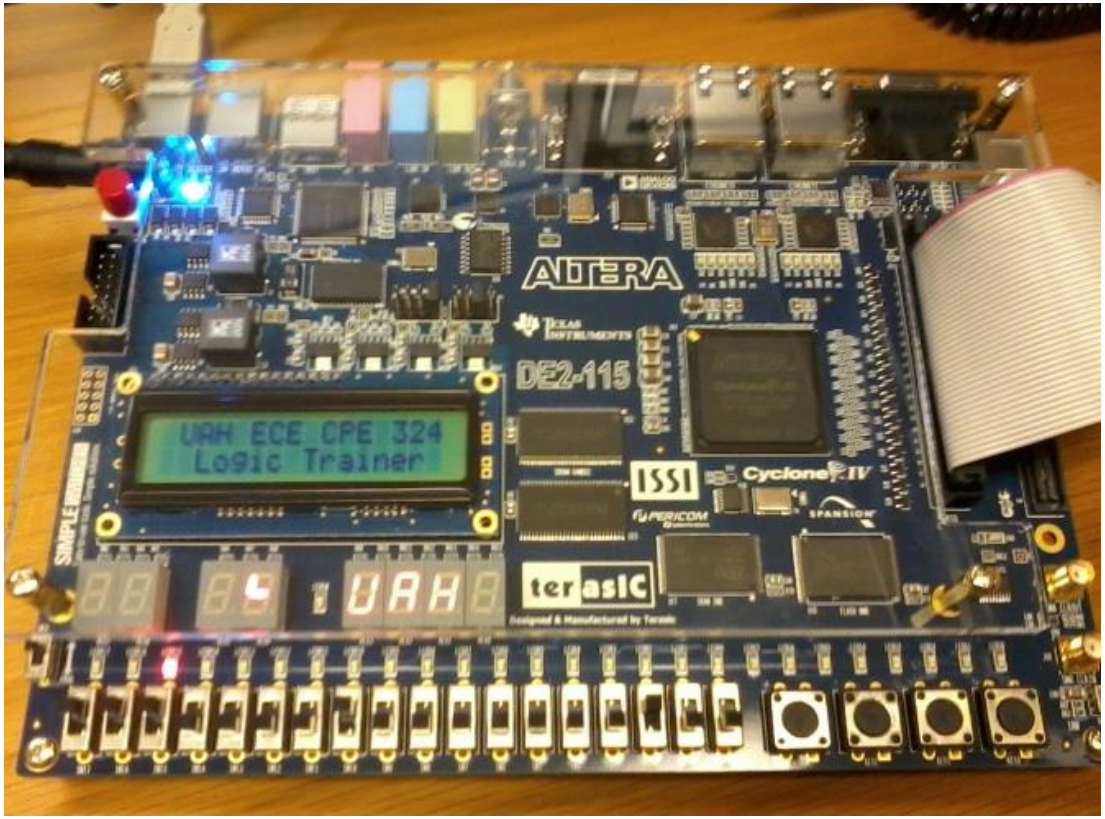
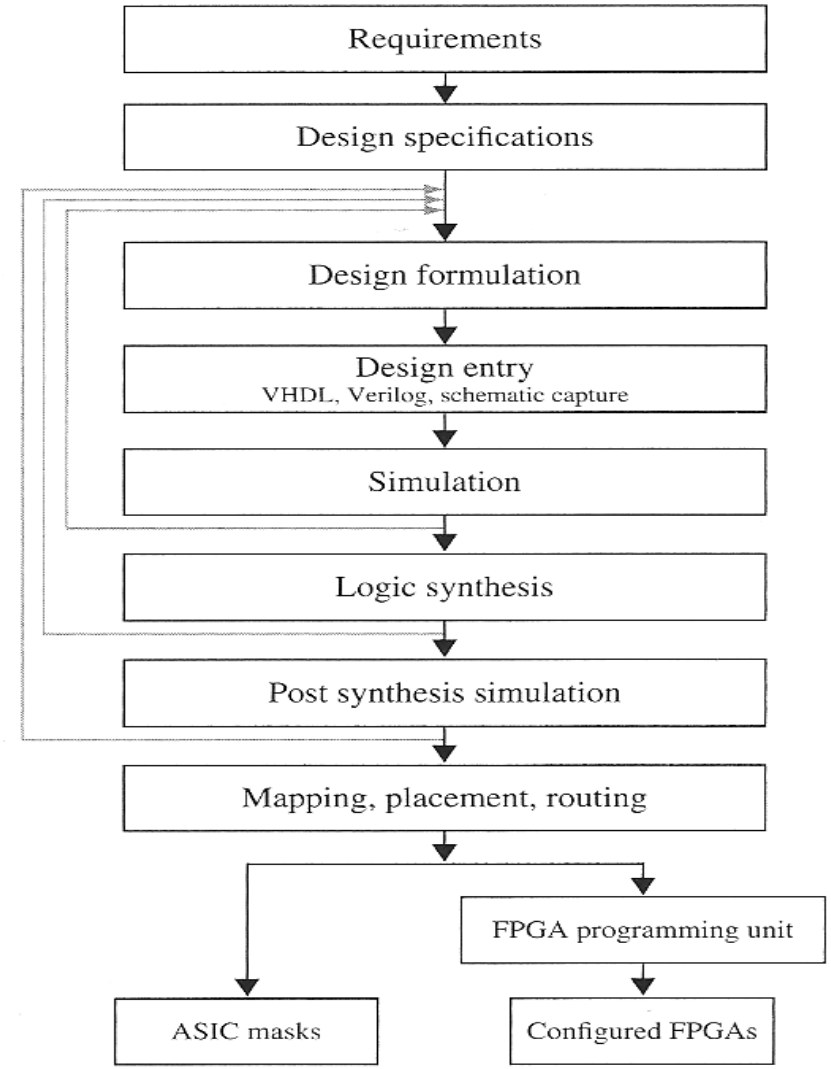
CPE 322
Digital Hardware Design Fundamentals

Electrical and Computer Engineering
UAH

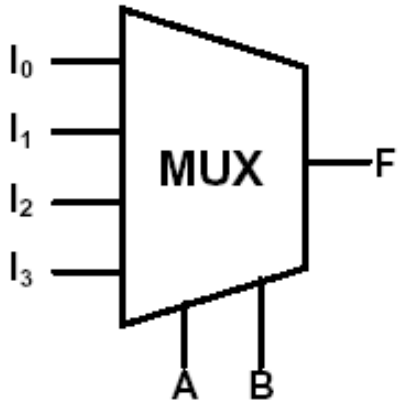
Introduction to Verilog Part II



Steps in Modern Digital System Design



Verilog Models for a MUX



Structural Design using AND, OR, NOT primatives

```
module MUX(input A, B, I0, I1, I2, I3, output F);

   wire A_,B_; // A negation, B negation nodes
   wire n0,n1,n2,n3; // internal nodes of AND terms

   not G0 (A_,A);
   not G1 (B_,B);
   and G2 (n0,A_,B_,I0);
   and G3 (n1,A_,B,I1);
   and G4 (n2,A,B_,I2);
   and G5 (n3,A,B,I3);
   or  G6 (F,n0,n1,n2,n3);

endmodule
```

Data flow design using continuous (concurrent signal) assignment statement

```
module MUX(input A, B, I0, I1, I2, I3, output F);

   assign F = (~A & ~B & I0) | (~A & B & I1) |
              (A & ~B & I2) | (A & B & I3);

endmodule
```

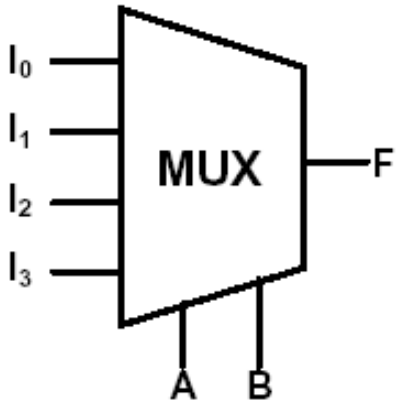
Verilog Models for a MUX

- Conditional continuous (concurrent signal) assignment statement format:
  - <condition> ? <if true> : <else>;

    If the Condition is true, the value of <if true> will be taken, otherwise the value of <else> will be taken.

