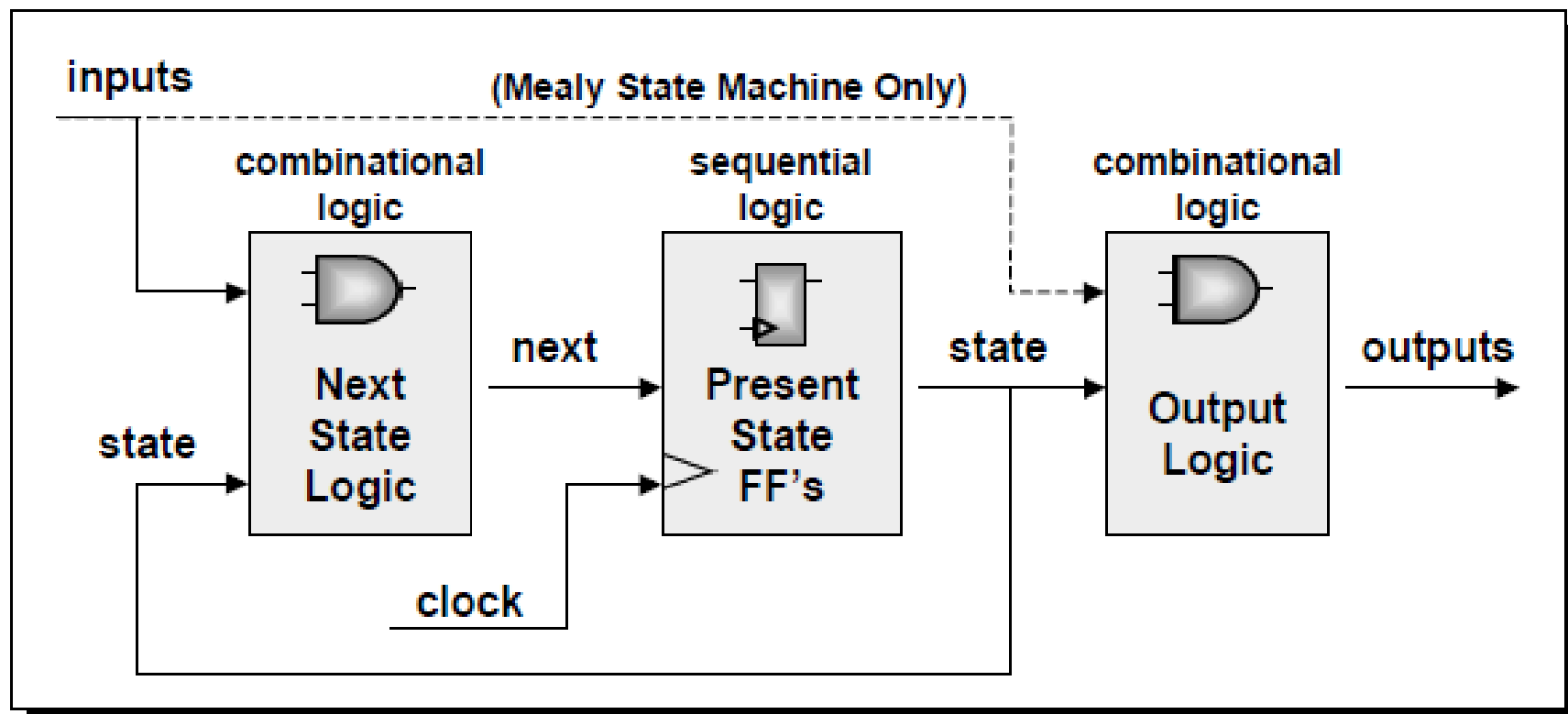


■ Automat (*Finite State Machine* – FSM)

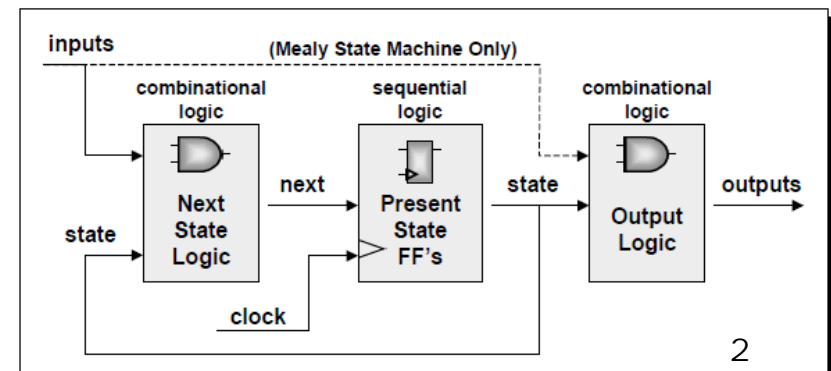
- *Next-state* kombinaciona logika
- Logika za praćenje trenutnog stanja (taktovana)
- Izlazna kombinaciona logika



■ Realizacija automata

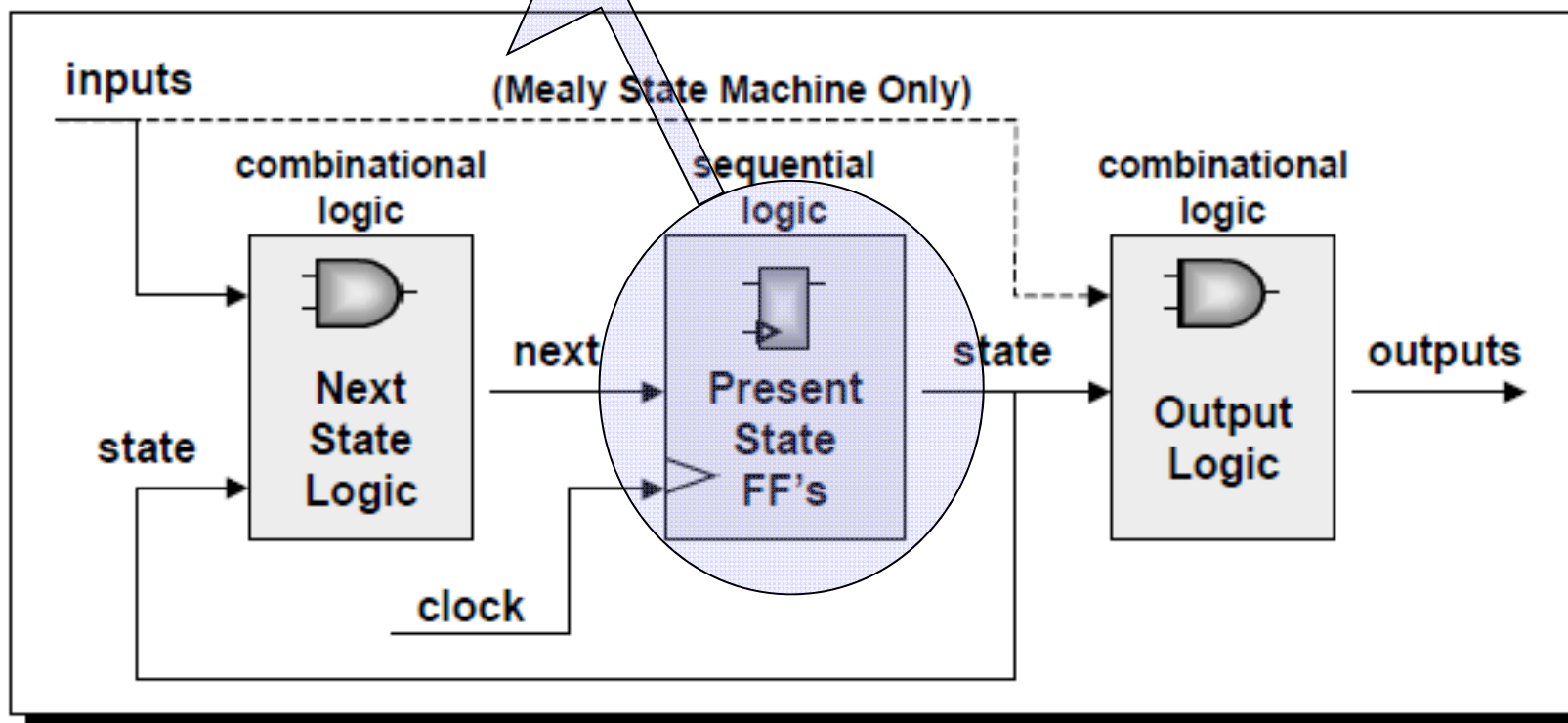
- 1. Kodirati sekvencijalni (taktovani) **always** blok koji reprezentuje trenutno stanje pomoću **vektorske registarske promjenljive**
- 2. Kodirati kombinatorni **always** blok koji reprezentuje *next-state* logiku
- 3. Kodirati kombinacionu mrežu za generisanje odgovarajućeg izlaza (output logika):
 - koristeći *continuous assignment* iskaze
 - ili
 - uključujući definisanje izlaza u okviru kombinacionog bloka za definisanje *next-state* logike (samo kod Mealy-jevog automata)

