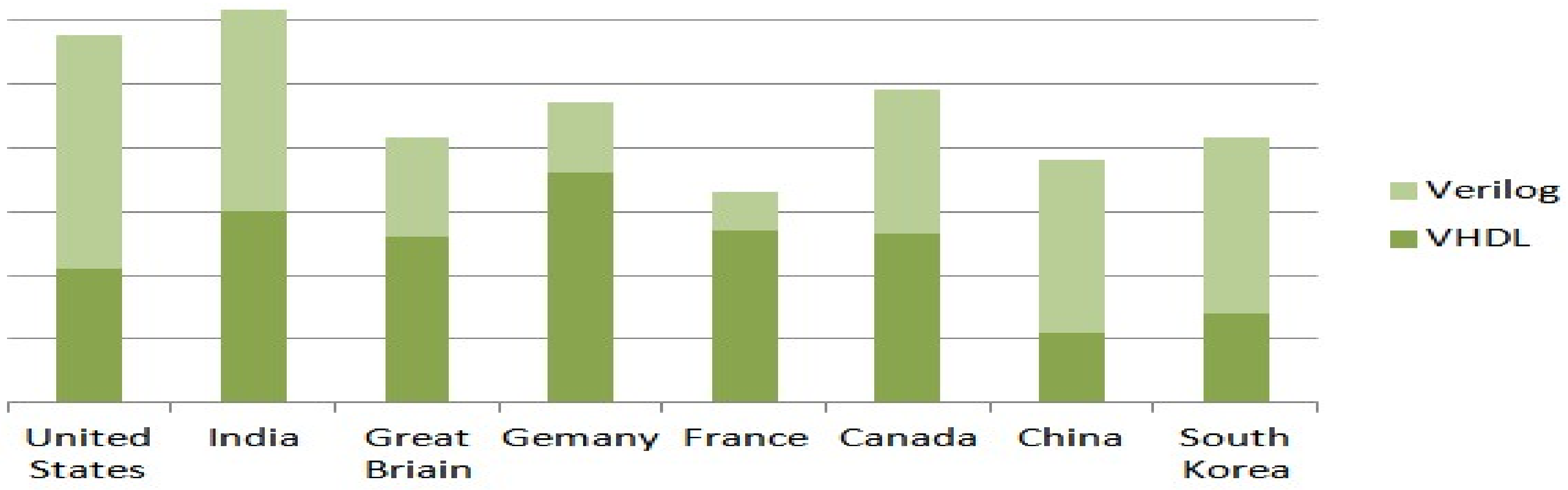
Verilog

Introduction

- Verilog HDL originated in 1983 at Gateway Design Automation.
- HDL are popular for logic verification
- Designers manually translate the HDL-based design into a schematic circuit with interconnections between gates
- Digital circuits could be described at a register transfer level (RTL) by use of an HDL.
- The details of gates and their interconnections to implement the circuit were automatically extracted by logic synthesis tools from the RTL description.
- Verilog standardized as IEEE 1364
- HDLs also began to be used for system-level design -→ ASIC / FPGA

VHDL vs Verilog

VHDL	Verilog
Library and Package requires	No library and Package
Case in-sensitive	Case Sensitive
Similar to ADA	Similar to C
IEEE 1164	IEEE 1364

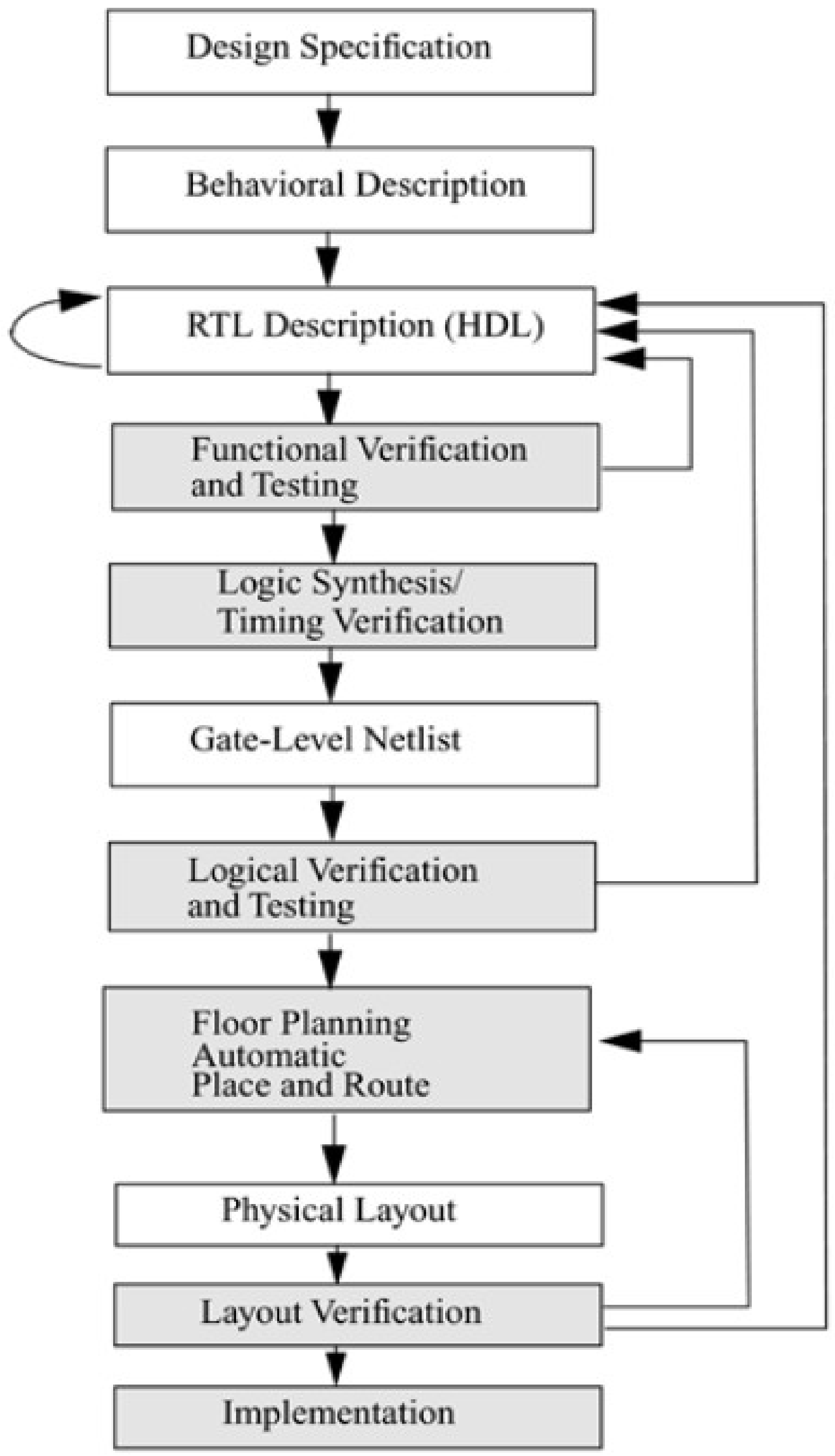


VLSI Design Flow

Specifications→ functionality, interface, and overall architecture of the digital circuit

behavioral description→ analyze the design in terms of functionality performance, compliance to standards and other high-level issues

