## UNIT-5
## SYNTHSIS OF COMBINATIONAL AND SEQUENTIAL LOGIC USING VERILOG

**Syllabus:** Synthesis of combinational logic, Net list of structured primitives, A set of continuous assignment statements and level sensitive cyclic behavior with examples, Synthesis of priority structures, Exploiting logic don't care conditions, Synthesis of sequential logic with latches, Accidental synthesis of latches and Intentional synthesis of latches, Synthesis of sequential logic with flip-flops, Synthesis of explicit state machines

**Design and synthesis**

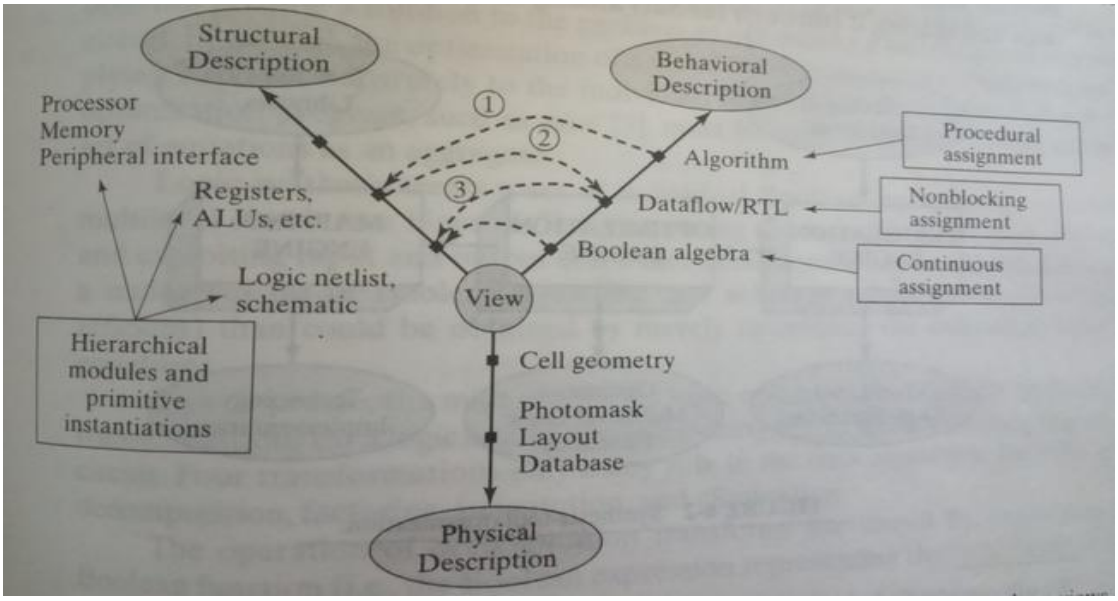
- Given an overall system architecture and partitioning is stable one can capture the design
- Work at many level typically at the register transfer level.
- Synthesis tools convert RTL to gate level circuits.
- Large designs require working at high level automation to minimize time to market -> RTL should be synthesizable
- Typically design at the RTL level of abstraction as lowest level of abstraction
    a) is used for functional simulation
    b) Synthesized to gate level for use by the CAD tools

