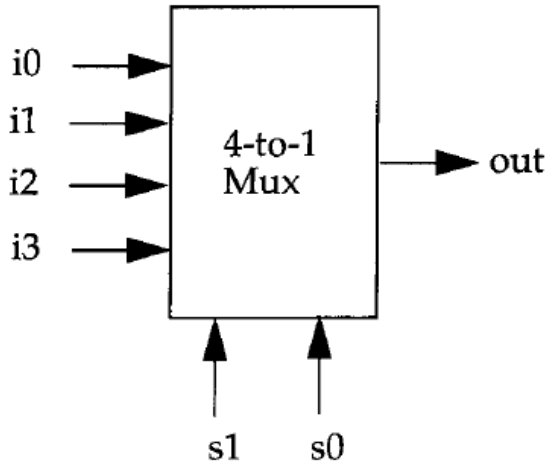
Primjer 1 (*behavioral* dizajn) – MUX 4/1

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;
// preračunati signal out ako se promijeni bilo koji od ulaznih signala
always @(s1 or s0 or i0 or i1 or i2 or i3)
      begin
         case ({s1, s0})
                  2'b00: out = i0;
                  2'b01: out = i1;
                  2'b10: out = i2;
                  2'b11: out = i3;
                  default: out = 1'bx;
         endcase
      end
endmodule
```



■ **Primjer 2 (*behavioral* dizajn)** – brojač (4 bita)

