



# Projektovanje digitalnih sistema

Operatori – nastavak

## ■ Verilog: operatori redukcije

Operator	Opis
&	AND
~&	NAND
	OR
~	NOR
^	XOR
^~ ili ~^	XNOR

- Imaju samo **jedan operand** (vektor)
- Izvršavaju operaciju nad bitovima vektora i daju **jednobitni** rezultat
- Operacija se izvršava bit po bit, sa desna na lijevo
- Redukcioni NAND, NOR i XNOR se izračunavaju invertovanjem rezultata redukcionog AND, OR i XOR, respektivno

## ■ Verilog: operatori redukcije – primjer

```
module redukcioni_operatori;
```

```
initial begin
```

```
// Bit Wise AND redukcija
```

```
$display (" & 4'b1001 = %b", (& 4'b1001));
```

```
$display (" & 4'bx111 = %b", (& 4'bx111));
```

```
$display (" & 4'bx101 = %b", (& 4'bx101));
```

```
$display (" & 4'bz111 = %b", (& 4'bz111));
```

```
// Bit Wise NAND redukcija
```

```
$display (" ~& 4'b1001 = %b", (~& 4'b1001));
```

```
$display (" ~& 4'bx001 = %b", (~& 4'bx001));
```

```
$display (" ~& 4'bx111 = %b", (~& 4'bx111));
```

```
$display (" ~& 4'bz001 = %b", (~& 4'bz001));
```

```
// Bit Wise OR redukcija
```

```
$display (" | 4'b1001 = %b", (| 4'b1001));
```

```
$display (" | 4'bx000 = %b", (| 4'bx000));
```

```
$display (" | 4'bz000 = %b", (| 4'bz000));
```

```
#10 $finish;
```

```
end
```

```
endmodule
```

Rezultat:

```
& 4'b1001 = 0
```

```
& 4'bx111 = x
```

```
& 4'bx101 = 0
```

```
& 4'bz111 = x
```

```
~& 4'b1001 = 1
```

```
~& 4'bx001 = 1
```

```
~& 4'bx111 = x
```

```
~& 4'bz001 = 1
```

```
| 4'b1001 = 1
```

```
| 4'bx000 = x
```

```
| 4'bz000 = x
```

## ■ Verilog: operatori redukcije – primjer

```
module redukcioni_operatori();
initial begin
  // Bit Wise OR redukcija
  $display (" ~| 4'b1001 = %b", (~| 4'b1001));
  $display (" ~| 4'bx001 = %b", (~| 4'bx001));
  $display (" ~| 4'bz001 = %b", (~| 4'bz001));
  // Bit Wise XOR redukcija
  $display (" ^ 4'b1001 = %b", (^ 4'b1001));
  $display (" ^ 4'bx001 = %b", (^ 4'bx001));
  $display (" ^ 4'bz001 = %b", (^ 4'bz001));
  // Bit Wise XNOR redukcija
  $display (" ~^ 4'b1001 = %b", (~^ 4'b1001));
  $display (" ~^ 4'bx001 = %b", (~^ 4'bx001));
  $display (" ~^ 4'bz001 = %b", (~^ 4'bz001));
  #10 $finish;
end
endmodule
```

Rezultat:

```
~| 4'b1001 = 0
~| 4'bx001 = 0
~| 4'bz001 = 0
^ 4'b1001 = 0
^ 4'bx001 = x
^ 4'bz001 = x
~^ 4'b1001 = 1
~^ 4'bx001 = x
~^ 4'bz001 = x
```

## ■ Verilog: operatori redukcije – primjer upotrebe

- Napraviti osmo-ulazno AND kolo

```
module and8 (y, a);  
    output y;  
    input [7:0] a;
```

```
    assign y = &a; // mnogo je lakše napisati y = &a nego
```

```
        // assign y = a[7] & a[6] & a[5] & a[4] & a[3] & a[2] & a[1] & a[0];
```