**Logic Synthesis:** Verilog is used to

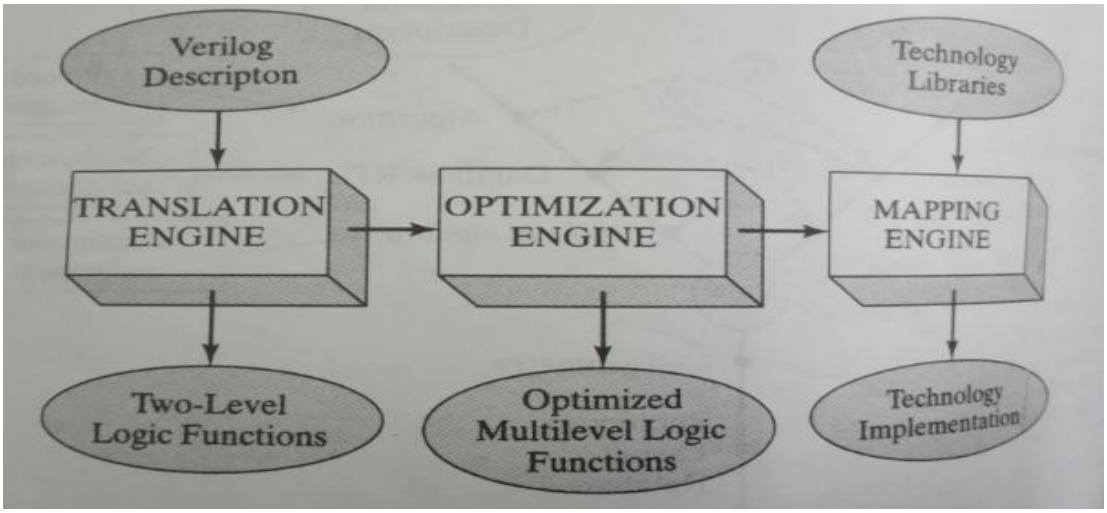
i. Model hardware for discrete-event simulation
ii. Input to logic synthesis.
iii. Construct test benches

- Only a subset of Verilog constructs can be synthesized into an efficient circuit
- Exactly what can synthesize depends on tools and technologies
- Logic synthesis involves
    i. Translating Verilog source to a net list
    ii. Optimization for speed and circuit size
        Detect and eliminate redundant logic
        Detect combinatorial feedback loops
        Exploit don't cares
        Collapse equivalent states
        Optimize for a particular technology

**A Y-chart representation of verilog constructs supporting synthesis activity in three domains is shown in below figure**



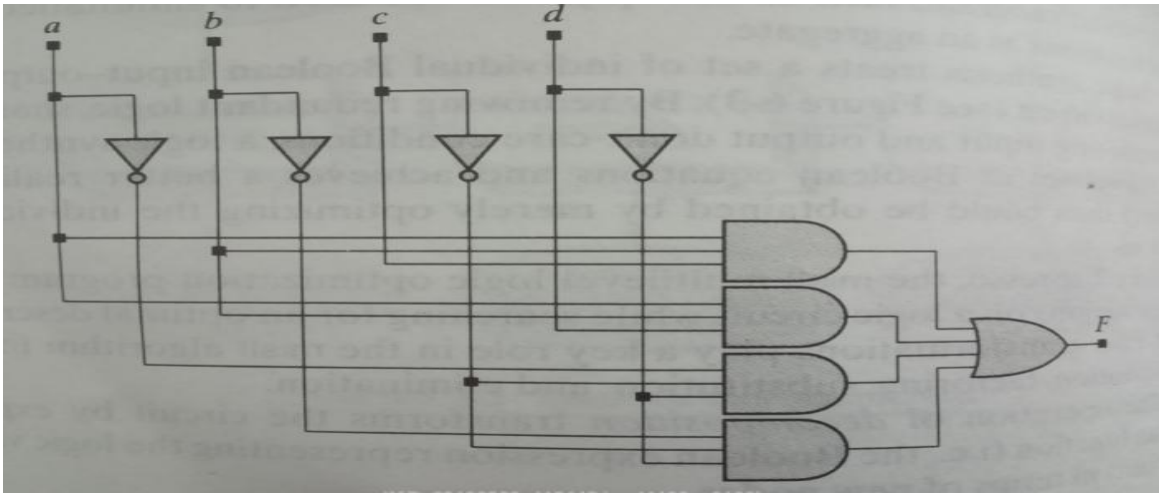

**Synthesis Tool Organization**




**Circuit Before Decomposition**

The following figure is to be decomposed in terms of new nodes X and Y.