```
module counter(Q, clock, clear);
output [3:0] Q;
input clock, clear;
reg [3:0] Q;
always @(posedge clear or negedge clock)
      begin
         if (clear)
                  Q = 4'd0;
         else
                  Q = (Q + 1) % 16;       // može i bez % 16
      end
endmodule
```

***Behavioral* dizajn** – rezime

- Digitalni sklop se opisuje u obliku algoritma koji taj sklop izvršava – ne mora da sadrži detalje hardverske implementacije
- Osnova modelovanja su proceduralne strukture initial i always: svi ostali iskazi se mogu naći samo u okviru ovih blokova
  - initial blok se *izvršava* jednom
  - always blok se *izvršava* neprekidno, dok se simulacija ne završi
- Vrijednosti registarskih promjenljivih se zadaju pomoću proceduralnog dodjeljivanja
  - Blokirajuće dodjeljivanje se mora završiti prije nego se *izvrši* sljedeći iskaz
  - Neblokirajuće dodjeljivanje "zakazuje" *izvršavanje* dodjeljivanja i nastavlja da procesuira sljedeći iskaz

***Behavioral* dizajn** – rezime

- Vremenska kontrola redoslijeda izvršavanja iskaza u Verilogu:
  - Delay-based
    - Regular delay
    - Zero delay
    - Intra-assignment delay
  - Event-based
    - Regular event
    - Named event
    - Event OR
  - Level-sensitive (wait)

***Behavioral* dizajn** – rezime

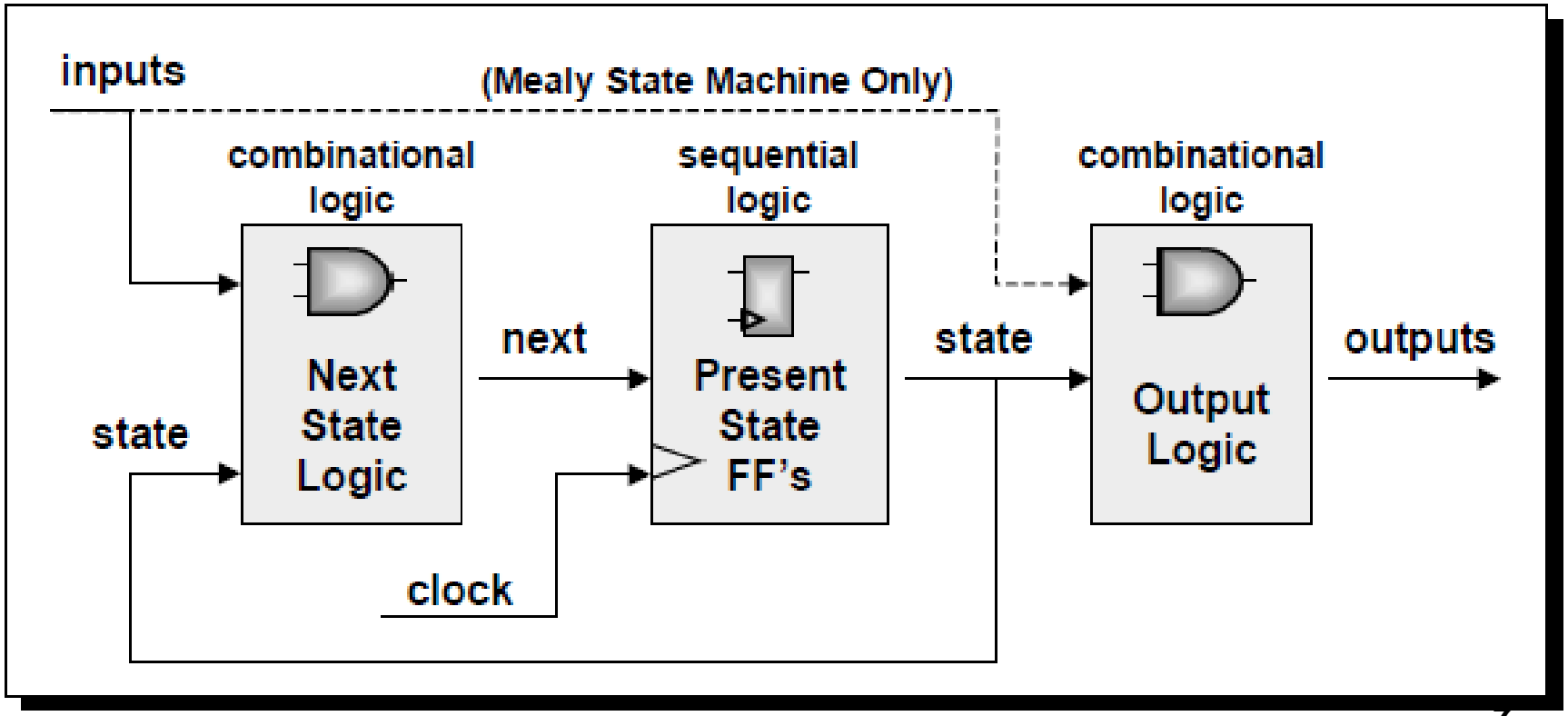
- Uslovni iskazi se modeluju sa **if** i **else**
- U slučaju višestrukih uslova, preporučuje se korišćenje **case** (casex, casez)
- Petlje se mogu realizovati na 4 načina:
  - while
  - for
  - repeat
  - forever
- Dva tipa blokova: sekvencijalni (begin – end) i paralelni (fork – join)
- Blokovi se mogu ugnijezditi i imenovati
- Ako je blok imenovan, može se isključiti sa bilo kog mjesta unutar dizajna
- Imenovani blokovi se referenciraju po hijerarhijskom imenu

Zadatak

