INTRODUCTION TO VERILOG

Verilog modules

```
module FullAdder(X, Y, Cin, Cout, Sum);
output Cout, Sum;

input X, Y, Cin;

  assign #10 Sum = X ^ Y ^ Cin;
  assign #10 Cout = (X && Y) || (X && Cin) || (X && Cin);
endmodule
```

Verilog module for full adder

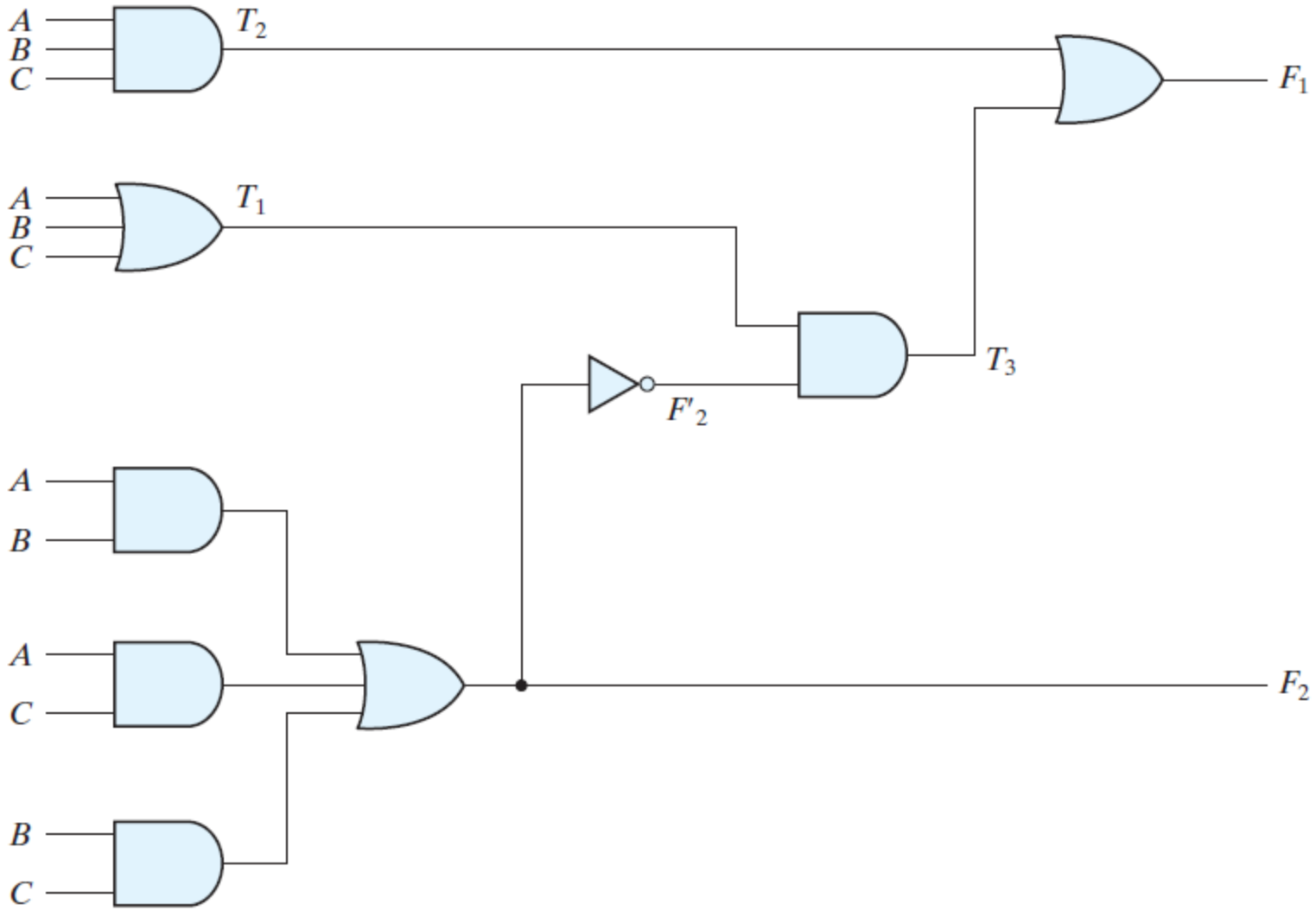
Verilog HDL operators

Symbol	Operation	Symbol	Operation
+	binary addition		
–	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT
= =	equality		
>	greater than		
<	less than		
{}	concatenation		
?:	conditional		

Data flow description of four bit adder

```
module binary_adder (output [3: 0] Sum, output C_out,
input [3: 0] A, B, input C_in);

assign {C_out, Sum}  =  A +  B + C_in;

endmodule
```

