



Projektovanje digitalnih sistema

Behavioral modelovanje



■ *Behavioral* modelovanje

- Kompleksniji dizajn => dizajner mora više da vodi računa o mnogim aspektima (prednosti i mane različitih arhitektura, algoritama, ...)
- Evaluacija se vrši na algoritamskom nivou umjesto u smislu logičkih kapija ili *dataflow*-a (ponašanje algoritma i njegove performanse)
- Dizajner vrši opis **ponašanja** kola – visok nivo apstrakcije
- Veoma podsjeća na programski jezik C i mnoge konstrukcije isto izgledaju
- Verilog kôd kojim se opisuje ponašanje (*behavioral* kôd) se nalazi unutar proceduralnih blokova:
 - **initial** (*izvršava se jednom i to u vremenskom trenutku 0*)
 - **always** (*izvršava se neprekidno, počev od trenutka 0*)
- Svaki **initial** i **always** blok predstavlja zaseban tok aktivnosti u Verilogu
- **initial** i **always** se ne smiju ugnježdavati



■ *Behavioral* modelovanje – initial

- Ako ima više *initial* blokova svi počinju *izvršavanje* u trenutku 0 (konkurentno)
- *Izvršavanje* se završava nezavisno od ostalih blokova
- Višestruki *behavioral* iskazi se grupišu, obično pomoću ključnih riječi *begin* i *end* (slično vitičastim zagradama u C-u)

■ Behavioral modelovanje – initial primjer

```
module stimulus;  
reg x,y,a,b,m;  
initial  
    m = 1'b0; // jedan izraz; ne treba grupisanje  
initial  
    begin  
        #5 a = 1'b1; // više izraza; moraju se grupisati  
        #25 b = 1'b0;  
    end  
initial  
    begin  
        #10 x = 1'b0;  
        #25 y = 1'b1;  
    end  
initial  
    #50 $finish;  
endmodule
```

Sekvenca izvršavanja:

0	m = 1'b0;
5	a = 1'b1;
10	x = 1'b0;
30	b = 1'b0;
35	y = 1'b1;
50	\$finish;



■ *Behavioral* modelovanje – **always**

- Modeluje blok aktivnosti u digitalnom kolu koje se neprekidno ponavljaju
- Primjer: takt generator – mijenja logički nivo na polovini periode i aktivan je sve dok je kolo pod napajanjem

```
module takt;  
  reg clock;  
  //Inicijalizacija u početnom trenutku  
  initial  
    clock = 1'b0;  
  // Promjena na polovini periode (perioda je 20)  
  always  
    #10 clock = ~clock;  
  // Trajanje simulacije  
  initial  
    #1000 $finish;  
endmodule
```



■ *Behavioral* modelovanje – *always*

- C programeri bi izveli analogiju između *always* bloka i beskonačne petlje
- Dizajneri hardvera *always* blok gledaju kao neprekidnu aktivnost unutar digitalnog kola koja počinje njegovim uključivanjem
- Ova aktivnost se zaustavlja isključivanjem (*\$finish*) ili prekidom (*\$stop*)



■ Verilog: proceduralna dodjeljivanja

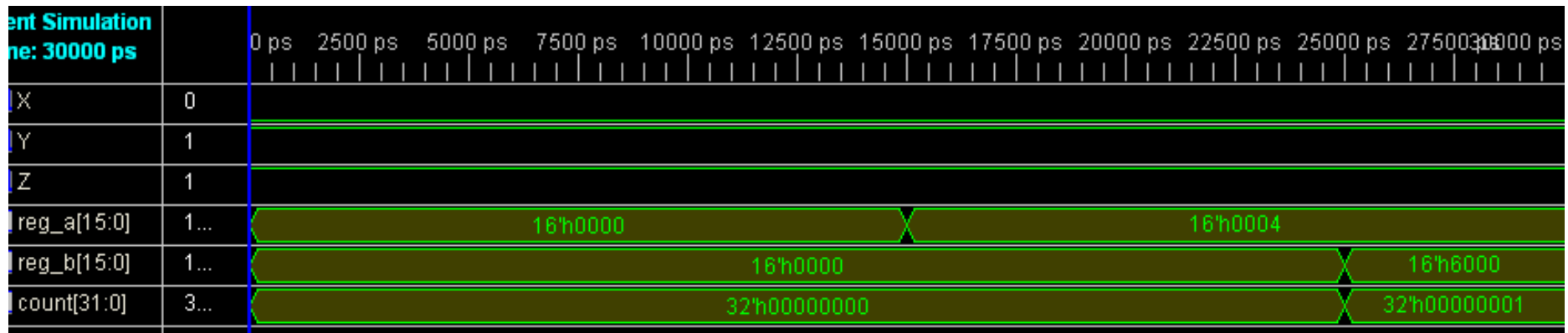
- Dodjeljivanje vrijednosti promjenljivima tipa **reg**, **integer**, **real**, **time**
- Dodijeljena vrijednost ostaje nepromijenjena dok se ne dodijeli nova vrijednost drugim proceduralnim dodjeljivanjem
- Sintaksa: <lvalue> = <izraz>
- <lvalue> može biti:
 - reg, integer, real ili time registarska promjenljiva ili memorijski element
 - pojedini bit ovih promjenljivih (npr. addr[0])
 - blok bitova ovih promjenljivih (npr. addr[31:16])
 - konkatencija nečeg od gore nabrojanog
- <izraz> može biti bilo šta što daje neku vrijednost
- Proceduralna dodjeljivanja mogu biti **blokirajuća** i **neblokirajuća**