Napomena: za automat koji želimo da sintetizujemo moramo obezbijediti **reset** i sinhronizovati promjene stanja sa jednom **ivicom** taktnog signala



■ Realizacija automata – primjer semafora

always @(posedge clock)
stanje = sljedece_stanje;



■ Realizacija automata – primjer semafora

```
always @(stanje or clear or X)
begin
```

```
  if (clear)
```

```
    sljedece_stanje = `S0;
```

```
  else
```

```
    case (stanje)
```

```
      `S0: if(X) sljedece_stanje = `S1;
```

```
           else sljedece_stanje = `S0;
```

```
      `S1: begin // sačekaj nekoliko pozitivnih ivica takta
```

```
                repeat(`ZUTO_CRVENO) @(posedge clock);
```

```
                sljedece_stanje = `S2;
```

```
            end
```

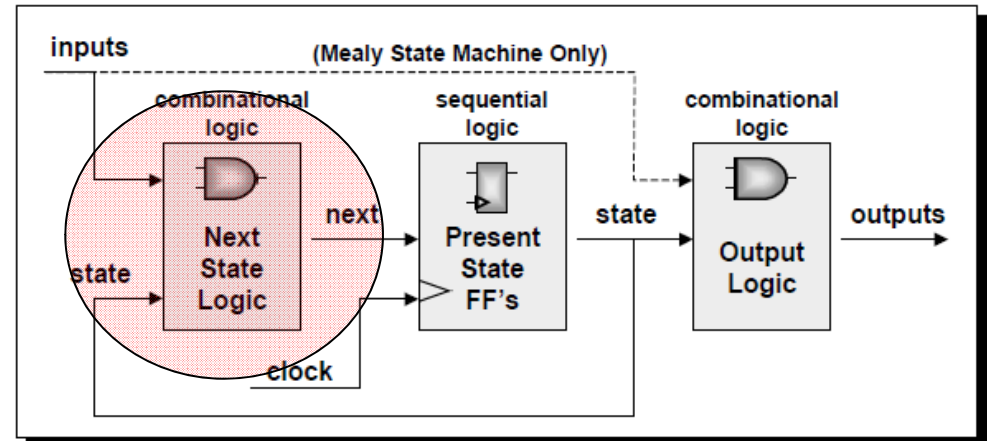
```
      `S2: begin // sačekaj nekoliko pozitivnih ivica takta
```

```
                repeat(`CRVENO_ZELENO) @(posedge clock);
```

```
                sljedece_stanje = `S3;
```

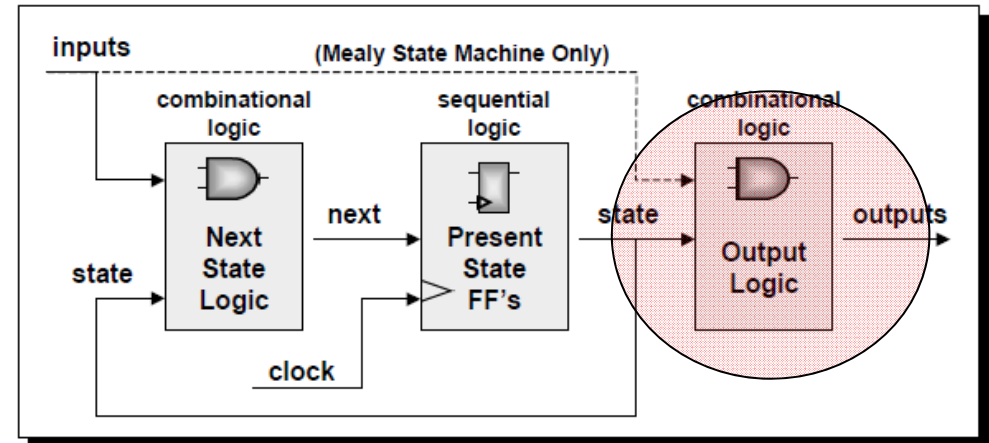
```
            end
```

```
      `S3: ...
```



■ Realizacija automata – primjer semafora

```
always @(stanje)
begin
  case(stanje)
    `S0: begin
          glavni = `ZELENO;
          sporedni = `CRVENO;
        end
    `S1: begin
          glavni = `ZUTO;
          sporedni = `CRVENO;
        end
    `S2: begin
          glavni = `CRVENO;
          sporedni = `CRVENO;
        end
    `S3: begin
          glavni = `CRVENO;
          sporedni = `ZELENO;
        end
  end ...
end
```





■ Task i funkcija

- Često se ista funkcionalnost (dio koda) mora implementirati na više mjesta unutar dizajna
- Umjesto da se ponavlja isti kod, potrebno ga je oblikovati u rutine koje će se umetati na odgovarajuća mjesta
- U Verilogu se, u tu svrhu, koriste *task* i *funkcija*
- *Task* ima *input*, *output*, i *inout* argumente; *funkcija* ima *input* argumente
- *Task* i *funkcija* su uključeni u hijerarhiju dizajna – mogu se adresirati pomoću hijerarhijskih imena, kao i imenovani blokovi
- I *task* i *funkcija* se moraju definisati unutar modula i lokalne su unutar tog modula
- *Task* i *funkcija* imaju različitu namjenu



■ Task i funkcija – razlike

- *Funkcija* može pozvati drugu funkciju, ali ne i neki *task*
- *Task* može pozvati neku funkciju ili drugi *task*
- ❖ *Funkcija* se izvršava trenutno (vrijeme trajanja je 0)
- ❖ *Task* se može izvršavati neko (ne-nulto) vrijeme
- *Funkcija* ne može sadržavati kašnjenja, događaje, ili iskaze za kontrolu tajminga
- *Task* može sadržavati kašnjenja, događaje, ili iskaze za kontrolu tajminga
- *Funkcija* mora imati bar jedan argument tipa input; može ih biti i više
- *Task* može da ima nula ili više argumenata tipa input, output ili inout
- *Funkcija* uvijek vraća jednu vrijednost
- *Task* ne vraća nikakvu vrijednost, ali može *proslijediti* više vrijednosti preko output ili inout argumenata