The original form F = abc + abd + a'c'd' + b'c'd'

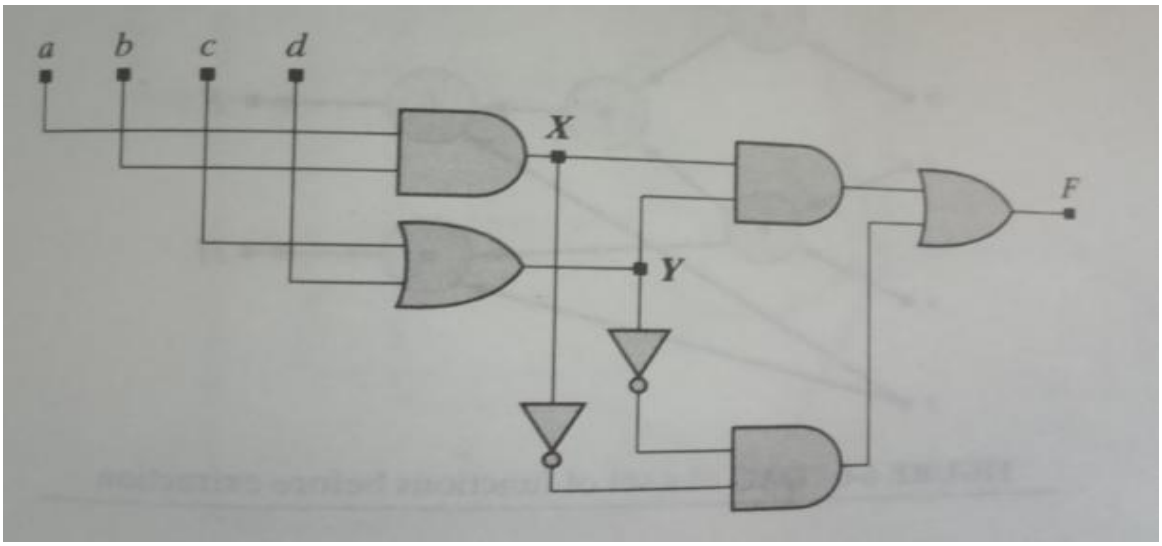


**Circuit After Decomposition**

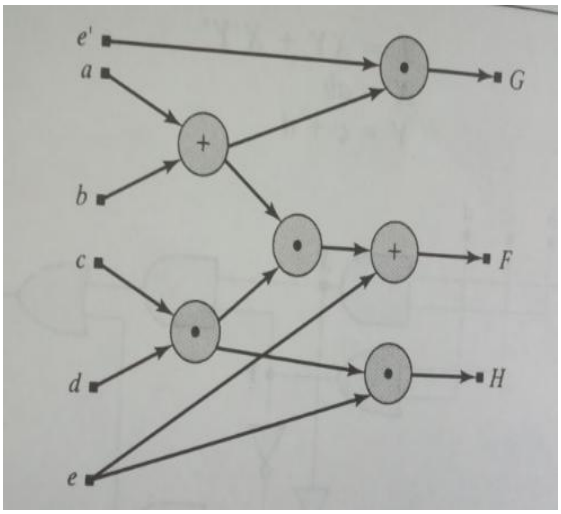
F = XY + X'Y'