■ Verilog: blokirajuće dodjeljivanje

- Blokirajuća dodjeljivanja se *izvršavaju* **redoslijedom kojim su specificirana** u sekvencijalnom bloku
- Blokirajuća dodjeljivanja **neće** blokirati *izvršavanje* izraza koji slijede u paralelnom bloku (o sekvencijalnom i paralelnom bloku – kasnije)
- Koristi se operator =

■ Verilog: blokirajuće dodjeljivanje – primjer



```

begin
  X=0; Y=1; Z=1; // skalarno dodjeljivanje
  count=0; // dodjeljivanje integer promjenljivoj
  reg_a=16'b0; reg_b=reg_a; // inicijalizacija vektora
  #15 reg_a[2] = 1'b1; // dodjeljivanje vrijednosti bitu, sa kašnjenjem
  #10 reg_b[15:13] = {X, Y, Z}; //dodjeljivanje vrijednosti grupi bitova
  count = count + 1; //dodjeljivanje integer-u (inkrement)
end
initial
  #30 $finish;
endmodule

```

■ Verilog: blokirajuće dodjeljivanje – primjer

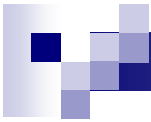
- U prethodnom primjeru izraz `Y=1` se *izvršava* nakon izraza `X=0` (sekvencijalno)
- Izraz `reg_b=reg_a` se *izvršava* nakon izraza `reg_a=16'b0`; **zato `reg_b` ima poznatu vrijednost**
- Izraz `count = count + 1` se *izvršava* posljednji
- Izrazi se *izvršavaju* u sljedećim vremenskim trenucima:
 - Svi izrazi od `X=0` do `reg_b=reg_a` u trenutku 0 (početak simulacije)
 - Izraz `reg_a[2] = 1'b1` u vremenskom trenutku 15
 - Izraz `reg_b[15:13] = {X, Y, Z}` u vremenskom trenutku 25
 - Izraz `count = count + 1` u vremenskom trenutku 25

```
X=0; Y=1; Z=1; count=0; reg_a=16'b0; reg_b=reg_a;  
#15 reg_a[2] = 1'b1;  
#10 reg_b[15:13] = {X, Y, Z};  
count = count + 1;
```

■ Verilog: blokirajuće dodjeljivanje – nastavak

- Ako sa desne strane jednakosti ima više bitova nego u registarskoj promjenljivoj sa lijeve strane, vrši se odsijecanje viška bitova – zadržavaju se bitovi manje težine (odbacuju se bitovi više težine)
- Ako sa desne strane ima manje bitova, popunjava se nulama na mjestima više težine