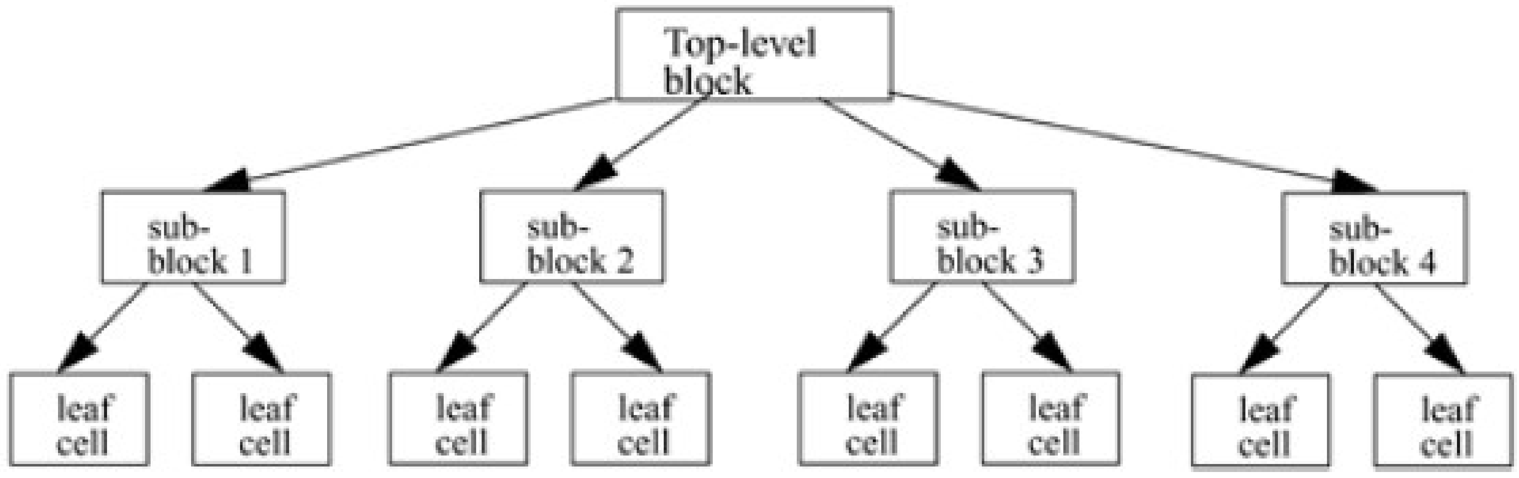
behavioral description is manually converted to an RTL description in an HDL


Design Specification
Behavioral Description
RTL Description (HDL)
Functional Verification and Testing
Logic Synthesis/ Timing Verification
Gate-Level Netlist
Logical Verification and Testing
Floor Planning Automatic Place and Route
Physical Layout
Layout Verification
Implementation

Design Methodologies

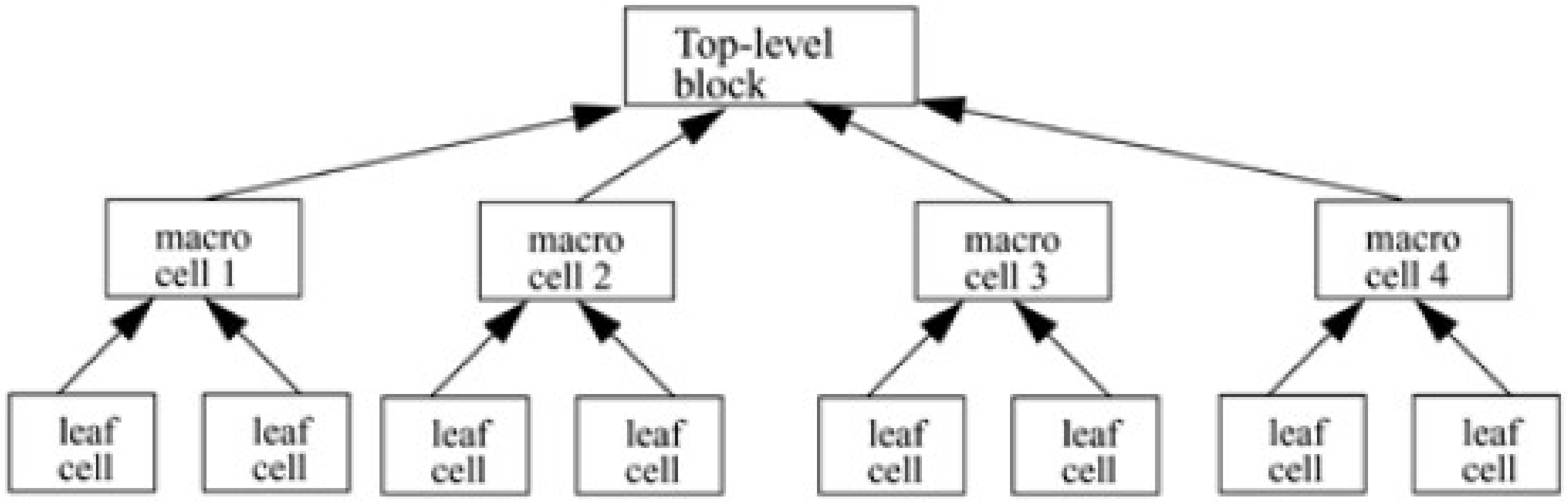
Top-down Design Methodology

define the top-level block and identify the sub-blocks necessary to build the top-level block.
We further subdivide the sub-blocks until we come to leaf cells
which are the cells that cannot further be divided



Bottom-up Design Methodology

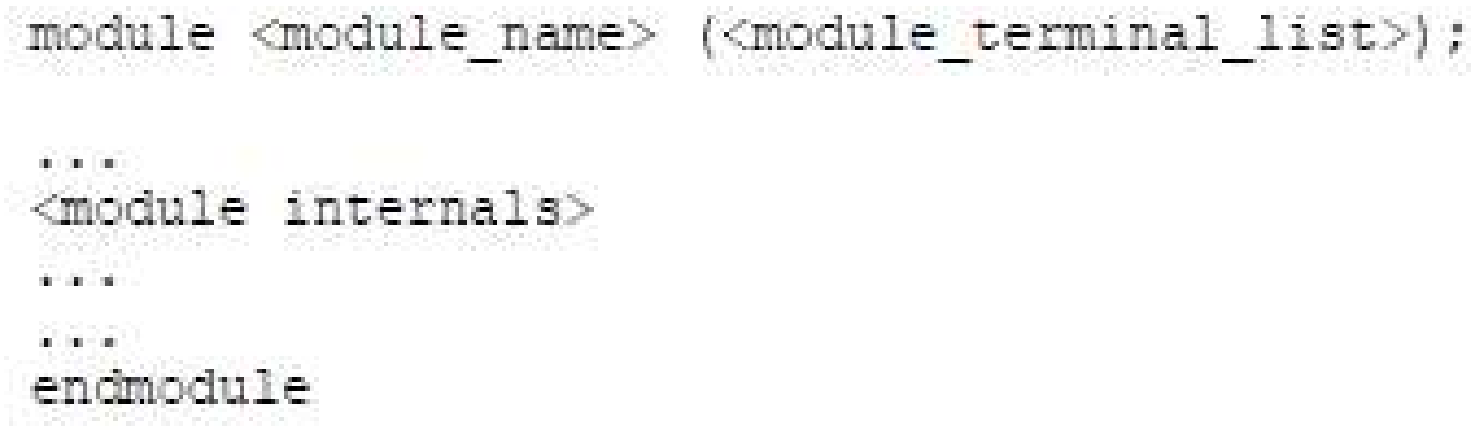
first identify the building blocks that are available to us.
We build bigger cells, using these building blocks.
These cells are then used for higher-level blocks until we build the top-level block in the design



Typically, a combination of top-down and bottom-up flows is used

Modules

- A module is the basic building block in Verilog.
- A module can be an element or a collection of lower-level design blocks.
- Typically, elements are grouped into modules to provide common functionality that is used at many places in the design.
- A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs)



```
module <module_name> (<module_terminal_list>);

...
<module internals>
...
...
endmodule
```

module can be defined at four levels of abstraction

1. Behavioral or algorithmic level

- This is the highest level of abstraction provided by Verilog HDL.
- A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details

2. Dataflow level

- module is designed by specifying the data flow.
- The designer is aware of how data flows between hardware registers

3. Gate level

- module is implemented in terms of logic gates and interconnections between these gates

4. Switch level

- This is the lowest level of abstraction provided by Verilog.
- A module can be implemented in terms of switches, storage nodes, and the interconnections between them.

Lexical Conventions

Whitespace Blank spaces (\b)

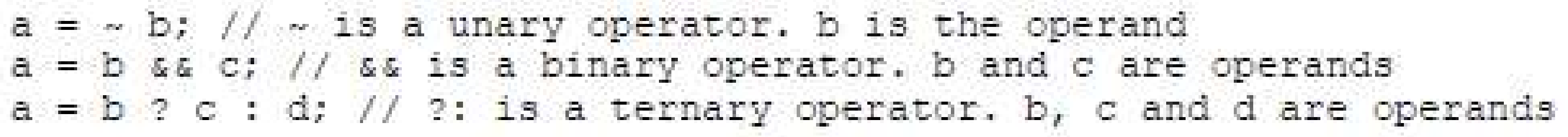
tabs (\t)

newlines (\n)

Comments one-line comment starts with "//“

multiple-line comment starts with "/*" and ends with "*/“

Operators



```
a = ~ b; // ~ is a unary operator. b is the operand
a = b && c; // && is a binary operator. b and c are operands
a = b ? c : d; // ?: is a ternary operator. b, c and d are operands
```

Number Specification

Sized numbers <size> '<base format> <number>

<size> in decimal and specifies the number of bits in the number.