Data flow design of MUX using **conditional** continuous (concurrent signal) assignment statement

```
module MUX(input A, B, I0, I1, I2, I3, output F);

   assign F = (A) ? (B ? I3 : I2) : (B ? I1 : I0);

endmodule
```



Behavioral Modeling using Procedural Constructs

- Two Procedural Constructs
  - initial Statement
  - always Statement
- initial Statement : Executes only once
- always Statement : Executes in a loop
- Example:

```
...
initial
  begin
    Sum = 0;
    Cout = 0;
  end
...
```

```
...
always @(A or B)
  begin
    Sum = A ^ B;
    Cout = A & B;
  end
...
```

Event Control

- Event Control
  - Edge Triggered Event Control
  - Level Triggered Event Control
- Edge Triggered Event Control

  @ (posedge CLK) //Positive Edge of CLK

  Curr_State = Next_state;

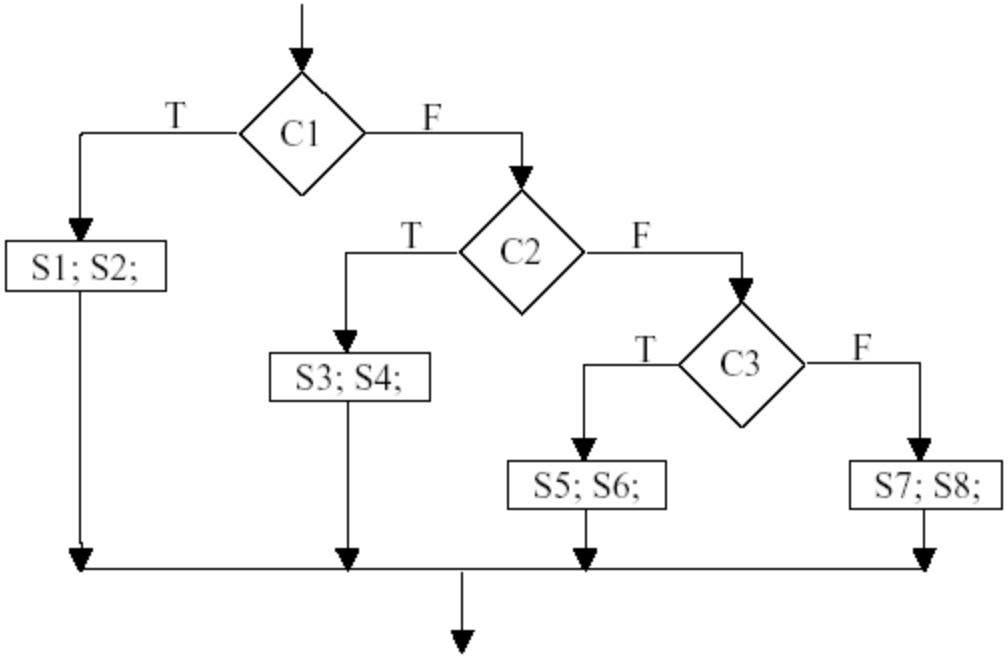
@ negedge	@ posedge
1 → x	0 → x
1 → z	0 → z
1 → 0	0 → 1
x → 0	x → 1
z → 0	z → 1

- Level Triggered Event Control

  @ (A or B) //change in values of A or B

  Out = A & B;

IF Statements in Verilog HDL

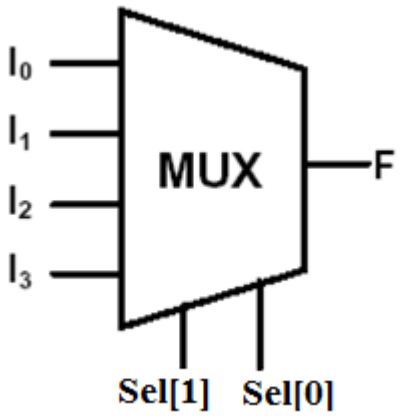



```
if (C1)
   begin
      S1;
      S2;
   end
else
   if (C2)
      begin
         S3;
         S4;
      end
   else
      if (C3)
         begin
            S5;
            S6;
         end
      else
         begin
            S7;
            S8;
         end
```

Verilog Models for a MUX

Data flow design of MUX using nested **if** statement placed inside an **always** block

```
module MUX(input [1:0] Sel, I0, I1, I2, I3, output reg F);

  always @ (Sel, I0, I1, I2, I3)
     begin
     if      (Sel == 2'b00) F = I0;
     else if (Sel == 2'b01) F = I1;
     else if (Sel == 2'b10) F = I2;
     else if (Sel == 2'b11) F = I3;
     end

endmodule
```



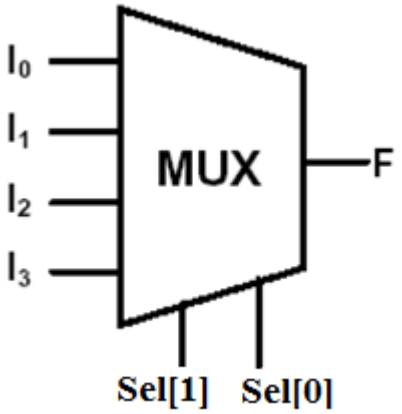
Verilog Models for a MUX

- The case statement has the general form:

case expression
    choice1 : sequential statements1
    choice2 : sequential statements2
    . . .
    [**default** : sequential statements]
endcase

Data flow design of MUX using a **case** statement placed inside an **always** block

```
module MUX(input [1:0] Sel, I0, I1, I2, I3, output reg F);

  always @ (Sel, I0, I1, I2, I3)
     case (Sel)
       2'b00 : F = I0;
       2'b01 : F = I1;
       2'b10 : F = I2;
       2'b11 : F = I3;
     endcase

endmodule
```

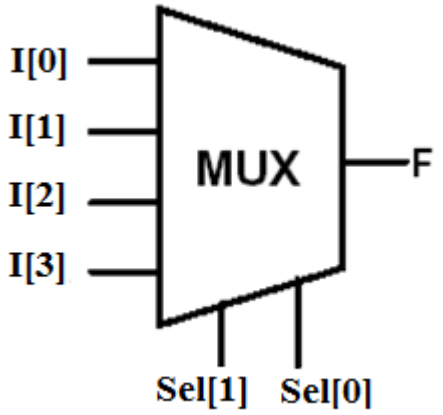


Verilog Models for a MUX

Data flow design of MUX using array declaration of the two sets of inputs

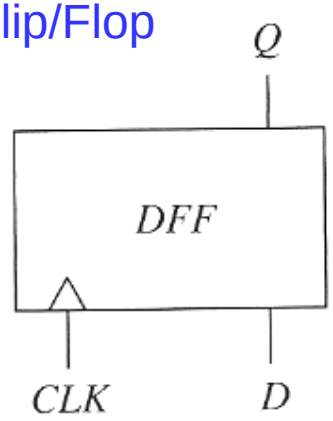
- The **Sel** input is used as the index into the **I** inputs.

```
module MUX(input [1:0] Sel, input [3:0] I, output reg F);

  always @ (Sel, I)
     begin
        F = I[Sel];
     end

endmodule
```



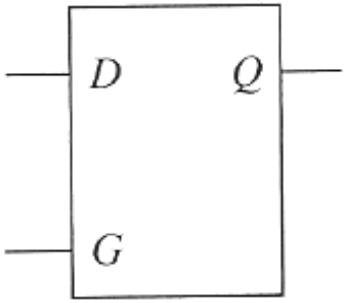
Modeling Simple Sequential Elements

D Flip/Flop




```
module DFF (input D, CLK, output reg Q);

  always @(posedge CLK)
     begin
     Q <= D;
     end

endmodule
```