■ Task i funkcija – nastavak

- *Task* se koristi za kod koji sadrži kašnjenje, tajming, događaje ili više output argumenata
- *Funkcija* se koristi za kod koji predstavlja kombinaciono kolo, izvršava se trenutno (trajanje je 0) i ima samo jedan izlaz
- *Funkcije* se tipično koriste za različite vrste proračuna
- *Task* i *funkcija* ne mogu imati *wire* promjenljive
- Sadrže samo *behavioral* izraze
- Ne sadrže *always* ili *initial* iskaze, ali mogu biti pozvani iz *always* bloka, *initial* bloka ili drugih *task*-ova i funkcija



■ Task

- *Task* se deklarira pomoću ključnih riječi **task** i **endtask**
- Moraju se koristiti ako je za rutinu ispunjen bilo koji od sljedećih uslova:
 - Postoje kašnjenja, kontrola tajminga, kontrola događaja
 - Nema *output* argumenata ili ih ima više od jednog
 - Nema *input* argumenata
- Za deklarisanje argumenata u *task*-u koriste se iste ključne riječi kao i za portove u modulu: *input*, *output* i *inout*, ali postoji razlika:
 - Portovi služe da se povežu spoljašnji signali sa modulom
 - Argumenti u *task*-u služe da se proslijede vrijednosti u/iz *task*-a

■ Task – primjer

```
module operacije; // modul operacije sadrži task bitwise_operacije
parameter delay = 10;
reg [15:0] A, B, AB_AND, AB_OR, AB_XOR;
always @(A or B) // kad A ili B promijeni vrijednost
begin // Pozvati task bitwise_operacije koji ima 2 input argumenta A i
// B i 3 output argumenta: AB_AND, AB_OR, AB_XOR. Argumenti se
// moraju specificirati u istom redoslijedu u kojem se pojavljuju
// prilikom deklaracije task-a
    bitwise_operacije(AB_AND, AB_OR, AB_XOR, A, B);
end

...
task bitwise_operacije; // definicija task-a bitwise_operacije
output [15:0] ab_and, ab_or, ab_xor; // izlazi
input [15:0] a, b; // ulazi
begin
    #delay ab_and = a & b; ab_or = a | b; ab_xor = a ^ b;
end
endtask
endmodule
```



■ Task – primjer 2 (asimetrični generator takta)

```
module takt;  
...  
reg clock;  
initial  
    inicijalizacija_takta; // poziv taska inicijalizacija_takta  
always  
begin  
    asimetricni_takt; // poziv taska asimetricni_takt  
end  
task inicijalizacija_takta; // task inicijalizacija_takta  
begin clock = 1'b0; end  
endtask  
task asimetricni_takt; // task asimetricni_takt  
begin #12 clock=1'b0; #5 clock=1'b1; #3 clock=1'b0; #10 clock=1'b1;  
end // radi direktno na promjenljivoj clock definisanoj u modulu  
endtask  
...  
endmodule
```



■ Funkcija

- *Funkcija* se deklarira pomoću ključnih riječi **function** i **endfunction**
- Koriste se ako su za rutinu ispunjeni svi sljedeći uslovi:
 - Nema kašnjenja, kontrole tajminga, kontrole događaja
 - Ima tačno jedan *output* argument
 - Ima bar jedan *input* argument
- Kod deklaracije funkcije *implicitno* se deklarira registarska promjenljiva koja ima isto ime kao funkcija
- Rezultat funkcije se prosleđuje preko te implicitne promjenljive
- Funkcija se poziva navođenjem njenog imena i ulaznih argumenata
- Na kraju izvršavanja funkcije, povratna vrijednost se smješta tamo gdje je funkcija pozvana
- Ako se ne navede opseg rezultata, podrazumijeva se da je jednobitan



■ Funkcija– primjer (kalkulator parnosti za 32 bita)

```
module parnost;
```

```
...
```

```
reg [31:0] addr;
```

```
reg parity;
```

```
always @(addr) // računaj parnost kad god se addr promijeni
```

```
begin
```

```
    parity = izracunaj_parnost(addr); // prvi poziv funkcije izracunaj_parnost
```

```
    $display("Izracunata parnost = %b", izracunaj_parnost(addr) );
```

```
end // drugi poziv
```

```
...
```

```
function izracunaj_parnost; // definicija funkcije za računanje parnosti
```

```
input [31:0] address;
```

```
begin // postavi odgovarajuću izlaznu vrijednost koristeći implicitnu
```

```
    // internu promjenljivu izracunaj_parnost
```

```
    izracunaj_parnost = ^address; //xor svih bitova promjenljive address
```

```
end
```

```
endfunction
```

```
...
```

```
endmodule
```

■ Funkcija– primjer 2 argumenta (pomjerački reg.)

```
module shifter;
`define LEFT_SHIFT 1'b0
`define RIGHT_SHIFT 1'b1
reg [31:0] addr, left_addr, right_addr;
reg control;
always @(addr)
begin // poziv funkcije koja obavlja pomjeranje
    left_addr = shift (addr, 'LEFT_SHIFT};
    right_addr = shift (addr, 'RIGHT_SHIFT};
end

...
function [31:0] shift; //definicija funkcije za pomjeranje, izlaz je 32-bitan
input [31:0] address;
input control;
begin
    shift = (control == `LEFT_SHIFT) ? (address << 1) : (address >>1);
end
endfunction
endmodule
```



Projektovanje digitalnih sistema

Neke korisne tehnike modelovanja



■ Neprekidno proceduralno dodjeljivanje

- Proceduralnim dodjeljivanjem se dodjeljuje vrijednost registarskoj promjenljivoj (između ostalih)
- Vrijednost tamo ostaje sve dok se drugim proceduralnim dodjeljivanjem ne stavi nova vrijednost u promjenljivu
- Neprekidno proceduralno dodjeljivanje (***Procedural Continuous Assignment***) je proceduralno dodjeljivanje kojim se omogućava da se vrijednost nekog izraza neprekidno (ali u ograničenom vremenskom trajanju) nalazi u datoj promjenljivoj (registarskoj ili *net*)
- Neprekidno proceduralno dodjeljivanje **nadjačava** efekat postojećih proceduralnih dodjeljivanja istoj promjenljivoj
- Neprekidno proceduralno dodjeljivanje je zapravo ekstenzija postojećim proceduralnim dodjeljivanjima
- Uobičajeno se koriste u ograničenim vremenskim intervalima



■ **assign i deassign**

- Koristi se za registarske promjenljive
- Sa lijeve strane jednakosti može biti samo registarska promjenljiva ili konkatencija registarskih promjenljivih (**ne** i niz ili dio registra)
- Sa lijeve strane jednakosti ne može biti **net**
- Primjer: D flip flop koji reaguje na silaznu ivicu taktnog signala, sa asinhronim resetom

```
module edge_dff(q, qbar, d, clk, reset);  
    output q,qbar;  
    input d, clk, reset;  
    reg q, qbar;  
    always @(negedge clk) // na negativnoj ivici takta dodijeli vrijednosti  
    begin  
        q = d;  
        qbar = ~d;  
    end  
    ...
```