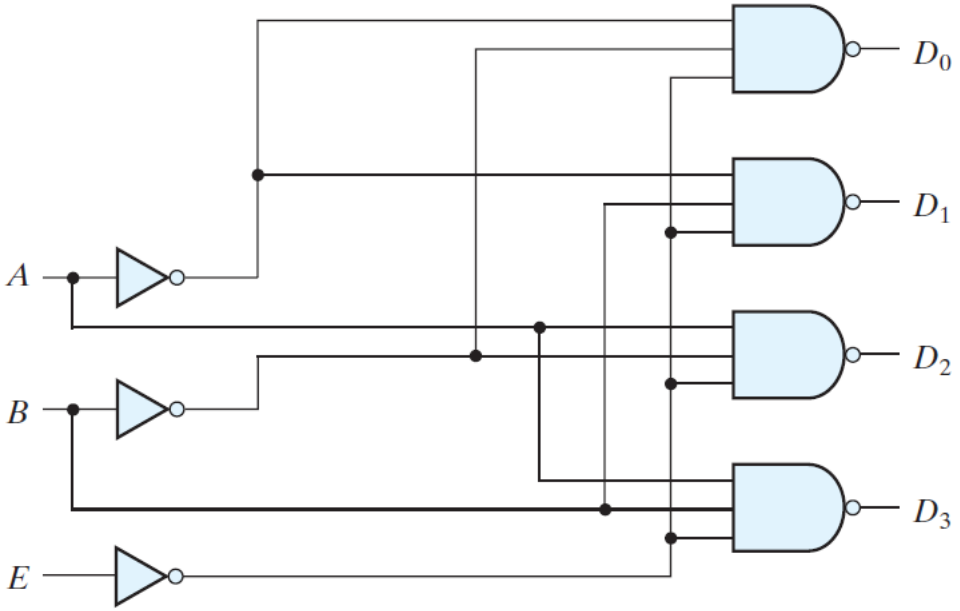
Gate-level modeling example



Gate-level modeling example

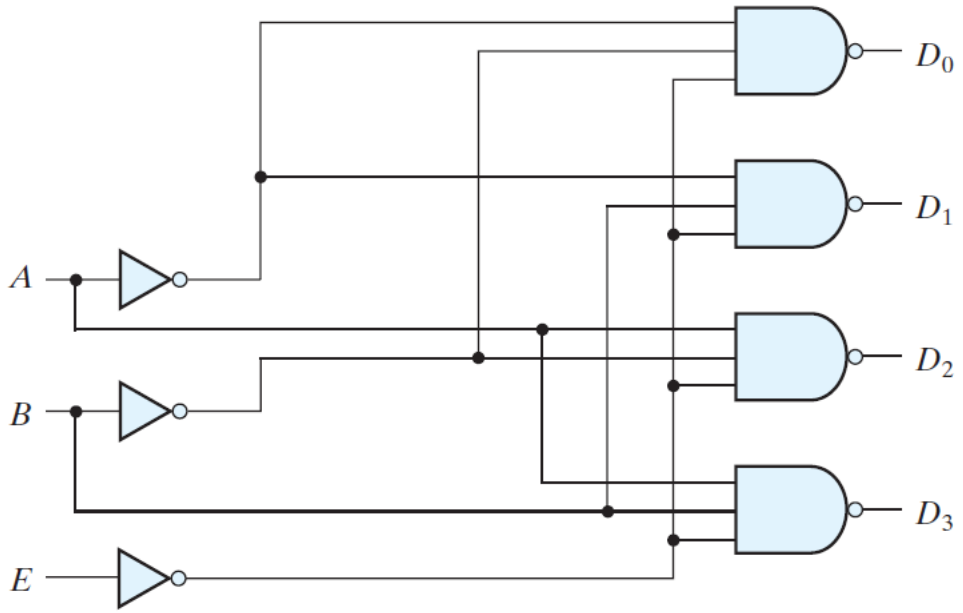
```
// Gate-level description of circuit

module Circuit_1 (A, B, C, F1, F2);

input A, B, C;
output F1, F2;

wire T1, T2, T3, F2_b, E1, E2, E3;

or g1 (T1, A, B, C);
and g2 (T2, A, B, C);
and g3 (E1, A, B);
and g4 (E2, A, C);
and g5 (E3, B, C);
or g6 (F2, E1, E2, E3);
not g7 (F2_b, F2);
and g8 (T3, T1, F2_b);
or g9 (F1, T2, T3);

endmodule
```

2-4 line decoder with enable input



(a) Logic diagram

2-4 line decoder with enable input



(a) Logic diagram

E	A	B	D_0	D_1	D_2	D_3
1	x	x	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

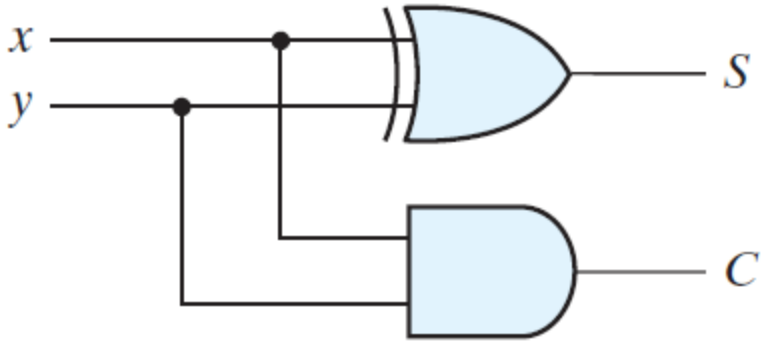
2-4 line decoder with enable input

```
module decoder_2x4_gates (D, A, B, enable);
 output      [0: 3]          D;
 input                       A, B;
 input                       enable;
 wire                        A_not,B_not, enable_not;

 not
   G1  (A_not, A),
   G2  (B_not, B),
   G3  (enable_not, enable);
 nand
   G4  (D[0], A_not, B_not, enable_not),
   G5  (D[1], A_not, B, enable_not),
   G6  (D[2], A, B_not, enable_not),
   G7  (D[3], A, B, enable_not);

endmodule
```

- Output always listed first in the port list of a primitive.