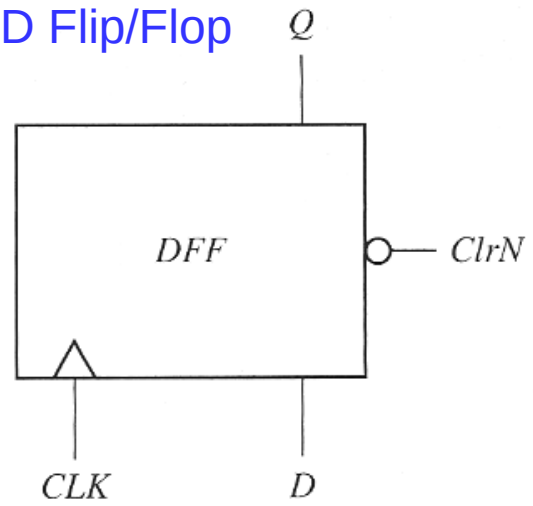
D Latch




```
module DLatch (input D, G, output reg Q);

  always @(D,G)
     begin
     if (G) Q <= D;
     end

endmodule
```

Modeling Simple Sequential Elements

D Flip/Flop




```
module DFF (input D, CLK, ClrN, output reg Q);

  always @(posedge CLK)
     begin
     if (~ClrN)
        Q <= 0;
     else
        Q <= D;
     end

endmodule
```

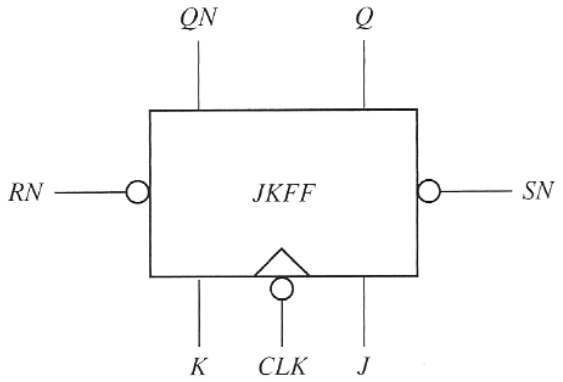
Synchronous ClrN

```
module DFF (input D, CLK, ClrN, output reg Q);

  always @(posedge CLK, negedge ClrN)
     begin
     if (~ClrN)
        Q <= 0;
     else
        Q <= D;
     end

endmodule
```

Asynchronous ClrN

- Note: should not mix level triggered and edge triggered inputs on sensitivity lists

Modeling Simple Sequential Elements



- Text incorrectly mixed level triggered and edge triggered inputs on the same sensitivity list which is not recommended practice for asynchronous case

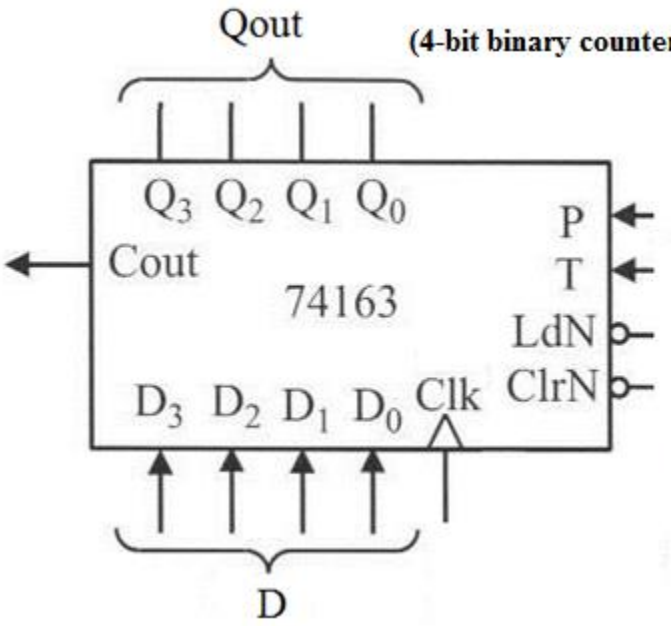
Asynchronous SN and RN

```
module JKFF (input SN, RN, J, K, CLK, output Q, QN);

  reg Qint;

  always @(negedge CLK, negedge RN, negedge SN)
     begin
     if (~RN)
        #8 Qint <= 0;
     else if (~SN)
        #8 Qint <= 1;
     else
        Qint <= #10 ((J & ~Qint) | (~K & Qint));
     end
   assign Q  = Qint;
   assign QN = ~Qint;

endmodule
```

Synchronous SN and RN

```
module JKFF (input SN, RN, J, K, CLK, output Q, QN);

  reg Qint;

  always @(negedge CLK)
     begin
     if (~RN)
        #8 Qint <= 0;
     else if (~SN)
        #8 Qint <= 1;
     else
        Qint <= #10 ((J & ~Qint) | (~K & Qint));
     end
   assign Q = Qint;
   assign QN = ~Qint;

endmodule
```

Modeling more Complex Sequential Elements



74163 FULLY SYNCHRONOUS COUNTER

Control Signals			Next State				
ClrN	LdN	P•T	$Q3^+$	$Q2^+$	$Q1^+$	$Q0^+$	
0	X	X	0	0	0	0	(clear)
1	0	X	D3	D2	D1	D0	(parallel load)
1	1	0	Q3	Q2	Q1	Q0	(no change)
1	1	1	present state + 1				(increment count)

If T=1, the counter generates a carry, C_{out}, when in state 15; consequently

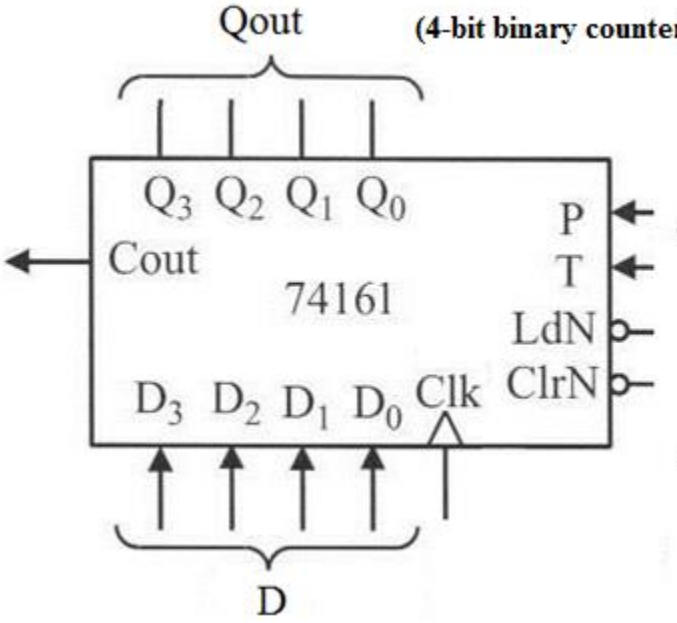
$$C_{out} = Q_3\, Q_2\, Q_1\, Q_0\, T$$

```
// 74163 FULLY SYNCHRONOUS COUNTER
module c74163(input LdN, ClrN, P, T, Clk, input reg [3:0] D, output Cout, output [3:0] Qout);

reg [3:0] Q;

assign Qout = Q;
assign Cout = Q[3] & Q[2] & Q[1] & Q[0] & T;

always @(posedge Clk)
   begin
   if (~ClrN) Q <= 4'b0000;
   else if (~LdN) Q <= D;
   else if (P & T) Q <= Q + 1;
   end
endmodule
```

Modeling more Complex Sequential Elements

- Synchronous inputs, such as the ClrN signal on the 74163 take effect after the active edge of the clock.
- Asynchronous inputs are level triggered not edge triggered.
- To ensure correct simulation properties, an always block should always contain the same type of triggering in the sensitivity list
  - If one signal is edge-sensitive, then all signals should be edge sensitive.

