# Projektovanje digitalnih sistema

Operatori – nastavak

# Verilog: relacioni operatori

| Operator | Opis |
|---|---|
| a < b | a manje od b |
| a > b | a veće od b |
| a <= b | a manje ili jednako b |
| a >= b | a veće ili jednako b |

- Izraz daje vrijednost logičke 1 ako je tačan, a logičke 0 ako je netačan
- Ako u operandima postoji bar jedan **x** ili **z** bit, izraz daje vrijednost **x**

# Verilog: relacioni operatori – primjer

```
module relacioni_operatori;
      integer A = 4, B = 3;
      reg [3:0] X=4'b1010, Y = 4'b1101, Z = 4'b0xxx;
      initial begin
        $display (" 5     <=  10 = %b", 5 <= 10);
        $display (" 5     >=  10 = %b", 5 >= 10);
        $display (" 1'bx  <=  10 = %b", 1'bx <= 10);
        $display (" 1'bz  <=  10 = %b", 1'bz  <= 10);
        $display (" A <= B (%0d <= %0d) = %b", A
        $display (" A > B (%0d > %0d) = %b", A, B
        $display (" Y >= X (%b >= %b) = %b", Y, X
        $display (" Y < Z (%b < %b) = %b", Y, Z, Y
        #10 $finish;
      end
endmodule
```

Rezultat:

```
5     <=  10 = 1
5     >=  10 = 0
1'bx  <=  10 = x
1'bz  <=  10 = x
A <= B (4 <= 3) = 0
A > B (4 > 3) = 1
Y >= X (1101 > 1010) = 1
Y < Z (1101 < 0xxx) = x
```

# Verilog: operatori jednakosti

| Operator | Opis |
|---|---|
| a === b | a jednako b, uključujući x i z (potpuna jednakost) |
| a !== b | a nije jednako b, uključujući x i z (nije potpuna jednakost) |
| a == b | a jednako b, rezultat može biti nepoznat (logička jednakost) |
| a != b | a nije jednako b, rezultat može biti nepoznat (logička nejednakost) |

- Upoređuju dva operanda bit po bit
- Ako su operandi nejednake dužine, kraći se dopunjava nulama
- Vraćaju logičku 1 ako je izraz tačan, a logičku 0 ako je netačan
- Operatori logičke (ne)jednakosti (**==** i **!=**) daju rezultat **x**, ako u nekom operandu ima bitova **x** i/ili **z**
- Operatori **===** i **!==** (*case equality*) daju kao rezultat logičku 1 ili logičku 0, zavisno da li se na svim mjestima nalaze iste vrijednosti, uključujući **x** i **z**

  => Oni nikada ne daju rezultat **x**

# Verilog: operatori jednakosti – primjer

```
module operatori_jednakosti();
initial begin
// Case Equality
  $display (" 4'bx001 ===  4'bx001 = %b", (4'bx001 === 4'bx001));
  $display (" 4'bx0x1 ===  4'bx001 = %b", (4'bx0x1 === 4'bx001));
  $display (" 4'bz0x1 ===  4'bz0x1 = %b", (4'bz0x1 === 4'bz0x1));
  $display (" 4'bz0x1 ===  4'bz001 = %b", (4
  // Case Inequality
  $display (" 4'bx0x1 !==  4'bx001 = %b", (4'
  $display (" 4'bz0x1 !==  4'bz001 = %b", (4'
  // Logical Equality
  $display (" 5       ==   10      = %b", (5 == 1(
  $display (" 5       ==   5       = %b", (5 == 5)
  $display (" 4'b0001 ==   4'bx001 = %b", (4'
  $display (" 4'bx001 ==   4'bx001 = %b", (4'
  // Logical Inequality
  $display (" 5       !=   5       = %b", (5 != 5));
  $display (" 5       !=   6       = %b", (5 != 6));
  #10 $finish;
end
endmodule
```