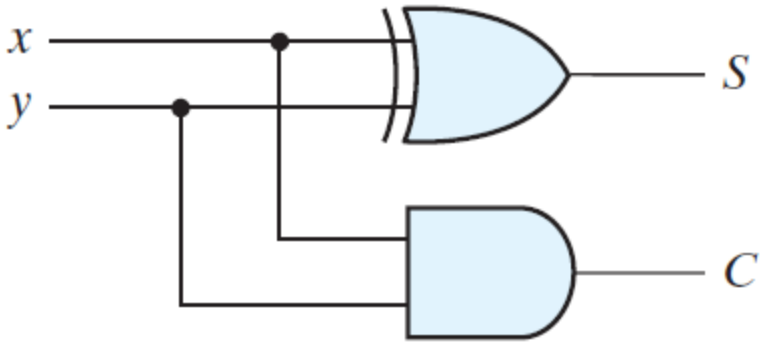
- not and nand written only once - commas at the end of each gates in the series except for the last statement.

Half adder – gate level description



$$S = x \oplus y$$
$$C = xy$$

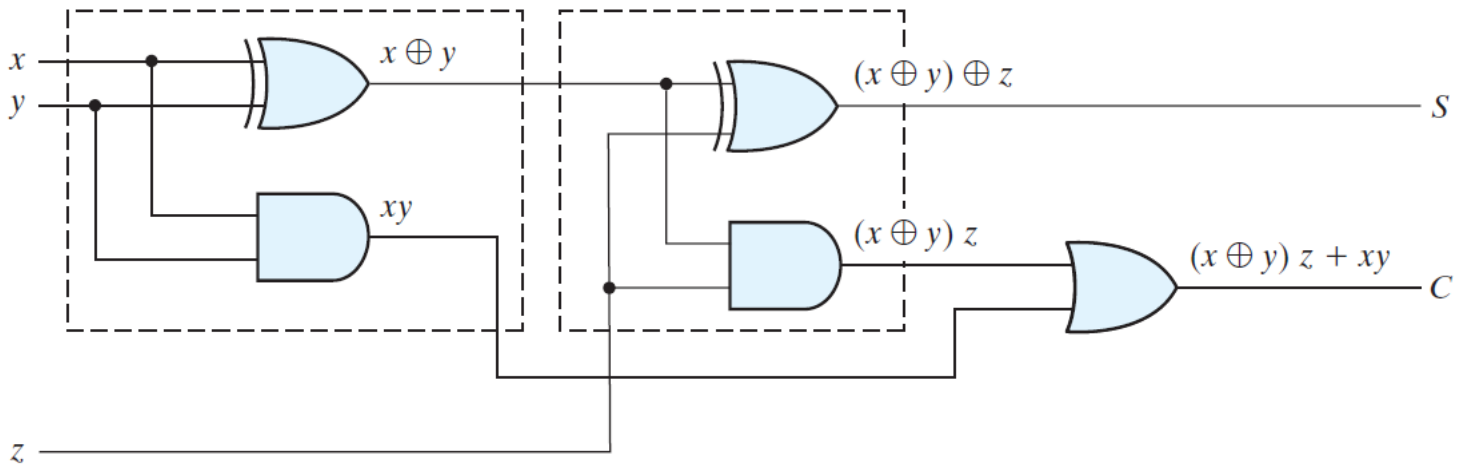
Half adder – gate level description



$S = x \oplus y$
$C = xy$

```
module half_adder (output S, C, input x, y);
// Instantiate primitive gates
  xor (S, x, y);
  and (C, x, y);
endmodule
```