Modeling more Complex Sequential Elements



74161 SYNCHRONOUS COUNTER

with asynchronous Clear (ClrN)

Control Signals			Next State				
ClrN	LdN	P•T	$Q3^+$	$Q2^+$	$Q1^+$	$Q0^+$	
0	X	X	0	0	0	0	(clear)
1	0	X	D3	D2	D1	D0	(parallel load)
1	1	0	Q3	Q2	Q1	Q0	(no change)
1	1	1	present state + 1				(increment count)

If T=1, the counter generates a carry, C_{out}, when in state 15; consequently

$$C_{out} = Q_3\, Q_2\, Q_1\, Q_0\, T$$

```
// 74161 SYNCHRONOUS COUNTER with asynchronous clear
module c74163(input LdN, ClrN, P, T, Clk, input reg [3:0] D, output Cout, output [3:0] Qout);

reg [3:0] Q;

assign Qout = Q;
assign Cout = Q[3] & Q[2] & Q[1] & Q[0] & T;

always @(posedge Clk, negedge ClrN)
   begin
   if (~ClrN) Q <= 4'b0000;
   else if (~LdN) Q <= D;
   else if (P & T) Q <= Q + 1;
   end
endmodule
```

Blocking versus Nonblocking Procedural Assignment Statements

Blocking Assignments

```
module dummy (input Trigger,
              output reg [15:0] sum=0);

  reg [15:0] var1=1,var2=2,var3=3;

  always @ (Trigger)
     begin
     var1 = var2 + var3;
     var2 = var1;
     var3 = var2;
     sum = var1 + var2 + var3;
     end

endmodule
```

Sum = 15 (16'd15)

Nonblocking Assignments

```
module dummy (input Trigger,
              output reg [15:0] sum=0);

  reg [15:0] var1=1,var2=2,var3=3;

  always @ (Trigger)
     begin
     var1 <= var2 + var3;
     var2 <= var1;
     var3 <= var2;
     sum <= var1 + var2 + var3;
     end

endmodule
```

Sum = 6 (16'd6)

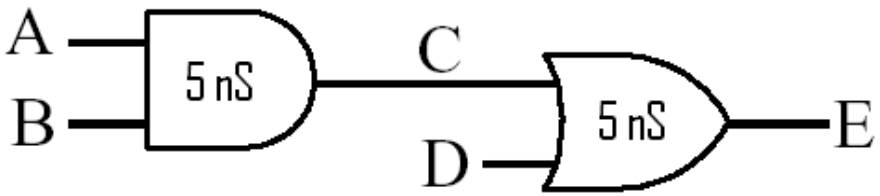
Data Flow Representation Verilog HDL

8421 BCD to Excess3 Code Converter (Mealy FSM)

$$Q_1^+ = Q_2'$$
$$Q_2^+ = Q_1$$
$$Q_3^+ = Q_1Q_2Q_3 + X'Q_1Q_3' + XQ_1'Q_2'$$
$$Z = X'Q_3' + XQ_3$$

```
// bcd to execess 3 converter
// Mealy Implementation
// Data Flow Model
module bcd_ex3_mealy(input X,CLK,output Z);

   reg Q1=0,Q2=0,Q3=0;

   // FF State Update Portion of design
   // Active only on rising edge of clock
   always @(posedge CLK)
      begin
         Q1 <= ~Q2;
         Q2 <= Q1;
         Q3 <= (Q1 & Q2 & Q3) | (~X & Q1 & ~Q3) | (X & ~Q1 & ~Q2);
      end

   // Output Equation -- Continuous Assignment that is
   // a function only of the state variables Q1,Q2,Q3
   assign Z = (~X & ~Q3) | (X & Q3);

endmodule
```

Data Flow Representation



- **If you are using Verilog HDL for simulation, then you can assign gate and wiring type delays:**
  - **Inertial (delays to operations, i.e. gate type delays)**
  - **Transport (delays between operations, i.e. wire delays)**
- **But user-specified delays are ignored if you are using Verilog HDL for synthesis (i.e. creating circuitry)!**

Types of Delay

- Transport: Signal will assume its new value after the specified delay time.

- Inertial: Signal will assume its new value after specified delay time if the signal is in the same state at least as long as the specified propagation delay.

Setting the Timing Scale

- Unless set by the **`timescale** construct units of time in Verilog are generic and measured in ‘time units’
- To associate these generic time-units with physical time use the *timescale* construct
  - `timescale time-unit/time-precision
    - Example `timescale 1ns/100 ps

      Sets the generic time unit specified in the Verilog file to equal 1 ns

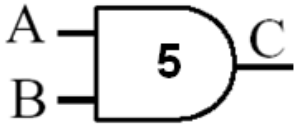
      Sets the precision of time to be 100ps (0.1 ns)

      Thus a time value of 10.52765 ns would be rounded to 10.5 ns

Inertial Delay
(using Continuous Assignment Statements)

- Provides for specification of the propagation delay and minimum input pulse width, i.e. 'inertia' of output:
  - *Continuous Assignment* statement example

```
wire <#delay> [net_name];
        or
assign <#delay> [net] = Boolean Expression;
```



Examples below have the same timing – delaying any change in output ***C*** by 5 time units

```
wire C;
assign #5 C = A & B; //delay in the continuous assignment

wire #5 C = A & B;   //delay in the implicit assignment

wire #5 C;           //delay in the wire declaration
assign C = A & B;
```

Inertial Delay
(using parametrized primitive instantiations)

- By default the timing delay for Verilog gate primitives is always zero.
- Rising and falling delays can be specified during primitive instantiation using the #(rise, fall) delay operator.
- Turn-off delay (transition to high impedance state Z) for tri-state primitives can also be defined by using the #(rise, fall, off) delay operator.

```
and #(10) G1(C,A,B); //rise=10,fall=10
and #(10,11) G2(C,A,B); //rise=10,fall=11
notif0 #(10,11,25) G3(c,d,en) //rise=10,fall=11,
                              //off=25 (when en==0)
```

Inertial Delay

- For each rise, fall, and turn-off delay times:
  - *Minimum*, Typical, and Maximum delay values can be entered using a colon to separate the values as shown below:

```
//min:typ:max values defined for the (rise, fall) delays
and #(6:10:12, 9:11:13) G2(C,A,B);
                or
assign #(6:10:12, 9:11:13) C = A & B;
```

- ModelSim™ the Typical delay values are normally used.

Min/Max/Typical Delay Selection in ModelSim™

- You can explicitly specify which delays are to be used at the start of the simulation.
- To do this utilize the following command line options

  +maxdelays {for utilization of maximum delay values)

  +mindelays {for utilization of minimum delay values)

  +typdelays {for utilization of typical delay values -- the default}

  – These can be placed on the vsim command line or by selecting *max*, *min*, or *typ* under *Delay Selection* pull down widget of the *Verilog* tab of the *Start Simulation Menu.*

Adding Delays to a module

- By default the timing inside a module is controlled by the design of the module itself.
- Many modules are created that have parameterized delays similar to the #(rise,fall) delay operator used with gate primitives.
- This is accomplished using the **parameter** keyword to create delay variables.
- Note that parameters can also be used to change other scalar values in the module.

Parameters

Parameters are run-time constant for storing integers, real numbers, time, delays, or ASCII strings. Parameters may be redefined for each instance of a module.