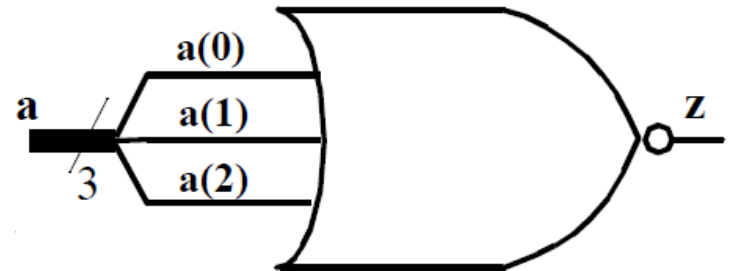
```
endmodule
```

- Napraviti detektor nule za trobitni vektor

```
module chk_zero (z, a);  
    output z;  
    input [2:0] a;
```

```
    assign z = ~| a; // Redukcija NOR
```

```
endmodule
```



## ■ Verilog: operatori pomjeranja

Operator	Opis
<<	pomjeranje ulijevo
>>	pomjeranje udesno

- Pomjeraju bitove vektorskog operanda za specificirani broj bitova
- Operandi su vektor i broj bitova za koje treba pomjeriti sadržaj vektora
- Upraznjena mjesta se popunjavaju nulama
- Korisno za implementaciju pomjeračkih registara, množenje sa stepenom dvojke, množenje sabiranjem,...

## ■ Verilog: operatori pomjeranja – primjer

```
module shift_operatori;
initial begin
    // Pomjeranje u lijevu stranu
    $display (" 4'b1001 << 1 = %b", (4'b1001 << 1));
    $display (" 4'b10x1 << 2 = %b", (4'b10x1 << 2));
    $display (" 4'b10z1 << 1 = %b", (4'b10z1 << 1));
    // Pomjeranje u desnu stranu
    $display (" 4'b1001 >> 1 = %b", (4'b1001 >> 1));
    $display (" 4'b10x1 >> 1 = %b", (4'b10x1 >> 1));
    $display (" 4'b10z1 >> 1 = %b", (4'b10z1 >> 1));
    #10 $finish;
end
endmodule
```

Rezultat:

```
4'b1001 << 1 = 0010
4'b10x1 << 2 = x100
4'b10z1 << 1 = 0z10
4'b1001 >> 1 = 0100
4'b10x1 >> 1 = 010x
4'b10z1 >> 1 = 010z
```

## ■ Verilog: operator konkatencije (nastavljanja)

Operator	Opis
{ }	konkatencija

- Obezbjeđuje mehanizam za “spajanje” više operanada
- Operandi **moraju** imati naznačenu veličinu
- Konkatencija se označava vitičastim zagradama pri čemu se operandi razdvajaju zarezima
- Operandi mogu biti skalarni *net* ili *reg*, vektorski *net* ili *reg*, selektovani bit, selektovana sekvenca bitova ili konstanta sa naznačenom dužinom

```
reg A;  
reg [1:0] B, C;  
reg [2:0] D;  
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;  
Y = {B , C} // Rezultat Y je 4'b0010  
Y = {A, B, C, D, 3'b001} // Rezultat Y je 11'b10010110001  
Y = {A, B[0], C[1]} // Rezultat Y je 3'b101
```



## ■ Verilog: operator replikacije

Operator	Opis
{ { } }	replikacija

- Uzastopno nastavljjanje istog broja
- Konstanta replikacije specificira koliko puta se ponavlja broj unutar vitičastih zagrada

```
reg A;  
reg [1:0] B, C;  
A = 1'b1; B = 2'b00; C = 2'b10;  
Y = { 4{A} } // Rezultat Y je 4'b1111  
Y = { 4{A}, 2{B} } // Rezultat Y je 8'b11110000  
Y = { 4{A}, 2{B}, C } // Rezultat Y je 10'b1111000010
```

## ■ Verilog: operatori konkatencije i replikacije – primjer

```
module operatori_konk_repl;
```

```
initial begin
```

```
    // konkatencija
```

```
    $display ("{4'b1001,4'b10x1} = %b", {4'b1001,4'b10x1});
```

```
    // replikacija
```

```
    $display ("{4{4'b1001}} = %b", {4{4'b1001}});
```

```
    // replikacija i konkatencija
```

```
    $display ("{4{4'b1001,1'bz}} = %b", {4{4'b1001,1'bz}});
```

```
    #10 $finish;
```

```
end
```

```
endmodule
```