■ **assign i deassign** – primjer

... // nastavak

```
always @(reset)
```

```
    if(reset)
```

```
        begin // ako je reset visok zamijeni regularna dodjeljivanja
```

```
                // promjenljivima q i qbar sa novim vrijednostima: 0 i 1
```

```
                assign q = 1'b0;
```

```
                assign qbar = 1'b1;
```

```
        end
```

```
    else
```

```
        begin // ako reset ode na nulu, ukloniti neprekidno dodjeljivanje
```

```
                // nakon ovoga, regularnim dodjeljivanjem q=d i qbar=~d će se
```

```
                // moći promijeniti sadržaj ovih promjenljivih na sljedećoj negativnoj
```

```
                // ivici taka
```

```
                deassign q;
```

```
                deassign qbar;
```

```
        end
```

```
endmodule
```



■ **force i release**

- Koristi se i za registarske i za *net* promjenljive
- Sa lijeve strane jednakosti može biti *net* ili dio (*bit-select* ili *part-select*) vektorskog *net*-a
- Sa lijeve strane jednakosti ne može biti dio (*bit-select* ili *part-select*) registarske promjenljive niti niz registara
- Tipično se koriste u procesu traženja grešaka u programu (debugovanje)
- Određene promjenljive se forsirano postavljaju na željene vrijednosti i posmatra se uticaj tih vrijednosti na djelove dizajna
- Preporučuje se da se **force** i **release** ne koriste u blokovima dizajna već samo u stimulusima ili kao iskazi u svrhu debugovanja
- **force** na **registarskoj promjenljivoj** nadjačava sva proceduralna i neprekidna proceduralna dodjeljivanja toj promjenljivoj, sve dok se ne izvrši **release**
- Nakon toga će promjenljiva nastaviti da čuva forsiranu vrijednost , ali će se ta vrijednost promijeniti sljedećim proceduralnim dodjeljivanjem

■ **force i release** – nastavak

- Primjer: D flip flop koji reaguje na silaznu ivicu taktnog signala, sa asinhronim resetom

```
module stimulus;  
    // instanciranje flip flopa  
    edge_dff dff(Q, Qbar, D, CLK, RESET);  
initial  
begin  
    // ovim iskazima se forsira vrijednost 1 na dff.q u vremenu između  
    // 50 i 100, bez obzira kakav je zaista izlaz flip flopa  
    #50 force dff.q = 1'b1; // forsira q=1 u trenutku 50  
    #50 release dff.q; // oslobađa promjenljivu q u trenutku 100  
end  
endmodule
```



■ **force i release** – nastavak

- **force** na *net promjenljivoj* nadjačava sva neprekidna dodjeljivanja (*continuous assignments*) toj promjenljivoj, sve dok se ne izvrši **release**
- *Net* promjenljiva može biti forsirana da uzme konstantnu vrijednost ili vrijednost izraza
- Kada se oslobodi, *net* promjenljiva će **odmah** da preuzme vrijednost koja se na nju dovodi regularnim dodjeljivanjem
- Primjer:

```
module top;
```

```
...
```

```
assign out = a & b & c; // continuous assignment
```


```
initial
```

```
    #50 force out = a | b & c; // između 50 i 100 ovo je vrijednost out-a
```

```
    #50 release out;
```

```
end
```

```
endmodule
```



■ Razlika između **assign-deassign** i **force-release**

- **force-release** se može primjeniti na *net* promjenljive, dok se **assign-deassign** primjenjuje samo na registarske promjenljive
- **force-release** konstrukcija je namijenjena za verifikaciju dizajna i nije je moguće sintetizovati
- Proceduralna **assign-deassign** konstrukcija je namijenjena za modelovanje hardvera (*behavioral* dizajn), ali je većina alata za sintetizovanje hardvera neće prevesti
- Xilinx ISE je podržava, ali uz mnoga ograničenja:
 - za svaki signal je dozvoljena samo jedna upotreba *assign-deassign*
 - mora biti u istom always bloku razdvojena sa *if-else*
 - ne može se primjenjivati na djelove vektora ili njihove bitove
- Raniji primjer D flip flopa je moguće sintetizovati

■ **assign i deassign** – primjer D ff sa set i reset, **koji se ne može sintetizovati**

```
module dfflop(RST, SET, STATE, CLOCK, DATA_IN); //višestruki
input RST, SET, CLOCK, DATA_IN; // assign-deassign na istoj promjenljivoj
output STATE;
reg STATE;
always @ (RST) // reset blok
    if(RST) assign STATE = 1'b0;
    else deassign STATE;

always @ (SET) // set blok
    if(SET) assign STATE = 1'b1;
    else deassign STATE;

always @ (posedge CLOCK)
    begin // takt blok
        STATE <= DATA_IN;
    end
endmodule
```

drugi način

```
...//nije kroz if-else
always @ (RST or SET)
    case ({RST,SET})
        2'b10: assign STATE = 1'b0;
        2'b11: assign STATE = 1'b0;
        2'b01: assign STATE = 1'b1;
        2'b00: deassign STATE;
    endcase
always @ (posedge CLOCK)
begin
    STATE <= DATA_IN;
end
endmodule
```



■ Parametri

- Parametre je moguće definisati prilikom definicije modula
- Prilikom kompajliranja Verilog modula moguće je promijeniti vrijednosti parametara za svaku instancu modula ponaosob
- Na taj način je sa istim kodom moguće realizovati različite sklopove – npr. za različite veličine nekog digitalnog sklopa se koristi isti kod
- Promjena *default* vrijednosti parametara se može obaviti na dva načina:
 - Pomoću **defparam** iskaza
 - Kroz dodjeljivanje vrijednosti parametru prilikom instanciranja modula

■ **defparam** iskaz

- Koristi se hijerarhijsko ime modula čiji se parametar mijenja

```
module hello_world; // definisanje modula hello_world
parameter id_num = 0; //definisanje identifikacionog broja (0)
initial           // prikazuje identifikacioni broj modula
    $display ( "hello_world identifikacioni broj je %d, id_num);
endmodule
```

```
module top;
// promjena vrijednosti parametra u instancama modula
defparam w1.id_num = 1, w2.id_num = 2;
// instanciranje dva hello_world modula
hello_world w1();
hello_world w2();
endmodule
```

Izlaz:

```
hello_world identifikacioni broj je 1
hello_world identifikacioni broj je 2
```



■ Dodjeljivanje vrijednosti parametru prilikom instanciranja modula

- Modifikacija prethodnog primjera:

```
module top;  
// instanciranje dva hello_world modula  
hello_world #(1) w1(); // prosljeđuje se vrijednost 1 parametru id_num  
hello_world #(2) w2(); // prosljeđuje se vrijednost 2 parametru id_num  
endmodule
```

Izlaz:

```
hello_world identifikacioni broj je 1  
hello_world identifikacioni broj je 2
```

- Ako ima više parametara, navode se u istom redoslijedu kao prilikom deklaracije modula
- Ako vrijednost parametra nije specificirana, koristiće *default* vrijednost, zadanu prilikom deklaracije modula



■ Dodjeljivanje vrijednosti parametru prilikom instanciranja modula – nastavak

■ Primjer višestrukog zadavanja parametara

```
module razna_kasnjenja;  
parameter delay1 = 2;  
parameter delay2 = 3;  
parameter delay3 = 7;  
... <unutrašnjost modula> ...  
endmodule
```

```
module top;  
// instanciranje modula sa novim vrijednostima za kasnjenja  
razna_kasnjenja #(4,5,6) b1(); //b1: delay1=4, delay2=5, delay3 =6  
razna_kasnjenja #(9,4) b2(); //b2: delay1=9, delay2=4, delay3=7(default)  
endmodule
```

