INTRODUCTION TO VERILOG

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;
```

- Any change in A and B will result in a delay of 5 ns – before the change in output visible.

Delays in Verilog

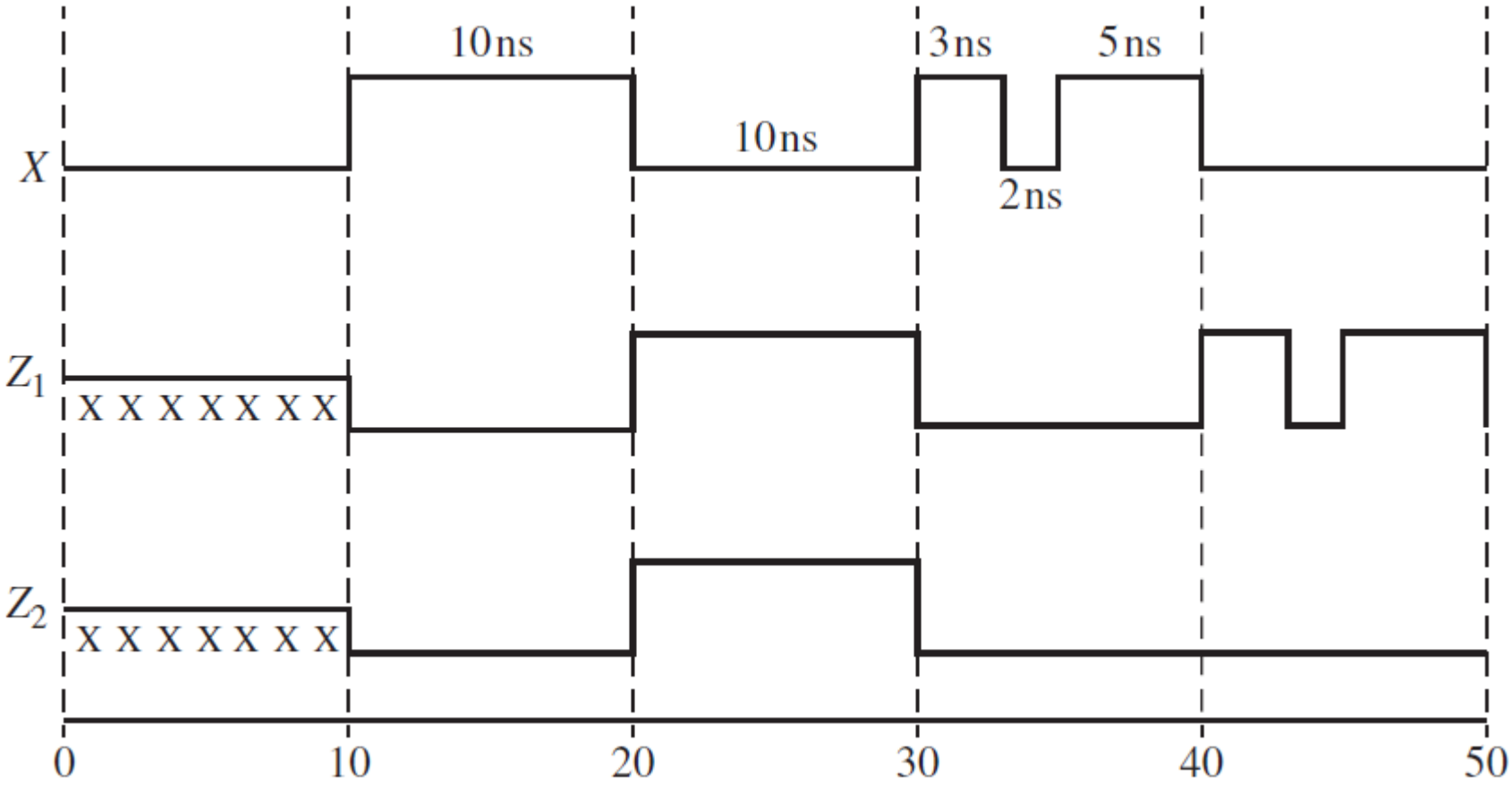
- Inertial delay – intended to model gates and other devices that do not propogate short pulses from the input to the output.
- Transport delay – intended to model the delay introduced by wiring.
- Simply delays an input signal by the specified delay time.
- Delay value specified in the right hand side of the statement.

Delays in Verilog

```
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.