X= ab

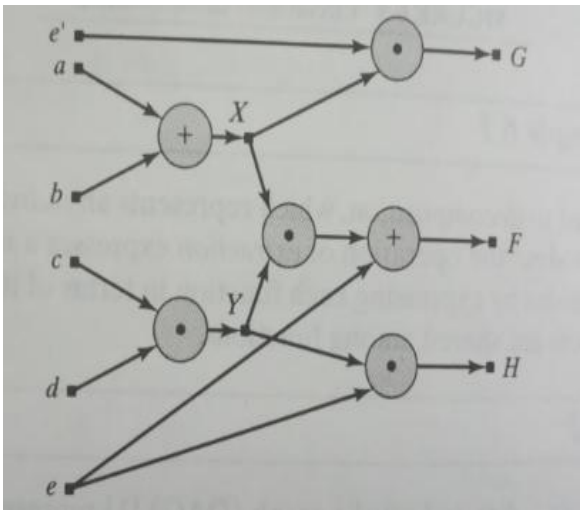
Y= c+d

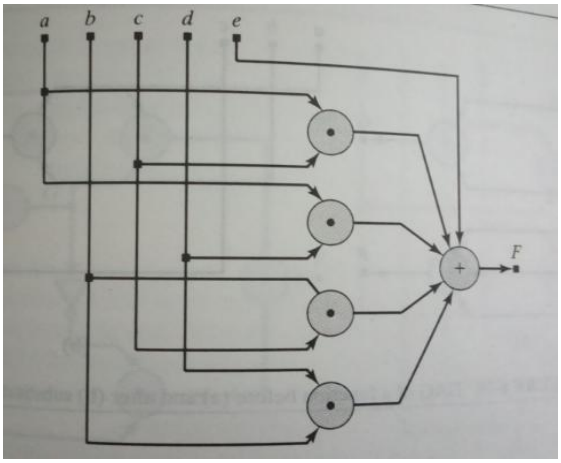


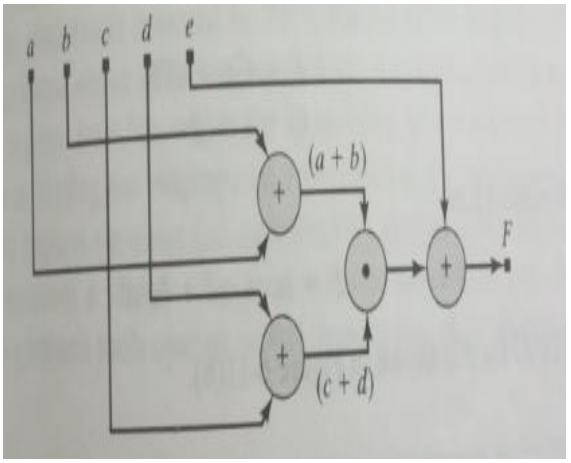

**Circuit Before Decomposition**




**Circuit After Decomposition**



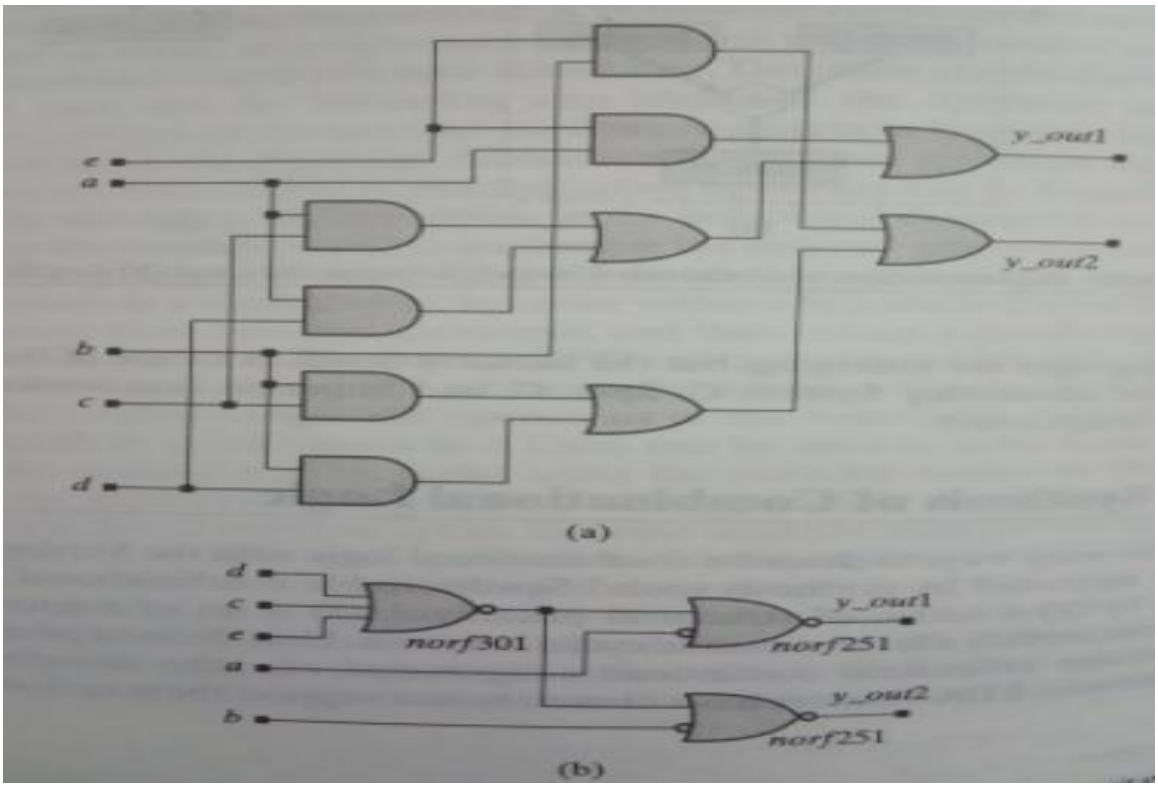



**Synthesis of Combinational Logic:** It can be described by

- A net list of structural primitives or combinatorial modules
- Continuous assignment statements
- Level sensitive cyclic behaviors

**A net list of structural primitives or combinatorial modules**

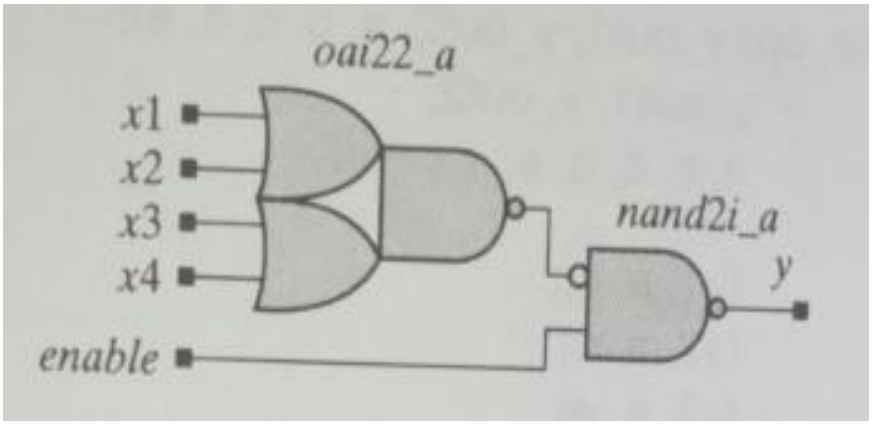



```
module boole_opt(y_out1, y_out2, a, b, c, d, e);
 output        y_out1, y_out2;
 input         a, b, c, d, e;

 and           (y1, a, c);
 and           (y2, a, d);
 and           (y3, a, e);
 or            (y4, y1, y2);
 or            (y_out1, y3, y4);
 and           (y5, b, c);
 and           (y6, b, d);
 and           (y7, b, e);
 or            (y8, y5, y6);