Rezultat:

```
{4'b1001,4'b10x1} = 100110x1
```

```
{4{4'b1001}} = 1001100110011001
```

```
{4{4'b1001,1'bz}} = 1001z1001z1001z1001z
```

## ■ Verilog: uslovni operator

Operator	Opis
? :	uslovni operator

- Uzima tri operanda: *izraz\_uslova* ? *izraz\_tačan* : *izraz\_netačan* ;
- Prvo se evaluira *izraz\_uslova*
  - Ako je rezultat tačan (log. 1) evaluira se dio *izraz\_tačan*
  - Ako je rezultat netačan (log. 0) evaluira se dio *izraz\_netačan*
  - Ako je rezultat neodređen (x) evaluiraju se i *izraz\_tačan* i *izraz\_netačan* i njihovi rezultati se porede bit po bit: na svakoj poziciji se dobija **x** ako su bitovi različiti, a vrijednost tih bitova ako su isti
- Ponašanje uslovnog operatora je slično radu multipleksera
- Može se uočiti sličnost sa *if – else* izrazom



## ■ Verilog: uslovni operator – nastavak

// modelovanje funkcionalnosti *tristate* bafera

```
assign addr_bus = drive_enable ? addr_out : 32'bz;
```

// modelovanje funkcionalnosti multipleksora 2/1

```
assign izlaz = select ? ulaz1 : ulaz0;
```

- Uslovni operatori se mogu ugnijezditi: i *izraz\_tačan* i *izraz\_netačan* mogu biti uslovni operatori

```
assign izlaz = (A == 3) ? ( select ? x : y) : ( select ? m : n) ;
```

## ■ Verilog: uslovni operator – primjer

```
module uslovni_operator;
wire out;
reg enable, data;
assign out = (enable) ? data : 1'bz; // Tri-state bafer
initial begin
    $display ("time\t enable data out");
    $monitor ("%g\t %b    %b  %b", $time, enable, data, out);
    enable = 0;
    data = 0;
    #1 data = 1;
    #1 data = 0;
    #1 enable = 1;
    #1 data = 1;
    #1 data = 0;
    #1 enable = 0;
    #10 $finish;
end
endmodule
```

Rezultat:

time	enable	data	out
0	0	0	z
1	0	1	z
2	0	0	z
3	1	0	0
4	1	1	1
5	1	0	0
6	0	0	z

## ■ Verilog: prioritet operatora

- Koristiti zagrade
- U suprotnom ugrađeni prioritet:

Operator	Simboli	Prioritet
Unarni, Množenje, Dijeljenje, Moduo	!, ~, *, /, %	Najviši ↑ ↓ Najniži
Sabiranje, Oduzimanje, Pomjeranje	+, -, <<, >>	
Relacioni, Jednakosti	<, >, <=, >=, ==, !=, ===, !==	
Redukcioni	&, !&, ^, ^~,  , ~	
Logički	&&,	
Uslovni	? :	



## ■ Verilog: skraćivanje evaluacije izraza (*short-circuit*)

- Poštujući pravila prioriteta operatora i njihove asocijativnosti, nekada je moguće izračunati rezultat prije nego se u obzir uzmu svi članovi izraza
- U tom slučaju nema potrebe za evaluacijom čitavog izraza