Delays in Verilog



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
```

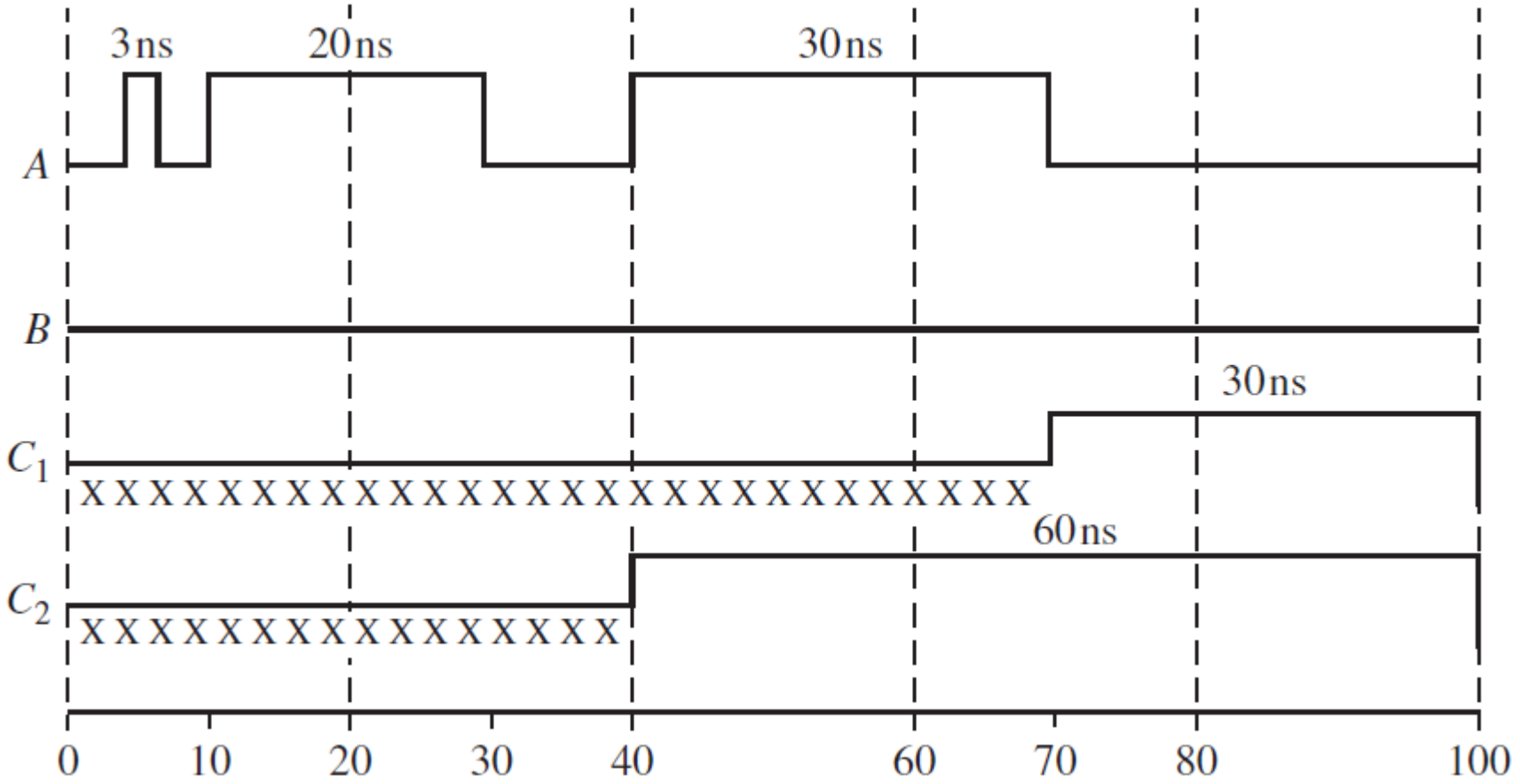
Delays in Verilog

- 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.

Delays in Verilog

Example of Net delays

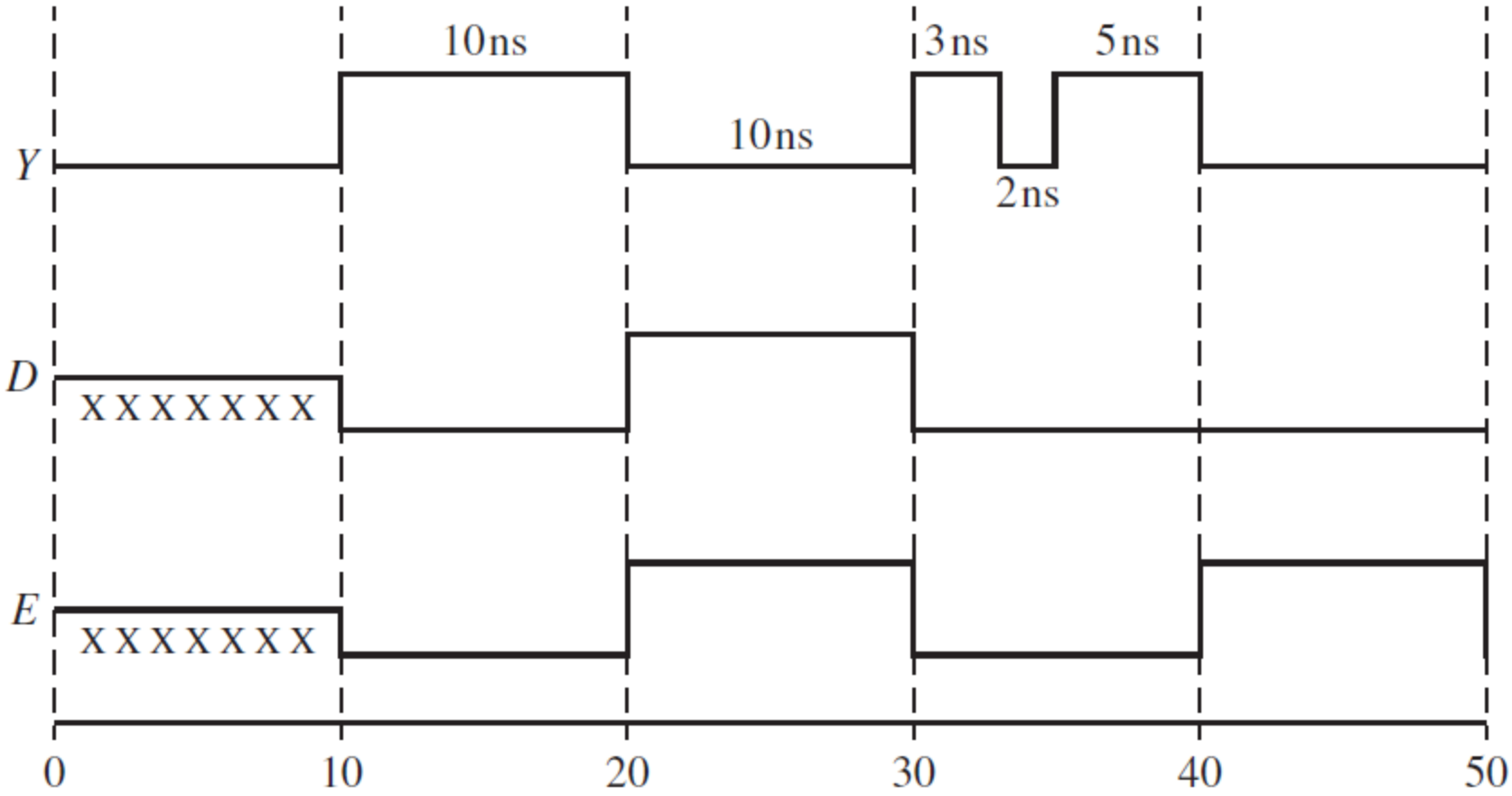


Delays in Verilog

C1 rejects narrow pulses less than 30 ns, C2 rejects only pulses less than 20 ns.