 or            (y_out2, y7, y8);
endmodule
```

**Continuous assignment statements**

- For combinational logic, behavioral descriptions take the form of Boolean equations
- Verilog equivalent is continuous assignment statement.
- Assigns net (wire) to output of multilevel combinational circuit




```
module or_nand (y, enable, x1, x2, x3, x4);
 output    y;
 input     enable, x1, x2, x3, x4;

 assign y = ~(enable & (x1 | x2) & (x3 | x4));
endmodule
```

**Level sensitive cyclic behaviors**

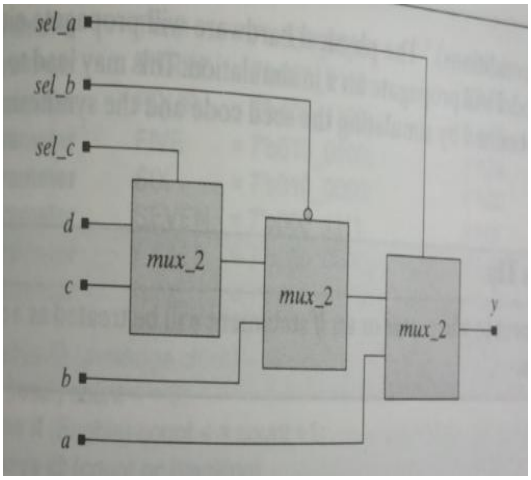
- Level sensitive cyclic behavior will synthesis to combinational logic if there is an output for every possible input combination (to inputs in sensitivity list)
    a) Sensitivity list must be sensitive to every input
    b) Every path through behavior must assign value to every output
- Defines functionality in device independent way but doesn't define realization, timing