Rezultat:

```
4'bx001 ===  4'bx001 = 1
4'bx0x1 ===  4'bx001 = 0
4'bz0x1 ===  4'bz0x1 = 1
4'bz0x1 ===  4'bz001 = 0
4'bx0x1 !==  4'bx001 = 1
4'bz0x1 !==  4'bz001 = 1
5       ==   10      = 0
5       ==   5       = 1
4'b0001 ==   4'bx001 = x
4'bx001 ==   4'bx001 = x
5       !=   5       = 0
5       !=   6       = 1
```

# Verilog: operatori nad bitovima *(bitwise)*

| Operator | Opis |
|---|---|
| ~ | negacija (NOT) |
| & | I (AND) |
| \| | ILI (OR) |
| ^ | ekskluzivno ILI |
| ^~ ili ~^ | ekskluzivno NILI |

- Bit po bit: uzimaju jedan bit operanda i izvršavaju operaciju sa odgovarajućim bitom drugog operanda
- Ako operandi nisu iste dužine, kraći se dopunjava nulama
- **z** se tretira kao **x**
- Negacija je unarna operacija – nema drugog operanda

# Verilog: operatori nad bitovima (*bitwise*) – primjer

```
module bitwise_operatori;
initial begin
  // Bit Wise NOT
  $display (" ~4'b0001          = %b", (~4'b0001));
  $display (" ~4'bx001          = %b", (~4'bx001));
  $display (" ~4'bz001          = %b", (~4'bz001));
  // Bit Wise AND
  $display (" 4'b0001 &  4'b1001 = %b", (
  $display (" 4'b1001 &  4'bx001 = %b", (
  $display (" 4'b1001 &  4'bz001 = %b", (
  // Bit Wise OR
  $display (" 4'b0001 |  4'b1001 = %b", (4
  $display (" 4'b0001 |  4'bx001 = %b", (4
  $display (" 4'b0001 |  4'bz001 = %b", (4
  #10 $finish;
end
endmodule
```

Rezultat:

```
~4'b0001          = 1110
~4'bx001          = x110
~4'bz001          = x110
4'b0001 &  4'b1001 = 0001
4'b1001 &  4'bx001 = x001
4'b1001 &  4'bz001 = x001
4'b0001 |  4'b1001 = 1001
4'b0001 |  4'bx001 = x001
4'b0001 |  4'bz001 = x001
```

# Verilog: operatori nad bitovima (*bitwise*) – primjer

```
module bitwise_operatori;
initial begin
// Bit Wise XOR
  $display (" 4'b0001 ^  4'b1001 = %b", (4'b0001 ^  4'b1001));
  $display (" 4'b0001 ^  4'bx001 = %b", (4'b0001 ^  4'bx001));
  $display (" 4'b0001 ^  4'bz001 = %b", (4'b0001 ^  4'bz001));
  // Bit Wise XNOR
  $display (" 4'b0001 ~^ 4'b1001 = %b", (4'b0001 ~^ 4'b1001));
  $display (" 4'b0001 ~^ 4'bx001 = %b", (4'b0001 ~^ 4'bx001));
  $display (" 4'b0001 ~^ 4'bz001 = %b", (4'b0001 ~^ 4'bz001));
  #10 $finish;
end
endmodule
```

Rezultat:

```
4'b0001 ^  4'b1001 = 1000
4'b0001 ^  4'bx001 = x000
4'b0001 ^  4'bz001 = x000
4'b0001 ~^ 4'b1001 = 0111
4'b0001 ~^ 4'bx001 = x111
4'b0001 ~^ 4'bz001 = x111
```

# Verilog: operatori nad bitovima *(bitwise)*

- Razlikovati od logičkih operatora:

```
// X = 4'b1010, Y = 4'b0000
X | Y  // bitwise operacija: rezultat je 4'b1010
X || Y // logička operacija: ekvivalentno sa 1 || 0, rezultat je 1 (tačno)
```