```
wire #3 D; // net delay on wire D
assign #7 D = Y; // statement 1 – inertial delay

wire #7 E; // net delay on wire E
assign #3 E = Y; // statement 1 – inertial delay
```

Delays in Verilog

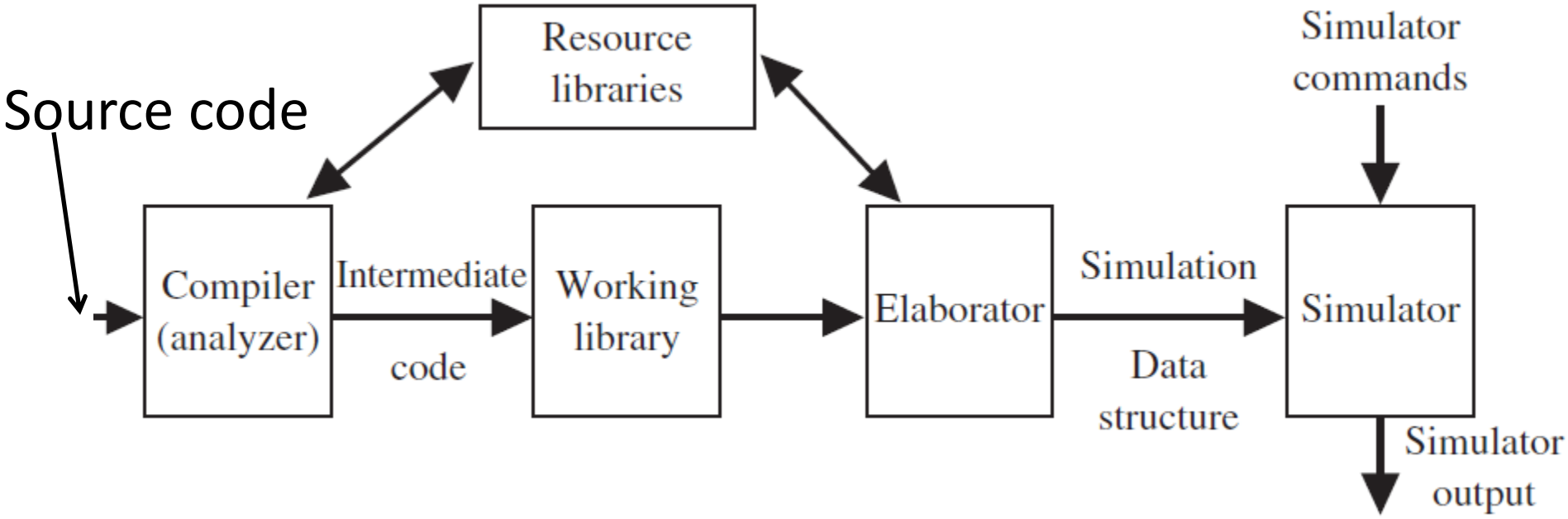


Delays in Verilog

- Understand the pulse rejection associated with inertial delays – improve your initial experience with Verilog – understand the output changes.

- Test sequence applied – wider than the inertial delays of the modeled devices.

Compilation, simulation and synthesis of Verilog code



- Instantiate the modules, link it to the modules defined, parameters propagated among various modules – elaboration.

Compilation, simulation and synthesis of Verilog code

- Instantiate the modules, link it to the modules defined, parameters propagated among various modules – elaboration.
- Driver – created for each signal – each driver holds the current value of a signal, queue of future signals.
- Each time a signal is scheduled to change in the future – new value placed in the queue along with the time at which the change is scheduled.