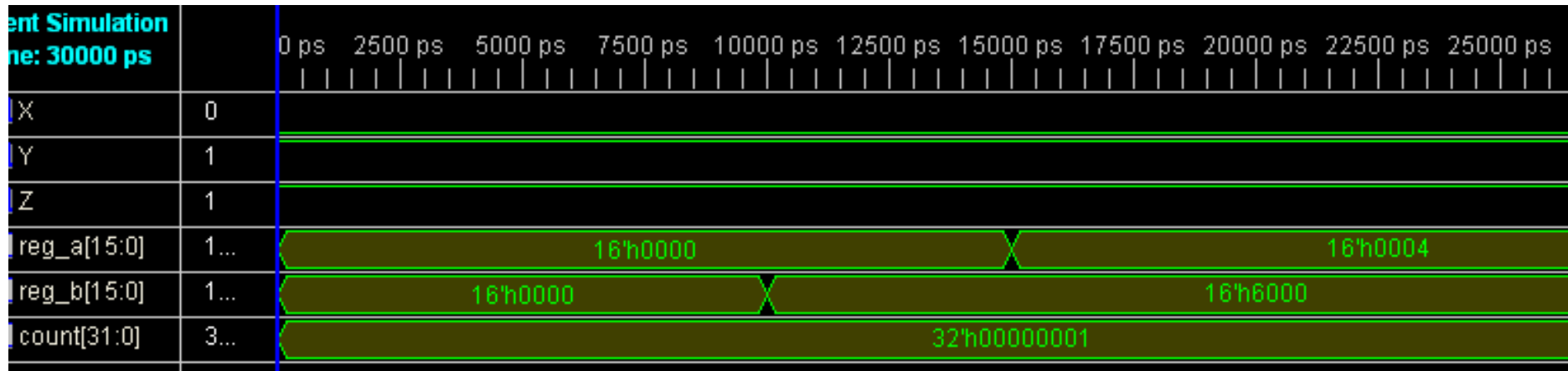
```
module blokirajuci2;  
  reg [15:0] reg_a, reg_b;  
  initial  
    begin  
      reg_a = 12'hfff; // dopunjava nulama: reg_a=16'h0fff  
      #5 reg_b = 16'b0;  
      #5 reg_b[7:0] = 16'h1234; // odsijecanje: reg_b=16'h0034  
    end  
  initial  
    #15 $finish;  
endmodule
```



■ Verilog: neblokirajuće dodjeljivanje

- Omogućava da se “zakaže” dodjeljivanje bez blokiranja narednih izraza u sekvencijalnom bloku
- Operator je `<=`
- Isti je simbol kao kod relacionog operatora “manje ili jednako” – interpretira se kao relacioni operator u izrazu, a kao operator dodjeljivanja u kontekstu neblokirajućeg dodjeljivanja

■ Verilog: neblokirajuće dodjeljivanje – primjer



```

begin
  X=0; Y=1; Z=1; // skalarno dodjeljivanje
  count=0; // dodjeljivanje integer promjenljivoj
  reg_a=16'b0; reg_b=reg_a; // inicijalizacija vektora
  reg_a[2] <= #15 1'b1; // dodjeljivanje vrijednosti bitu, sa kašnjenjem
  reg_b[15:13] <= #10 {X, Y, Z}; //dodjeljivanje vrijednosti grupi bitova
  count = count + 1; //dodjeljivanje integer-u (inkrement)
end
initial
  #30 $finish;
endmodule

```



■ Verilog: neblokirajuće dodjeljivanje – primjer

- Svi izrazi od `X=0` do `reg_b=reg_a` se izvršavaju sekvencijalno u trenutku 0 (početak simulacije)
- Nakon toga se tri neblokirajuća dodjeljivanja obrađuju u istom vremenskom trenutku:
 - Izvršavanje `reg_a[2]=0` se zakazuje za momenat nakon 15 vremenskih jedinica (*time=15*)
 - Izvršavanje `reg_b[15:13] = {X, Y, Z}` se zakazuje za momenat nakon 10 vremenskih jedinica (*time=10*)
 - Izvršavanje `count = count + 1` se zakazuje bez kašnjenja (*time=0*)

```
X=0; Y=1; Z=1; count=0; reg_a=16'b0; reg_b=reg_a;  
reg_a[2] <= #15 1'b1;  
reg_b[15:13] <= #10 {X, Y, Z};  
count = count + 1;
```



■ Verilog: primjena neblokirajućeg dodjeljivanja

- Modeluje nekoliko konkurentnih transfera podataka koji se dešavaju nakon nekog zajedničkog događaja
- Primjer: tri konkurentna transfera podataka nakon uzlazne ivice takta


```
always @(posedge clock)
begin
    reg1 <= #1 in1;
    reg2 <= @(negedge clock) in2 ^ in3;
    reg3 <= #1 reg1;
end
```

- Na svakoj pozitivnoj ivici signala *clock* obavlja se sljedeća sekvenca:
 - Izvršava se očitavanje svake promjenljive sa desne strane operatora (in1, in2, in3 i reg1), izrazi se izračunavaju i smještaju interno u simulatoru

■ Verilog: primjena neblokirajućeg dodjeljivanja

```
always @(posedge clock)
begin
    reg1 <= #1 in1;
    reg2 <= @(negedge clock) in2 ^ in3;
    reg3 <= #1 reg1;
end
```