■ Parametri – nastavak

- Primjer parametrizovanog dekodera $N:2^N$ (veličinu biraemo prilikom instanciranja)

```
module dekode(a, y);  
    parameter N = 3;  
    input [N-1:0] a;  
    output [2**N-1:0] y; // 2**N je 2 na N – Verilog 2001 (ne postoji ranije)  
    reg [2**N-1:0] y;  
  
    always @ (a)  
    begin // veliki dekode je teško specificirati sa case direktivom  
        y = 0; // blokirajuće dodjeljivanje! Prvo mora uzeti vrijednost 0  
        y[a] = 1;  
    end  
endmodule
```

Može i ovako:

```
module dekode # (parameter N = 3) (a, y);  
    input [N-1:0] a;  
    output [2**N-1:0] y;  
    reg [2**N-1:0] y;
```



■ Uslovno kompajliranje

- Dio Verilog koda može biti pogodan za jedno okruženje, a da nije za drugo
- Dizajneru nije zgodno da pravi dvije verzije Verilog dizajna, za svako okruženje posebno
- Umjesto toga, dizajner može da specificira da li će se neki dio koda kompajlirati, u zavisnosti od nekog uslova (*flega*) – *uslovno kompajliranje*
- Uslovno kompajliranje se obavlja pomoću ključnih riječi: **`ifdef**, **`else**, i **`endif**
- **`ifdef** se može pojaviti bilo gdje unutar dizajna i odnositi se na iskaze, module, blokove, deklaracije i ostale direktive
- **`else** je opciono i može biti najviše jedno
- **`ifdef** se uvijek završava sa odgovarajućim **`endif**
- Postavljanje *flega* se vrši direktivom **`define**

■ Uslovno kompajliranje – nastavak

■ *Primjer:*

```
`ifdef TEST // kompajliraj modul test samo ako je TEST definisano
    module test;
    . . .
endmodule
`else // po default-u kompajliraj modul stimulus
    module stimulus;
    . . .
endmodule
`endif // kraj `ifdef iskaza
```

■ *Primjer 2:*

```
module top;
    bus_master b1(); // bezuslovno instanciranje modula
    `ifdef DODAJ_B2
        bus_master b2 (); //b2 se instancira samo ako je definisan DODAJ_B2
    `endif
endmodule
```

- Bulovi izrazi (npr. TEST && DODAJ_B2) nisu dozvoljeni u `ifdef iskazy



■ Upis u fajl

- Izlaz normalno ide na *standard output*, ali ga je moguće redirektovati u proizvoljan fajl

- Fajl se otvara sistemskim *task*-om **\$fopen**:

`<file_handle> = $fopen("<ime_fajla>");`

- **\$fopen** vraća 32-bitnu vrijednost koja se zove *multikanalni deskriptor*
- U njemu je setovan samo jedan bit
- Kod standardnog izlaza to je bit 0 (*standard output* je kanal 0) i on je uvijek otvoren
- Svaki naredni poziv \$fopen otvara novi kanal sa setovanim bitom na jednoj više poziciji
- Redni broj kanala odgovara poziciji setovanog bita u multikanalnom deskriptoru

■ Upis u fajl – nastavak

■ Primjer:

// Multikanalni deskriptori

integer handle1, handle2, handle3; // integer-i su 32-bitne vrijednosti

// *standard output* je otvoren; deskriptor = 32'h0000_0001 (bit 0 setovan)

initial

begin

 handle1=\$fopen("file1.out"); //handle1 = 32'h0000_0002 (bit 1 setovan)

 handle2=\$fopen("file2.out"); //handle2 = 32'h0000_0004 (bit 2 setovan)

 handle3=\$fopen("file3.out"); //handle3 = 32'h0000_0008 (bit 3 setovan)

end

- Pomoću multikanalnih deskriptora je moguće istovremeno selektivno upisivati u više fajlova
- Za upisivanje se koriste sistemske direktive: `$fdisplay`, `$fmonitor`, ...
- Rade isto kao `$display` i `$monitor`, samo što upisuju u fajl

■ Upis u fajl – nastavak

■ Primjer:

```
integer handle1, handle2, handle3, desc1, desc2, desc3;  
initial  
begin  
    handle1=$fopen("file1.out"); //handle1 = 32'h0000_0002 (bit 1 setovan)  
    handle2=$fopen("file2.out"); //handle2 = 32'h0000_0004 (bit 2 setovan)  
    handle3=$fopen("file3.out"); //handle3 = 32'h0000_0008 (bit 3 setovan)  
end  
initial  
begin  
    desc1 = handle1 | 1; // bitwise or; desc1 = 32'h0000_0003  
    $fdisplay(desc1, "Display 1"); // upisuje u fajl file1.out i na standardni izlaz  
    desc2 = handle2 | handle1; //desc2 = 32'h0000_0006  
    $fdisplay(desc2, "Display 2"); // upisuje u fajlove file1.out i file2.out  
    desc3 = handle3; // desc3 = 32'h0000_0008  
    $fdisplay(desc3, "Display 3"); // upisuje samo u fajl file3.out  
end
```



■ Upis u fajl – nastavak

- Zatvaranje fajlova se obavlja direktivom **\$fclose**

- Primjer:

```
$fclose(handle1);
```

- Nakon zatvaranja nije više moguće upisivati u fajl, jer je odgovarajući bit u multikanalnom deskriptoru postavljen na nulu
- Za novo upisivanje je potrebno ponovo otvoriti fajl sa \$fopen

■ \$strobe

- Radi slično kao **\$display** uz jednu bitnu razliku: ako se više iskaza izvršava u istom vremenskom trenutku kao i \$display task – redoslijed izvršavanja (iskaza i \$display-a) nije determinisan
- Ako se koristi **\$strobe** on će se izvršiti **poslije** svih ostalih iskaza koji se izvršavaju u istom vremenskom trenutku
- Primjer:

```
always @(posedge clock)
begin
    a = b;
    c = d;
end
always @(posedge clock)
    $strobe("a=%b, c=%b", a, c); // prikazivanje vrijednosti na posedge
```

- Vrijednosti će biti ispisane **nakon** izvršavanja iskaza **a = b;** i **c = d;**
- Da je korišćen \$display, rezultati bi mogli biti drugačiji