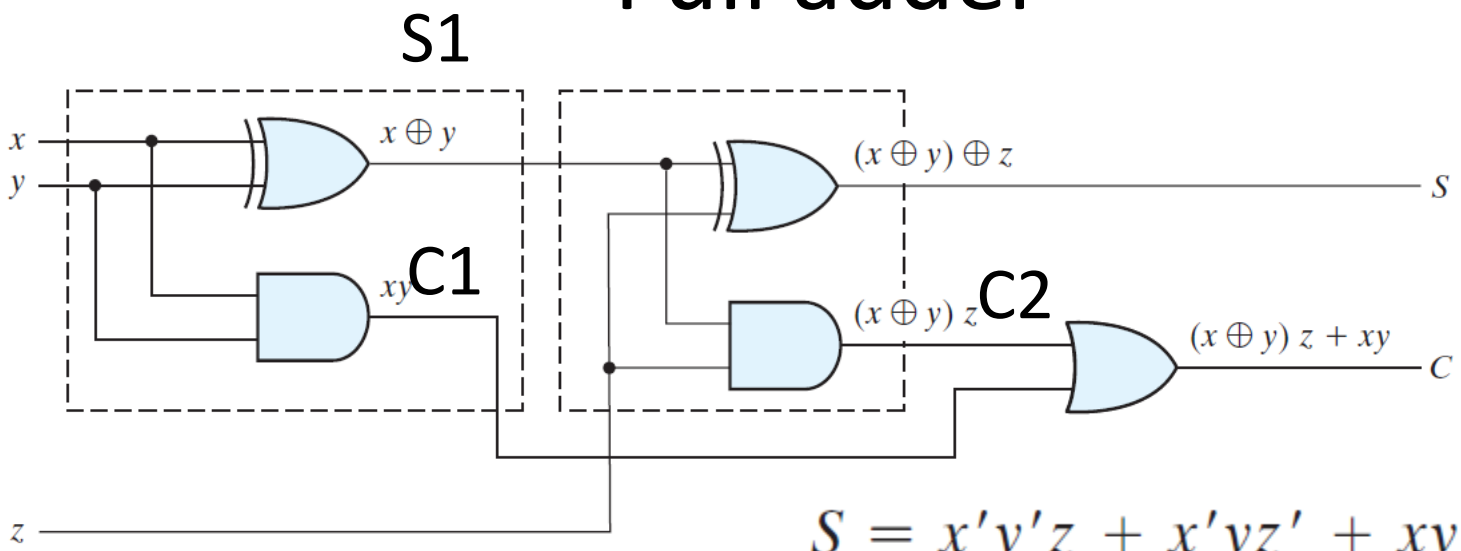
Full adder



$$S = x'y'z + x'yz' + xy'z' + xyz$$

$$C = xy + xz + yz$$

Full adder

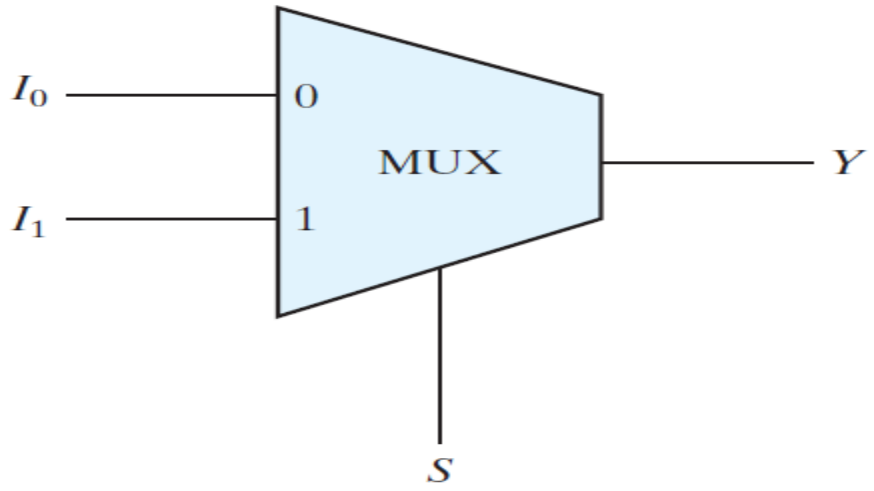


$$S = x'y'z + x'yz' + xy'z' + xyz$$

$$C = xy + xz + yz$$

```
module full_adder (output S, C, input x, y, z);
  wire S1, C1, C2;

// Instantiate half adders
  half_adder HA1 (S1, C1, x, y);
  half_adder HA2 (S, C2, S1, z);
  or G1 (C, C2, C1);
endmodule
```

2x1 multiplexer



specifies the condition that $OUT = A$ if $select = 1$, else $OUT = B$ if $select = 0$.

```
// Dataflow description of two-to-one-line multiplexer

module mux_2x1_df(m_out, A, B, select);
  output      m_out;
  input       A, B;
  input       select;

  assign m_out = (select)? A : B;
endmodule
```

2x1 multiplexer

```
// Behavioral description of two-to-one-line multiplexer

module mux_2x1_beh (m_out, A, B, select);

output m_out;
input A, B, select;
reg m_out;

always @(A or B or select)
if (select == 1) m_out  =  A;
else m_out = B;
endmodule
```

4x1 multiplexer

```
module mux_4x1_beh( output reg m_out, input in_0,
in_1, in_2, in_3, input [1: 0] select);

always @ (in_0, in_1, in_2, in_3, select)

case (select)
2'b00: m_out  in_0;
2'b01: m_out  in_1;
2'b10: m_out  in_2;
2'b11: m_out  in_3;
endcase

endmodule
```

HDL models of sequential circuits

Behavioral models – abstract representations of the functionality of digital hardware.

Description of the circuit behavior – do not specify the internal details of the circuit.

Abstract description – by truth tables, state tables and state diagrams.

Behavior declared by initial keyword – single pass behavior – expires after the associated statement executes.

HDL models of sequential circuits

Always keyword – declares cyclic behavior - executes and re-executes indefinitely until the simulation is stopped.

Module – may contain arbitrary number of initial and always behavioral statements.

Execute concurrently w.r.t each other starting at t = 0, may interact therough common variables.

HDL models of sequential circuits

Two possible ways to provide a free running clock – operates for a specified number of cycles:

```
Initial

 begin

clock = 1'b0;
repeat (30)
#10 clock = ~clock;

end
```

```
initial

begin
clock = 1'b0;
end

initial 300 $ finish ;
always #10 clock = ~clock;
```

HDL models of sequential circuits

Specified loop re-executes 30 times – toggles the value of clock every 10 time units.

15 clock cycles – clock cycle time = 20 time units.

```
Initial

 begin

clock = 1'b0;
repeat (30)
#10 clock = ~clock;

end
```

HDL models of sequential circuits

Second initial statement – declares a stopwatch for the simulation.

Simulation stops unconditionally after 300 time units elapsed.

Provides a clock generator – cycle time of 20 time units.

```
initial

begin
clock = 1'b0;
end
```

```
initial 300 $ finish ;
always #10 clock = ~clock;
```

HDL models of sequential circuits

Another way to describe a free running clock:

initial begin clock = 0; **forever** #10 clock = ~clock; **end**

Executes an indefinite loop.