***Example:***

```
module comparator (AltB, AgtB, AeqB, A1, A0, B1, B0);
input      A1, A0, B1, B0;
output   AltB, AgtB, AeqB;
reg       AltB, AgtB, AeqB;
always @ (A0 or A1 or B0 or B1)
begin
AltB =({A1,A0} < {B1,B0});
AgtB =({A1,A0} > {B1,B0});
```

```
AeqB =({A1,A0} == {B1,B0});
end
endmodule
```

**Synthesis of priority structures**




```
module mux_4pri (y, a, b, c, d, sel_a, sel_b, sel_c);
 output  y;
 input   a, b, c, d, sel_a, sel_b, sel_c;
 reg     y;

  always @ (sel_a or sel_b or sel_c or a or b or c or d)
   begin
    if   (sel_a == 1)          y = a; else
    if   (sel_b == 0)          y = b; else
    if   (sel_c == 1)          y = c; else
                               y = d;
   end
endmodule
```

**EXPLOITING LOGICAL DON'T CARE CONDITIONS**

- When **case**, conditional branch(**if**), or conditional assignment (**?... :**) statements are used in verilog behavioral description of combinational logic then the synthesized net list should produce the same simulation results if the code has **default** assignments that are purely 0 or 1.
- Simulation results will differ if the default or branch statements makes an explicit assignment of an **X** or a **Z**.
- A synthesis tool will treat **casex** and **casez** statements as **case** statements.
- Those **case** items that decode to explicit assignment of X or Z will be treated as don't care conditions for the purpose of logical minimization of the equivalent Boolean expressions

**Synthesis of sequential Logic with latches**

- Latches are synthesized in two ways as intentionally and accidentally
- A feedback free net list of combinational primitives or continuous assignments will synthesize into latch-free combinational logic
- If the assign statement or level sensitive always @ has *feedback* a latch is synthesized

```
assign Q = s ? D:Q;
```

- Feedback free net list of combinational logic will form latch free combinatorial logic
  - REG variables do not always mean flip-flops
- **Accidental synthesis of latches** can occur in procedural combinational logic blocks

- If an output is not specified for some combination of inputs in the sensitivity list Verilog assumes current values of variables set in procedural assignments is maintained and latch will be inferred.
- Likewise all RHS operands of procedural assignments should be in sensitivity list
- Also RHS operand of procedural assignment must not be on LHS of the expression (explicit feedback)

**Accidental synthesis of latches**

```
module mux_latch (yout, sela, selb, datax, datay);
output yout;
input sela, selb, datax, datay;
reg yout;
always @ (sela or selb or datax or datay)
 case ({ sela, selb })
        2'b10:yout=datax;
        2'b01:yout=datay;
endcase
endmodule
```

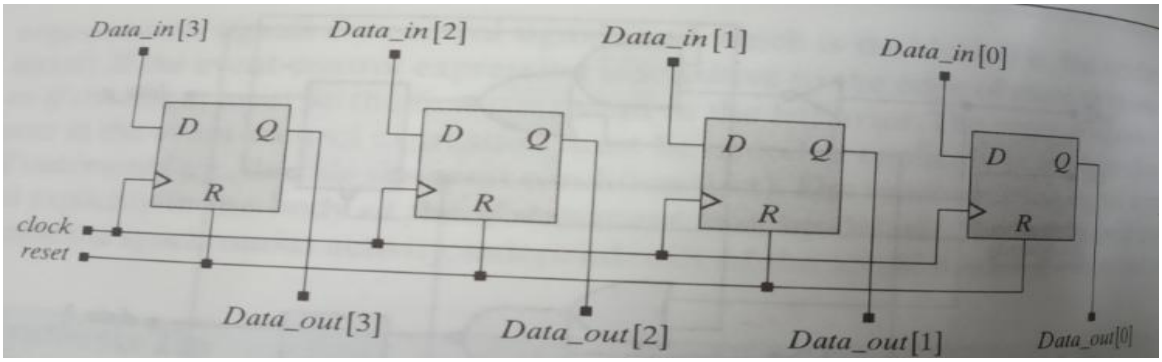
**Intentional synthesis of latches**

```
module latch_if (datout, datain, latchenable);
output [3:0] dataout;
input   [3:0] datain;
input   latchenable;
reg     [3:0] dataout;
always @ (latchenable or datain)
        if (latchenable) dataout=datain;
        else dataout=dataout;
endmodule
```

