrukcija su ekvivalentne istim funkcijama iz C jezika. Za definisanje formata pročitane vrednosti ili vrednosti koja se zapisuje tipično se koriste *%b* za binarni format, *%h* za heksadecimalni format i *%d* za decimalni format. Primer učitavanja stimulusa iz ulaznog fajla i upisivanje vrednosti u izlazni fajl za 4x4 svič:

```
`timescale 1ns / 1ps
//testbench za 4x4 svic
module testbench();
	parameter duzina_magistrale = 16;
	//simulirani ulazi u svic
	//simulirani data ulazi
	reg [(duzina_magistrale -1):0] stimulus_ulaz1;
	reg [(duzina_magistrale -1):0] stimulus_ulaz2;
	reg [(duzina_magistrale -1):0] stimulus_ulaz3;
	reg [(duzina_magistrale -1):0] stimulus_ulaz4;
	//simulirani kontrolni ulazi
	reg [1:0] stimulus_sel1;
	reg [1:0] stimulus_sel2;
	reg [1:0] stimulus_sel3;
	reg [1:0] stimulus_sel4;
	//za posmatranje izlaza svica
	wire [(duzina_magistrale -1):0] test_izlaz1;
	wire [(duzina_magistrale -1):0] test_ izlaz2;
	wire [(duzina_magistrale -1):0] test_ izlaz3;
	wire [(duzina_magistrale -1):0] test_ izlaz4;
	//takt i brojac signali
	reg clk;
	reg [2:0] brojac=3'b000;
	reg [7:0] global_brojac=8'h00; //za globalno merenje vremena u simulaciji
	//signali za fajlove
	integer handle1,handle2;//pokazivaci na fajlove
	integer EOF;//za odredjivanje kraja fajla koji se cita
	//interna struktura testbencha
	initial //initial blok za generisanje signala takta
	begin
		clk=1'b0;
		forever
		    #5 clk = ~clk;
	end
	initial //initial blok za otvaranje fajlova
	begin
		//difolt direktorijum je direktorijum projekta
		//putanja se navodi realtivno u odnosu na difolt direktorijum
		handle1 = $fopen("Sources/input.txt","r");
```

```
            handle2 = $fopen("Sources/output.txt","w");
end
always @(posedge clk) //always blok za generisanje stimulusa
begin
    case(brojac)
    3'b000:
    begin
            EOF = $fscanf(handle1, "%h %h %h %h %b %b %b %b",stimulus_ulaz1,stimulus_ ulaz2,stimulus_
ulaz3,stimulus_ ulaz4,stimulus_sel1,stimulus_sel2,stimulus_sel3,stimulus_sel4);
            if(EOF==1)
                $stop;
            end
    3'b001:
    begin
            EOF = $fscanf(handle1, "%h %h %h %h %b %b %b %b",stimulus_ulaz1,stimulus_ ulaz2,stimulus_
ulaz3,stimulus_ ulaz4,stimulus_sel1,stimulus_sel2,stimulus_sel3,stimulus_sel4);
            if(EOF==1)
                $stop;
    end
    3'b011:
    begin
            EOF = $fscanf(handle1, "%h %h %h %h %b %b %b %b",stimulus_ulaz1,stimulus_ ulaz2,stimulus_
ulaz3,stimulus_ ulaz4,stimulus_sel1,stimulus_sel2,stimulus_sel3,stimulus_sel4);
            if(EOF==1)
                $stop;
    end
    3'b100:
    begin
            EOF = $fscanf(handle1, "%h %h %h %h %b %b %b %b",stimulus_ulaz1,stimulus_ ulaz2,stimulus_
ulaz3,stimulus_ ulaz4,stimulus_sel1,stimulus_sel2,stimulus_sel3,stimulus_sel4);
            if(EOF==1)
                $stop;
    end
    3'b101:
    begin
            EOF = $fscanf(handle1, "%h %h %h %h %b %b %b %b",stimulus_ulaz1,stimulus_ ulaz2,stimulus_
ulaz3,stimulus_ ulaz4,stimulus_sel1,stimulus_sel2,stimulus_sel3,stimulus_sel4);
            if(EOF==1)
                $stop;
    end
    default:
    begin
    end
    endcase
    brojac=brojac+1'b1;
    //ispis izlazne vrednosti
    $fwrite(handle2, "%h\t%h\t%h\t%h\t%h\n", global_brojac, test_izlaz1, test_izlaz2, test_izlaz3, test_izlaz4);
    global_brojac=global_brojac+1'b1;
end

//switch instance that is tested
switch4x4
#(.N (duzina_magistrale))
switch_instanca
(
            .ulaz1(stimulus_ulaz1),
            .ulaz2(stimulus_ulaz2),
            .ulaz3(stimulus_ulaz3),
            .ulaz4(stimulus_ulaz4),
            .sel1(stimulus_sel1),
```

```
        .sel2(stimulus_sel2),
        .sel3(stimulus_sel3),
        .sel4(stimulus_sel4),
        .izlaz1(test_izlaz1),
        .izlaz2(test_izlaz2),
        .izlaz3(test_izlaz3),
        .izlaz4(test_izlaz4)
    );

endmodule
```

Iz datog primera se može videti da se pokazivači na fajlove deklarišu kao *integer* tipovi. Takođe, poželjno je definisati i signal koji će se koristiti za ispitivanje dostizanja kraja fajla koji se čita (*EOF* signal u primeru), koji je takođe integer tipa. Princip mapiranja promenjivih (signala) na pročitane vrednosti, odnosno na vrednosti koje se ispisuju je istog principa kao u C jeziku što se može videti iz datog primera. Instrukcija *$stop* prekida izvršavanje simulacije.