- delay control operator.
@ - event control operator

always @ (event control expression) **begin**
// Procedural assignment statements that execute when the condition is met
end

HDL models of sequential circuits

always @ (A **or** B **or** C), **always @**(**posedge** clock, **negedge** reset)

Posedge or negedge – accomplish the functionality of an edge triggered device.

HDL models of sequential circuits

```
// Description of D latch
module D_latch (Q, D, enable);

output Q;
input D, enable;
reg Q;

always @ (enable or D)
if (enable) Q <= D;                    // Same as: if (enable == 1)
endmodule
```

HDL models of sequential circuits

```
module D_latch (output reg Q, input enable, D);

always @ (enable, D)

if (enable) Q <= D;          // No action if enable not asserted

endmodule
```

HDL models of sequential circuits

```
// D flip-flop without reset

module D_FF (Q, D, Clk);

output Q;
input D, Clk;
reg Q;

always @ ( posedge Clk)
Q <= D;
endmodule
```

HDL models of sequential circuits

```
// D flip-flop without reset

module D_FF (Q, D, Clk);

output Q;
input D, Clk;
reg Q;

always @ ( posedge Clk)
Q <= #5 D;
endmodule
```

HDL models of sequential circuits

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

always @ ( posedge Clk, negedge rst)

if (!rst) Q <= 1'b0;                    // Same as: if (rst == 0)
else Q <= D;

endmodule
```

- Event expression after the @ symbol in the always statement – may have any number of edge events.

HDL models of sequential circuits

```
// Functional description of JK flip-flop

module JK_FF ( input J, K, Clk, output reg Q, output Q_b);

assign Q_b = ~ Q ;

always @ ( posedge Clk)
case ({J,K})
2'b00: Q <= Q;
2'b01: Q <= 1'b0;
2'b10: Q <= 1'b1;
2'b11: Q <= !Q;
endcase
endmodule
```

HDL models of sequential circuits

Case multi-way branch condition

Checks for 2 bit binary number – obtained by concatenating J and K bits.

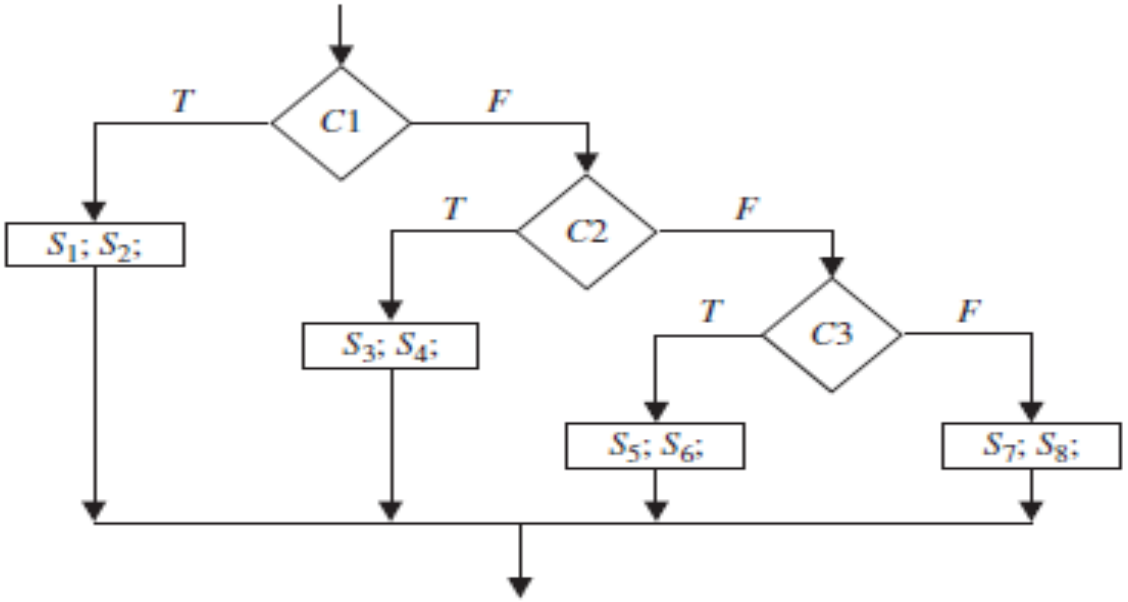
HDL models of sequential circuits

```
if (condition)
   sequential statements1
else
   sequential statements2
```

Form of the basic if statements.

```
if (condition)
  sequential statements
  // 0 or more else if clauses may be included
else if (condition)
  sequential statements}
[else  sequential statements]
```

HDL models of sequential circuits




```
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
```

Equivalent representation of a flowchart using nested ifs and else-ifs.

HDL models of sequential circuits

- Always block using event controlled statements
- Alternative form of always block uses wait or event controlled statements instead of a sensitivity list.
- Sensitivity list can be omitted at the always keyword – delays or time controlled events must be specified inside the always block.
- Always block cannot have both sensitivity list and wait statements.

HDL models of sequential circuits