Dopunski materijal

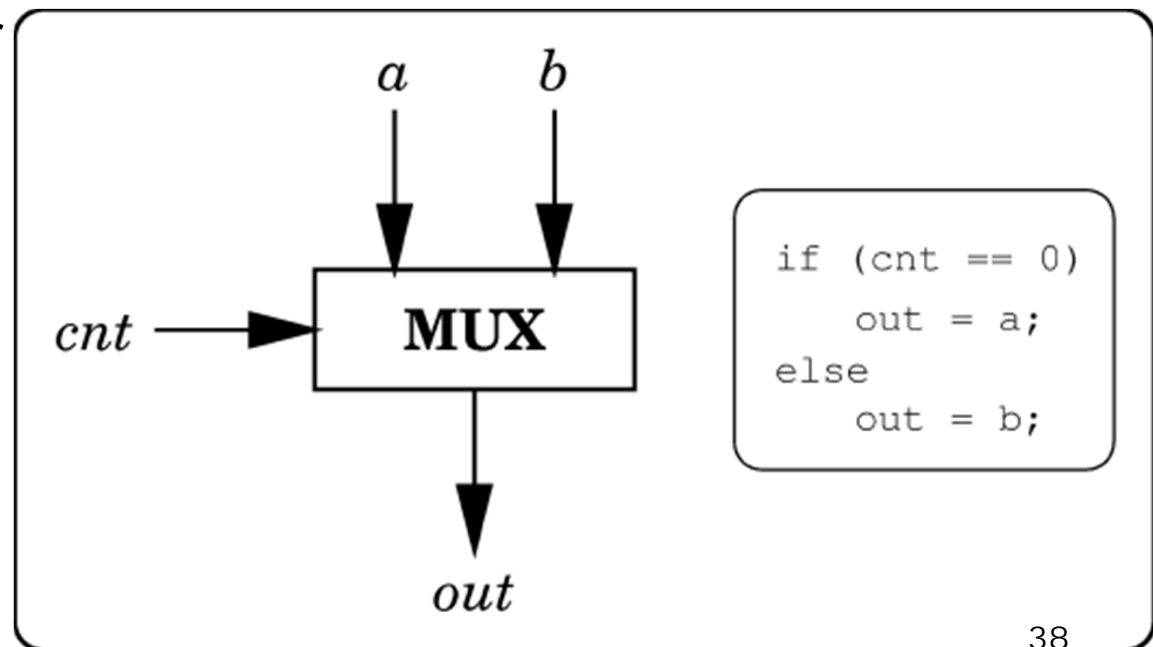


■ Sinteza

- Automatska translacija HDL koda u odgovarajuću *netlist*-u digitalnih ćelija
- Alatke su, u suštini, kolekcije programa sa *vještačkom inteligencijom* koji interpretiraju, optimizuju i formiraju dizajn, **na osnovu prethodnih iskustava eksperata** u sintezi
- Sinteza se, u principu, obavlja u nekoliko koraka:
 1. Prepoznavanje struktura u kodu u smislu ili apstraktnih koncepata u dizajnu (recimo automati) ili logičkih funkcija
 2. Primjena raznih tehnika za optimizaciju dizajna (kriterijum može biti brzina, prostor, kašnjenja, potrošnja, ...):
 - Minimizacija Bulovih funkcija (uključujući uklanjanje konstantnih signala)
 - Izbor arhitekture (recimo izbor binarnog sabirača)
 - Minimizacija dijagrama stanja (automata)
 - Generisanje i distribucija signala (npr. taktni signal)
 - Automatsko umetanje među-stepena (*pipeline*)

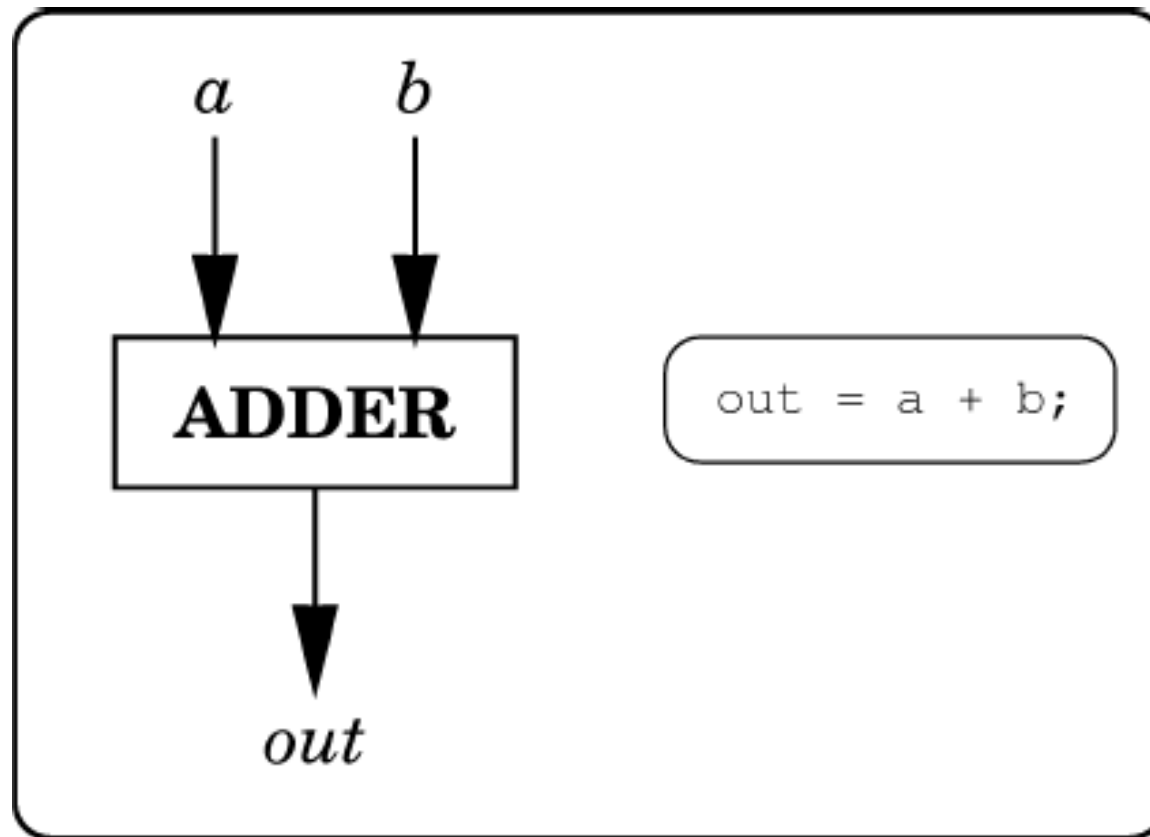
■ Sinteza – nastavak

- 3. Mapiranje dizajna na konkretnu tehnologiju – poenta HDL dizajna je da bude potpuno nezavisno od ciljne tehnologije – alati za sintezu se brinu o tome; isti dizajn se može koristiti na različitim tehnologijama
- U praksi se često prožimaju faza optimizacije i mapiranja, jer karakteristike ćelija utiču na algoritme optimizacije
- Primjer translacije Verilog koda u digitalnu logiku:
 - **if-else** u multiplekser