- Realizovati multiplekser MUX 8/1 i testirati njegov rad sa svim mogućim kombinacijama ulaznih signala.

Automat (*Finite State Machine* – FSM)

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



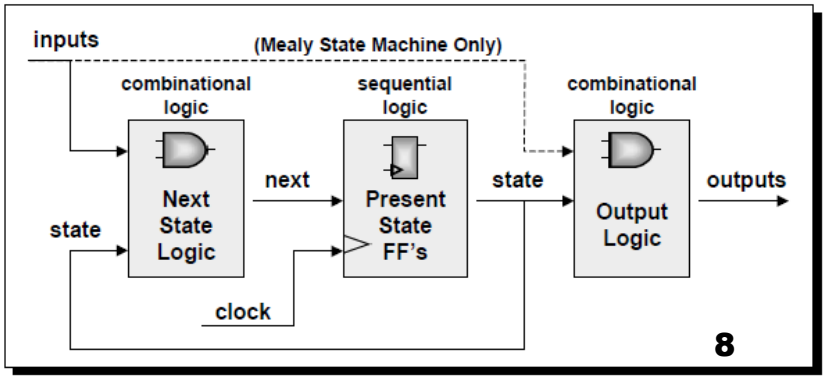
Realizacija automata

- **1.** Kodirati sekvencijalni (taktovani) **always** blok koji reprezentuje trenutno stanje pomoću **vektorske registarske promjenljive**
- **2.** Kodirati kombinacioni **always** blok koji reprezentuje *next-state* logiku
- **3.** Kodirati kombinacionu mrežu za generisanje odgovarajućeg izlaza (output logika):
  - koristeći *continuous assignement* 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

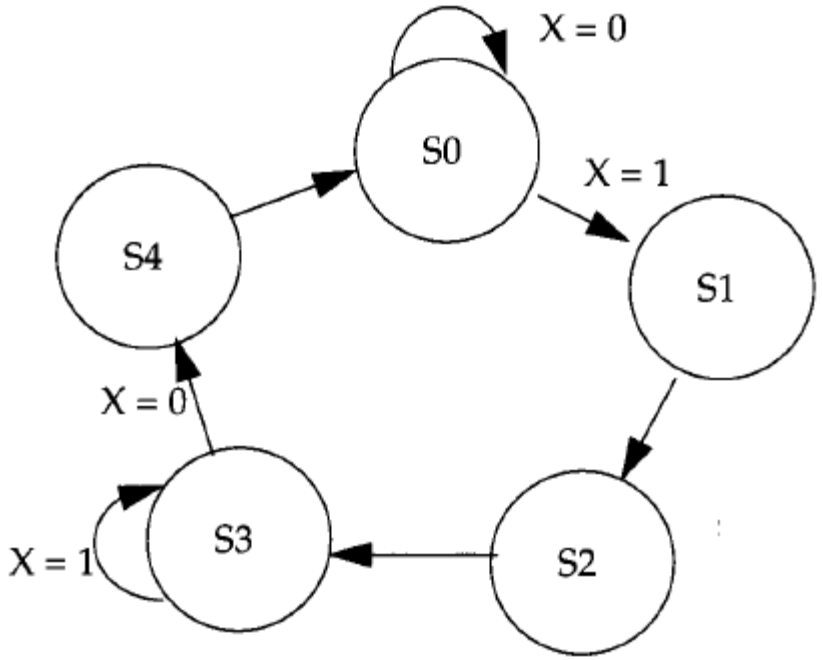


Primjer (*behavioral* dizajn automata) – kontroler za semafor

- Ukrštanje glavnog i sporednog puta (put sa jako malom frekvencijom saobraćaja)
- Glavni put ima prioritet (u startu je zeleno svijetlo)
- Zeleno svijetlo na sporednom putu traje samo toliko da se propuste vozila koja se tamo nalaze
- Čim tamo nema više vozila pali se žuto, pa crveno svijetlo, a na glavnom putu se pali zeleno svijetlo
- Na sporednom putu se nalazi senzor: X=1 ako ima vozila na sporednom putu; X=0 ako tamo nema vozila

- Problem se rješava upotrebom automata (Moor)

Primjer (kontroler za semafor) – nastavak



Prelazak iz stanja S1 u S2, iz stanja S2 u S3 i iz stanja S4 u S0 se mora obavljati sa nekim kašnjenjem

- Stanje S0: glavni – zeleno; sporedni – crveno
- Stanje S1: glavni – žuto; sporedni – crveno
- Stanje S2: glavni – crveno; sporedni – crveno
- Stanje S3: glavni – crveno; sporedni – zeleno
- Stanje S4: glavni – crveno; sporedni – žuto

Primjer (kontroler za semafor) – nastavak

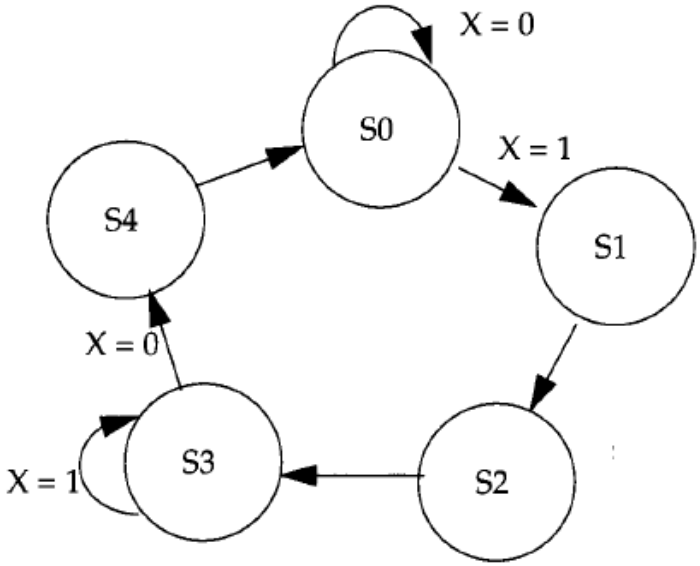
- Umjesto brojeva treba koristiti razumljive oznake:

```
`define TRUE 1'b1
`define FALSE 1'b0
// definicije signala
`define CRVENO 2'd0
`define ZUTO 2'd1
`define ZELENO 2'd2
// definicije stanja: glavni  sporedni
`define S0 3'd0  //   zeleno  crveno
`define S1 3'd1  //    zuto   crveno
`define S2 3'd2  //   crveno  crveno
`define S3 3'd3  //   crveno  zeleno
`define S4 3'd4  //   crveno   zuto
// kašnjenja
`define ZUTO_CRVENO 3
`define CRVENO_ZELENO 2
```

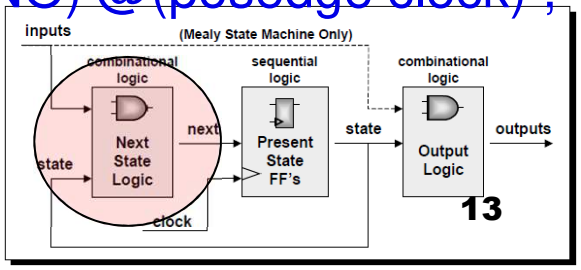
Primjer (kontroler za semafor) – nastavak

```
module semafor(glavni, sporedni, X, clock, clear);
    output [1:0] glavni, sporedni; // 3 moguća svijetla (C,Ž,Z)
    reg [1:0] glavni, sporedni;
    input X; // ako je TRUE ima vozila na sporednom putu
    input clock, clear;

    // pomocne promjenljive (za stanja automata)
    reg [2:0] stanje;
    reg [2:0] sljedece_stanje;

    // semafor počinje u stanju S0
    initial // inicijalizacija
        begin
            stanje = `S0;
            sljedece_stanje = `S0;
            glavni = `ZELENO;
            sporedni = `CRVENO;
        end  ...
```

Realizacija automata – primjer semafora

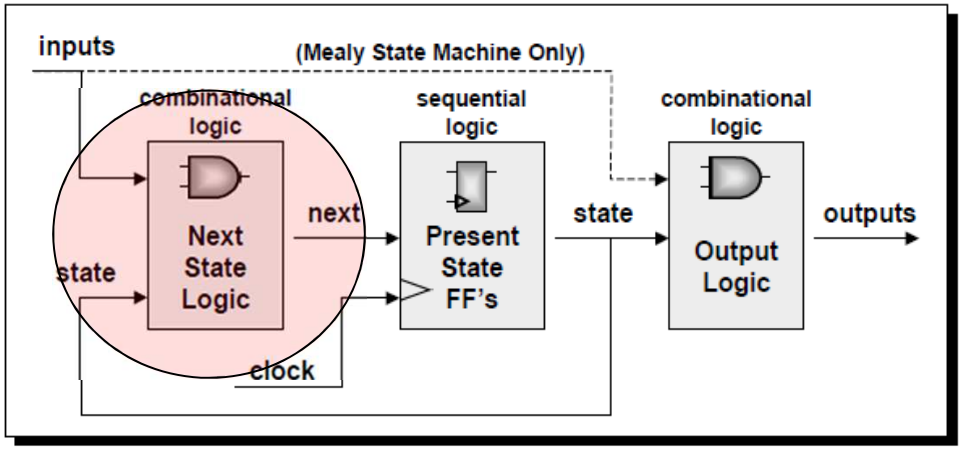


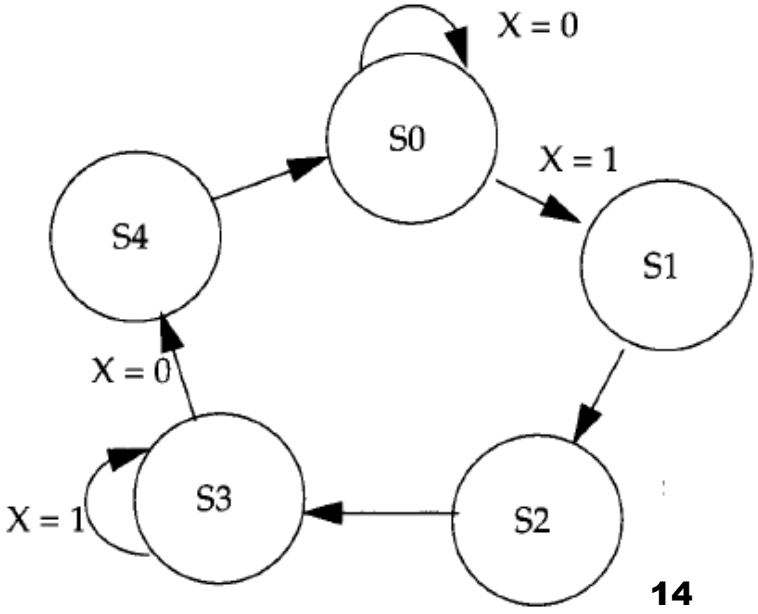