- Mechanism established for executing the Verilog statements in the proper sequence

Compilation, simulation and synthesis of Verilog code

- Simulation – discrete event simulation.
- Time unit discretised.
- Initilaization phase – initial value to the signal
- Specify initial values in Verilog – to facilitate correct initialization.
- Every change in value of a net or a variable in the circuit being simulated – update event.

- Update event executed – processes sensitive to that

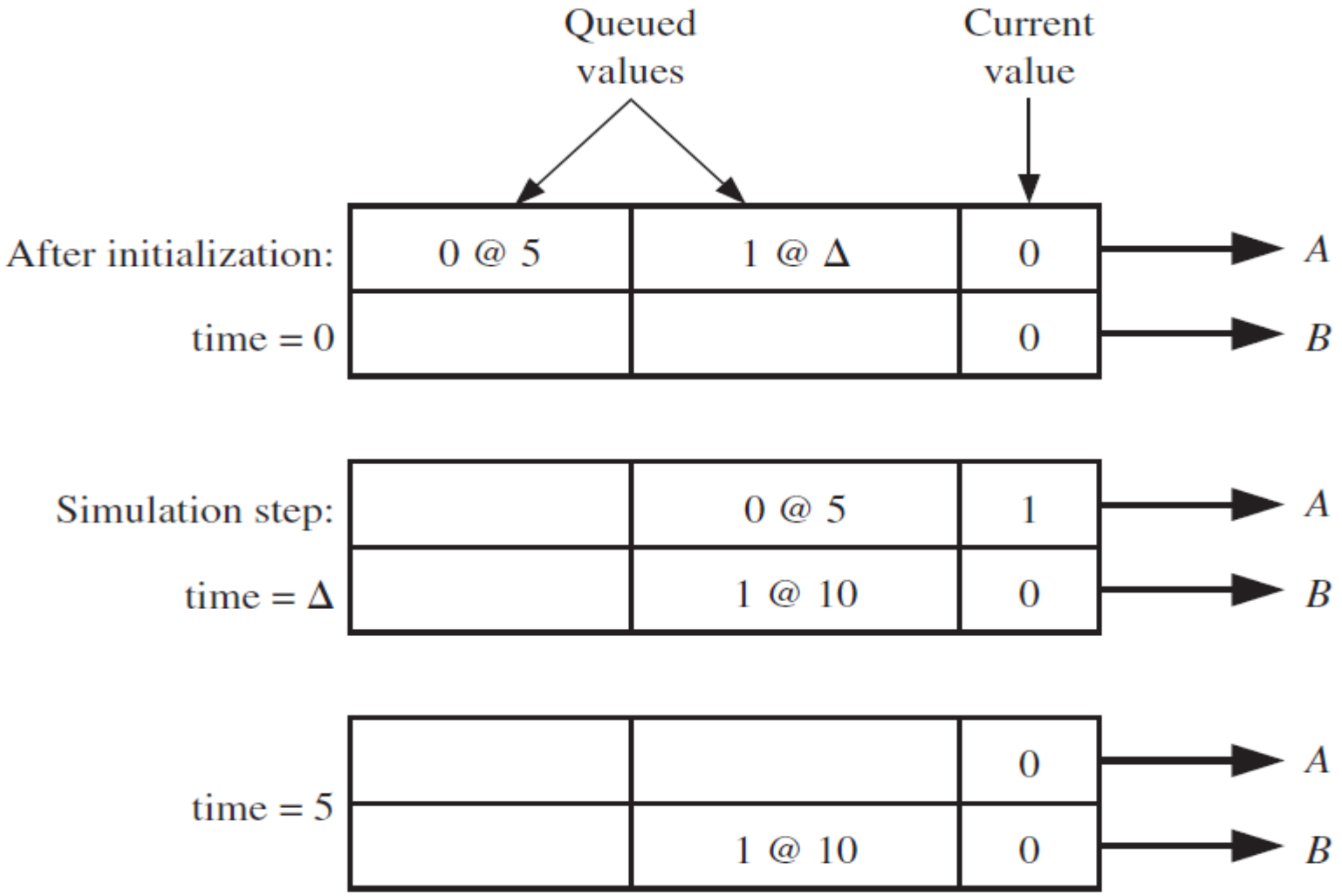
Compilation, simulation and synthesis of Verilog code

- To keep track of events and events processed in the correct order – managed by event queue.
- Model with more than one process – concurrent execution of all processes.
- Concurrent statements outside always block – concurrent execution.
- Statement inside always block – execute sequentially.

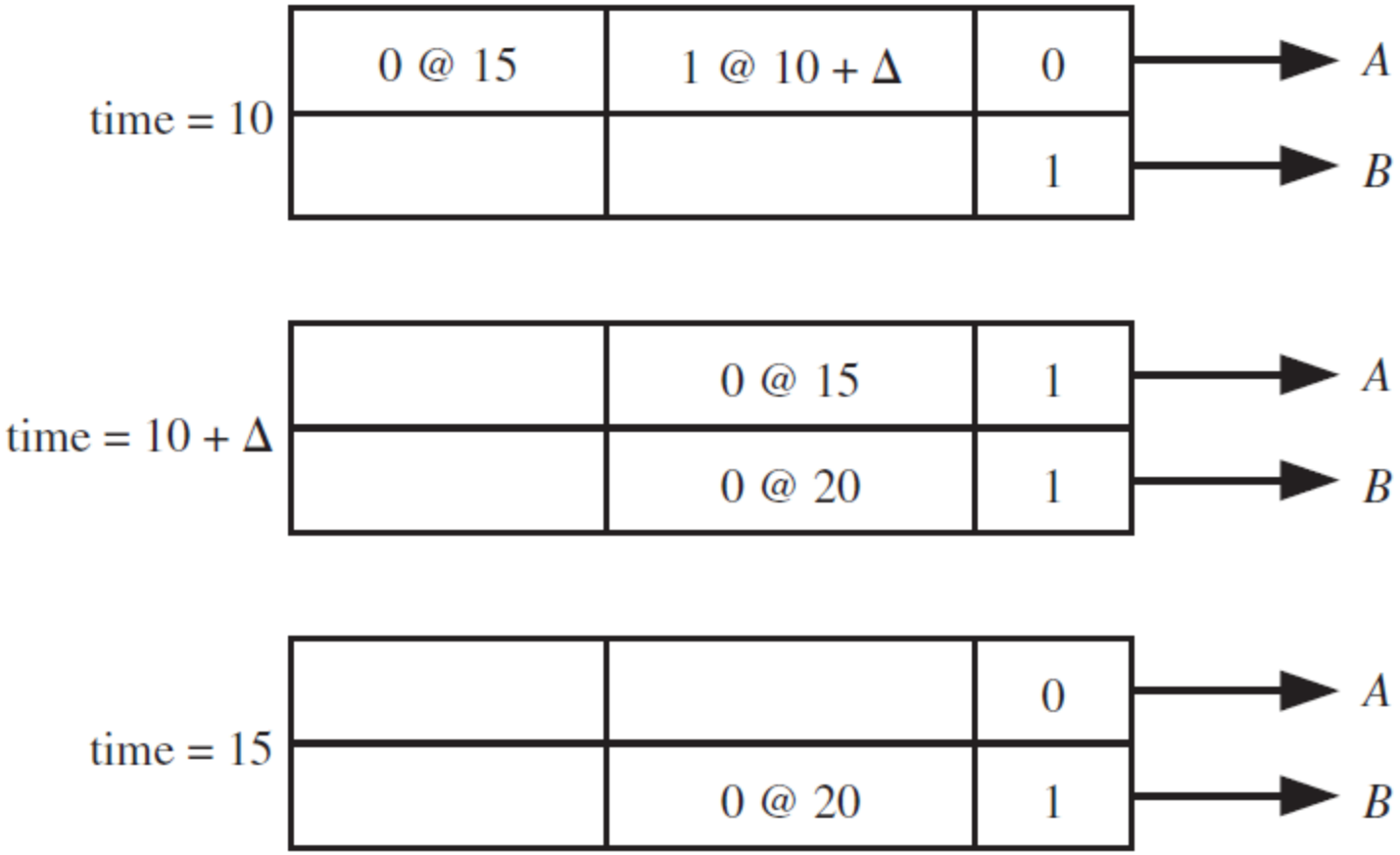
Compilation, simulation and synthesis of Verilog code