- Operacije upisa u promjenljive sa lijeve strane operatora se “zakazuju” za vremenski trenutak specificiran kašnjenjem: **reg1** nakon 1 vr.jed., **reg2** na sljedeću silaznu ivicu takta, **reg3** nakon 1 vr.jed.
- Operacije upisa se izvršavaju u “zakazanim” vremenskim trenucima
- **Redoslijed** u kojem se operacije izvršavaju **nije važan** jer su vrijednosti sa desne strane interno zapamćene
- Npr. **reg3** će poprimiti staru vrijednost **reg1**, koja je sačuvana nakon faze čitanja, iako je u fazi upisivanja u **reg1** upisana nova vrijednost – prije upisivanja u **reg3**

- 
- Verilog: neblokirajuće dodjeljivanje – primjer 2
 - Zamjeniti sadržaje registara a i b, na svakoj pozitivnoj ivici *clock*-a
 - Varijanta 1 (sa blokirajućim dodjeljivanjem):

```
always @(posedge clock)
a = b;
always @(posedge clock)
b = a;
```

- Varijanta 2 (sa neblokirajućim dodjeljivanjem):

```
always @(posedge clock)
a <= b;
always @(posedge clock)
b <= a;
```

- 
- Verilog: neblokirajuće dodjeljivanje – primjer 2
 - Zamjeniti sadržaje registara **a** i **b**, na svakoj pozitivnoj ivici **clock-a**
 - Varijanta 1 (sa blokirajućim dodjeljivanjem):

```
always @(posedge clock)
a = b;
always @(posedge clock)
b = a;
```

- *Race condition*: ili će **a = b** biti izvršeno prije **b = a**, ili će biti obrnuto – zavisno od implementacije simulatora
- Rezultat: vrijednosti registara **a** i **b** neće biti zamijenjene, već će oba registra imati istu vrijednost (prethodnu vrijednost **a** ili **b** – zavisno od implementacije simulatora)



■ Verilog: neblokirajuće dodjeljivanje – primjer 2

- Zamjeniti sadržaje registara **a** i **b**, na svakoj pozitivnoj ivici **clock-a**

- Varijanta 2 (sa neblokirajućim dodjeljivanjem):

```
always @(posedge clock)
```

```
a <= b;
```

```
always @(posedge clock)
```

```
b <= a;
```

- Eliminira *race condition*
- Na pozitivnoj ivici **clock-a** promjenljive sa desne strane se “pročitaju”, izrazi se izračunaju i rezultati smještaju u privremene promjenljive
- Potom se, u fazi upisa, vrijednosti iz privremenih promjenljivih dodjeljuju promjenljivima sa lijeve strane
- Zahvaljujući odvajanju faza čitanja i upisa, vrijednosti registara **a** i **b** su zamijenjene, bez obzira na redoslijed *izvršavanja* operacija upisa



■ Verilog: neblokirajuće dodjeljivanje – zaključak

- Koristiti neblokirajuće dodjeljivanje umjesto blokirajućeg kada treba izvršiti konkurentne prenose podataka nakon zajedničkog *događaja*
- Za razliku od blokirajućeg dodjeljivanja, kod neblokirajućeg dodjeljivanja konačan rezultat ne zavisi od redoslijeda kojim se dodjeljivanja evaluiraju
- Neblokirajuća dodjeljivanja mogu degradirati performanse simulatora i uvećati zahtjev za memorijskim resursima
- *Rule of thumb*: blokirajuće dodjeljivanje koristiti kod kombinacione logike, a neblokirajuće dodjeljivanje kod sekvencijalne logike



■ Verilog: kontrola tajminga

- Postoje različite konstrukcije za kontrolu tajminga u *behavioral* modelovanju
- Ako nema iskaza za kontrolu tajminga, vrijeme simulacije ne “napreduje”
- Kontrole tajminga omogućavaju da se specificira vremenski trenutak u kome proceduralni izraz treba da se “izvrši”
- Tri su metoda za kontrolu tajminga:
 - ❖ *Delay-based* (bazirano na kašnjenju)
 - ❖ *Event-based* (bazirano na događaju)
 - ❖ *Level-sensitive* (bazirano na nivou)



■ *Delay-based* kontrola tajminga

- Specificira vremensko trajanje između trenutka kad se naišlo na neki iskaz i kad se on *izvršava*
- Kašnjenje se specificira pomoću simbola #
- Može se specificirati korišćenjem broja, identifikatora ili izraza (min:typ:max)
- Tri su tipa kontrole kašnjenja kod proceduralnih dodjeljivanja:
 - Regularna kontrola kašnjenja (*regular delay control*)
 - Kontrola kašnjenja unutar dodjeljivanja (intra-assignment delay control)
 - Nulta kontrola kašnjenja (zero delay control)



■ Regularna *delay* kontrola tajminga

- Nenulta vrijednost kašnjenja se specificira na lijevoj strani proceduralnog dodjeljivanja