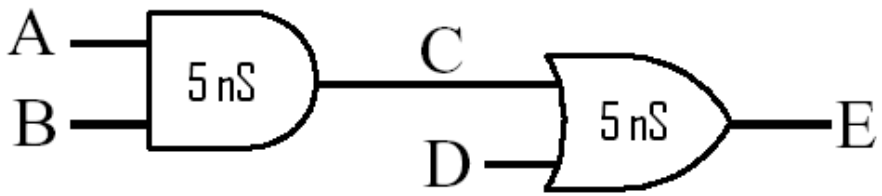
- Syntax parameter declaration
  - **parameter** identifier1 = constant_expression,
    identifier2 = constant_expression, …, ;
- Syntax parameter redefinition
  - ***Explicit***

```
defparam heirarchy_path.parameter_name = value;
```

  - ***Implicit Parameter Redefinition***

```
module_name #(value) instance_name (signals);
```

Changing Timing Parameters of Modules



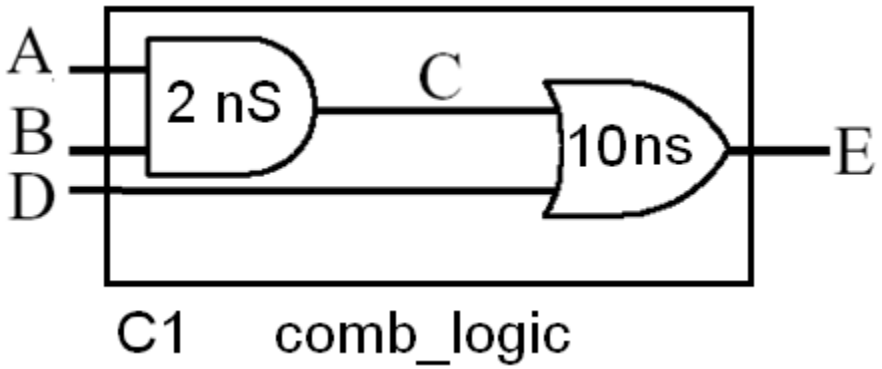
for

`timescale 1ns/100 ps;

```
module comb_logic(input A,B,D, output E);
   parameter and_delay = 5, // default rise/fall delays of 5
                    or_delay = 5;   // for both internal gates
    assign #or_delay    C = A & B;
    assign #and_delay E = C & D;
endmodule
```

- When the module is instantiated then you can choose to override the delay values using the #(f*irst_parameter*, *second_parameter*) notation

Changing Timing Parameters of Modules



for

`timescale 1ns/100 ps;

- For example to specify a 2ns rise/fall for the internal OR gate and a 10ns rise/fall for the internal AND gate you could instantiate the component as follows:

```
comb_logic  #(2,10)  C1(A,B,D,E);
```

Assigning Inertial Delay to Complex Combinational Logic that is Modeled Behaviorally

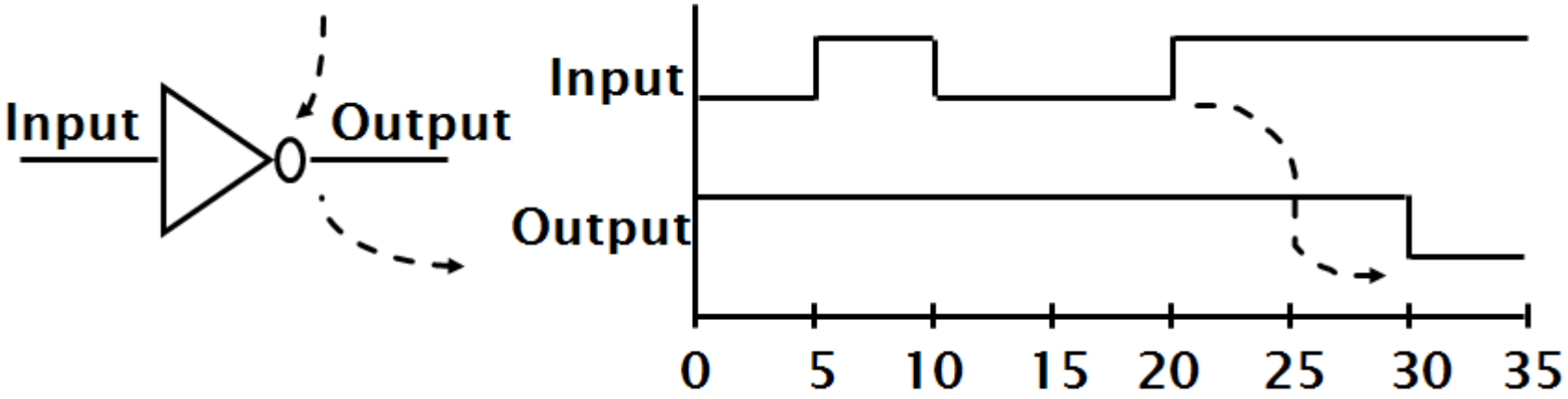
Best Practices:

- When using an **always** block to model complex combinational logic behaviorally
  - Model this logic without assigning delay parameters (i.e. zero delay logic).
  - The outputs from this block should then be passed through as inputs to continuous assignment statements.
  - Assign the appropriate delays to these statements.
- The delay values should then reflect true inertial type delays.

Inertial Delay Example

```
assign #10 Output = ~Input; // using continuous assignments
       or
not #(10) I1(Output,Input);     // using primitives
```

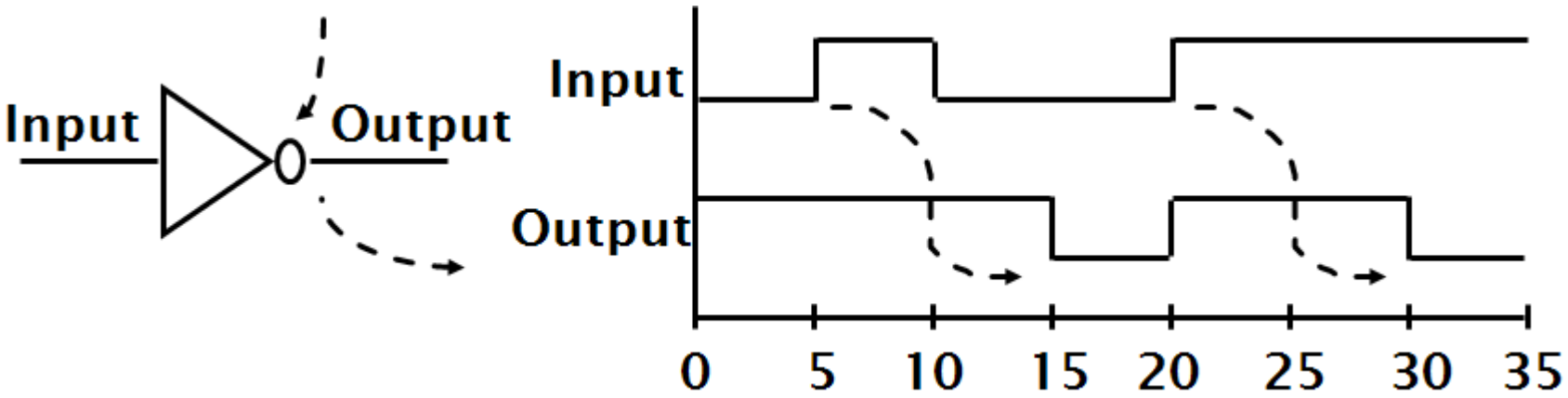
- Example of gate with ‘inertia’ smaller than propagation delay
  - e.g. Inverter with propagation delay of 10ns which suppresses pulses shorter than 5ns