```
module twoprocess

reg A,B;

initial
begin
A = 0;
B = 0;
End

// process P1
always @(B)
begin
A <= 1;
A <= #5 0;
end

// process P2
always @(A)
begin
if (A)
B <= #10 ~B;
end
```

Compilation, simulation and synthesis of Verilog code



Compilation, simulation and synthesis of Verilog code



Compilation, simulation and synthesis of Verilog code

Event driven simulation.

Change in signal – referred as an event.

Each time an event occurs – processes waiting on the event executed resulting signal change queued up to occur at future time.

All active processes finished executing – simulation time advanced to the time of next scheduled event.

Continues – either no more events scheduled, simulation time reached.

Compilation, simulation and synthesis of Verilog code

Inertial delay – each input change causes the simulator to schedule a change – scheduled to occur after the specified delay.

Another input change happens before the specified delay has elapsed.

First change de-queued from the simulation driver queue.

Pulses wider than the specified delay – appear at the output.

Verilog Data Types and Operators

Bitwise and Logical Operators:

Operator type	Operator symbols	Operation performed
Bitwise	~	Bitwise NOT (1's complement)
	&	Bitwise AND
	|	Bitwise OR
	^	Bitwise XOR
	~^ or ^~	Bitwise XNOR
Logical	!	NOT
	&&	AND
	||	OR

Verilog Data Types and Operators

Reduction and shift operators:

Operator type	Operator symbols	Operation performed
Reduction	&	Reduction AND
	~&	Reduction NAND
	|	Reduction OR
	~|	Reduction NOR
	^	Reduction XOR
	~^ or ^~	Reduction XNOR
Shift	>>	Logical right shift
	<<	Logical left shift
	>>>	Arithmetic right shift
	<<<	Arithmetic left shift

Verilog Data Types and Operators

Relational, logical and bitwise operators:

Operator type	Operator symbols	Operation performed
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal to
	<=	Less than or equal to
Logical and bitwise	==	Logical equality
	!=	Logical inequality
	===	Case equality
	!==	Case inequality

Verilog Data Types and Operators

Arithmetic, concatenation, replication, conditional operators:

Operator type	Operator symbols	Operation performed
Arithmetic	+	Addition
	-	Subtraction
	-	2's complement
	*	Multiplication
	/	Division
	**	Exponentiation
Concatenation	{ }	Concatenation
Replication	{{}}	Replicate value m for n times
Conditional	? :	Conditional

Verilog Data Types and Operators

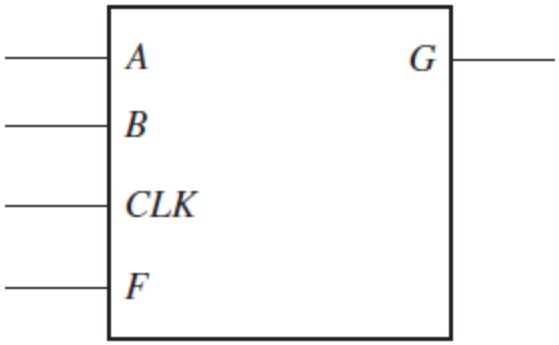
({A, ~B} | C >> 2 & D) == 110010

>>, &, |

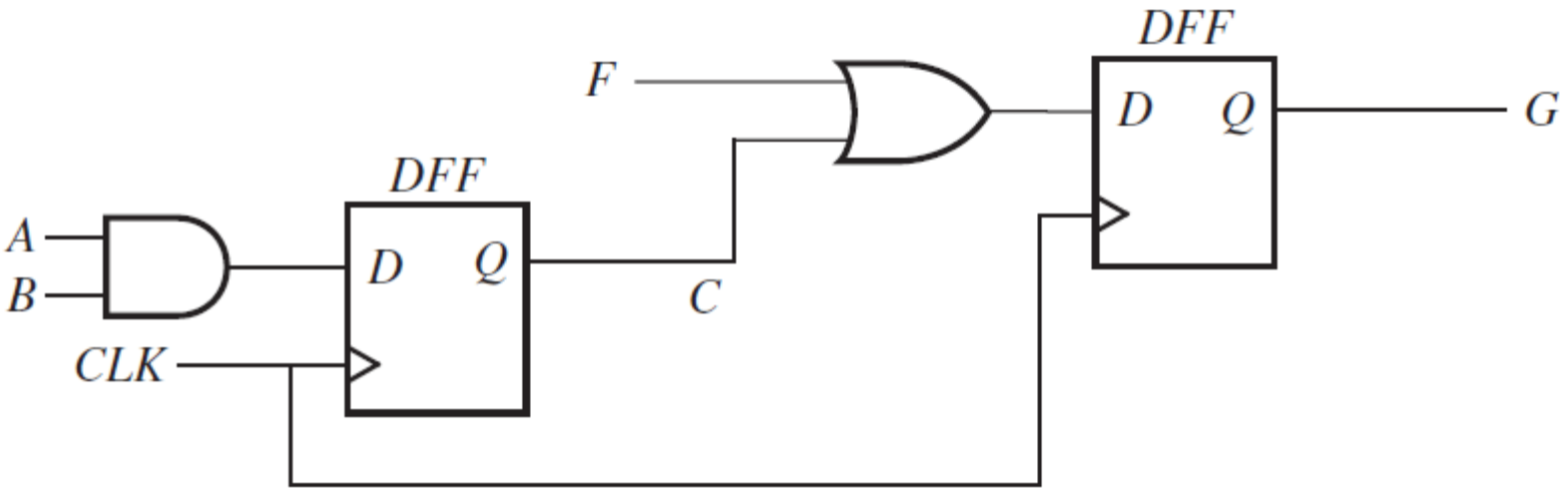
Synthesis examples