<base format> are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O).

<number> specified as consecutive digits from 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f

```
4'b1111 // This is a 4-bit   binary number
12'habc // This is a 12-bit  hexadecimal number
16'd255 // This is a 16-bit  decimal number.
```

Unsized numbers

without a <base format> specification are decimal numbers by default.
Numbers that are written without a <size> specification have a default number of bits that is simulator- and machine-specific (must be at least 32).

```
23456 // This is a 32-bit   decimal number by default
'hc3 // This is a 32-bit   hexadecimal number
'o21 // This is a 32-bit   octal number
```

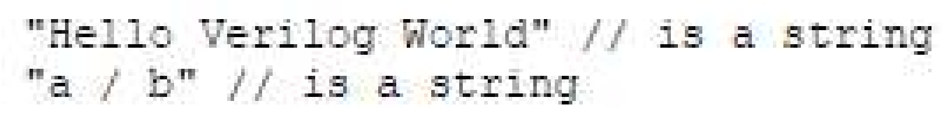
```
12'h13x // This is a 12-bit hex number; 4 least significant bits
unknown
6'hx // This is a 6-bit hex number
32'bz // This is a 32-bit high impedance number
```

Negative numbers minus sign before the size for a constant number.

-25 ➔ -5’b11001

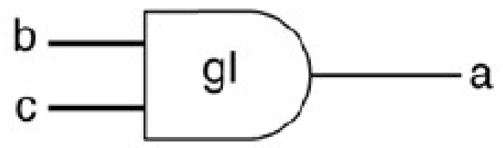
Strings

- sequence of characters that are enclosed by double quotes.
- string is that it must be contained on a single line
- It cannot be on multiple lines.



```
"Hello Verilog World" // is a string
"a / b" // is a string
```

Nets Nets represent connections between hardware elements.



Nets are declared primarily with the keyword wire.
Nets are one-bit values by default
The default value of a net is z
Nets get the output value of their drivers. If a net has no driver, it gets the value z.

Verilog Logic Value

Verilog is 4 valued logic

0 – False

1 – True

X –Unknown

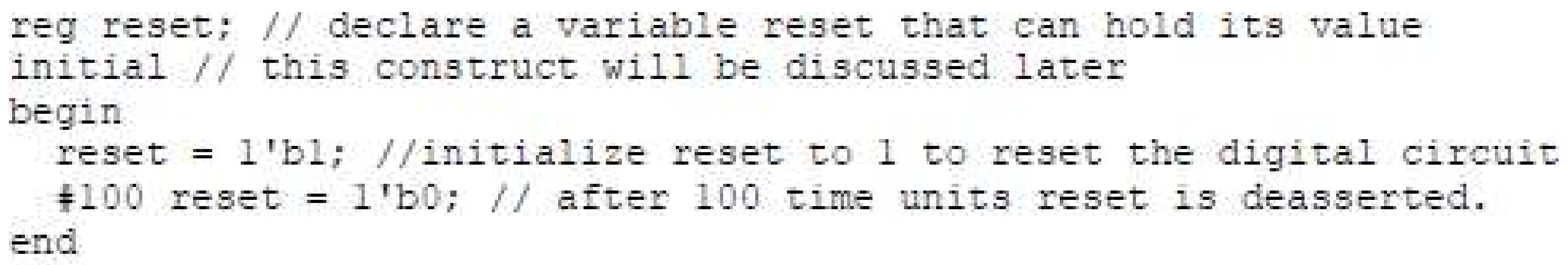
Z- High impedance

Registers

- Registers represent data storage elements.
- Registers retain value until another value is placed onto them.

  reg ➔ variable that can hold a value.

- default value for a reg data type is x

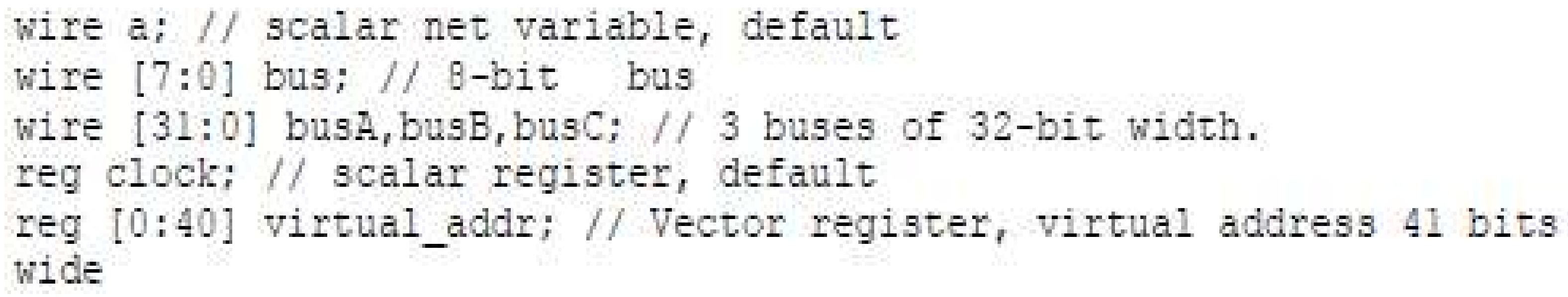


```
reg reset; // declare a variable reset that can hold its value
initial // this construct will be discussed later
begin
  reset = 1'b1; //initialize reset to 1 to reset the digital circuit
  #100 reset = 1'b0; // after 100 time units reset is deasserted.
end
```

Reg	Wire
Variable data type	Net data type
Can store value	Connection between part of a design
Default value X	Default value Z
Used in Procedural block	Used in concurrent assignment

Vectors

Nets or reg data types can be declared as vectors (multiple bit widths). If bit width is not specified, the default is scalar (1-bit).

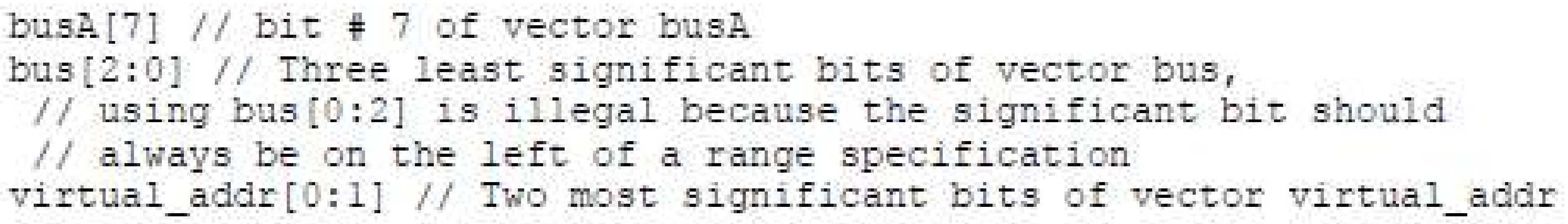


```
wire a; // scalar net variable, default
wire [7:0] bus; // 8-bit   bus
wire [31:0] busA,busB,busC; // 3 buses of 32-bit width.
reg clock; // scalar register, default
reg [0:40] virtual_addr; // Vector register, virtual address 41 bits
wide
```