```
always @(stanje or clear or X)
begin
   if (clear)
          sljedece_stanje = `S0;
   else
          case (stanje)
            `S0:   if(X) sljedece_stanje = `S1;
                   else sljedece_stanje = `S0;
            `S1:   begin // sačekaj nekoliko pozitivnih ivica takta
                          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:  ...
```



Primjer (kontroler za semafor) – nastavak

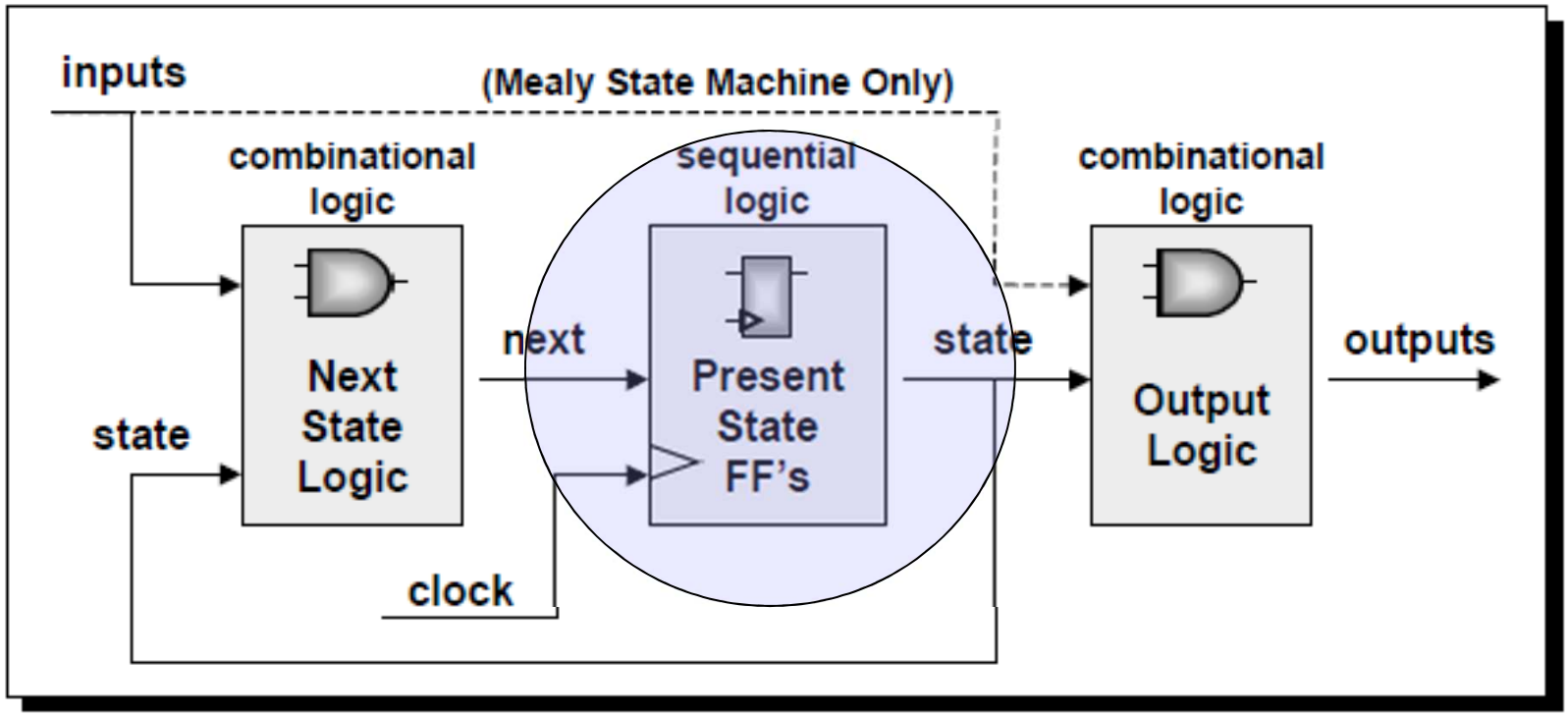
```
        `S3:   if(X) sljedece_stanje = `S3;
               else sljedece_stanje = `S4;
        `S4:   begin // sačekaj nekoliko pozitivnih ivica takta
                        repeat(`ZUTO_CRVENO) @(posedge clock) ;
                        sljedece_stanje = `S0;
               end
      default: sljedece_stanje = `S0;
    endcase
end
```



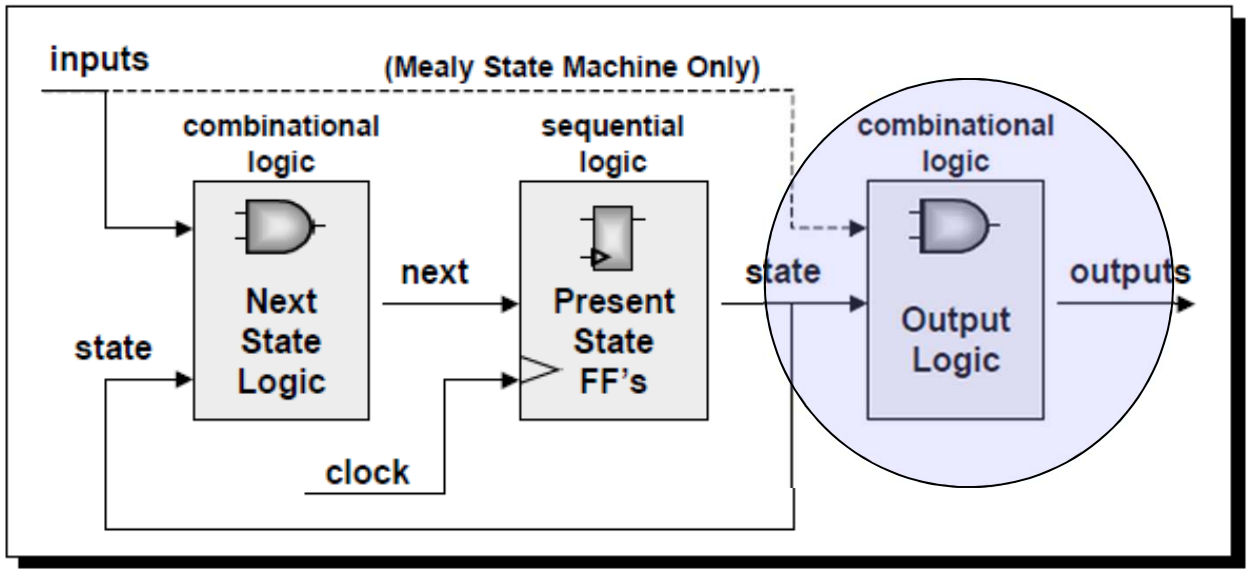


Primjer (kontroler za semafor) – nastavak