Transport Delay

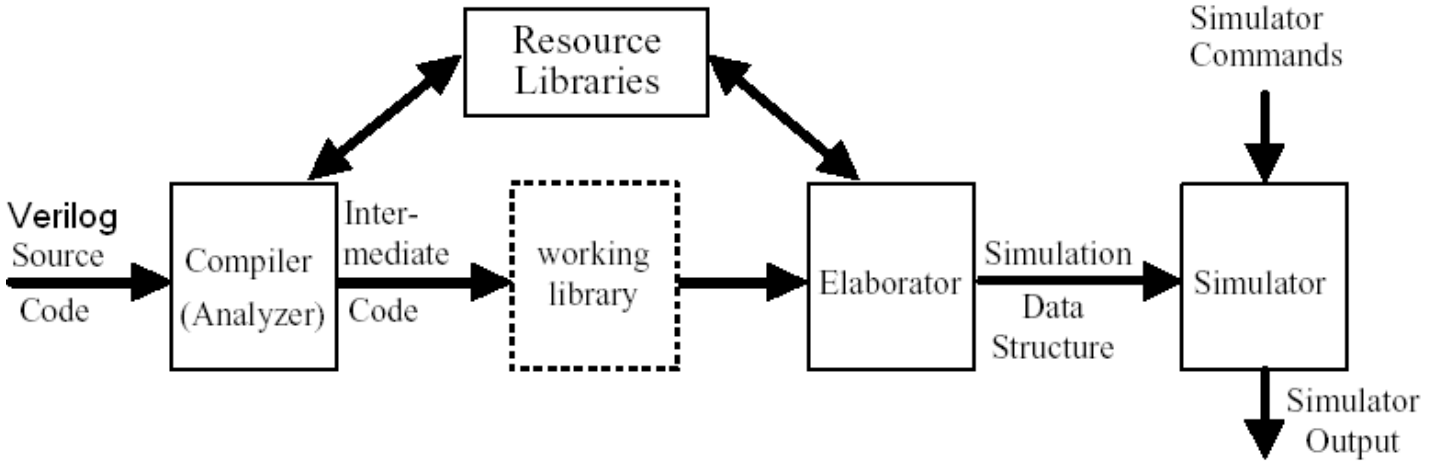
- Best practices for modeling Transport delays in Verilog is to place the delay on the RHS of a non-blocking procedural assignment statement.

```
// Transport delay example
always @(Input)
       Output <= #10 ~Input;
```



Compilation &Simulation of Verilog HDL Models

- Compiler (Semantic Analyzer) – checks the Verilog HDL source model
  - does it conforms with Verilog HDL syntax and semantic rules
  - are references to libraries correct
- Elaboration
  - creates an intermediate form used by a simulator or by a synthesizer
  - create ports, allocate memory storage, create interconnections, ...
  - establish mechanism for executing of Verilog HDL processes



Logic Simulation

- **Discrete Event Simulation** -- where the passage of time is simulated in discrete event driven steps.
  - **Simulation time** – the time value maintained by the simulator to model the actual time it would take for the circuit being simulated
  - Event driven
    - **Update Event** – an actual change in value of a net or variable in the circuit being simulated.
    - **Evaluation Event** – evaluation of a possible future change in value of a net or variable.
  - All process that are sensitive to the update event are evaluated in an arbitrary order.

Simulation

- **Event Queue** –list of pending events that is ordered based upon increasing simulation time that the event is to occur.
  - The Verilog event queue is segmented into five different regions
- **Scheduling an event** – the placement of an event on one of the regions of the event queue.
- **Simulation Cycle** – the processing of all active events.

Verilog Event Queue Regions

- **Active Event Region**: Events that are scheduled to occur at the current simulation time.
  - This is the only region that events can be retired.
  - *Continuous assignment & procedural continuous assignment* constructs add events to this region
- **Inactive Event Region:** Events that occur at the current simulation time but are required to wait until all events in the active region have been processed.
  - When all active events are processed events in this region are placed in active event region.
  - *Blocking assignments* with zero delay are placed here.

Verilog Event Queue Regions

- **Non-blocking Assign Update Region**: Events that were evaluated at some previous simulation time but are scheduled to occur at the current simulation time.
  - Events in this region are processed only after events in the **active** and **inactive** regions have been retired.
- **Monitor Event Region:** Events scheduled for the current time step that are continuously reenabled in every successive time step by the *$monitor* and *$strobe* system tasks.
  - Events in this region are processed only after events in the **active**, **inactive,** and **non-blocking** regions have been retired.

Verilog Event Queue Regions

- **Future Event Region:** Events that occur at some future simulation time.

  Two sub-regions:

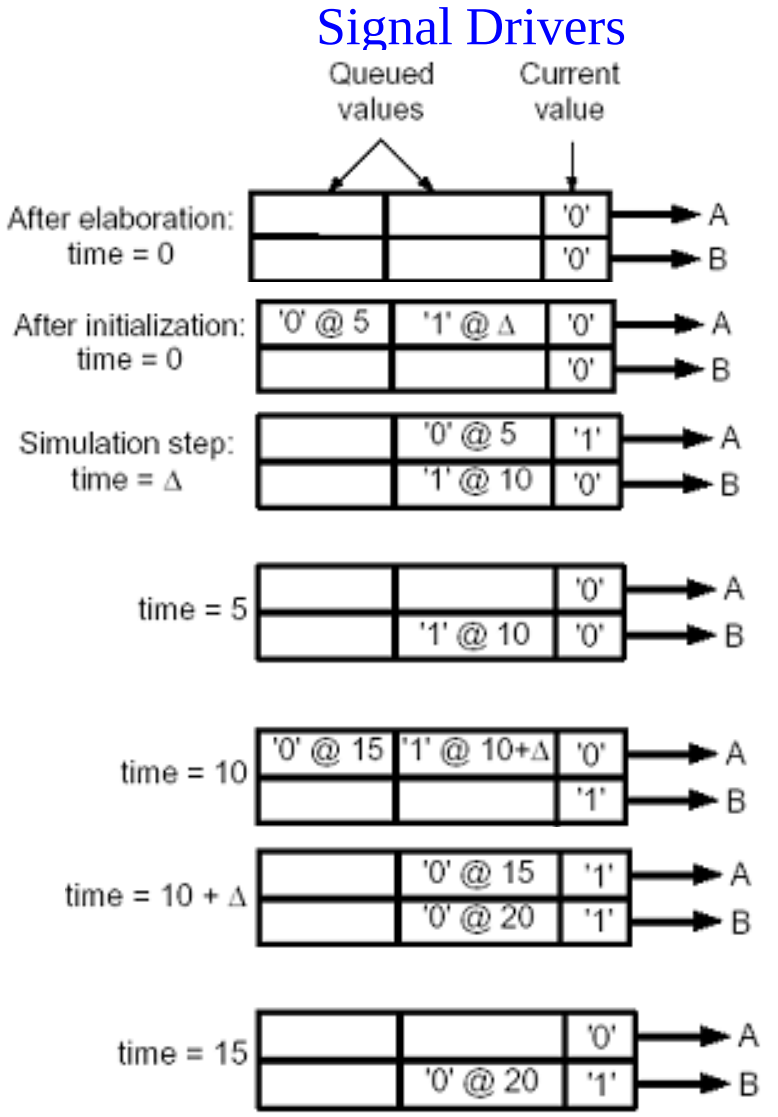
  – **Future Inactive Events**:
    - Events generate by Blocking assignments with nonzero delay

  – **Future Non-blocking Assignment Update Events**:
    - Events generated by Non-blocking assignments with nonzero delay