- Na primjer:

```
assign out = ((a > b) & (c | d));
```

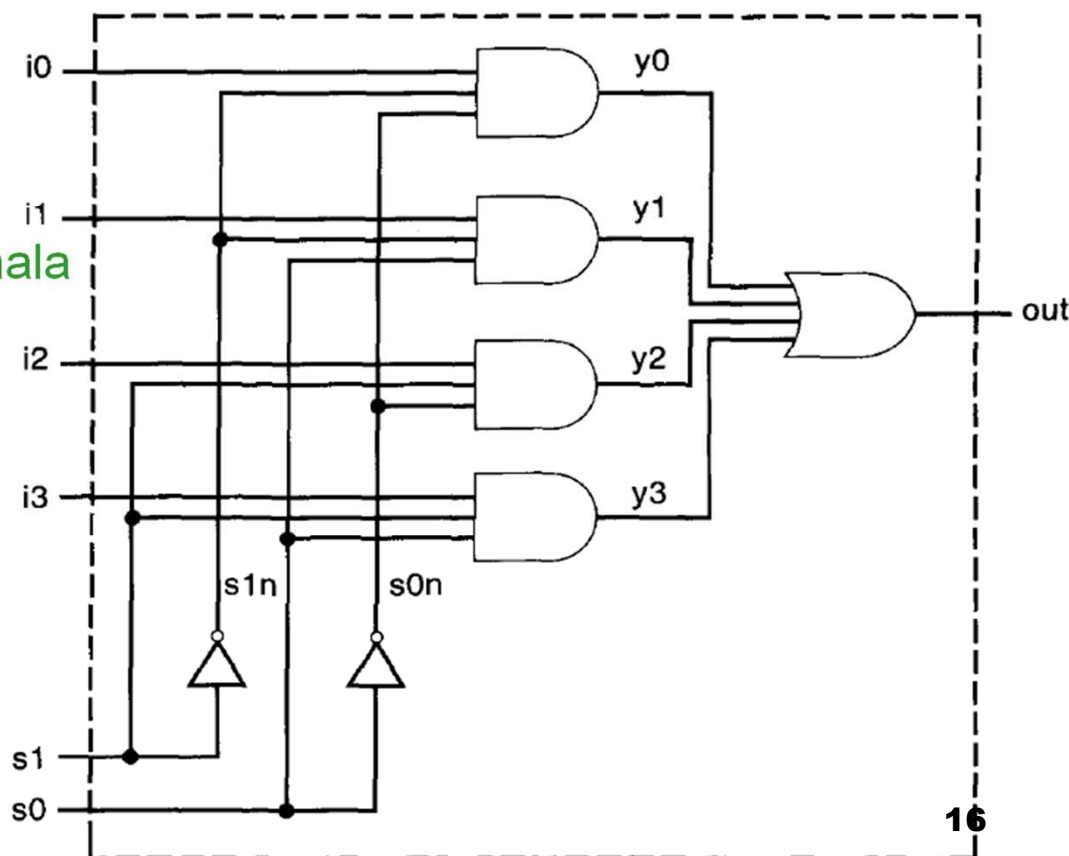
- Ako je rezultat  $(a > b)$  netačan (1'b0), rezultat AND operacije će biti 0 bez obzira na vrijednost preostalog dijela izraza
- Dakle, tada nema potrebe vršiti evaluaciju izraza  $(c | d)$

## ■ Verilog: primjeri *dataflow* modelovanja

### ■ Multiplexer 4/1

- Podsjećanje na nivo logičkih kapija:

```
module mux4_to_1(out, i0, i1, i2, i3, s1, s0);  
output out;  
input i0, i1, i2, i3, s1, s0;  
// unutrašnje linije  
wire s1n, s0n;  
wire y0, y1, y2, y3;  
// kreiranje s1n i s0n signala  
not (s1n, s1);  
not (s0n, s0);  
// 3-ulazna and kola  
and (y0, i0, s1n, s0n);  
and (y1, i1, s1n, s0);  
and (y2, i2, s1, s0n);  
and (y3, i3, s1, s0);  
// 4-ulazno or kolo  
or (out, y0, y1, y2, y3);  
endmodule
```

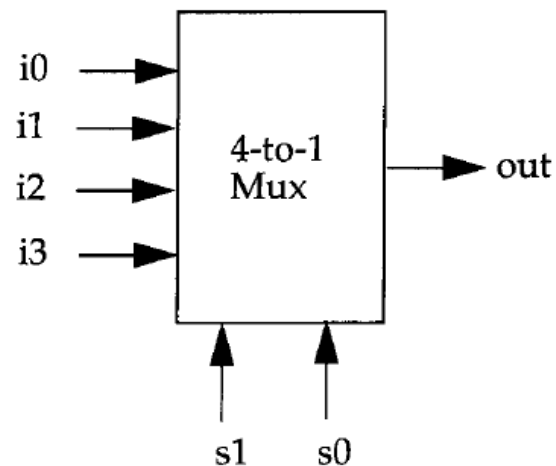




- Verilog: primjeri *dataflow* modelovanja
- Multiplexer 4/1 (prvi način – logička jednačina)