[high# : low#] or [low# : high#]

left number in the squared brackets is always the MSB of the vector

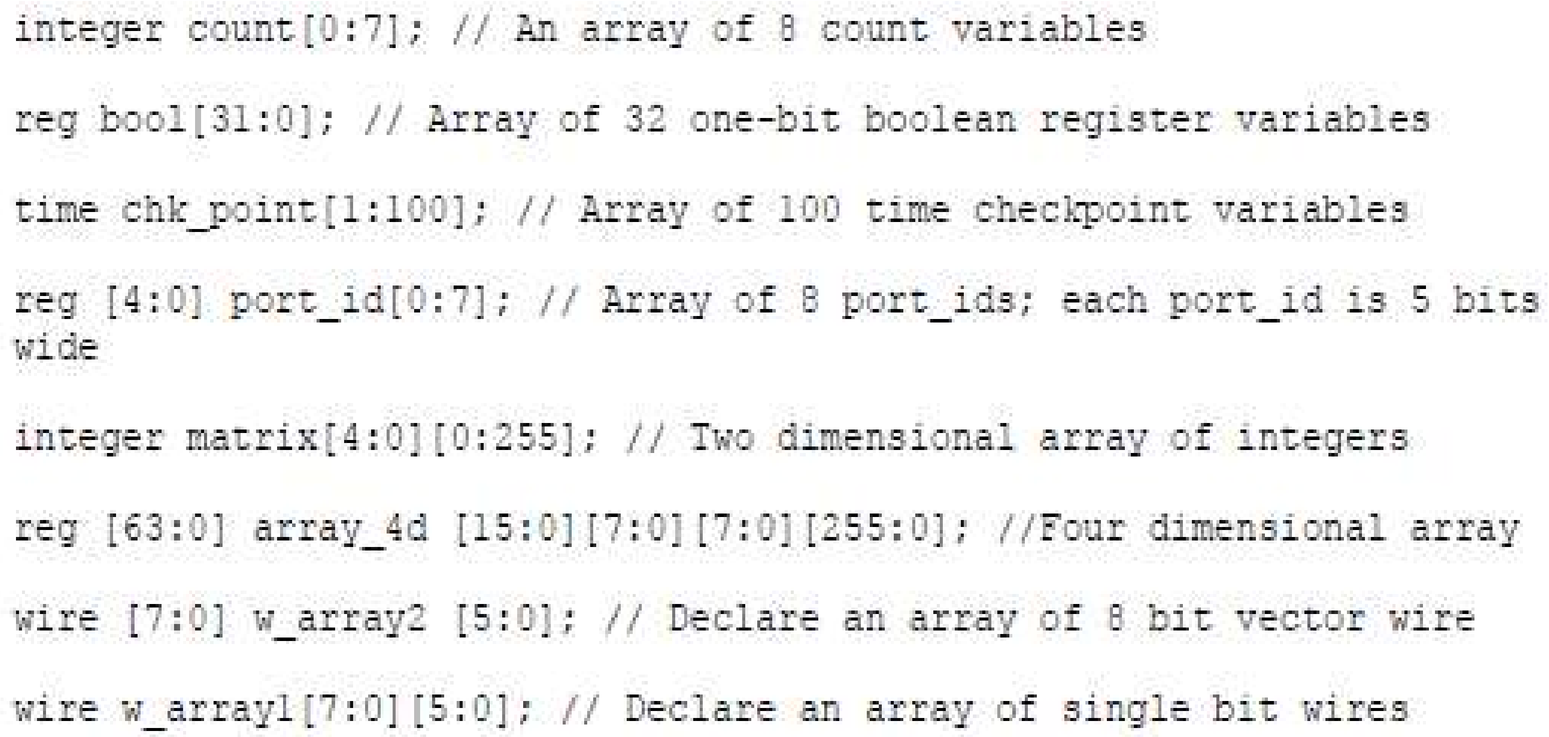
Vector Part Select



```
busA[7] // bit # 7 of vector busA
bus[2:0] // Three least significant bits of vector bus,
 // using bus[0:2] is illegal because the significant bit should
 // always be on the left of a range specification
virtual_addr[0:1] // Two most significant bits of vector virtual_addr
```

Arrays array is a collection of variables

Arrays are accessed by <array_name>[<subscript>]



```
integer count[0:7]; // An array of 8 count variables

reg bool[31:0]; // Array of 32 one-bit boolean register variables

time chk_point[1:100]; // Array of 100 time checkpoint variables

reg [4:0] port_id[0:7]; // Array of 8 port_ids; each port_id is 5 bits
wide

integer matrix[4:0][0:255]; // Two dimensional array of integers

reg [63:0] array_4d [15:0][7:0][7:0][255:0]; //Four dimensional array

wire [7:0] w_array2 [5:0]; // Declare an array of 8 bit vector wire

wire w_array1[7:0][5:0]; // Declare an array of single bit wires
```

QUIZ

The default value for reg data type is _______.

A. 0

B. z

C. 1

D. x

QUIZ

To introduce delays in a circuit, we can use a ___________

A. Buffer
B. EXOR gate
C. Inverter
D. Flip-flops

Integer

integer is a general purpose register data type used for manipulating quantities.

The default width for an integer is the host-machine word size 32 bits

```
integer counter; // general purpose variable used as a counter.
initial
counter = -1;
```

Time

time register data type is used in Verilog to store simulation time.

system function $time is invoked to get the current simulation time.

```
time save_sim_time; // Define a time variable save_sim_time
initial
save_sim_time = $time; // Save the current simulation time
```

Parameters

constants to be defined in a module by the keyword parameter

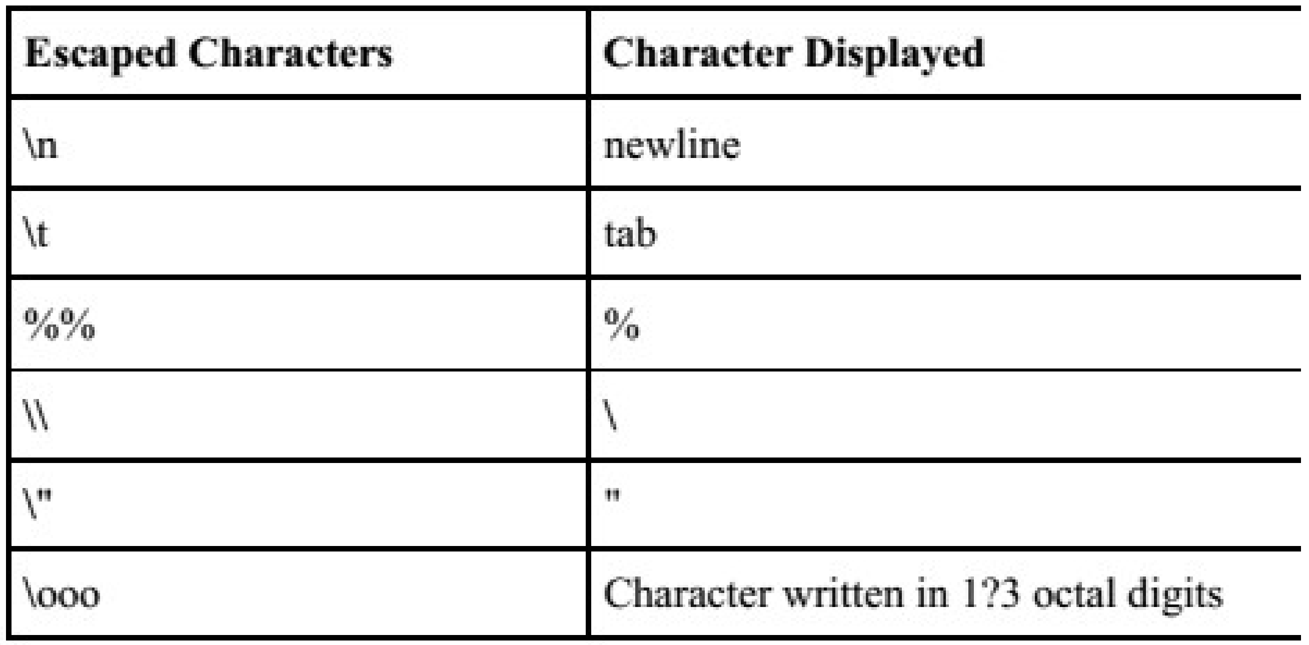
```
parameter port_id = 5;
```

Memories RAM/ROM are modelled as 1D array

Each element of the array is element or word and is addressed by a single array index.

Each word can be one or more bits

```
reg mem1bit[0:1023];          // Memory mem1bit with 1K 1-bit words
reg [7:0] membyte[0:1023];    // Memory membyte with 1K 8-bit words
membyte[511]                  // Fetches 1 byte word whose address is 511.
```



Escaped Characters	Character Displayed
\n	newline
\t	tab
%%	%
\\	\
\"	"
\ooo	Character written in 1?3 octal digits

Practice

Which of the following statements is true for Verilog modules?

a. A module can contain definitions of other modules.

b. When a module X is called multiple numbers of times from some other module, only one copy of module X is included in the hardware after synthesis.

c. More than one module can be instantiated within another module.

d. If a module X is instantiated 4 times within another module, 4 copies of X are created.