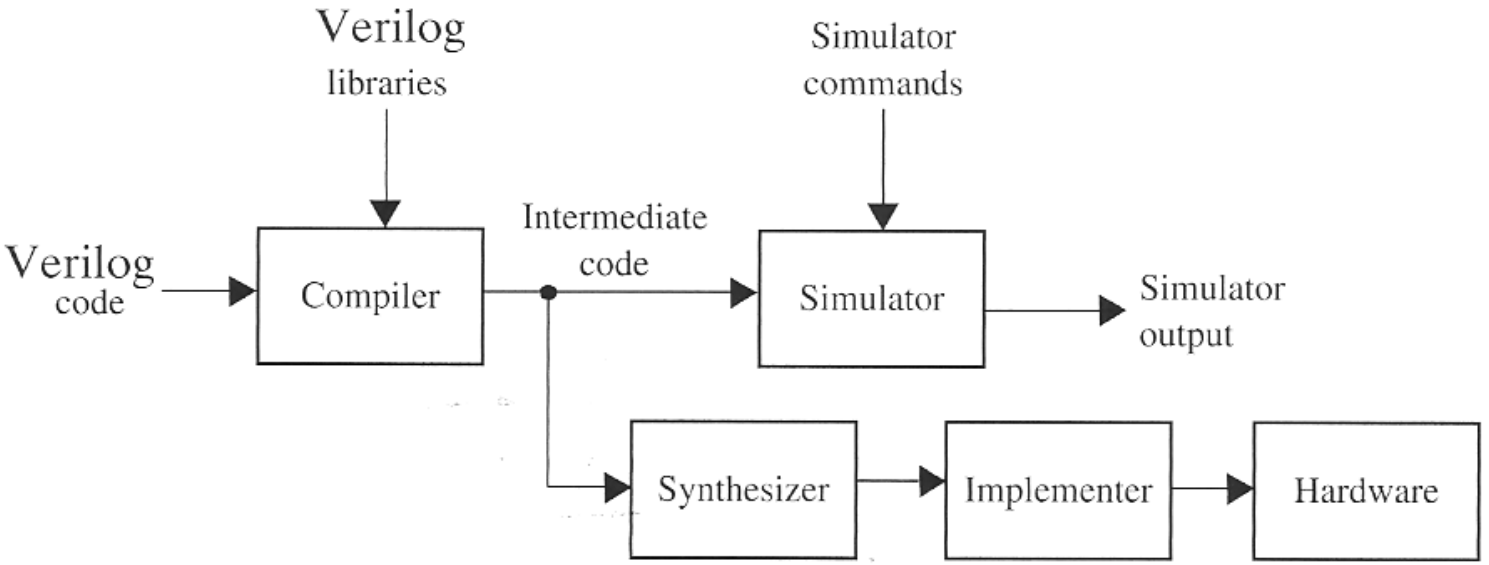
  – Processed after all other regions with events chosen based upon those that have the earliest scheduled simulation time.

Simulation Example

```
`timescale 1ns/100ps

module simulation_example;
   reg A=0,B=0;

   always @ (B)
      begin
         A <= 1;
         A <= #5 0;
      end

   always @ (A)
      if (A) B <= #10 ~B;

endmodule
```

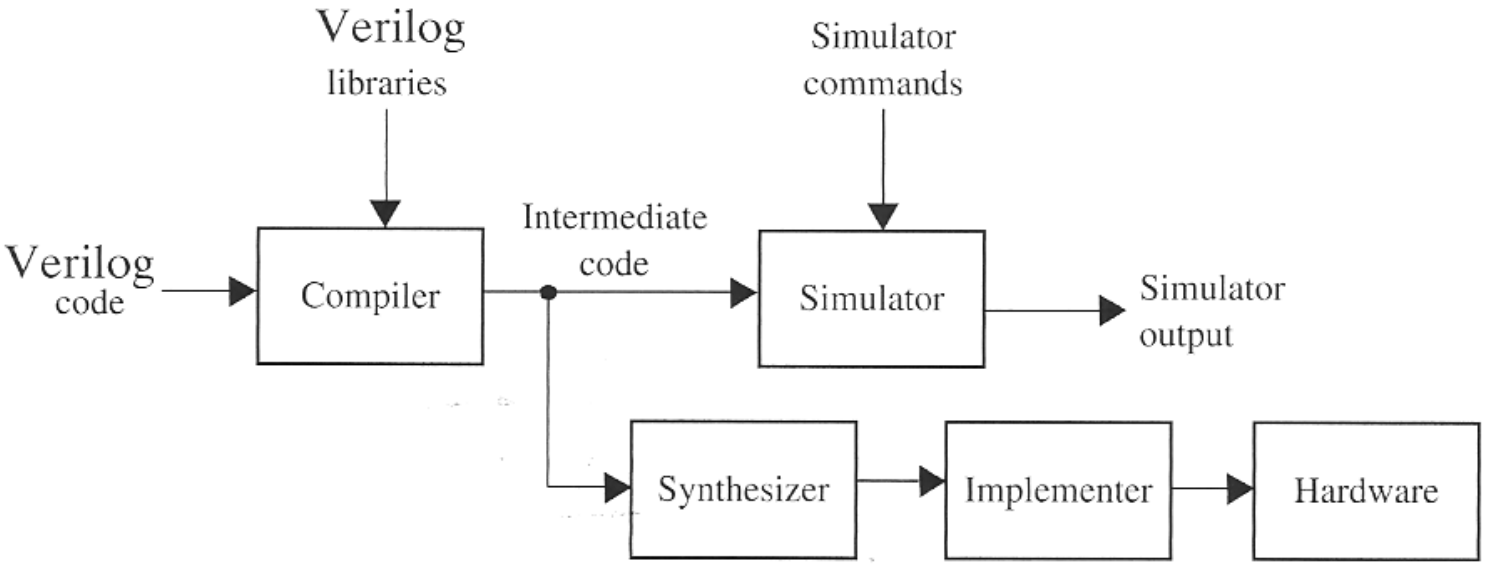


Synthesis of Verilog Code



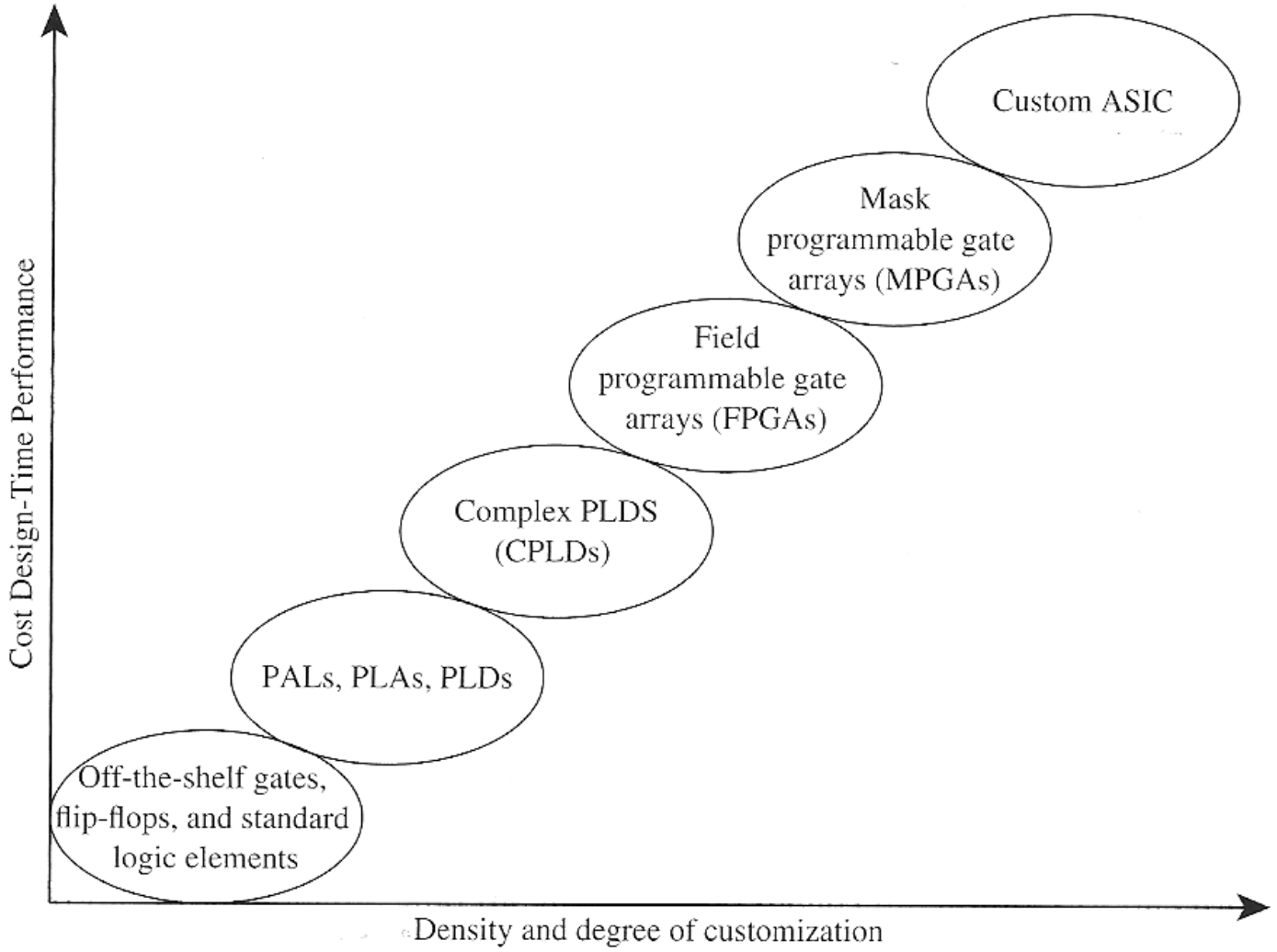
- Synthesis – automatic create hardware from a Verilog Description
  - Intermediate code is created through the same steps of analysis and elaboration used in simulation
  - Often simulation is performed first to catch errors before hardware is created

Synthesis of Verilog Code



- Synthesis
  - Produces a **netlist** that includes a list of required components and their interconnections
  - Output of **synthesizer** is used by the targeted **implementer** module to create the specific CPLD, FPGA, or ASIC hardware.