- Primjer:

```
//definicija parametara
parameter latency = 20;
parameter delta = 2;
// definicija registarskih promjenljivih
reg x, y, z, p, q;
initial
begin
    x = 0; // nema kontrole kašnjenja
    #10 y = 1; // specificirano brojem; kašnjenje 10 jedinica
    #latency z = 0; //specificirano identifikatorom; kašnjenje 20 jedinica
    # (latency + delta) p = 1; // specificirano izrazom
    #y x = x + 1; // specificirano identifikatorom; uzima vrijednost y
    #(4:5:6) q = 0; // minimalna, tipična i maksimalna vrijednost kašnjenja
end
```



■ *Intra-assignment delay* kontrola tajminga

- Kašnjenje se specificira sa desne strane operatora dodjeljivanja

- Primjer:

// definicija registarskih promjenljivih

reg x, y, z;

initial

begin

x = 0; z = 0;

y = #5 x + z; // uzima vrijednosti x i z u *time=0*, izračunava x+z i

// čeka 5 vremenskih jedinica da dodijeli vrijednost promjenljivoj y

end

// ekvivalentan metod pomoću privremene promjenljive i regularne kontrole

initial

begin

x = 0; z = 0;

temp_xz = x + z;

#5 y = temp_xz; // ako se x i z promijene to neće uticati na rezultat

end

■ *Zero delay* kontrola tajminga

- Proceduralni iskazi u različitim *always-initial* blokovima se mogu evaluirati u istom vremenskom trenutku simulacije
- Redoslijed *izvršavanja* takvih iskaza je neodređen
- *Zero delay* kontrola tajminga se koristi da osigura da se iskaz *izvrši* poslednji, nakon svih ostalih koji se *izvršavaju* u istom trenutku
- Time se eliminiše *race condition*
- Ako ima više *zero delay* iskaza, njihov redoslijed nije određen

■ Primjer:

initial

begin

x = 0; y = 0;

end

initial

begin

#0 x = 1; #0 y = 1;

end

x=1 i y=1 će se izvršiti nakon *x=0 i y=0*, iako se svi izvršavaju u *time=0*

Dakle, zahvaljujući *#0*, *x* i *y* će sigurno imati vrijednost 1

Ovo je dato kao ilustracija: dodjeljivanje različitih vrijednosti istoj promjenljivoj u istom trenutku treba izbjegavati!



■ *Event-based* kontrola tajminga

- *Događaj* je svaka promjena vrijednosti registarske ili *net* promjenljive
- Događaji se mogu iskoristiti da aktiviraju *izvršavanje* iskaza ili bloka iskaza
- Postoje 4 tipa *event-based* kontrole tajminga:
 - *Regular event control*
 - *Named event control*
 - *Event OR control*
 - *Level-sensitive control*



■ *Regular event control*

- Specificira se uz pomoć simbola @
- Iskazi se mogu *izvršavati* na promjenu vrijednosti signala ili na pozitivnu odnosno negativnu tranziciju vrijednosti signala

@(clock) q = d; // q=d se izvršava kad god *clock* promijeni vrijednost

@(posedge clock) q = d; // q=d se izvršava na pozitivnu tranziciju
// clock-a : sa 0 na 1, x ili z; sa x na 1; sa z na 1

@(negedge clock) q = d; // q=d se izvršava na negativnu tranziciju
// clock-a : sa 1 na 0, x ili z; sa x na 0; sa z na 0

q = @(posedge clock) d; // d se evaluira odmah, a dodjeljuje se
// promjenljivoj q na pozitivnu tranziciju clock-a



■ *Named event control*

- Deklariše se događaj pomoću ključne riječi **event**
- Događaj se aktivira pomoću simbola ->
- Aktiviranje događaja se prepoznaje pomoću simbola @

Primjer bafera za podatke u koji se podaci smještaju nakon što pristigne posljednji paket podataka:

```
event received_data; // definisanje događaja po imenu received_data
```

```
always @(posedge clock) // provjeri na svakoj pozitivnoj tranziciji clock-a  
begin
```

```
    if (last_data_packet) // ako je ovo posljednji paket  
        ->received_data; // aktiviraj događaj received_data
```

```
end
```

```
always @(received_data) // cekanje da se aktivira događaj received_data  
    data_buf={data_pkt[0] , data_pkt[1] , data_pkt[2] , data_pkt[3]};
```



■ *Event OR control*

- Nekad tranzicija bilo kog od više posmatranih signala treba da aktivira *izvršavanje* iskaza ili bloka iskaza
- Ključna riječ **or** se koristi za specificiranje višestrukih “okidača”
- Lista događaja ili signala izražena pomoću **or** se naziva *sensitivity list*