```
always
begin
#10 clk <= ~clk;
end
```

```
always
begin
   rst = 1; // sequential statements
   @(posedge CLK); //wait until posedge CLK
   // more sequential statements
end
```

Delays in Verilog

- assign #5 D = A && B;
- Models an AND gate with a propagation delay of 5 ns.
- If suppose, the inputs change very often in comparison to the gate delay (1 ns to 4 ns) – simulation output will not show changes.
- Two models of delay in Verilog
  - Inertial delay
  - Transport delay.

Delays in Verilog

- Inertial delay for combinational blocks

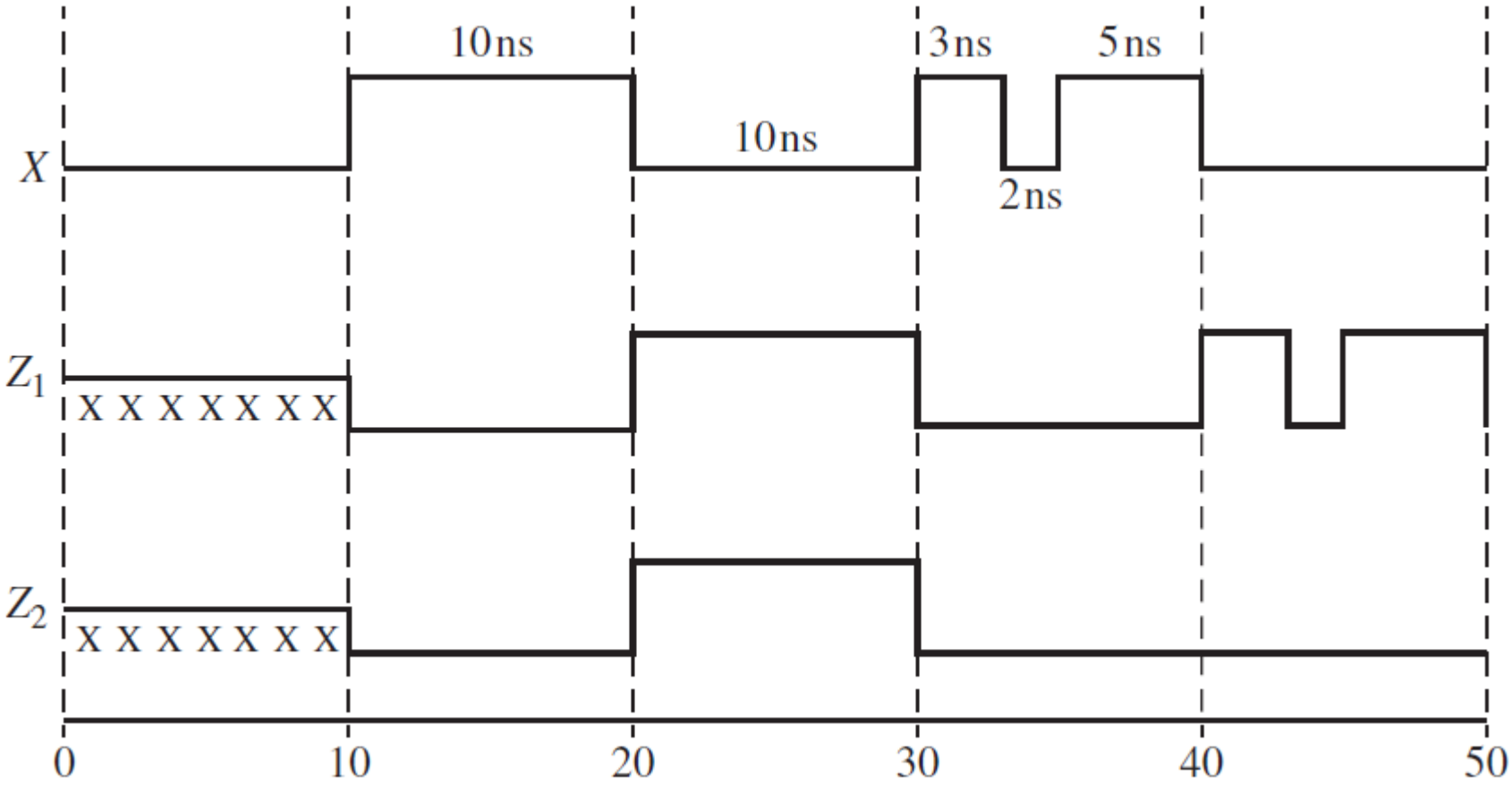
```
// explicit continuous assignment
wire D;
assign #5 D = A && B;

// implicit continuous assignment
wire #5 D = A && B;
```

- Inertial delay – intended to model gates and other devices that do not propogate short pulses from the input to the output.

Delays in Verilog

- Transport delay – intended to model the delay introduced by wiring.
- Simply delays an input signal by the specified delay time.
- Delay value specified in the right hand side of the statement.

Delays in Verilog



Delays in Verilog

```
always @ (X)
begin
  Z1 <= #10 (X);      // transport delay
end
assign #10 Z2 = X;   // inertial delay
```

- Inertial delay – any pulse which has a pulse width less than 10 ns will not be propagated to the output.
- In transport delay – all the pulses has been propagated.
- Expression on the RHS evaluated – but not assigned to Z1 until the delay has elapsed.

Delays in Verilog

- Expression on the RHS evaluated – but not assigned to Z1 until the delay has elapsed. – delayed assignment.

$$Z_1 <= \#10\ X;$$

$$\#10\ Z_1 <= X;$$

Delay of 10ns elapses first, then the expression evaluated – delayed evaluation

```
assign a = #10 b;
```

Illegal - placement of delay on rhs with continuous assignment.

Delays in Verilog

Net delay