Correct answers are (c) and (d).

Practice

What does the statement   assign f = (a & b) | (a ^ b) signify?

a. A gate level netlist consisting of one AND gate, one OR gate, and one XOR gate.
b. A behavioral description of the function f.
c. A structural description of the function f.
d. A continuous assignment of the function realized by the right hand side to the net type variable on the left hand side.

Correct answers are (b) and (d).

Practice

For the following Verilog code segment, if the initial value of IR is ABCD3456 (in hexadecimal), the value of “data” in decimal will be …………. (Note that “data” is a 4-bit variable)

```
wire [31:0] IR;
wire [3:0] data;
wire [15:0] d1;
wire [31:16] d2;
assign d1 = IR[31:16];
assign d2 = IR[15:0];
assign data = d1[11:8] + d2[19:16] + d2[31:28];
```

System Tasks➔ $<keyword>

system tasks for certain routine operations

$display is system task for displaying values of variables or strings or expressions.

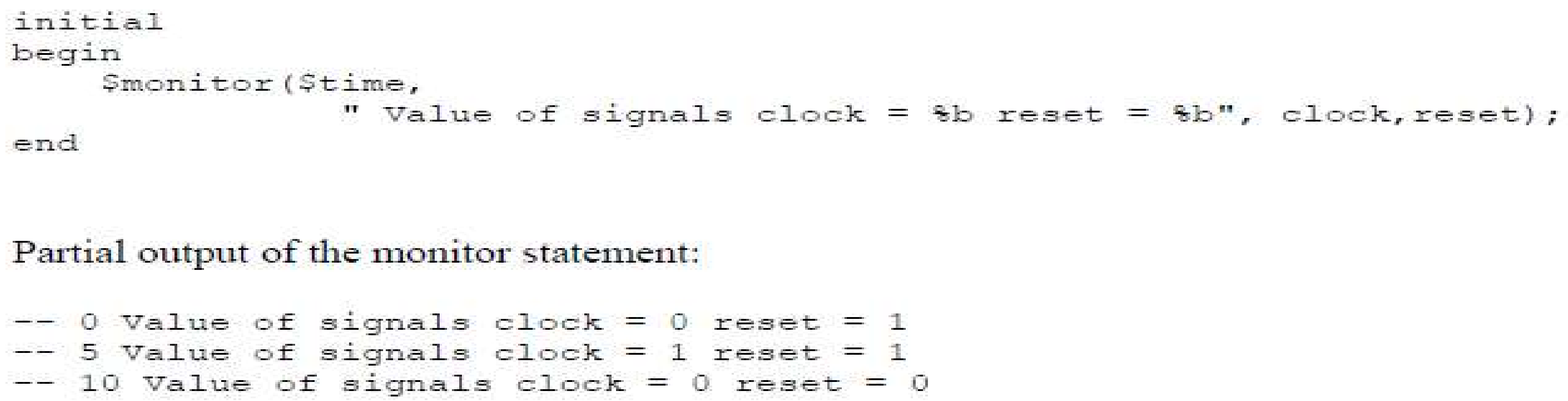
Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name (no argument required)
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format

```
//Display the string in quotes
$display("Hello Verilog World");
-- Hello Verilog World

//Display value of current simulation time 230
$display($time);
-- 230

//Display value of 41-bit virtual address 1fe0000001c at time 200
reg [0:40] virtual_addr;
$display("At time %d virtual address is %h", $time, virtual_addr);
-- At time 200 virtual address is 1fe0000001c

//Display value of port_id 5 in binary
reg [4:0] port_id;
$display("ID of the port is %b", port_id);
-- ID of the port is 00101

//Display x characters
//Display value of 4-bit bus 10xx (signal contention) in binary
reg [3:0] bus;
$display("Bus value is %b", bus);
-- Bus value is 10xx
```

$monitor task   monitor a signal when its value changes

continuously monitors the values of the variables or signals specified in the parameter list

displays all parameters in the list whenever the value of any one variable or signal changes



```
initial
begin
    $monitor($time,
                  " Value of signals clock = %b reset = %b", clock,reset);
end
```

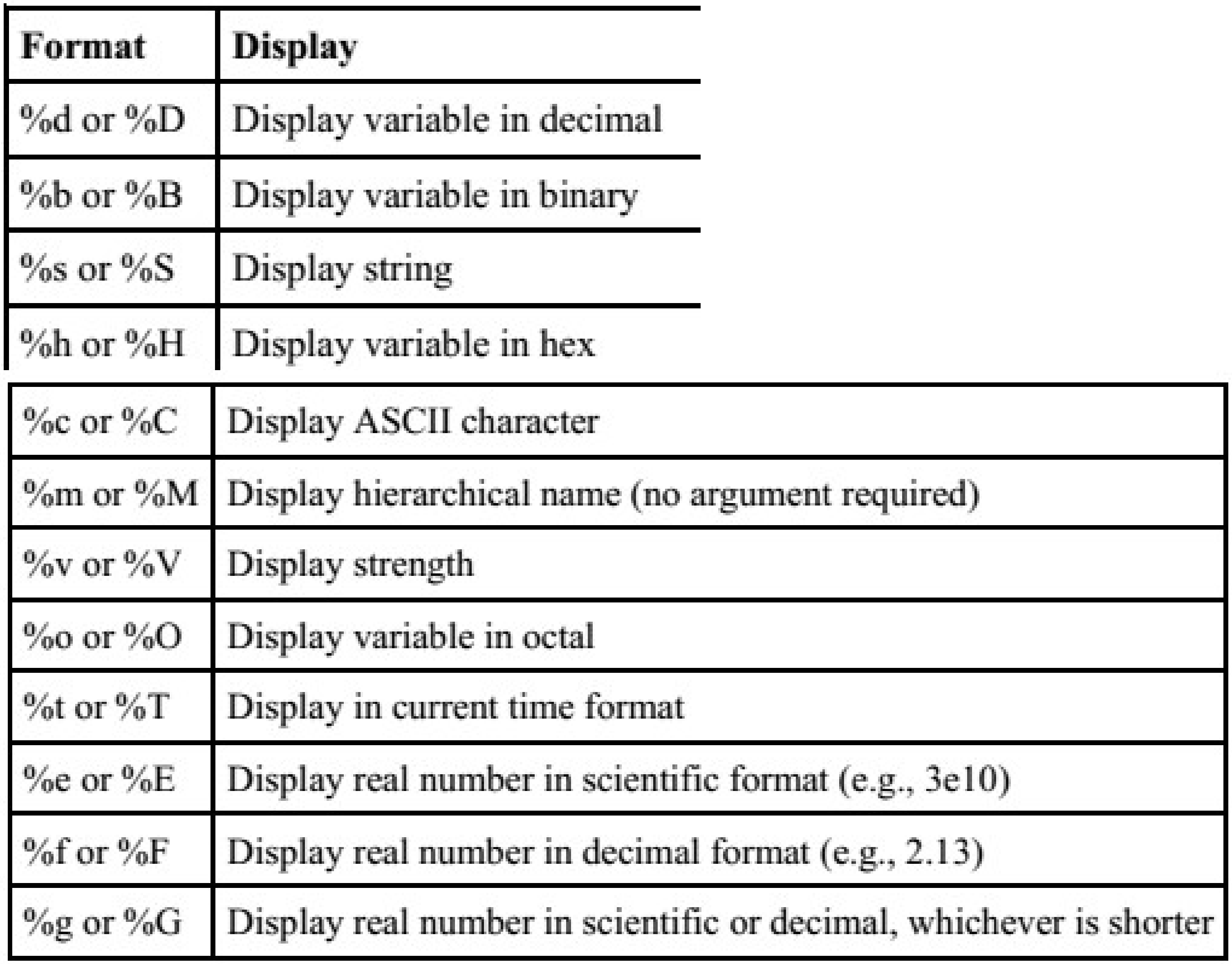
Partial output of the monitor statement:

```
-- 0 Value of signals clock = 0 reset = 1
-- 5 Value of signals clock = 1 reset = 1
-- 10 Value of signals clock = 0 reset = 0
```

$display s-- prints the value (once) whenever it is executed.