// level-sensitive latch sa asinhronim resetom

always @(reset or clock or d) // čeka na promjenu signala reset, clock i d

begin

if (reset) // ako je reset signal visok postavi q na 0

q = 1'b0;

else if(clock) // ako je clock visok upiši stanje na ulazu

q = d;

end



■ *Level-sensitive control*

- Simbol @ omogućava kontrolu na bazi **promjene** (ivice) signala
- Nekad je potrebno sačekati da se ispuni određeni uslov da bi se *izvršio* iskaz ili blok iskaza
- Koristi se ključna riječ **wait**

always

```
wait (count_enable) #20 count = count + 1;
```

- Vrijednost signala *count_enable* se neprestano nadgleda
- Ako je njegova vrijednost 0, iskaz se neće *izvršiti*
- Ako je njegova vrijednost 1, iskaz *count = count + 1* će se *izvršiti* nakon 20 vremenskih jedinica
- Ako je njegova vrijednost i dalje 1, *count* će se inkrementirati svakih 20 vremenskih jedinica



■ Uslovni iskazi

- Odlučivanje, na osnovu određenih uslova, da li će se neki iskaz(i) *izvršiti*
- Ključne riječi su **if** i **else**
- Tri su tipa uslovnih iskaza:

- Nema **else** – iskaz(i) se *izvršava(ju)* ili se ne *izvršava(ju)*

if (<neki_izraz>) iskazi_za_tacan_izraz;

- Ima jedno **else** – *izvršava(ju)* se ili iskaz(i) za tačan uslov ili za netačan uslov

if (<neki_izraz>) iskazi_za_tacan_izraz; else iskazi_za_netacan_izraz;

- Ugnježdjeni **if-else** – višestruki izbor, samo se jedan *izvršava*

if (<izraz1>) iskazi_za_tacan_izraz1;
else if (<izraz2>) iskazi_za_tacan_izraz2;
else if (<izraz3>) iskazi_za_tacan_izraz3;
else iskazi_za_sve_gornje_netacno;



■ Uslovni iskazi – nastavak

- Kod uslovnih iskaza prvo se evaluira <izraz> uz **if**
 - Ako je tačan (1 ili nenulta vrijednost) *izvršava(ju)* se iskaz(i) za tačan izraz
 - Ako je netačan (0) ili nepoznat (x ili z) *izvršava(ju)* se iskaz(i) za netačan izraz
- Iskaz (za tačno ili netačno) može biti pojedinačni iskaz ili grupa (blok) iskaza
- Blok se mora grupisati ključnim riječima **begin** i **end**
- Pojedinačni iskaz se ne mora grupisati



■ Uslovni iskazi – primjeri

// prvi tip uslovnog iskaza

if(!lock) buffer = data;

if(enable) izlaz = ulaz;

// drugi tip uslovnog iskaza

if (broj_baferovanih < MAX_DUZINA)

begin

bafer = data;

broj_baferovanih = broj_baferovanih + 1;

end

else

\$display ("Bafer je popunjen");



■ Uslovni iskazi – primjeri

// treći tip uslovnog iskaza

```
if (alu_control == 0)
```

```
    y = x + z;
```

```
else if(alu_control == 1)
```

```
    y = x - z;
```

```
else if(alu_control == 2)
```

```
    y = x * z;
```

```
else
```

```
    $display("Neispravan ALU controlni signal");
```



■ Uslovni iskazi – nastavak

- Ako ima mnogo ugnježđenih **if-else-if**, kod trećeg tipa uslovnih iskaza, postaje nepregledno pratiti sve moguće uslove
- Ovakvi slučajevi se lakše zapisuju pomoću **case** direktive
- Koriste se ključne riječi **case**, **endcase** i **default**

```
case (izraz)
    varijanta1: iskaz1;
    varijanta2 : iskaz2;
    varijanta3 : iskaz3;
    default: podrazumijevani_iskaz; // nijedna od varijanti nije tačna
endcase
```

- `iskaz1, iskaz2, ..., podrazumijevani_iskaz` mogu biti pojedinačni iskazi ili blokovi iskaza uokvireni sa **begin** i **end**
- **izraz** se poredi sa **varijantama** onim redoslijedom kojim su navedene
- *Izvršava* se iskaz koji pripada prvoj varijanti koja se podudara sa izrazom



■ Uslovni iskazi – nastavak

- Ako se izraz ne podudara ni sa jednom varijantom, *izvršava* se iskaz koji odgovara **default**-u
- **Default** je opcion – nije obavezan
- **case** iskazi se mogu ugniježdavati
- Primjer:

```
// izvršavanje iskaza zavisno od vrijednosti ALU kontrolnog signala  
reg [1:0] alu_control;
```

```
...
```

```
case (alu_control)  
    2'd0 : y = x + z;  
    2'd1 : y = x - z;  
    2'd2 : y = x * z;  
    default: $display("Neispravan ALU controlni signal");  
endcase
```