Spectrum of Possible Technologies for Design Synthesis

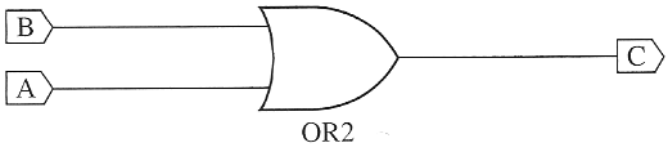


Synthesis Issues and Examples

What logic circuit will be synthesized?

```
module Q1 (input A, B, output reg C);

  always @(A)
     C = #5 A | B;

endmodule
```

Most synthesizers will give a warning that B is not in sensitivity list but synthesize an OR gate.



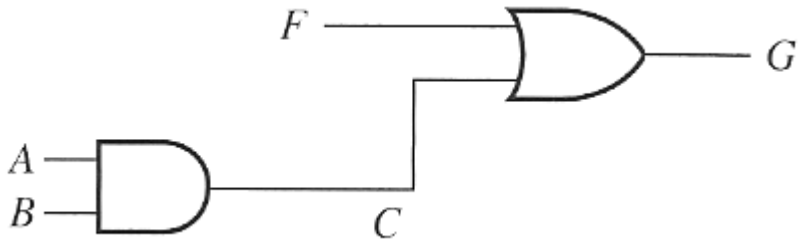
- In this case the synthesizer “guessed” at what the user wanted.
- This means the simulation of the circuit will behave differently from the actual circuit that is synthesized!
  - Check the warnings – synthesizer may have helped or it may have hurt you
- (The synthesizer will also ignore the #5 delay)

Synthesis Issues and Examples

What logic circuit will be synthesized?

```
module Q1 (input A, B, F, CLK, output reg G);

  reg C;

  always @(posedge CLK)
     begin
     C <= A & B;
     G <= C | F;
     end

endmodule
```

Is it?



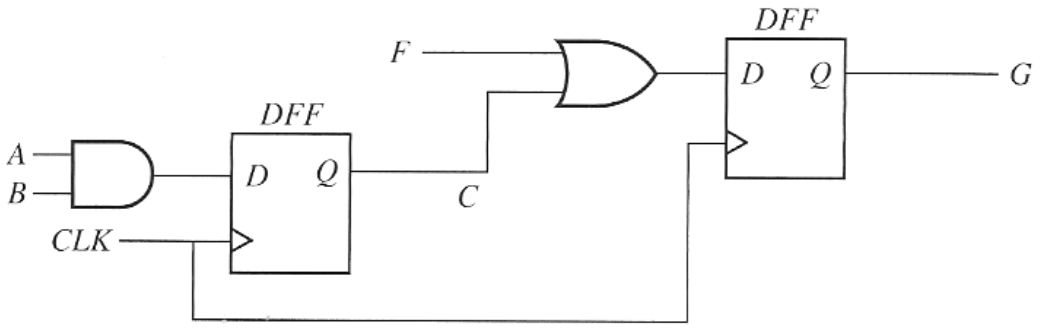
- An edge-triggered clock is implied (by the **posedge** or **negedge**)
- This means LHS variables C and G will need to be remembered after the clocks edge.
- Non-blocking assignments mean that both statements function concurrently

Synthesis Issues and Examples

What logic circuit will be synthesized?

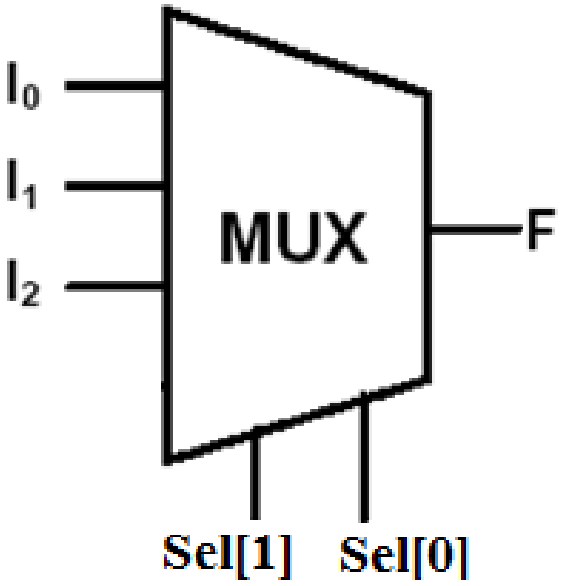
```
module Q1 (input A, B, F, CLK, output reg G);

  reg C;

  always @(posedge CLK)
     begin
     C <= A & B;
     G <= C | F;
     end

endmodule
```

Actual Circuit



Avoiding Unwanted Latches

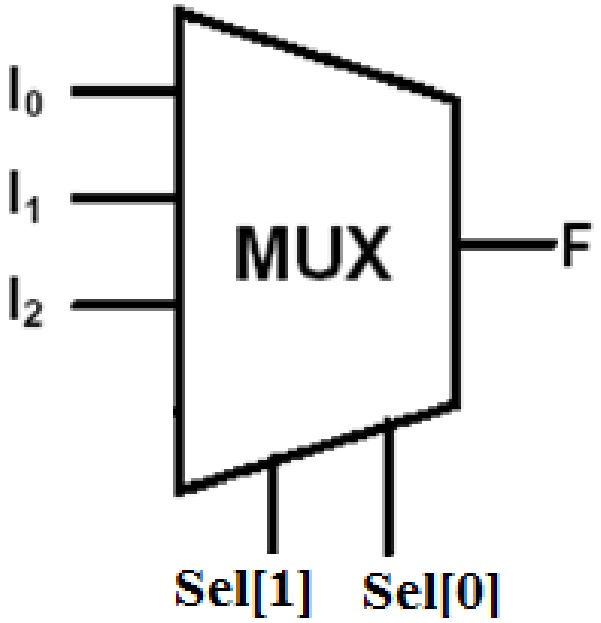
```
module MUX(input [1:0] Sel, I0, I1, I2,output reg F);

  always @ (Sel, I0, I1, I2)
     begin
     if       (Sel == 2'b00) F = I0;
     else if  (Sel == 2'b01) F = I1;
     else if  (Sel == 2'b10) F = I2;
     end

endmodule
```



Unwanted latch created

Avoiding Unwanted Latches

```
module MUX(input [1:0] Sel, I0, I1, I2, output reg F);

  always @ (Sel, I0, I1, I2)
     begin
     if      (Sel == 2'b00) F = I0;
     else if (Sel == 2'b01) F = I1;
     else if (Sel == 2'b10) F = I2;
     else F = 0;
     end

endmodule
```



Avoiding Unwanted Latches

```
module MUX(input [1:0] Sel, I0, I1, I2,output reg F);

  always @ (Sel, I0, I1, I2)
     begin
     F = 0;
     if      (Sel == 2'b00) F = I0;
     else if (Sel == 2'b01) F = I1;
     else if (Sel == 2'b10) F = I2;
     end

endmodule
```