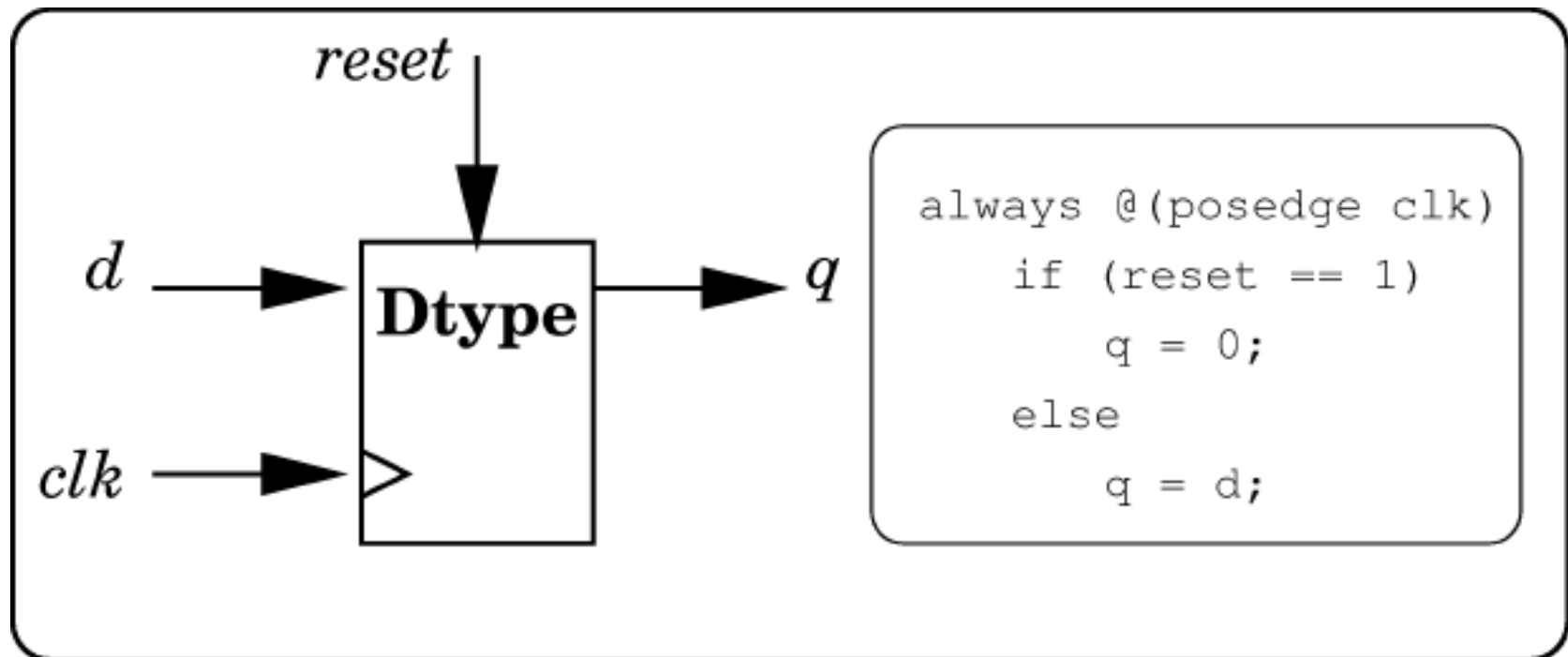
■ Sinteza – nastavak

- Osnovne aritmetičke operacije se preslikavaju u odgovarajuće sklopove



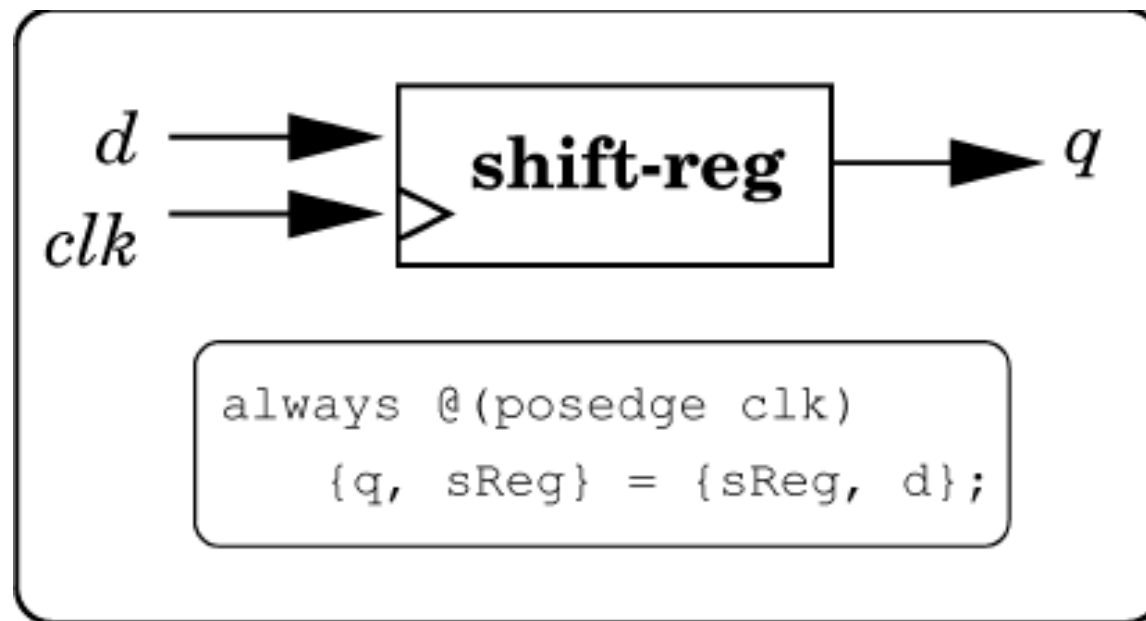
■ Sinteza – nastavak

- Ako je **if-else** uokvireno sa **@(posedge ...)**, onda neće biti preslikano u multiplekser, već u memorijski element



■ Sinteza – nastavak

- Konkatenacija preslikana u pomjerački registar:



■ Sinteza – nastavak

- Iz primjera se moglo vidjeti da se relativno složeni moduli mogu realizovati pomoću vrlo jednostavnog koda
- Međutim, ako u alatu za sintetizaciju nije ugrađeno “znanje” za određeni oblik Verilog koda, postaje problem sintetizovati ga na najbolji način
- Tada način kodiranja igra važnu ulogu
- **Primjer:** binarni sabirač/oduzimač – operacija se bira kontrolnim signalom

```
module adder(out, contr, a, b);  
  output [3:0] out;  
  input contr;  
  input [3:0] a, b;  
  
  assign out = contr ? a - b : a + b;  
  
endmodule
```

Alat će uslovni iskaz preslikati u multipleksor koji bira jedan od izlaza druga dva modula: jednog koji sabira a i b i drugog koji ih oduzima



■ Sinteza – nastavak

- Kompetentan dizajner zna da se oduzimanje $a - b$ može svesti na sabiranje a i dvojnog komplementa od b ; dvojni komplement se dobija komplementiranjem bitova operanda b i sabiranjem sa 1
- Na osnovu ovoga se može napisati optimalniji kod:

```
module adder(out, contr, a, b);
```

```
output [3:0] out;
```

```
input contr;
```

```
input [3:0] a, b;
```

```
assign out = contr + a + ({contr, contr, contr, contr} ^ b);
```

```
endmodule
```

- Ovaj kod će rezultirati korišćenjem samo jednog sabirača



■ Obratiti pažnju!

- Treba se čuvati od korišćenja registarske promjenljive sa malim brojem bitova kao promjenljive u petlji
- Takođe treba paziti na testiranje registarske promjenljive da li ima negativnu vrijednost
- Sabiranje i oduzimanje tretiraju registarsku promjenljivu kao **unsigned**
- Neopreznim korišćenjem registarskih promjenljivih se lako može proizvesti beskonačna petlja

- *Primjeri:*

`reg [2:0] i; // i je uvijek između 0 i 7`

`for (i=0; i<8; i=i+1) ... // beskonačna petlja!!!`

`for (i=-4; i<0; i=i+1) ... // ne izvršava se!!!`

- U ovakvim situacijama treba koristiti **integer** za promjenljivu u petlji (i)

■ *Race condition*

- Javlja se kada dva ili više iskaza, koji treba da se izvrše u istom trenutku simulacije, daju različite rezultate ako se promijeni njihov redoslijed izvršavanja

```
module race(out1, out2, clk, rst);  
  output out1, out2;  
  input clk, rst;  
  reg out1, out2;  
  always @(posedge clk or posedge rst)  
    if (rst) out1 = 0;  
    else out1 = out2;  
  always @(posedge clk or posedge rst)  
    if (rst) out2 = 1;  
    else out2 = out1;  
endmodule
```

- Ako se prvi *always* blok izvrši prvi (na prvom sljedećem taktu nakon reset), oba izlaza (*out1* i *out2*) će imati vrijednost 1; ako se drugi *always* blok izvrši prvi, oba izlaza će imati vrijednost 0 => *race condition*