# 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 NOR 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
```

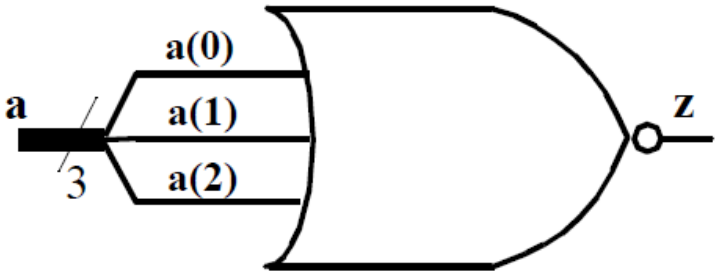
# Verilog: operatori redukcije – primjer upotrebe

- 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
- Upražnjena 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 konkatenacije (nastavljanja)

| Operator | Opis |
|---|---|
| { } | konkatenacija |

- Obezbjeđuje mehanizam za “spajanje” više operanada
- Operandi **moraju** imati naznačenu veličinu
- Konkatenacija 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 nastavljanje 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 konkatenacije i replikacije – primjer

```
module operatori_konk_repl;

initial begin
  // konkatenacija
  $display ("{4'b1001,4'b10x1} = %b", {4'b1001,4'b10x1});
  // replikacija
  $display ("{4{4'b1001}} = %b", {4{4'b1001}});
  // replikacija i konkatenacija
  $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 |
| Sabiranje, Oduzimanje, Pomjeranje | +, - , <<, >> | |
| Relacioni, Jednakosti | <,>,<=,>=,==,!=,===,!== | |
| Redukcioni | &, !&,^,^~,\|,~\| | |
| Logički | &&, \|\| | |
| Uslovni | ? : | Najniži |

# 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:

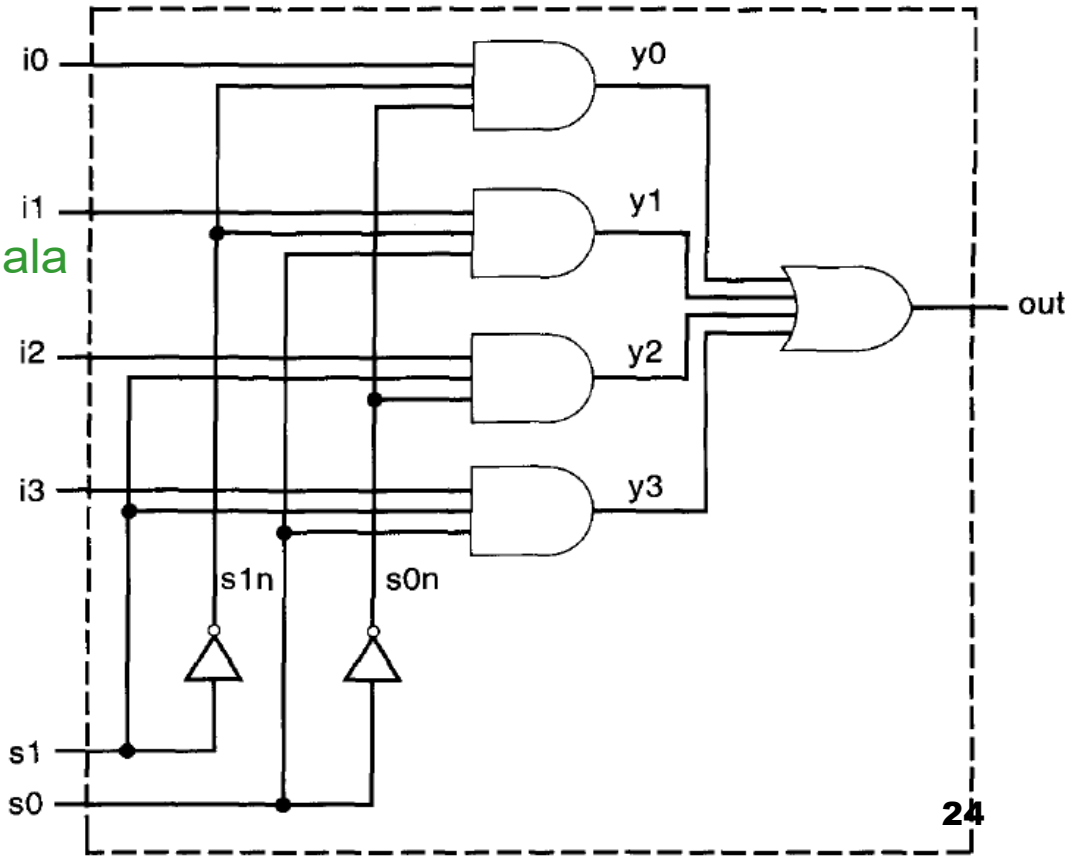
```
assign out = ((a > b) & (c | d));
```