- A continuous assignment that uses the conditional operator (? :) with feedback will synthesize to a latch. Ex: SRAM

  assign data_out= (CS==0) ? (WE==0) ? Data_in : data_out : 1'bz;
- Here the signals CS and WE are active low chip select and write enable functions of the cell
- If the chip is selected and WE==0, data_out follows data_in (Transparent mode) but when WE switches to 1 then data_out=data_out (Latched mode).
- Synthesis tool infer the behavior of latch because the output of device is not effected by data_in while WE=1 and data_out holds the remaining value.
- If CS==1 then the cell is in three state high impedance condition

**Synthesis of sequential logic with flip flops**




```
module d_reg(dataout, clock, reset, datain);
output [3:0] dataout;
input  [3:0] datain;
input  clock, reset;
reg    [3:0] dataout;
always @ (posedge clock or posedge reset)
begin
        if (reset==1'b1) dataout<=4'b0;
        else dataout<=datain;
end
endmodule
```

- The behavior in emptycircuit assigns value to the register variable dout. But in the below verilog program dout is not referenced outside the scope of the behavior. Consequently, a synthesis tool will eliminate dout.

```
module emptycircuit (din, clk);
input  din;
input  clk;
reg    dout;
always @ (posedge clk)
begin
dout<=din;
end
endmodule
```

- If emptycircuit is modified to declare dout as an output port then dout will be synthesized as the output of a flip flop

Sensitivity list of an always block can be sensitive to clock edges (posedge or negedge). Clock, clk etc are not keywords and need to be defined and assigned. Synthesis tool must infer the synchronising signal and whether a flip flop is required.

**// D Flipflop**

```
module my_dff (q, qbar,d, clk,clr);
input d, clk, clr;
output q, qbar;
assign qbar = ~q;
always @ (posedge clk)
begin
if (clr == 0) q <= 0;
else q <=d;
end
endmodule
```

**// asynchronous clear D flipflop**

```
module my_dff (q, qbar,d, clk,clr);
input d, clk, clr;
output q, qbar;
assign qbar = ~q;
always @ (posedge clk or negedge clr)
begin
if (clr == 0) q <= 0;
else q <=d;
end
endmodule
```

**SHIFT REGISTERS:** Registers can be modeled as vectored flipflops

```
module shiftreg (A, E, clk, rst);
output A;
input  E;
input  clk,rst;
reg    A,B,C,D;
always @ (posedge clk or posedge rst)
begin
if (rst)
begin A <= 0; B <= 0; C <= 0; D <= 0; end
else
begin
A <= B;        //D <= E;
B <= C;        //C <= D;
C <= D;        //B <= C;
D <= E;        //A <= B;
end
end
endmodule
```

**Synthesis of explicit state machines**

- ***For state machine implementation***
  - State based on control operations (counting up,down, idle) rather than count itself. Can often be simplified to counting/idle.
  - More flexible
  - Generalizes to other word lengths
  - Count maintained in separate register than the state
- Explicit state machines have explicit storage of state variables and explicit logic to generate next state. Typically at least two (or three) always blocks
- Level sensitive, next state (and output) generator
  - Use = for combinations logic
  - Decode all possible states
- Edge sensitive, synchronous state transitions
  - Use <= for synchronous transitions
- Usually restrictions on state machine implementation for synthesis
  - Can assign a variable in state assignments
  - Must set entire state register at once
- Avoid having one variable assigned in more than one always block.