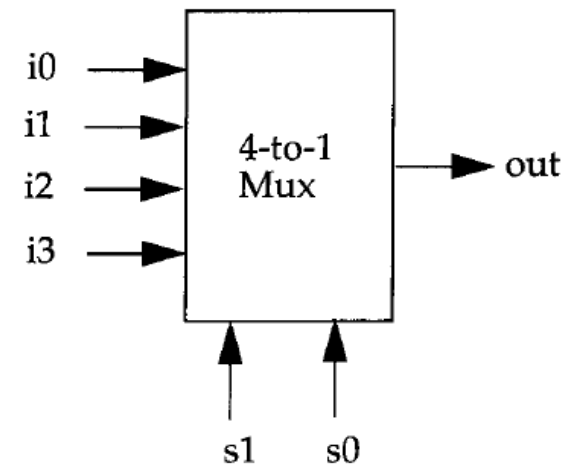
$$out = i_0 \bar{s}_1 \bar{s}_0 + i_1 \bar{s}_1 s_0 + i_2 s_1 \bar{s}_0 + i_3 s_1 s_0$$

```
module mux4_to_1(out, i0, i1, i2, i3, s1, s0);  
    output out;  
    input i0, i1, i2, i3, s1, s0;  
  
    assign out =    (~s1 & ~s0 & i0) |  
                   (~s1 &  s0 & i1) |  
                   (  s1 & ~s0 & i2) |  
                   (  s1 &  s0 & i3);  
  
endmodule
```



- Verilog: primjeri *dataflow* modelovanja
- **Multiplekser 4/1 (drugi način – uslovni operator)**

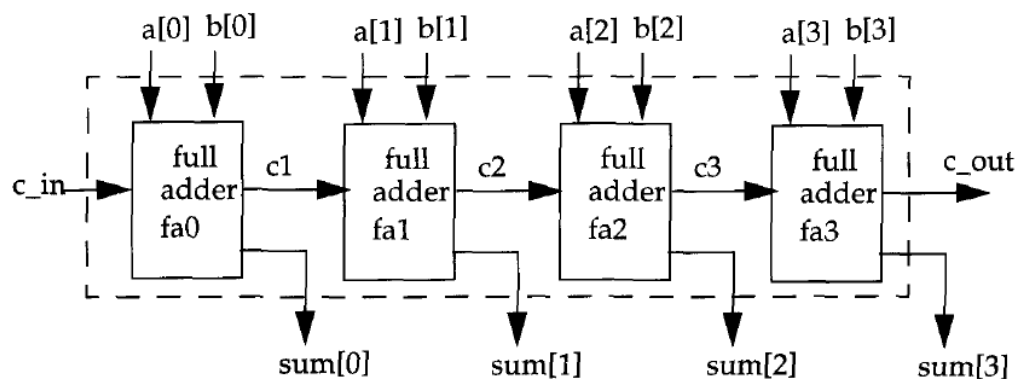
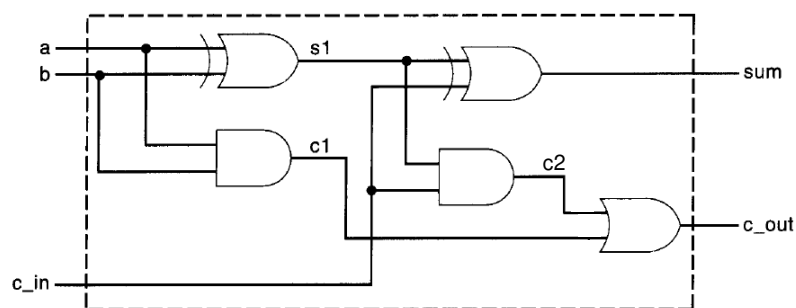
```
module mux4_to_1(out, i0, i1, i2, i3, s1, s0);  
    output out;  
    input i0, i1, i2, i3, s1, s0;  
  
    assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0);  
  
endmodule
```