```
wire C1;
wire #10 C2; // net delay on wire C2

assign #30 C1 = A || B;   // statement 1 - inertial delay
assign #20 C2 = A || B;   // statement 2 - inertial delay
                             will be
                          // added to net delay before being
                          // assigned to wire C2
```

- Net delay – refers to the time it takes from any driver on the net to change value to the time when the net value is updated and propagated further.

- After statement 2 processes its delay of 20 ns, net delay of 10 ns added to it.

Hierarchical modeling concepts

Components of a simulation.

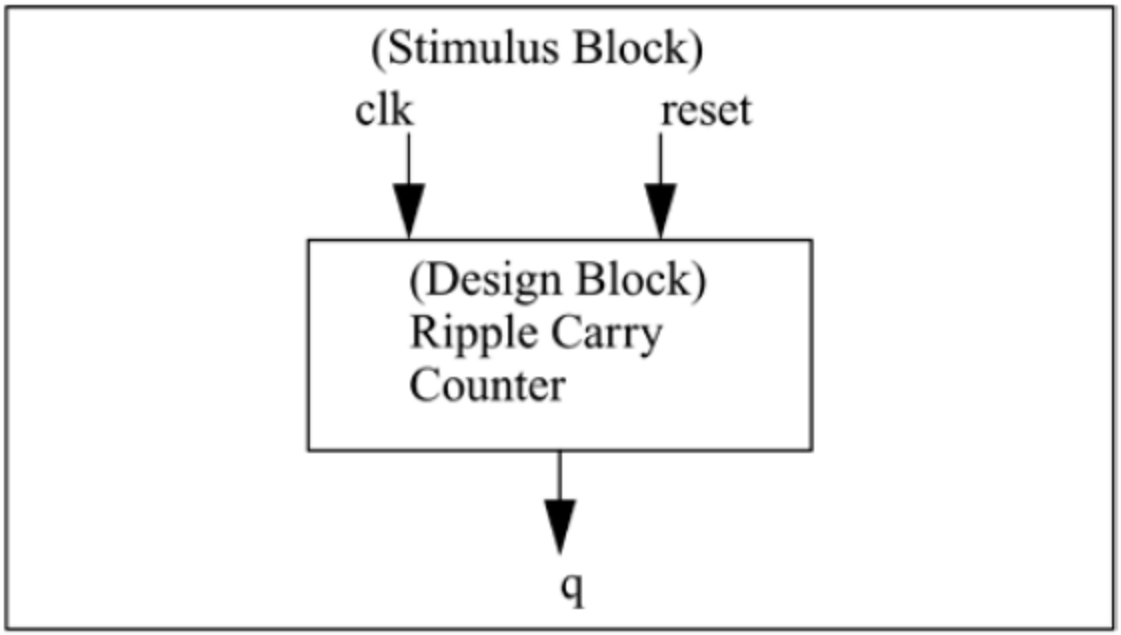
Test the functionality of the design block – applying stimulus and checking results.

Stimuls block – commonly called the test bench.

Stimulus and design block are kept seperate. – good practice in the design.

Hierarchical modeling concepts

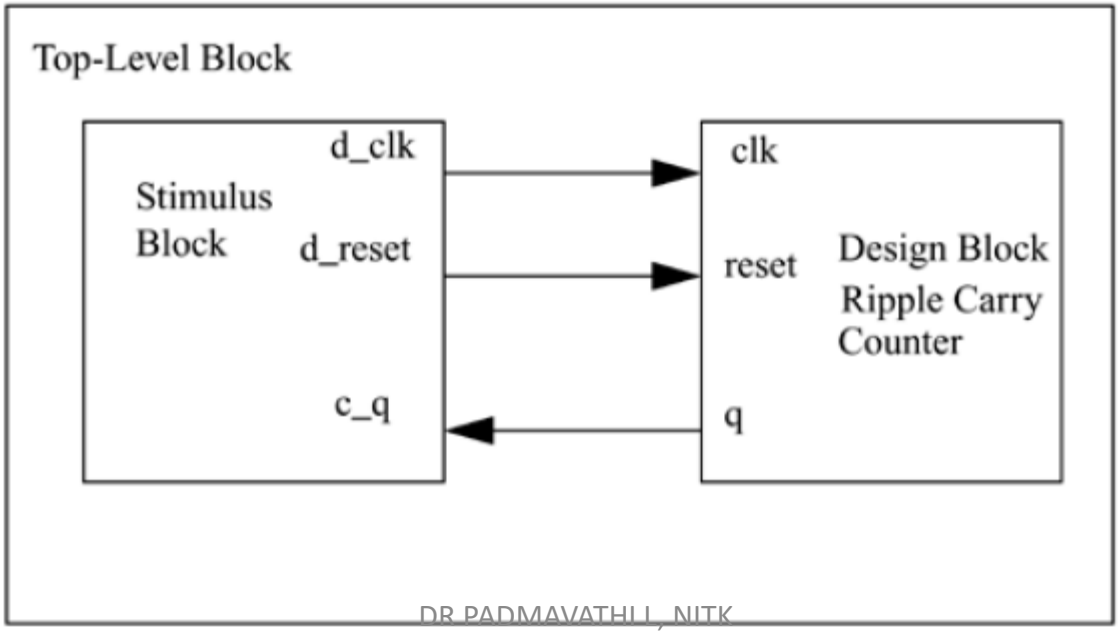
Stimulus block instantiating design block.



Hierarchical modeling concepts

Instantiating both the stimulus and design blocks in a top-level dummy module.

Stimulus block interacts with the design block through interface.
Top level block simply instantiates the design and stimulu block.



Writing test bench

Test bench for describing and applying stimuls to an HDL model of the circuit – test and observe its response during smulation.

Test bencehes – van be quiet complex, lengthy.

Care must be taken to –write stimuli that will test a circuit thoroughly

Exercise all the operating features that are specified.

Writing test bench