■ Uslovni iskazi – nastavak

■ Primjer: Multiplekser 4/1

```
module mux4_to_1(out, i0, i1, i2, i3, s1, s0);  
    output out;  
    input i0, i1, i2, i3, s1, s0;  
    reg out;  
    always @(s1 or s0 or i0 or i1 or i2 or i3)  
        case ({s1, s0}) // izbor se zasniva na konkatenciji signala  
            2'd0 : out = i0;  
            2'd1 : out = i1;  
            2'd2 : out = i2;  
            2'd3 : out = i3;  
            default: $display("neispravni kontrolni signali");  
        endcase  
    endmodule
```



■ Uslovni iskazi – nastavak

- Porede se 0, 1, x i z vrijednosti u izrazu i u ponuđenim varijantama, bit po bit
- Ako su izraz i varijanta nejednake dužine (broja bita) kraći se dopunjava nulama
- Ako postoji više varijanti vrijednosti izraza za koje treba da se *izvrši* isti blok, mogu se staviti zajedno, odvojeni zarezima:
 - umjesto:
2'd0 : out = i0;
2'd1 : out = i0;
2'd2 : out = i0;
 - može:
2'd0, 2'd1, 2'd2 : out = i0;

■ Uslovni iskazi – nastavak

- Primjer: demultiplekser 1/4 sa potpunim definisanjem selekcionih signala

```
module demultiplexer1_to_4 (out0, out1, out2, out3, in, s1, s0);  
... // deklaracije portova  
always @(s1 or s0 or in)  
case ({s1, s0})  
    2'b00 : begin out0 = in; out1 = 1'bz; out2 = 1'bz; out3 = 1'bz; end  
    2'b01 : begin out0 = 1'bz; out1 = in; out2 = 1'bz; out3 = 1'bz; end  
    2'b10 : begin out0 = 1'bz; out1 = 1'bz; out2 = in; out3 = 1'bz; end  
    2'b11 : begin out0 = 1'bz; out1 = 1'bz; out2 = 1'bz; out3 = in; end  
    2'bx0, 2'bx1, 2'bxz, 2'bxx, 2'b0x, 2'b1x, 2'bzx : // x ima prioritet nad z  
        begin out0 = 1'bx; out1 = 1'bx; out2 = 1'bx; out3 = 1'bx; end  
    2'bz0, 2'bz1, 2'bzz, 2'b0z, 2'b1z :  
        begin out0 = 1'bz; out1 = 1'bz; out2 = 1'bz; out3 = 1'bz; end  
    default: $display("Nespecificirani kontrolni signali");  
endcase  
endmodule
```



■ Uslovni iskazi – nastavak

- Postoje dvije podvarijante **case** direktive: **casex** i **casez**
- **casez** tretira sve **z** vrijednosti u *case* alternativama ili *case* izrazu kao “bilo šta” vrijednosti; pozicije sa vrijednošću **z** se mogu predstaviti i sa ?
- **casex** tretira sve **x** i **z** vrijednosti u *case* alternativama ili *case* izrazu kao “bilo šta” vrijednosti
- Pomoću **casex** i **casez** se mogu porediti samo pozicije koje nisu **x** ili **z** u *case* izrazu i *case* alternativama

■ Uslovni iskazi – nastavak

- **Primjer.** dekodiranje bitova u četvorobitnom podatku – samo se jedan bit posmatra da se definiše sljedeće stanje, a ostali bitovi se ignorišu

```
reg [3:0] podatak;  
integer sljedece_stanje;  
case (podatak) // vrijednost x predstavlja "bilo šta" bit  
    4'b1xxx : sljedece_stanje = 3;  
    4'bx1xx : sljedece_stanje = 2;  
    4'bxx1x : sljedece_stanje = 1;  
    4'bxxx1 : sljedece_stanje = 0;  
    default : sljedece_stanje = 0;  
endcase
```

- Vrijednost `podatak = 4'b10xz` bi rezultiralo sa `sljedece_stanje = 3`



■ Petlje

- Postoje 4 tipa petlji u Verilogu: **while**, **for**, **repeat** i **forever**.
- Sintaksa je vrlo slična kao u programskom jeziku C
- Petlje se mogu pojaviti samo unutar **initial** ili **always** bloka
- Unutar petlji se mogu nalaziti izrazi za unošenje kašnjenja