## ■ Verilog: primjeri *dataflow* modelovanja

### ■ 4-bitni potpuni binarni sabirač

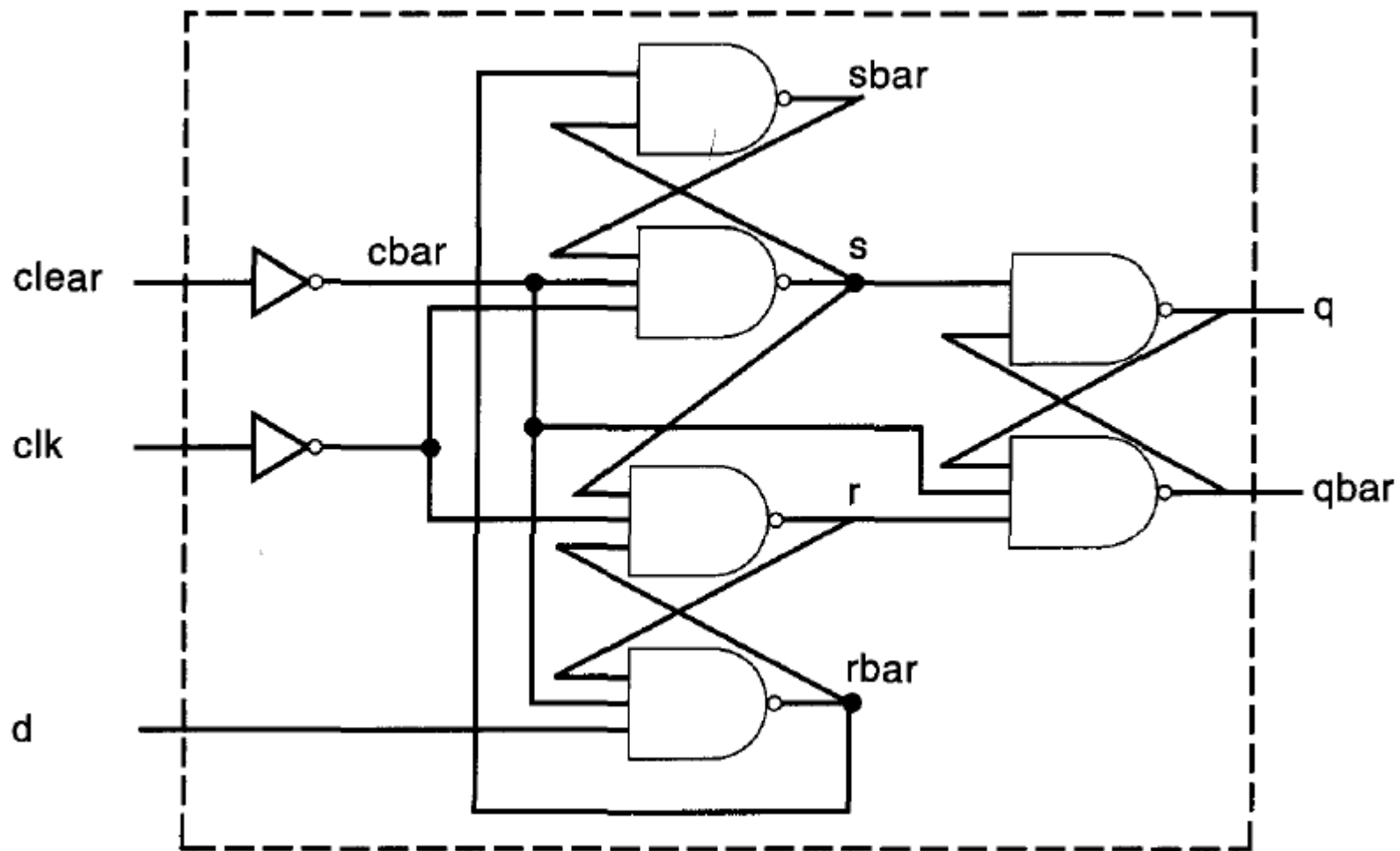
- Podsjećanje – projektovani smo jednobitni FA, pa pomoću njega 4-bitni