```
initial
begin
A  0; B  0;
#10 A  1;
#20 A  0; B  1;
End

initial
begin
D  3'b000;
repeat (7)
#10 D  D  3'b001;
end
```

Stimulus test module

module test_module_name;
// Declare local **reg** and **wire** identifiers.
// Instantiate the design module under test.
// Specify a stopwatch, using $finish to terminate the simulation.
// Generate stimulus, using **initial** and **always** statements.
// Display the output response (text or graphics (or both)).
endmodule

Stimulus test module

Test module like any other module – but typically has no inputs or outputs.

Local reg data types, local reg wire types.

Instantiate the module under test – using local identifiers in the port list.

Simulator associates the local identifiers withihn the test bench with the formal identifiers of the module.

Response to the stimuls – appear in text fomat as standard output and as waveforms – timing diagrams.

Stimulus test module

Numerical outputs displayed by system tasks

Built in system functions – recognised by keywords that begin with $ symbol.

Some system tasks useful for display:
$display —display a one-time value of variables or strings with an end-of-line return,
$write —same as **$display ,** but without going to next line,
$monitor —display variables whenever a value changes during a simulation run,
$time —display the simulation time,
$finish —terminate the simulation.

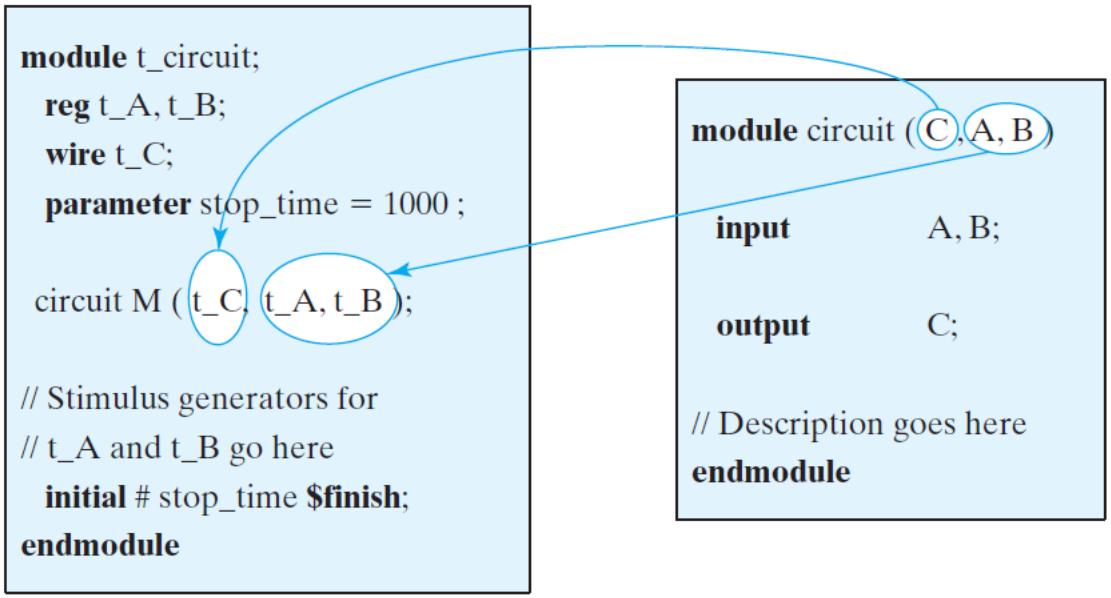
Stimulus test module

$display ("%d %b %b", C, A, B);

base may be binary, decimal, hexadecimal, or octal - identified with the symbols %b, %d, %h, and %o. (%B, %D, %H, and %O are valid too).

No commas in the format specification – argument list seperated by commas.

Stimulus test module



Interaction between stimulus and design modules

Stimulus test module

```
// Dataflow description of two-to-one-line multiplexer

module mux_2x1_df(m_out, A, B, select);

output m_out;
input A, B;
input select;

assign m_out  (select)? A : B;

endmodule
```

Stimulus test module

```
// Test bench with stimulus for mux_2x1_df

module t_mux_2x1_df;

wire t_mux_out;
reg t_A, t_B;
reg t_select;
parameter stop_time  = 50;

mux_2x1_df M1 (t_mux_out, t_A, t_B, t_select);
                        // Instantiation of circuit to be tested

initial # stop_time $finish;
```

Stimulus test module

```
initial begin                                   // Stimulus generator

t_select = 1; t_A = 0; t_B = 1;
#10 t_A  = 1; t_B  = 0;
#10 t_select = 0;
#10 t_A = 0; t_B = 1;

end

initial begin                                   // Response monitor
// $display (“ time Select A B m_out “);
// $monitor ( $time ,,  “%b %b %b %b” , t_select, t_A, t_B,
t_m_out);
```

Stimulus test module

$monitor (“time = “ , **$time** ,, “select = %b A = %b B = %b OUT = %b” , t_select, t_A, t_B, t_mux_out);
end
endmodule

Stimulus test module

Two types of verification – functional and timing verification.

Functional verification – study the circuit logical operation independent of timiing considerations.

Timing verification – study the circuit operation, including the effect of delays through the gates.