■ **While** petlja

- Izvršava se dok uslov petlje ne postane netačan
- Ako se u petlju uđe dok je uslov netačan, petlja se neće uopšte izvršiti
- Ako je unutar petlje više iskaza, grupišu se sa **begin** i **end**



■ Primjer **while** petlje

```
// Inkrementira promjenljivu brojac od 0 do 127
// Izlazi iz petlje za brojac==128.
// Stalno prikazuje vrijednost promjenljive brojac
integer brojac;
initial
begin
    brojac = 0;
    while (brojac < 128)
    begin
        $display("brojac = %d", brojac);
        brojac = brojac + 1;
    end
    $finish;
end
```

■ Primjer **while** petlje

// Nadji prvi bit sa log. 1 u vektorskoj promjenljivoj

`define TRUE 1'b1;

`define FALSE 1'b0;

reg [15:0] flag;

integer i; // brojac

reg continue; // za prekid petlje

initial

begin

flag = 16'b 0010_0000_0000_0000; i = 0; continue = `TRUE;

while((i < 16) && continue)

begin

if (flag[i])

begin

\$display("Nadjena log. jedinica na bitu broj %d", i);

continue = `FALSE;

end

i = i + 1;

end

end



■ For petlja

- Sadrži tri dijela:

- Inicijalizaciju

- Provjeru tačnosti uslova za prekid petlje

- Proceduralno dodjeljivanje vrijednosti kontrolnoj promjenljivoj

- Primjer brojača sa while petljom preko for petlje:

```
integer brojac;
```


```
initial
```

```
for (brojac=0; brojac < 128; brojac = brojac + 1)
```

```
    $display("brojac = %d", brojac);
```

- Inicijalizacija i promjena kontrolne promjenljive su uključeni u tijelo **for** petlje i ne moraju se navoditi odvojeno => kompaktnija struktura nego kod **while** petlje

- **While** petlja, sa druge strane, ima šire polje upotrebe



■ For petlja – nastavak

■ *Primjer:* inicijalizacija niza

```
`define MAX_DUZINA 32
integer mem[0: `MAX_DUZINA - 1];
integer i;
initial
begin
    for(i = 0; i < 32; i = i + 2) // inicijalizacija parnih lokacija sa nulom
        state[i] = 0;
    for(i = 1; i < 32; i = i + 2) // inicijalizacija neparnih lokacija sa 1
        state[i] = 1;
end
```

■ Repeat petlja

- Repeat petlja se *izvršava* konstantan broj puta
- Ne može se koristiti za petlju koja zavisi od logičkog uslova
- Repeat konstrukcija mora sadržati broj ponavljanja, u obliku konstante, promjenljive ili vrijednosti signala
- Ako je u pitanju promjenljiva ili signal, **vrijednost se evaluira samo prilikom započinjanja petlje**, a ne tokom njenog izvršavanja
- **Primjer** ranijeg brojača preko *repeat* petlje:

```
initial
begin
    brojac= 0;
    repeat(128)
        begin
            $display(" brojac = %d", brojac);
            brojac = brojac + 1;
        end
    end
end
```



■ Repeat petlja - primjer

```
module data_buffer (data_start, data, clock);  
    parameter cycles = 8; // Bafer za podatke koji ih prihvata na pozitivnoj  
    input data_start, clock; // ivici takta u toku 8 ciklusa nakon što primi  
    input [15:0] data; // start signal  
    reg [15:0] buffer [0:7];  
    integer i;  
    always @(posedge clock)  
    begin  
        if (data_start) // start signal  
        begin  
            i = 0;  
            repeat (cycles) // smješta podatke na sljedećih 8 ivica takta  
            begin  
                @(posedge clock) buffer[i] = data;  
                i = i + 1;  
            end  
        end  
    end  
end  
endmodule
```



■ Forever petlja

- Forever petlja ne sadrži nikakve izraze i izvršava se neprekidno
- Ekvivalentno sa `while(1)`
- Tipično se koristi zajedno sa konstrukcijama za kontrolu tajminga
- U suprotnom bi simulator beskonačno izvršavao ovaj iskaz bez napredovanja u vremenskom domenu => ostatak dizajna nikad se ne bi “izvršio”
- *Primjer:* generator takta

```
reg clock;  
initial  
begin  
    clock = 1'b0;  
    forever #10 clock = ~clock; // perioda je 20 vremenskih jedinica  
end
```



■ Forever petlja – nastavak

- *Primjer.* sinhronizovati vrijednosti dva registra na svakoj pozitivnoj ivici takta

```
reg clock;
```

```
reg X, Y;
```

```
initial
```

```
    forever @(posedge clock) X = Y;
```