■ ***Race condition*** – nastavak

- Nastaje i kada se istoj promjenljivoj dodjeljuje vrijednost iz više od jednog *always* bloka, čak i pomoću neblokirajućeg dodjeljivanja
- Da bi izbjegli najčešći razlog za pojavu *race condition*-a:
 - nikada ne kombinovati blokirajuća i neblokirajuća dodjeljivanja (*blocking* i *nonblocking assignments*)
 - nikada ne dodjeljivati vrijednost istoj promjenljivoj iz više od jednog *always* bloka
- Sintetizator neće sintetizovati dizajn koji ne poštuje ova pravila, ali će simulator izvršiti simulaciju
- Treba *latch*-ovati sve izlaze važnih komponenti dizajna (velikih blokova ili modula) – flip flopovi na izlazima garantuju da će se na ivici sljedećeg taktnog impulsa ispravni signali predati ostalim komponentama sistema
- Ne koristiti zero-delay (#0) osim eventualno u modulu za testiranje (*stimulus*, *testbench*)



■ ***Deadlock***

- Kad proces A čeka da bude omogućen od strane procesa B, a proces A treba da omogući proces B
- Kod može biti sintaksno ispravan, a da ipak ima *deadlock* situaciju
- Može se dogoditi i u sinhronim i u asinhronim sistemima
- Primjer asinhronog kola sa *deadlock*-om:

```
module deadlock;  
  reg reg1, reg2;  
  initial begin  
    reg1 = 1'b0;  
    wait @ (reg2==1'b1)  
  end  
  always @(reg1) begin  
    if (reg1==1'b1)  
      reg2 = 1'b1;  
  end  
endmodule
```



■ Sintetizovanje dizajna – napomene

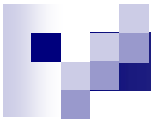
- Nije moguće sve direktive iz Veriloga sintetizovati
- Da bi bili sigurniji da će ono što simuliramo biti moguće sintetizovati treba poštovati neka okvirna pravila:
 - ne miješati detektovanje ivice i detektovanje promjene nivoa u listi događaja (*sensitivity* listi)
 - ako se uvodi kašnjenje u *always* bloku, onda koristiti samo blokirajuća dodjeljivanja (*blocking assignments*)
 - ako se kašnjenje kodira za potrebe simulacije: izbjegavati kašnjenje unutar *always* blokova – umjesto toga procijeniti ukupna kašnjenja na izlazima tog bloka i implementirati ih na samim izlazima pomoću neprekidnog dodjeljivanja (*continuous assignment*)



■ Sintetizovanje dizajna – napomene

■ Primjer posljednjeg pravila:

```
module proba(output X, Y, ostatak sensitivity liste);  
    lokalne deklaracije  
    ...  
    assign #5 X = Xreg; // procijenjeno ukupno kašnjenje = 5  
    assign #7 Y = Yreg; // procijenjeno ukupno kašnjenje = 7  
    ...  
    always@(posedge clock) // neki always blok  
    begin  
        x1 = (a && b) ^ c;  
        Xreg = x1 | x2;  
        Yreg = &(y1 + y2);  
    end  
endmodule
```



■ Najčešća greška početnika

- Razmišlja o Verilogu kao računarskom programu umjesto kao o načinu da se opiše digitalni sklop
- Ako dizajner makar približno ne zna u šta bi se trebao sintetizovati njegov kod, vjerovatno mu se neće dopasti rezultat:
 - napraviće se mnogo više hardvera nego što je bilo neophodno, ili
 - simulacija će se odvijati korektno, ali nije moguće izvršiti hardversku implementaciju (sintetizaciju)
- Zato o dizajnu treba razmišljati u obliku blokova kombinacionih kola, registara i automata
- Skicirati ove blokove na papiru i vidjeti kako ih treba povezati prije nego što se počne sa pisanjem koda



■ Napomene – nepoznata/neispravna vrijednost

- Na početku simulacije se izlazi flip flopova (memorijskih elemenata) inicijalizuju na nepoznatu vrijednost (x)
- Ovo je korisno da bi se uočile greške uzrokovane propustom dizajnera da izvrši resetovanje sistema prije nego što počne da se koristi
- Ako se na ulazu logičkog kola nađe *plivajući* signal, na izlazu se može naći vrijednost x (kada ne može da se odredi ispravna vrijednost)
- Ako se u simulaciji uoči neka **x** vrijednost, to je najčešće posljedica neke greške ili loše tehnike kodiranja
- U sintetizovanom kolu to korespondira sa plivajućim ulazom kola, neinicijalizovanom stanju ili nadmetanju više različitih signala na istomvodu
- U realnom kolu će na tom mjestu biti logička 0 ili 1 – po slučajnom “izboru” logičkog kola, što će dovesti do nepredvidivog ponašanja dizajniranog sistema



■ Napomene – kašnjenja

- Kašnjenja koja se navode u verilog kodu su pogodna da bi se za vrijeme simulacija moglo predvidjeti koliko brzo će sklop raditi (ako su zadana smisljena kašnjenja)
- Pogodna su i u procesu traženja grešaka, da se razdvoje uzrok i posljedica (traženje uzroka nekog pogrešnog izlaznog signala je teško ako se svi ulazi mijenjaju istovremeno i u istom trenutku kao i izlaz)
- Ova kašnjenja se prilikom sintetizacije ignorišu – kašnjenje logičkih kola koja su rezultat sintetizacije zavisi od ***tpd*** specifikacija, a ne od brojeva u Verilog kodu

■ D flip flop – razlika sinhronog i asinhronog reseta

```
module flopr (q, clk, reset, d);  
input clk, reset;  
input d;  
output q;  
reg q;  
// asinhroni reset  
always @ (posedge clk or posedge reset)  
    if (reset) q <= 1'b0;  
    else q <= d;  
endmodule  
// reaguje odmah na pojavu reseta
```

```
module flopr (q, clk, reset, d);  
input clk, reset;  
input d;  
output q;  
reg q;  
// sinhroni reset  
always @ (posedge clk)  
    if (reset) q <= 1'b0;  
    else q <= d;  
endmodule  
// resetuje se tek na ivicu clk-a
```

■ Primjer zamjene *sensitivity* liste džoker znakom

```
module fulladder (a, b, cin, s, cout);  
input a, b, cin;  
output s, cout;  
reg s, cout;  
reg p, g; // pošto se nalaze sa lijeve strane '=' moraju biti reg  
always @ (*) // Verilog 2001 (ne postoji u originalnom standardu)  
begin  
    p = a ^ b;  
    g = a & b;  
    s = p ^ cin;  
    cout = g | (p & cin);  
end  
endmodule
```

Koriste se blokirajuća dodjeljivanja da bi se prvo izračunali **p**, **g** i **s**, a na kraju **cout**

- U ovom slučaju bi @(a, b, cin) bilo ekvivalentno sa @ (*)
- Upotrebom džokera se izbjegava česta greška – zaboraviti da se neki signal stavi u listu ...



■ ... nastavak

- `always @ (*)` će ponovo evaluirati iskaze unutar *always* bloka svaki put kad se neki od signala sa desne strane '=' ili '<=' promijeni
- Znači da je `@ (*)` bezbjedan način da se modeluje kombinatorno kolo
- *Primjer 2:*

```
module inverter (a, y);  
  input [3:0] a;  
  output [3:0] y;  
  reg [3:0] y;  
  always @ (*)  
    y = ~a;  
endmodule
```

- `y` mora biti deklarisan kao **reg** jer se nalazi sa lijeve strane kod proceduralnog dodjeljivanja u *always* bloku
- Međutim, `y` je izlaz kombinatornog kola, a ne registar