```
// promjena stanja se vrši na pozitivnoj ivici takta
always @(posedge clock)
   stanje = sljedece_stanje;
```



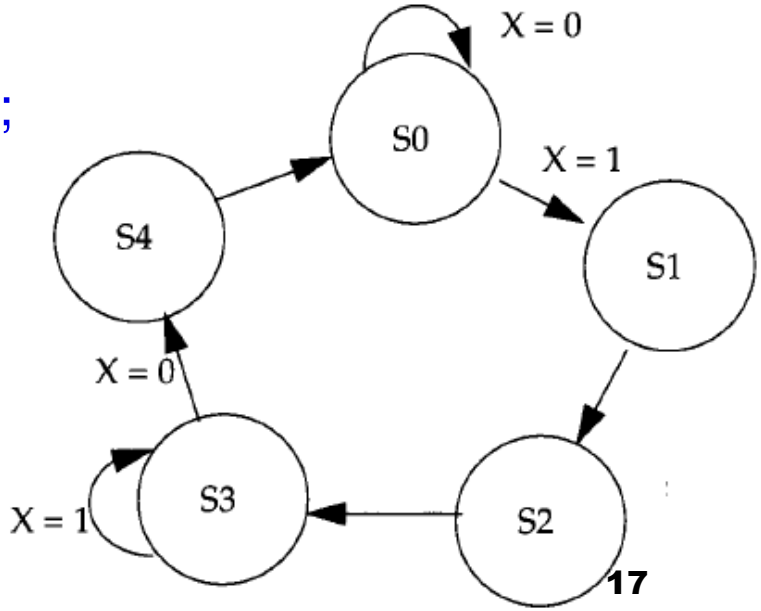
Primjer (kontroler za semafor) – nastavak




