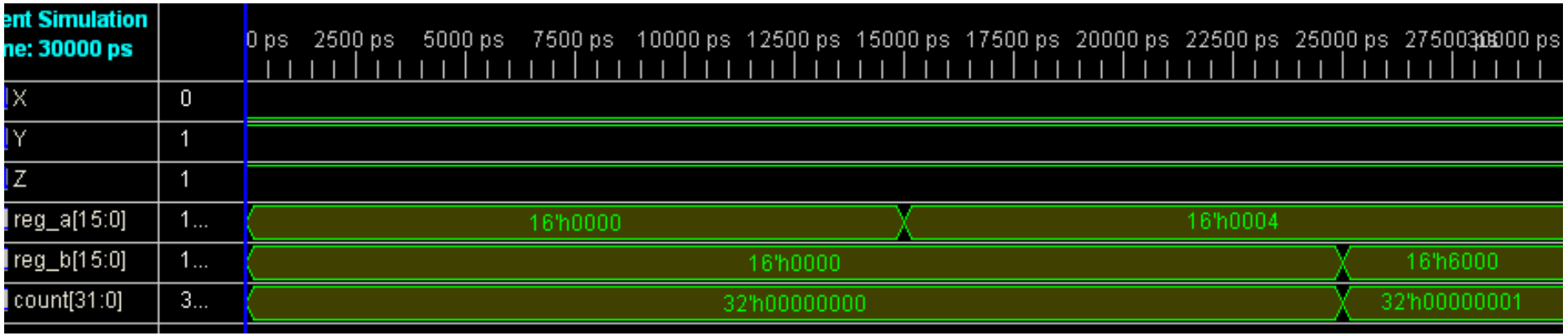
Verilog: **proceduralna dodjeljivanja**

- 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])
  - konkatenacija 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

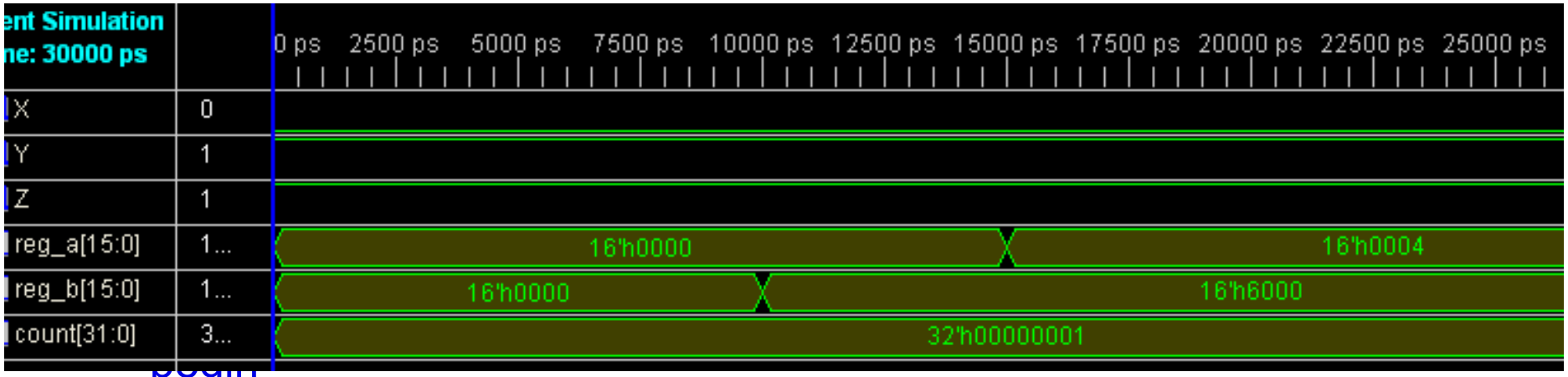
- Ako sa desne strane znaka jednako ima više bitova nego u registarskoj promjenljivoj sa lijeve strane, vrši se odsijecanje viška bitova – zadržavaju se bitovi manje težine (odbacuju se bitovi više težine)
- Ako sa desne strane jednakosti ima manje bitova, popunjava se nulama na mjestima više težine

```
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
```

Ako bi se kašnjenje stavilo ispred dodjeljivanja:

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

svelo bi se na blokirajuće dodjeljivanje

Verilog: neblokirajuće dodjeljivanje – primjer

- Svi izrazi od X=0 do reg_b=reg_a se izvršavaju sekvencijaljo u trenutku 0 (početak simulacije)
- Nakon toga se tri neblokirajuća dodjeljivanja obrađuju u istom vremenskom trenutku:
  - *Izvršavanje* reg_a[2]=1 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 signala *clock*, **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;
```

- Eliminiše *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 – primjer 3

- sa neblokirajućim dodjeljivanjem:

  Početne vrijednosti: A=B=C=E=1

  A <= B + C;

  D <= A + E;

  Nakon dodjeljivanja:

  A=2, D=2

- sa blokirajućim dodjeljivanjem:

  Početne vrijednosti: A=B=C=E=1

  A = B + C;

  D = A + E;

  Nakon dodjeljivanja:

  A=2, D=3

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; // minim., 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 3 tipa *event-based* kontrole tajminga:
  - *Regular event control*
  - *Named event control*
  - *Event OR 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 stigao_podatak; // definisanje događaja po imenu stigao_podatak

always @(posedge clock) // provjeri na svakoj pozitivnoj tranziciji clock-a
      begin
         if (poslednji_paket_podatka) // ako je ovo posljednji paket
             ->stigao_podatak;  // aktiviraj događaj stigao_podatak
      end

always @(stigao_podatak) // čekanje da se aktivira događaj stigao_podatak
   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 signal visok upiši stanje sa ulaza
            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

Edge-Sensitive vs. Signal-Sensitive always blokovi

- Ako želimo da implementiramo registar, koristimo *edge-sensitive always* blok (posedge, negedge)
- Ako želimo da implementiramo kombinacionu logiku, koristimo *signal-sensitive always* blok; ili alternativno – *assign* iskaz
  - *Kada koristimo signal-sensitive always* blok, umjesto iskaza

    always @ (a or b or c), može se koristiti iskaz always @(*)
  - * automatski uključuje sve signale koji se koriste unutar *always* bloka u *sensitivity* listu
- *always* blok ne može sadržati istovremeno *edge-sensitive* i *signal-sensitive* signale: always @(a or posedge b) je nelegalni iskaz (jedan *always* blok opisuje ili registar ili kombinacionu logiku
  - Napomena: u nekim implementacijama Verilog alata je ovo ipak dozvoljeno, ali treba izbjegavati

Edge-Sensitive vs. Signal-Sensitive always blokovi

- Nema smisla specificirati nekompletnu *sensitivity* listu
- Sledeći kod je legalan, ali nema smisla:

```
reg izlaz;
always @(a)
    begin
        izlaz = a & b & c;
    end
```

- Treba imati na umu da *signal-sensitive always* blok opisuje logičke kapije: ne postoji troulazno AND kolo čiji se izlaz mijenja kad se mijenja samo jedan ulaz
- Od alata zavisi kako će se ponašati u ovakvoj situaciji
- Nota bene: *izlaz* nije registar; *reg izlaz* znači samo da će *izlaz*-u biti dodijeljena vrijednost u *always* bloku; da li će zaista biti registar zavisi od tipa *always* bloka (edge-sensitive -> registar; signal-sensitive -> kombinaciono kolo)