```
module Q3 (A, B, F, CLK, G);

input A;
input B;
input F;
input CLK;
output G;

reg G;
reg C;

always @(posedge CLK)
begin
    C <= A & B;          // statement 1
    G <= C | F;          // statement 2
end
endmodule
```

Synthesis examples



Synthesis examples



Synthesis examples

```
module no_syn (A, B, CLK, D);
input    A;
input    B;
input    CLK;
output   D;

reg      C;

always @(posedge CLK)
   C <= A & B;

endmodule
```

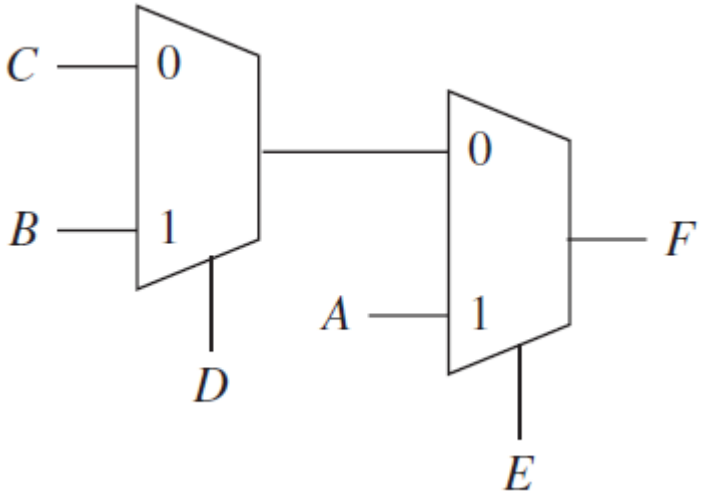
Synthesis examples

```
assign signal_name = condition ? expression_T : expression_F;
```

```
assign F = E ? A : ( D ? B : C );
// nested conditional assignment
```

Synthesis examples

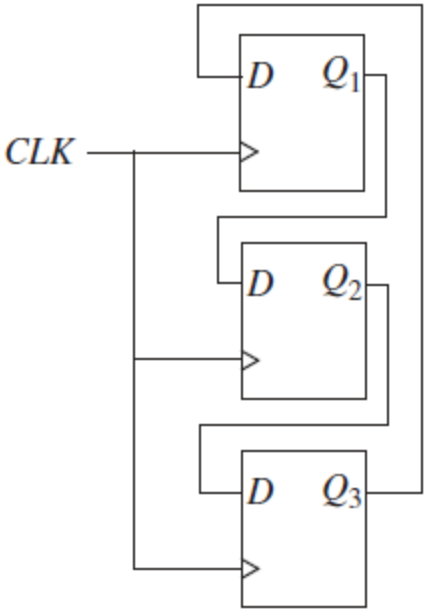
Cascaded 2 to 1 MUXes using conditional assignment




```
assign F = E ? A : ( D ? B : C );
// nested conditional assignment
```

Synthesis examples

Cyclic shift register




```
always @ (posedge CLK)
begin
   Q1 <= #5 Q3;
   Q2 <= #5 Q1;
   Q3 <= #5 Q2;
end
```

Synthesis examples

At the rising edge of the clock – all of the D inputs loaded into the Flip-flops – state change does not occur until after a propagation delay.

Non-blocking assignment – order of the statements not important.

Synthesis examples

```
module reg3 (Q1,Q2,Q3,A,CLK);
input   A;
input   CLK;
output  Q1,Q2,Q3;

reg     Q1,Q2,Q3;

always @(posedge CLK)
begin
  Q3 = Q2;  // statement 1
  Q2 = Q1;  // statement 2
  Q1 = A;  // statement 3

end

endmodule
```

Synthesis examples

```
module reg3 (Q1,Q2,Q3,A,CLK);
input   A;
input   CLK;
output  Q1,Q2,Q3;

reg     Q1,Q2,Q3;

always @(posedge CLK)
begin
  Q3 = Q2;  // statement 1
  Q2 = Q1;  // statement 2
  Q1 = A;  // statement 3