$monitor -- prints the values whenever at least one of the variables in the list is modified

String Format Specifications



Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name (no argument required)
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format (e.g., 3e10)
%f or %F	Display real number in decimal format (e.g., 2.13)
%g or %G	Display real number in scientific or decimal, whichever is shorter

$stop designer wants to suspend the simulation and examine the values of signals in the design.
$finish task terminates the simulation

```
initial // to be explained later. time = 0
begin
clock = 0;
reset = 1;
#100 $stop; // This will suspend the simulation at time = 100
#900 $finish; // This will terminate the simulation at time = 1000
end
```

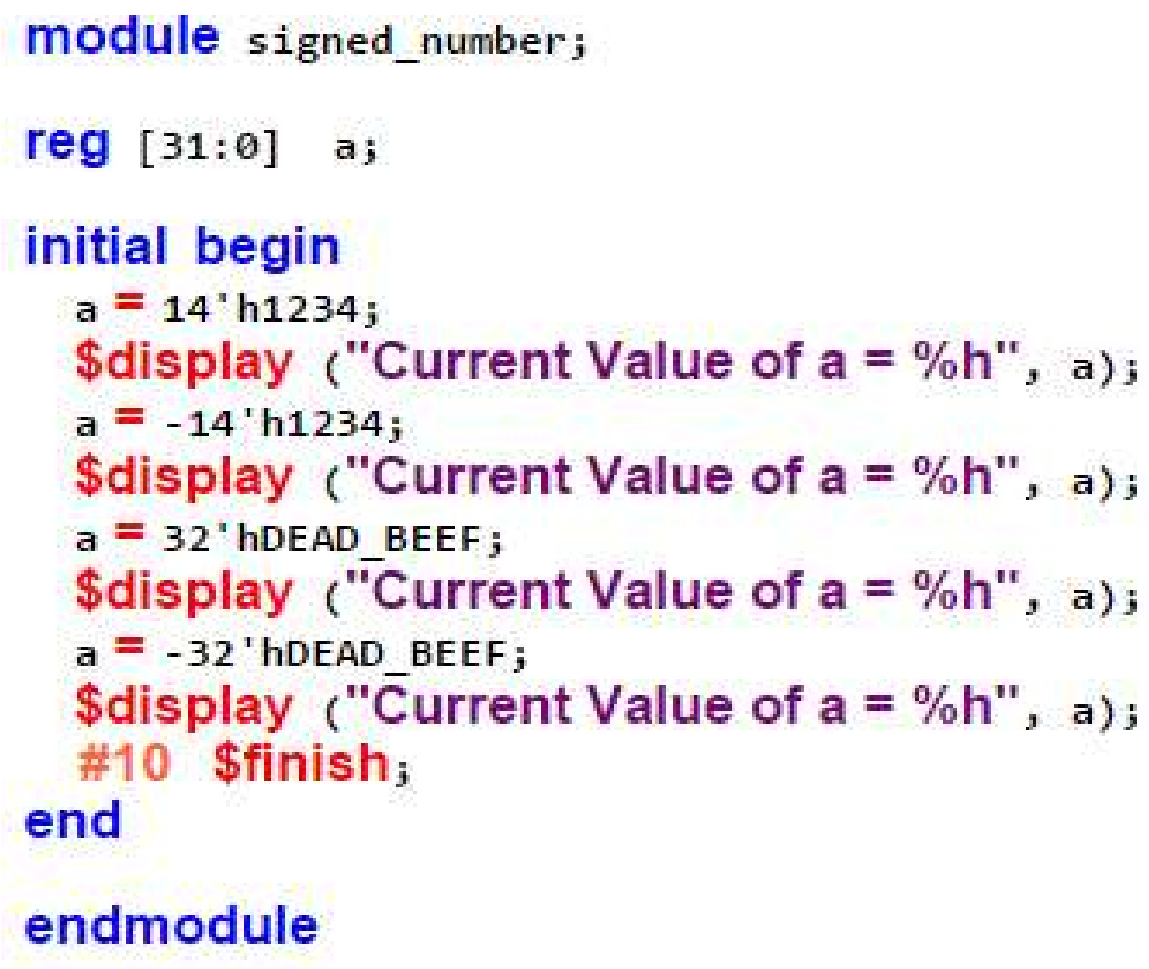
```
latch = 4'd12;
$display("The current value of latch = %b\n", latch);

in_reg = 3'd2;
$monitor($time, " In register value = %b\n", in_reg[2:0]);

`define MEM_SIZE 1024
$display("The maximum memory size is %h", 'MEM_SIZE);
```

QUIZ

To suspend a simulation, you can use this system task command.

A. **$finish**
B. **$stop**
C. **$end**
D. **$close**

Example

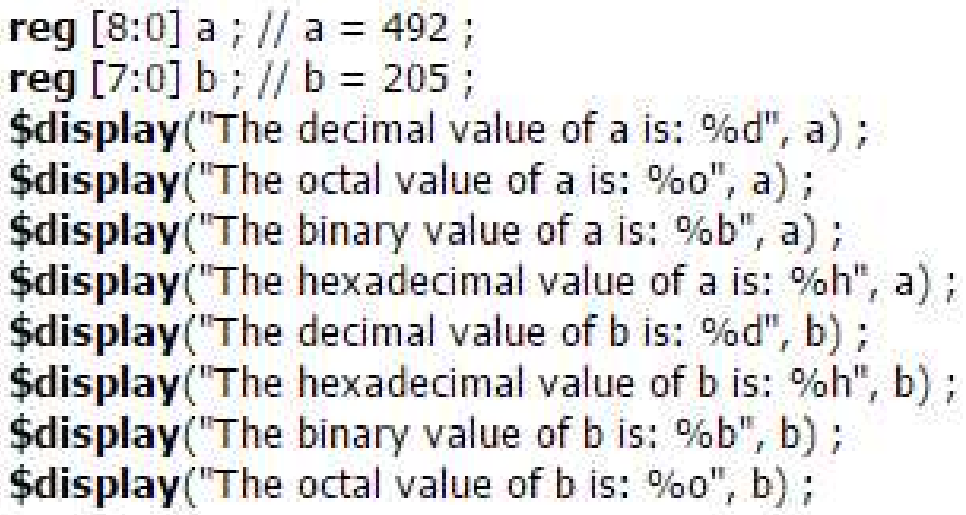


```
module signed_number;

reg [31:0]  a;

initial begin
  a = 14'h1234;
  $display ("Current Value of a = %h", a);
  a = -14'h1234;
  $display ("Current Value of a = %h", a);
  a = 32'hDEAD_BEEF;
  $display ("Current Value of a = %h", a);
  a = -32'hDEAD_BEEF;
  $display ("Current Value of a = %h", a);
  #10  $finish;
end

endmodule
```

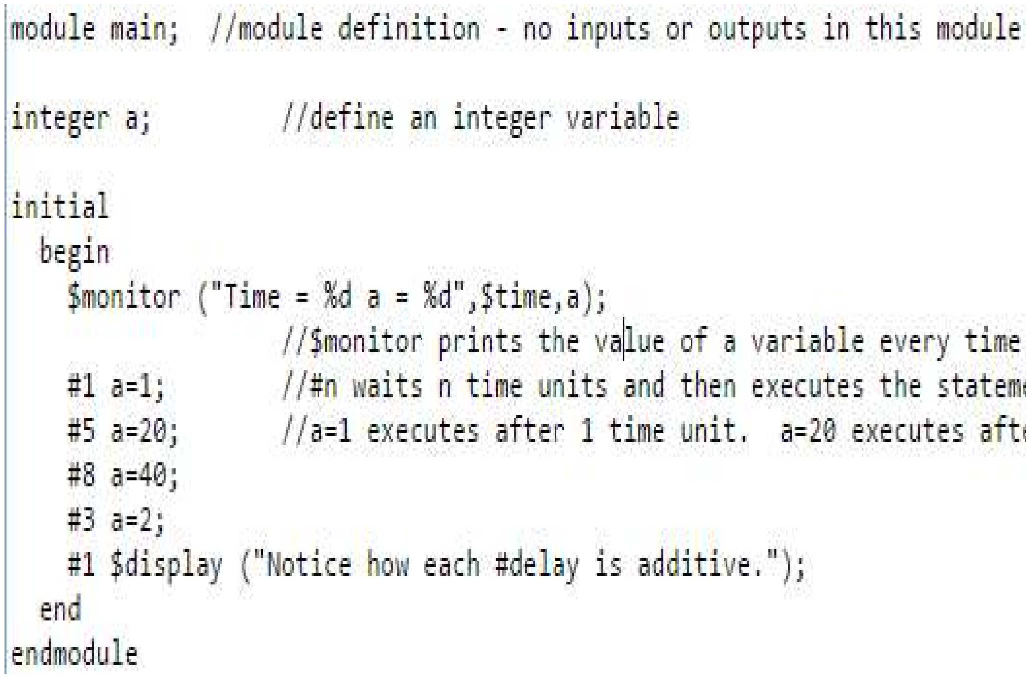
QUIZ

When does the $monitor statement in a Verilog test bench print the specified values?