```
// Izračunati vrijednost
// svjetlosnih signala
always @(stanje)
begin
   case(stanje)
         `S0:     begin
                        glavni = `ZELENO;
                        sporedni = `CRVENO;
                  end
         `S1:     begin
                        glavni = `ZUTO;
                        sporedni = `CRVENO;
                  end
         `S2:     ...
```

- **Primjer** (kontroler za semafor) – nastavak

```
        `S2:    begin
                        glavni = `CRVENO;
                        sporedni = `CRVENO;
                end
        `S3:    begin
                        glavni = `CRVENO;
                        sporedni = `ZELENO;
                end
        `S4:    begin
                        glavni = `CRVENO;
                        sporedni = `ZUTO;
                end
    endcase
end
endmodule
```



Primjer (kontroler za semafor) – nastavak

```
module stimulus;
wire [1:0] GLAVNI, SPOREDNI;
reg VOZILO_NA_SPOREDNOM;
reg CLOCK, CLEAR;

// instanciranje kontrolera
semafor S(GLAVNI, SPOREDNI, VOZILO_NA_SPOREDNOM, CLOCK, CLEAR);

initial
   $monitor ($time, " Glavni = %b Sporedni = %b X = %b",
            GLAVNI, SPOREDNI, VOZILO_NA_SPOREDNOM);

initial
      begin
         CLOCK = `FALSE;
         forever #5 CLOCK = ~CLOCK;
      end
```

...

Primjer (kontroler za semafor) – nastavak

```
initial // resetovanje kontrolera
        begin
            CLEAR = `TRUE;
            repeat (5) @(negedge CLOCK);
            CLEAR = `FALSE;
        end
// stimulus
initial
        begin
            VOZILO_NA_SPOREDNOM = `FALS
            #200 VOZILO_NA_SPOREDNOM =
            #100 VOZILO_NA_SPOREDNOM =
            #200 VOZILO_NA_SPOREDNOM =
            #100 VOZILO_NA_SPOREDNOM =
            #200 VOZILO_NA_SPOREDNOM =
            #100 VOZILO_NA_SPOREDNOM =
            #100 $finish;
        end
endmodule
```