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

# Verilog: primjeri *dataflow* modelovanja

- **Multiplekser 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
```

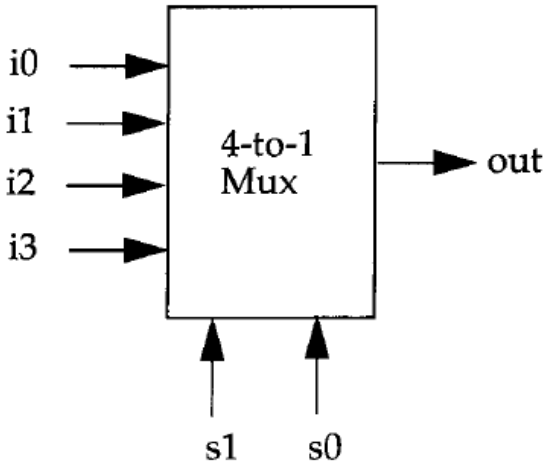


# Verilog: primjeri *dataflow* modelovanja

- **Multiplekser 4/1 (prvi način – logička jednačina)**

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

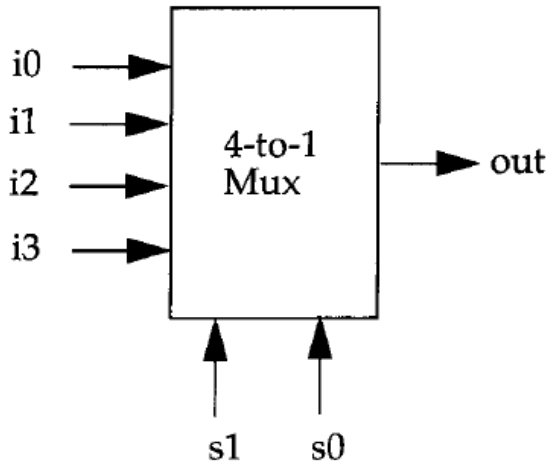
```
module mux4_to_1(out, i0, i1, i2, i3, s1, s0);
      output out;
      input i0, i1, i2, i3, s1, s0;