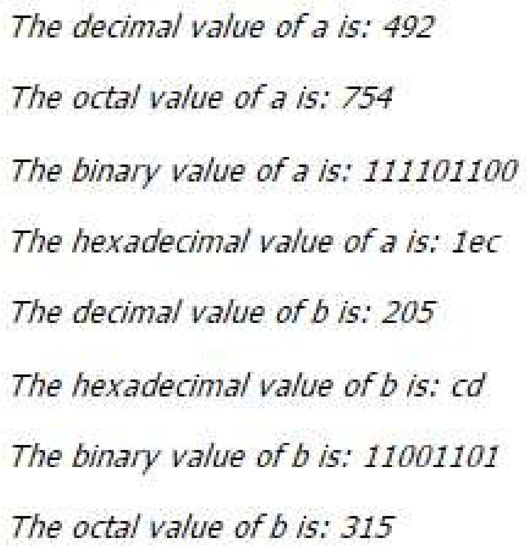
a. At the beginning of the simulation.
b. When the $monitor statement is first encountered.
c. Whenever the value of any of the specified variables change.
d. None of the above



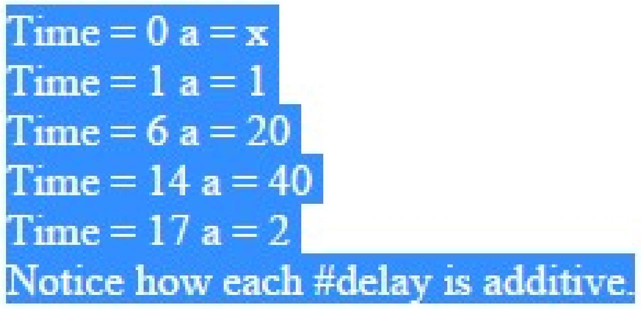
```
reg [8:0] a ; // a = 492 ;
reg [7:0] b ; // b = 205 ;
$display("The decimal value of a is: %d", a) ;
$display("The octal value of a is: %o", a) ;
$display("The binary value of a is: %b", a) ;
$display("The hexadecimal value of a is: %h", a) ;
$display("The decimal value of b is: %d", b) ;
$display("The hexadecimal value of b is: %h", b) ;
$display("The binary value of b is: %b", b) ;
$display("The octal value of b is: %o", b) ;
```



```
module main;  //module definition - no inputs or outputs in this module

integer a;          //define an integer variable

initial
  begin
    $monitor ("Time = %d a = %d",$time,a);
                    //$monitor prints the value of a variable every time
    #1 a=1;         //#n waits n time units and then executes the stateme
    #5 a=20;        //a=1 executes after 1 time unit.  a=20 executes afte
    #8 a=40;
    #3 a=2;
    #1 $display ("Notice how each #delay is additive.");
  end
endmodule
```

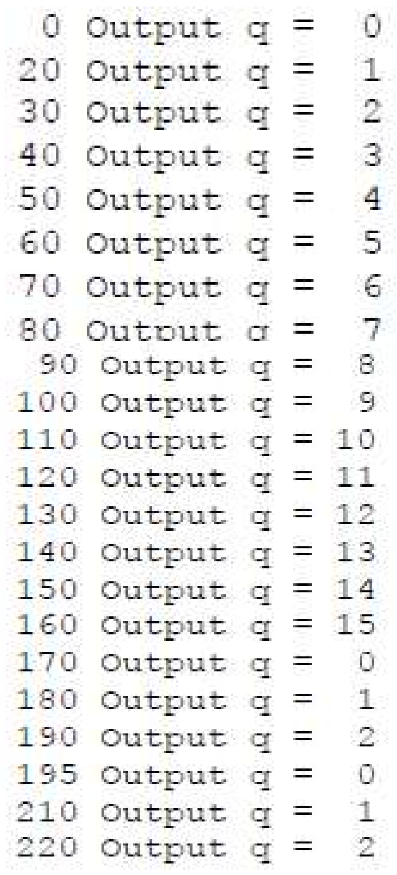


```
The decimal value of a is: 492

The octal value of a is: 754

The binary value of a is: 111101100

The hexadecimal value of a is: 1ec

The decimal value of b is: 205

The hexadecimal value of b is: cd

The binary value of b is: 11001101

The octal value of b is: 315
```



```
Time = 0 a = x
Time = 1 a = 1
Time = 6 a = 20
Time = 14 a = 40
Time = 17 a = 2
Notice how each #delay is additive.
```

```
module stimulus;
reg clk;
reg reset;
wire[3:0] q;
ripple_carry_counter r1(q, clk, reset);
initial
clk = 1'b0;
always
#5 clk = ~clk;
initial
begin
reset = 1'b1;
#15 reset = 1'b0;
#180 reset = 1'b1;
#10 reset = 1'b0;
#20 $finish;
End
initial
$monitor($time, " Output q = %d", q);
endmodule
```



```
  0 Output q =  0
 20 Output q =  1
 30 Output q =  2
 40 Output q =  3
 50 Output q =  4
 60 Output q =  5
 70 Output q =  6
 80 Output q =  7
 90 Output q =  8
100 Output q =  9
110 Output q = 10
120 Output q = 11
130 Output q = 12
140 Output q = 13
150 Output q = 14
160 Output q = 15
170 Output q =  0
180 Output q =  1
190 Output q =  2
195 Output q =  0
210 Output q =  1
220 Output q =  2
```

QUIZ

When does the $monitor statement in a Verilog test bench print the specified values?

a. At the beginning of the simulation.
b. When the $monitor statement is first encountered.
c. Whenever the value of any of the specified variables change.
d. None of the above