- Dataflow:

```
module fulladd4(sum, c_out, a, b, c_in);  
    output [3:0] sum;  
    output c_out;  
    input[3:0] a, b;  
    input c_in;  
  
    assign {c_out, sum} = a + b + c_in;  
endmodule
```

- Verilog: primjeri *dataflow* modelovanja
- D flip flop koji reaguje na silaznu ivicu i ima *clear* signal



## ■ Verilog: primjeri *dataflow* modelovanja

### ■ D flip flop koji reaguje na silaznu ivicu i ima *clear* signal

```
module nedge_dff(q, qbar, d, clk, clear);
```

```
output q, qbar;
```

```
input d, clk, clear;
```

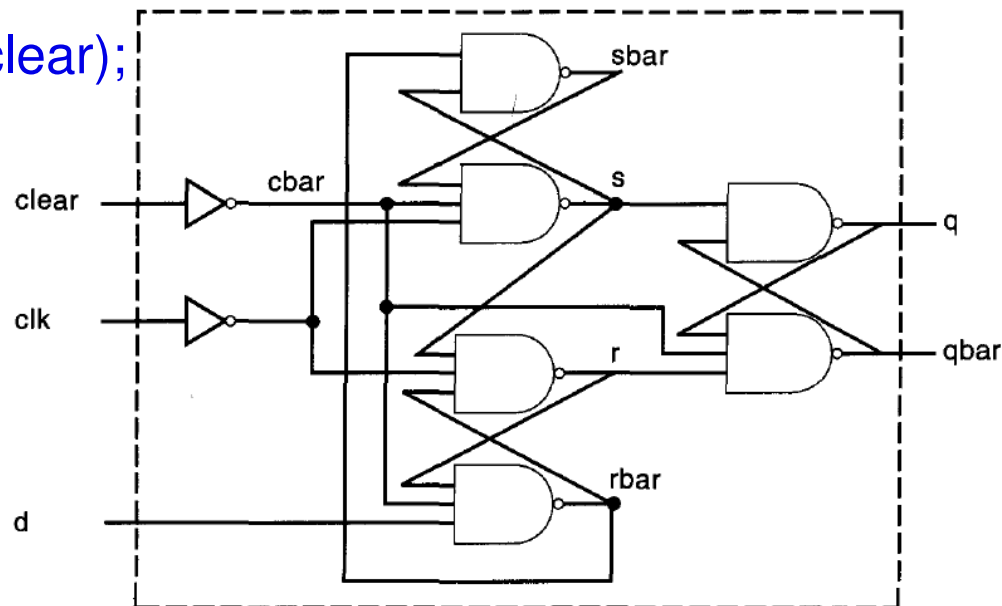
```
wire s, sbar, r, rbar, cbar;
```

```
assign cbar = ~clear;
```

```
assign sbar = ~(rbar & s),  
s = ~(sbar & cbar & ~clk),  
r = ~(rbar & ~clk & s),  
rbar = ~(r & cbar & d);
```

```
assign q = ~(s & qbar),  
qbar = ~(q & r & cbar);
```

```
endmodule
```





# 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 kod kojim se opisuje ponašanje (*behavioral* kod) 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**