```
0 Glavni = 10 Sporedni = 00 X = 0
200 Glavni = 10 Sporedni = 00 X = 1
205 Glavni = 01 Sporedni = 00 X = 1
235 Glavni = 00 Sporedni = 00 X = 1
255 Glavni = 00 Sporedni = 10 X = 1
300 Glavni = 00 Sporedni = 10 X = 0
305 Glavni = 00 Sporedni = 01 X = 0
335 Glavni = 10 Sporedni = 00 X = 0
500 Glavni = 10 Sporedni = 00 X = 1
505 Glavni = 01 Sporedni = 00 X = 1
535 Glavni = 00 Sporedni = 00 X = 1
555 Glavni = 00 Sporedni = 10 X = 1
600 Glavni = 00 Sporedni = 10 X = 0
605 Glavni = 00 Sporedni = 01 X = 0
635 Glavni = 10 Sporedni = 00 X = 0
800 Glavni = 10 Sporedni = 00 X = 1
805 Glavni = 01 Sporedni = 00 X = 1
...
```

Izbjegavanje najčešćih grešaka kod automata

- Da bi se izbjegli latch-evi “pokriti” sve moguće skokove, IF i CASE directive:

```
always @ (state or A or B or C...);
   begin
        next_state = S1; // default vrijednost;
        if (A | B&C) next_state = S3;
        if ((~A)&(~B)&C) next_state=S2;
        . . .
```

- Alternativa je da se default vrijednost pojavljuje u svakom else:

```
always @ (state or A or B or C...);
   begin
        if (A | B&C) next_state = S3; else next_state = S1;
        if ((~A)&(~B)&C) next_state=S2; else next_state = S1;
        . . .
```

Izbjegavanje najčešćih grešaka kod automata

- state mora biti promjenljiva u triger listi, kao i sve promjenljive unutar IF uslova:

```
always @ (state or A or B or C...);
   begin
        case (state)
```

- Sve promjenljive unutar IF uslova moraju biti u triger listi:

```
always @ (state or A or B or C...);
   begin
        if (A | B&C) next_state = S3;  . . .
```

- Promjenljiva sa lijeve strane znaka '=' ne smije biti u triger listi:

```
always @ (state or A or B or C...);
   begin
        A = B+C; // odmah trigeruje always proceduru – beskonačna petlja
```

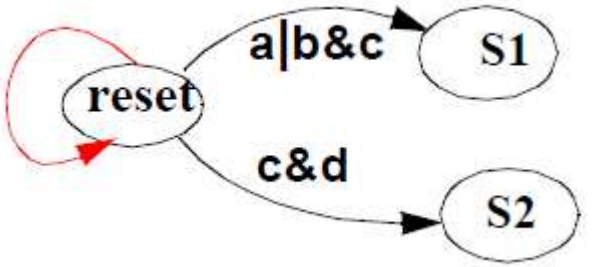
Izbjegavanje najčešćih grešaka kod automata

- Svaki CASE završiti sa default, bez obzira da li je to neophodno:

```
case (state)
        . . .
        default: next_state = reset;
endcase
```

- Uvijek staviti "povratnu petlju" u dijagram stanja:

```
case (state)
        reset: if (a|b&c) next_state=S1;
                else if (c&d) next_state=S2;
                else next_state = reset;
        . . .
```



Zadaci

- Realizovati digitalno kolo koje će detektovati pojavu tri uzastopne logičke jednice na svom ulazu. Detektovanje se signalizira logičkom jedinicom na izlazu kola, u trajanju jednog taktnog impulsa. Nakon detektovanja kolo prestaje sa radom. Ponovo počinje da radi nakon resetovanja posebnim ulaznim signalom. Napraviti odgovarajući stimulus koji će detaljno provjeriti funkcionisanje ovog kola.
- Realizovati digitalno kolo koje će detektovati pojavu sekvence 101 na svom ulazu. Detektovanje se signalizira logičkom jedinicom na izlazu kola, u trajanju jednog taktnog impulsa. Moguće je preklapanje sekvenci. Napraviti odgovarajući stimulus koji će detaljno provjeriti funkcionisanje ovog kola.

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 deklariše 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 // mora se uokviriti sa begin – end jer je više iskaza
               #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
      asimetricni_takt;  // poziv taska asimetricni_takt
    task inicijalizacija_takta; // task inicijalizacija_takta
      clock = 1'b0;
    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 deklariše 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 deklariše 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 // nije neophodno jer je samo jedan iskaz
           shift = (control == `LEFT_SHIFT) ? (address << 1) : (address >>1);
        end
    endfunction
endmodule
```