end

endmodule
```

3 bit shift register.

Blocking operator – execution of statements from top to bottom in order.

Synthesis examples

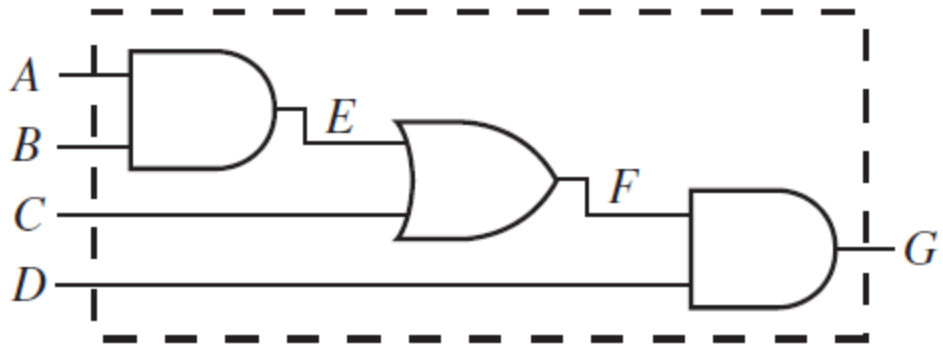
```
module reg31 (Q1,Q2,Q3,A,CLK);
input   A;
input   CLK;
output  Q1,Q2,Q3;

reg     Q1,Q2,Q3;

always @(posedge CLK)
begin

  Q1 = A;  // statement 1
  Q2 = Q1;  // statement 2
  Q3 = Q2;  // statement 3

end

endmodule
```

Synthesis examples

```
module reg31 (Q1,Q2,Q3,A,CLK);
input   A;
input   CLK;
output  Q1,Q2,Q3;

reg     Q1,Q2,Q3;

always @(posedge CLK)
begin

   Q1 = A;  // statement 1
   Q2 = Q1;  // statement 2
   Q3 = Q2;  // statement 3

end

endmodule
```

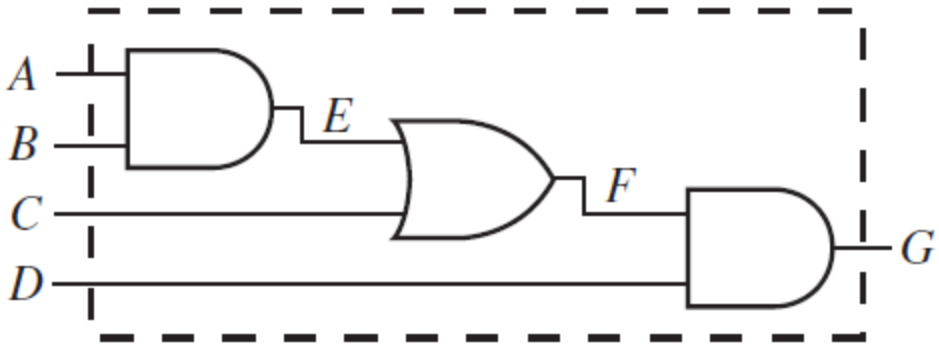
Single flip-flop

Synthesis examples



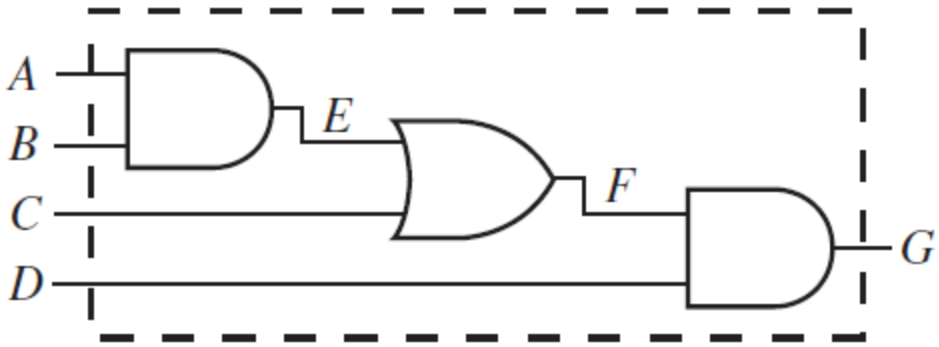
1. Verilog code using concurrent statements.
2. Using an always block with sequential statements.

Synthesis examples




```
module circuit(A, B, C, D, G);

input A, B, C, D;
output G;
wire E, F;

assign E = A && B;
assign F = E || C;
assign G = D && F;

endmodule
```

Synthesis examples




```
module circuit(A, B, C, D, G);

input A, B, C, D;
output reg G;
reg E, F;