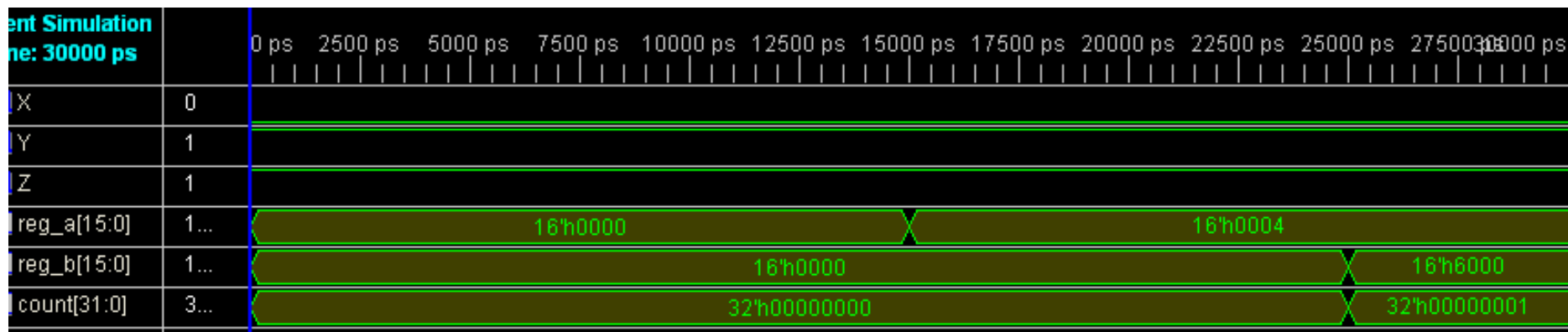
- Dodjeljivanje vrijednosti promjenljivima tipa **reg, integer, real, time**
- Dodijeljena vrijednost ostaje nepromijenjena dok se ne dodijeli nova vrijednost drugim proceduralnim dodjeljivanjem
- Sintaksa: `<lvalue> = <izraz>`
- `<lvalue>` može biti:
  - `reg, integer, real` ili `time` registarska promjenljiva 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 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
```

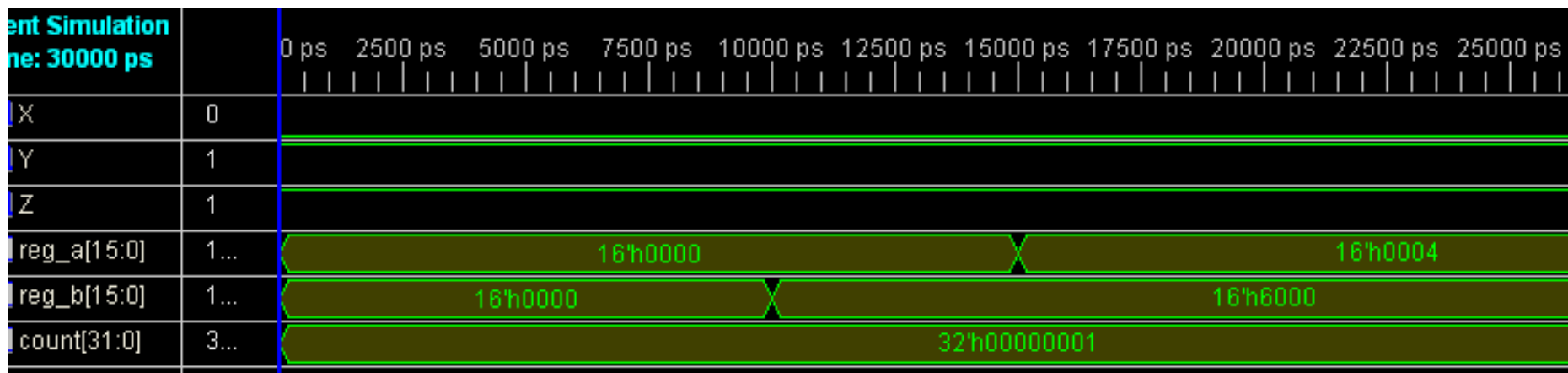




## ■ Verilog: **neblokirajuće** dodjeljivanje

- 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



```

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  $\text{reg\_b}=\text{reg\_a}$  se izvršavaju sekvencijalno u trenutku 0 (početak simulacije)
- Nakon toga se tri neblokirajuća dodjeljivanja obrađuju u istom vremenskom trenutku:
  - *Izvršavanje*  $\text{reg\_a}[2]=1$  se zakazuje za momenat nakon 15 vremenskih jedinica ( $\text{time}=15$ )
  - *Izvršavanje*  $\text{reg\_b}[15:13] = \{X, Y, Z\}$  se zakazuje za momenat nakon 10 vremenskih jedinica ( $\text{time}=10$ )
  - *Izvršavanje*  $\text{count} = \text{count} + 1$  se zakazuje bez kašnjenja ( $\text{time}=0$ )

```
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;
```

- Eliminirane *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:  $A=B=C=E=1$

$A \leq B + C;$

$D \leq A + E;$

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 *dogadjaja*
- 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