      assign out =      (~s1 & ~s0 & i0) |
                        (~s1 &   s0 & i1) |
                        (  s1 & ~s0 & i2) |
                        (  s1 &   s0 & i3);
endmodule
```



# Verilog: primjeri *dataflow* modelovanja

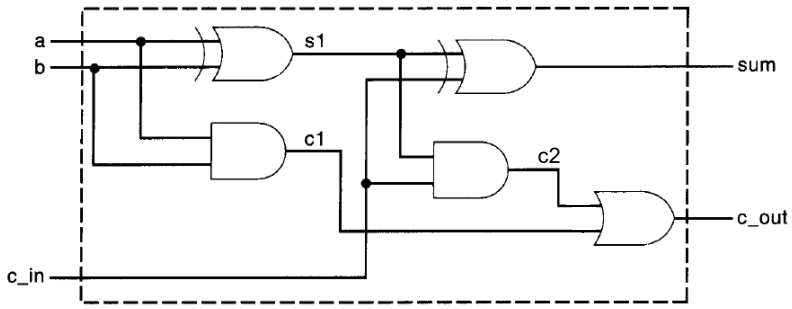
- **Multiplekser 4/1 (drugi način – uslovni operator)**

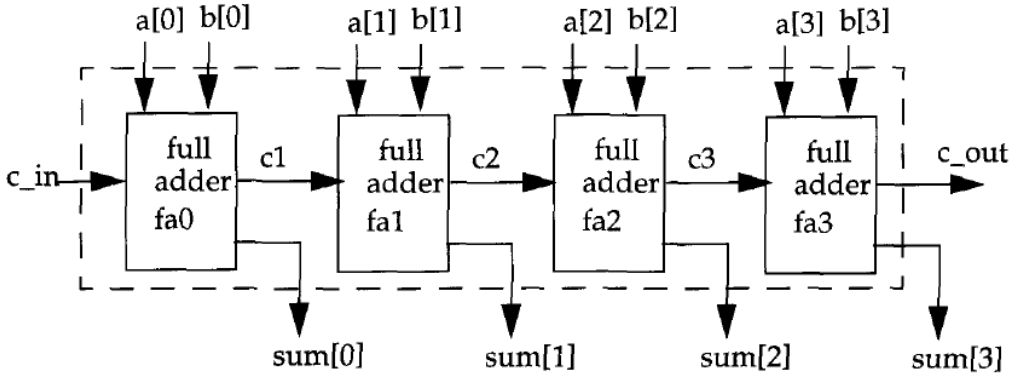
```
module mux4_to_1(out, i0, i1, i2, i3, s1, s0);
    output out;
    input i0, i1, i2, i3, s1, s0;

    assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0);

endmodule
```



# Verilog: primjeri *dataflow* modelovanja

- **4-bitni potpuni binarni sabirač**
  - Podsjećanje – projektovali smo jednobitni FA, pa pomoću njega 4-bitni



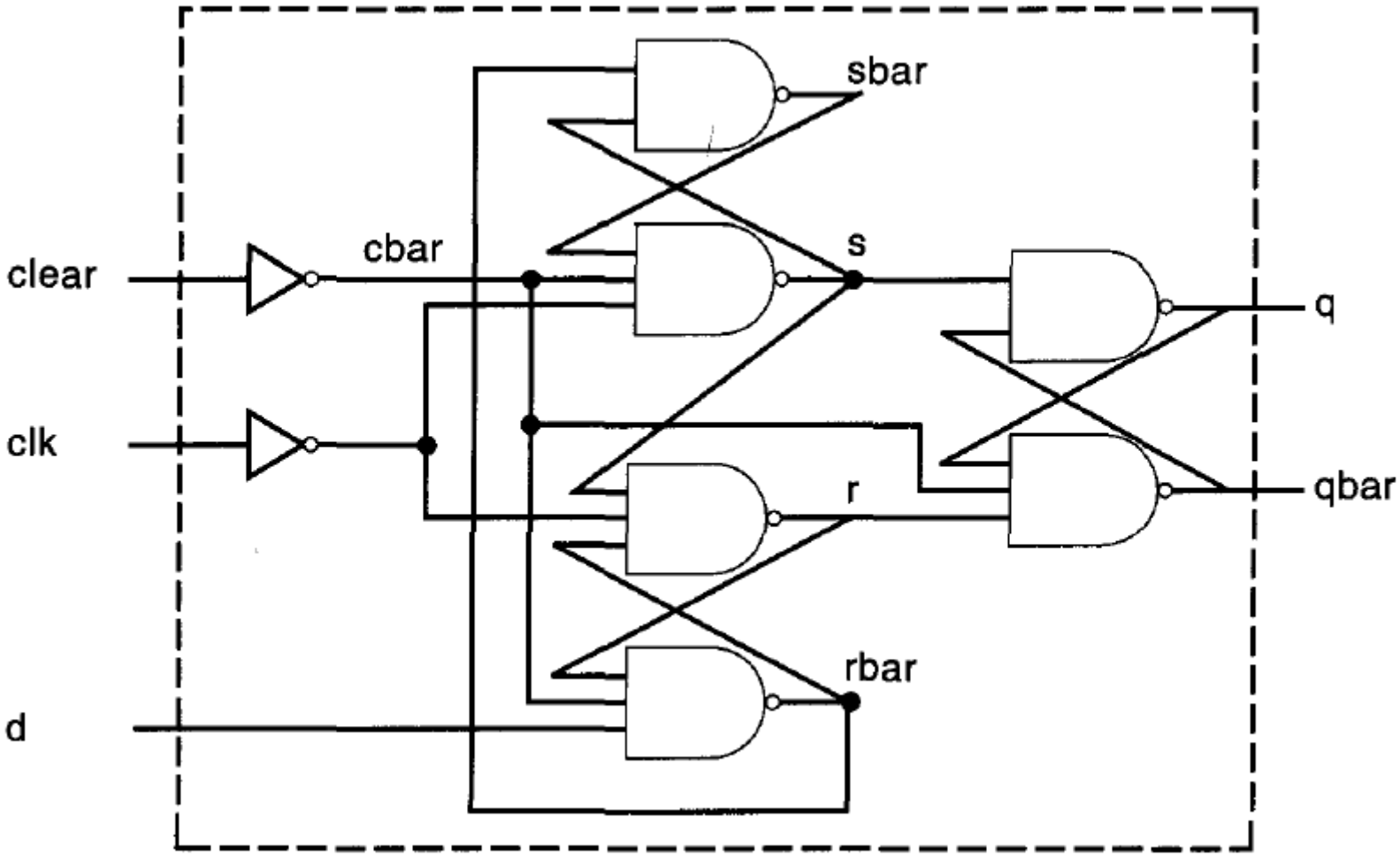




  - Dataflow:

```
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

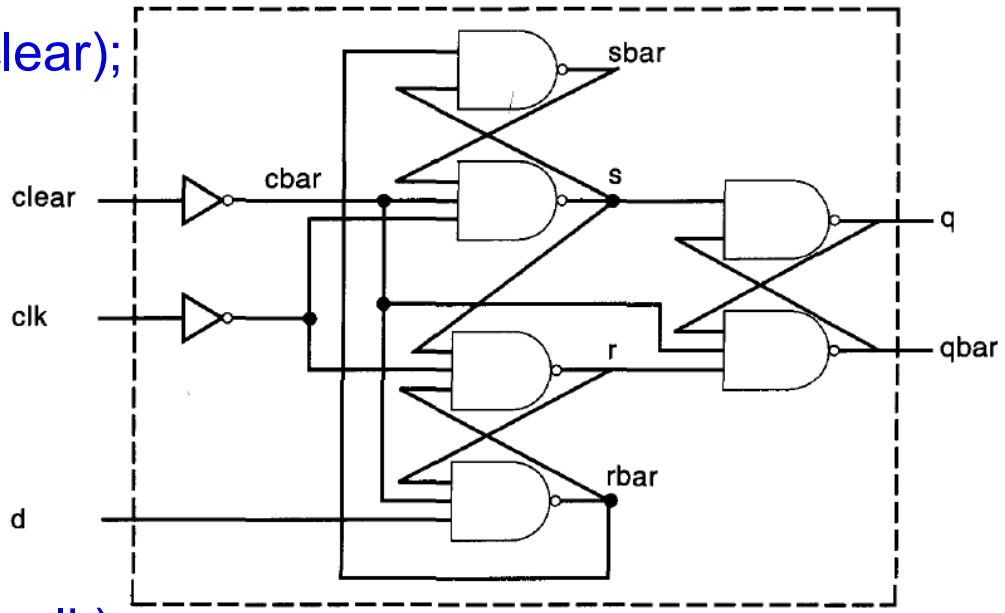
- **D flip flop koji reaguje na silaznu ivicu i ima *clear* signal**



# 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)