initial begin
E <= 0;
F <= 0;
G <= 0;
end

always @(*)
begin
E <= A & & B;
F <= E | | C;
G <= F & & D;
end

endmodule
```

Writing test bench

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

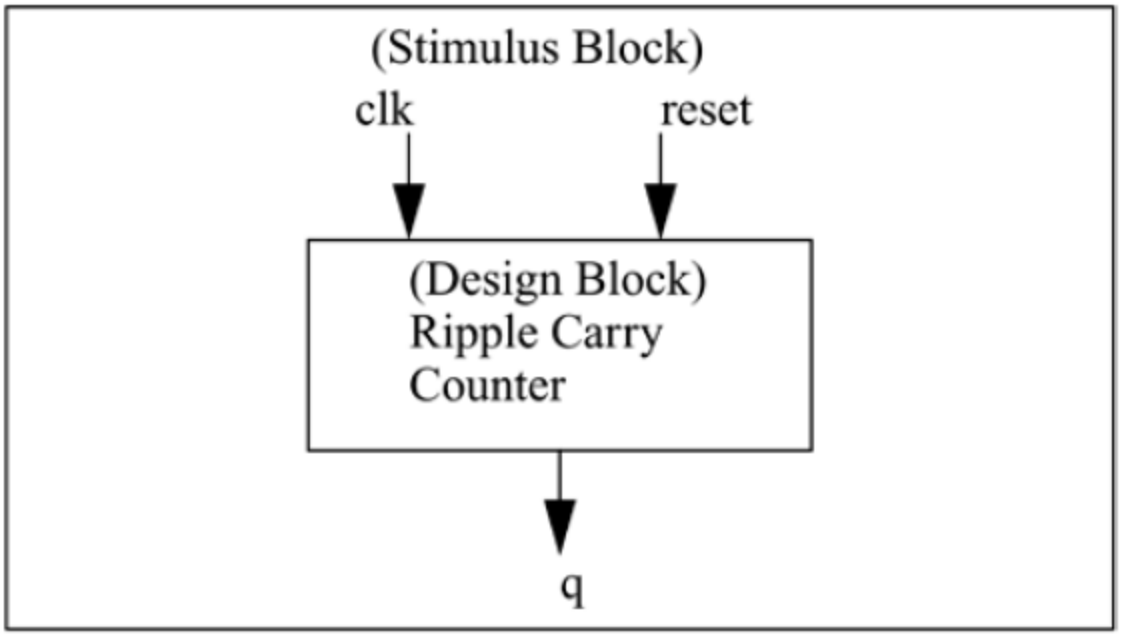
Test benches – van be quiet complex, lengthy.

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

Exercise all the operating features that are specified.

Hierarchical modeling concepts

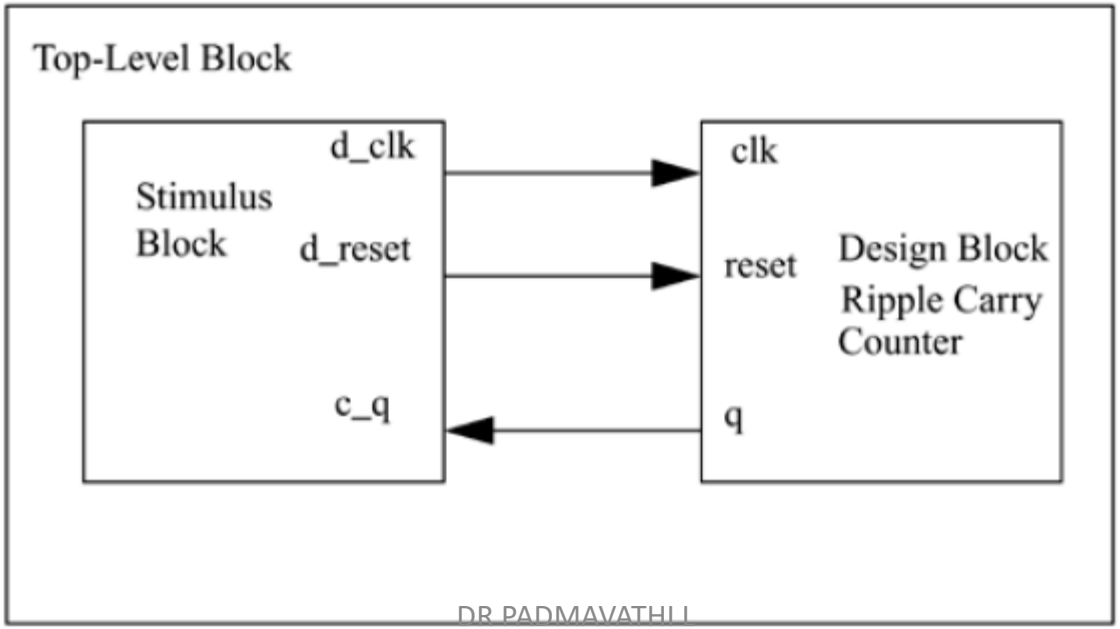
Stimulus block instantiating design block.



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

```
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.

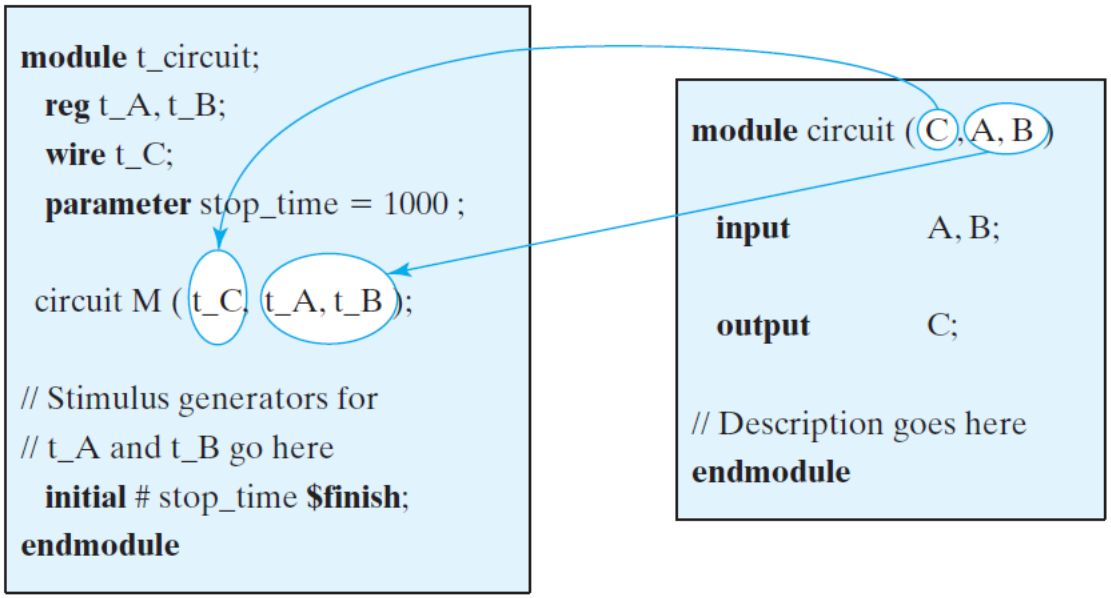
Stimulus test module

$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.