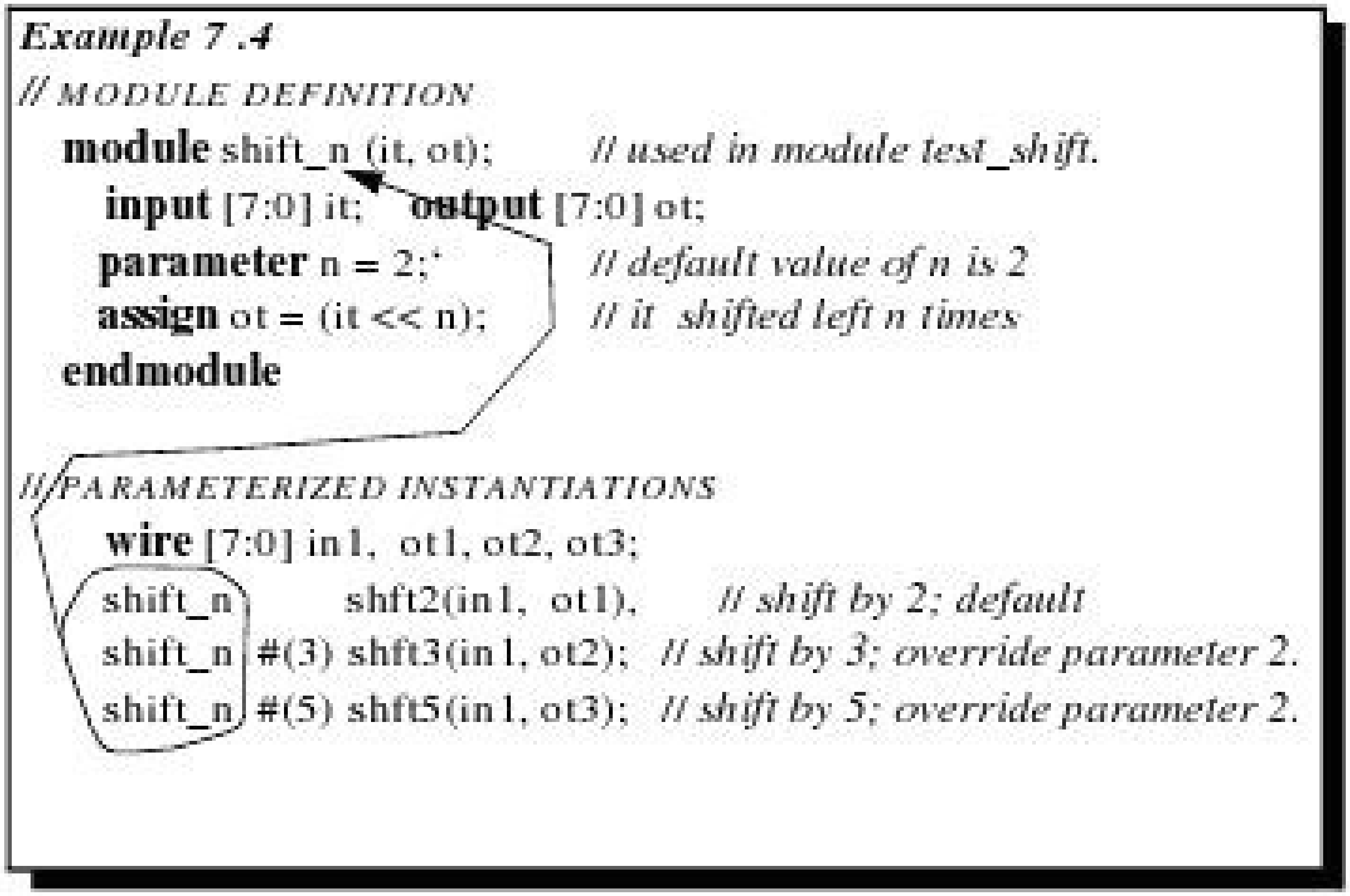
Instantation

A module provides a template from which you can create actual objects.

When a module is invoked, Verilog creates a unique object from the template.

Each object has its own name, variables, parameters, and I/O interface.

The process of creating objects from a module template is called instantiation, and the objects are called instances



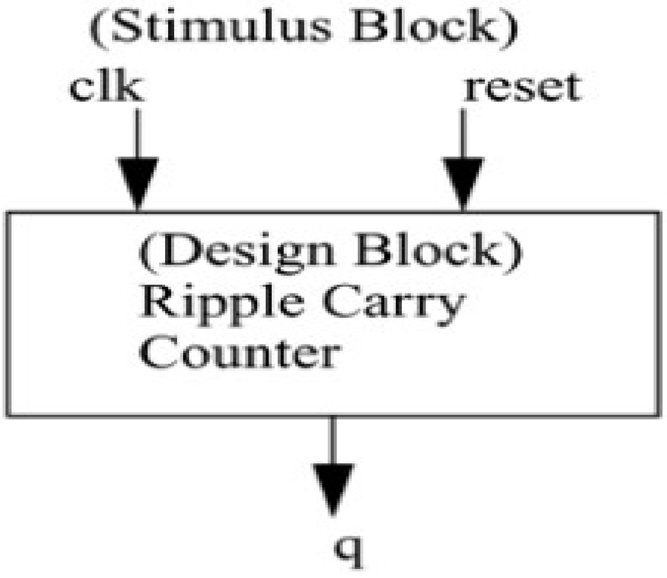
Example 7 .4

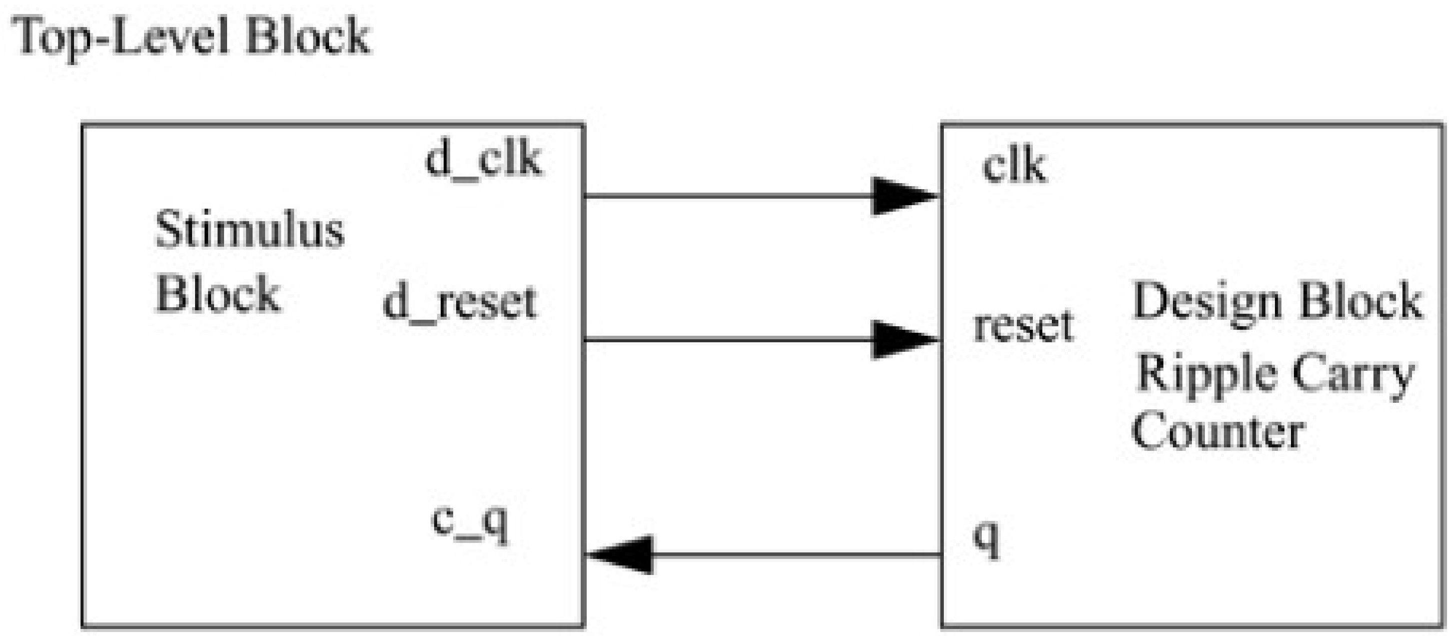
```
// MODULE DEFINITION
  module shift_n (it, ot);        // used in module test_shift.
    input [7:0] it;  output [7:0] ot;
    parameter n = 2;              // default value of n is 2
    assign ot = (it << n);        // it shifted left n times
  endmodule

//PARAMETERIZED INSTANTIATIONS
    wire [7:0] in1, ot1, ot2, ot3;
    shift_n      shft2(in1, ot1),    // shift by 2; default
    shift_n #(3) shft3(in1, ot2);  // shift by 3; override parameter 2.
    shift_n #(5) shft5(in1, ot3);  // shift by 5; override parameter 2.
```

Components of a Simulation

- Once a design block is completed, it must be tested.
- The functionality of the design block can be tested by applying stimulus and checking results
- Stimulus or Test bench
- Simulation – Process of functionality verification
- stimulus and design blocks separate

- stimulus block instantiates the design block and drives the signals in the design block





d_clk and d_reset, connected to clk and reset
c_q connected to q

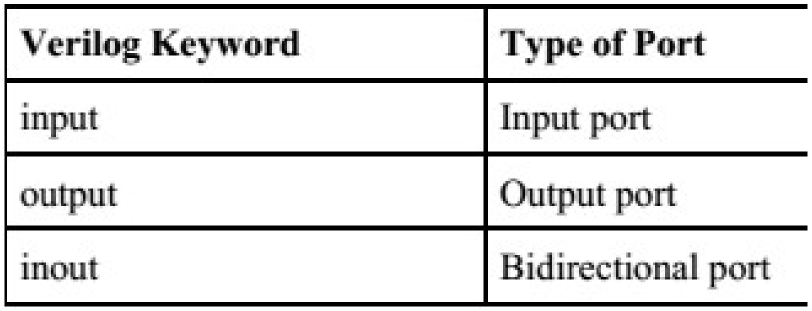
In stimulus block   Top module
Input declared as reg
Output declared as wired
Top module instantioned in stimulus block

Components of a Verilog Module

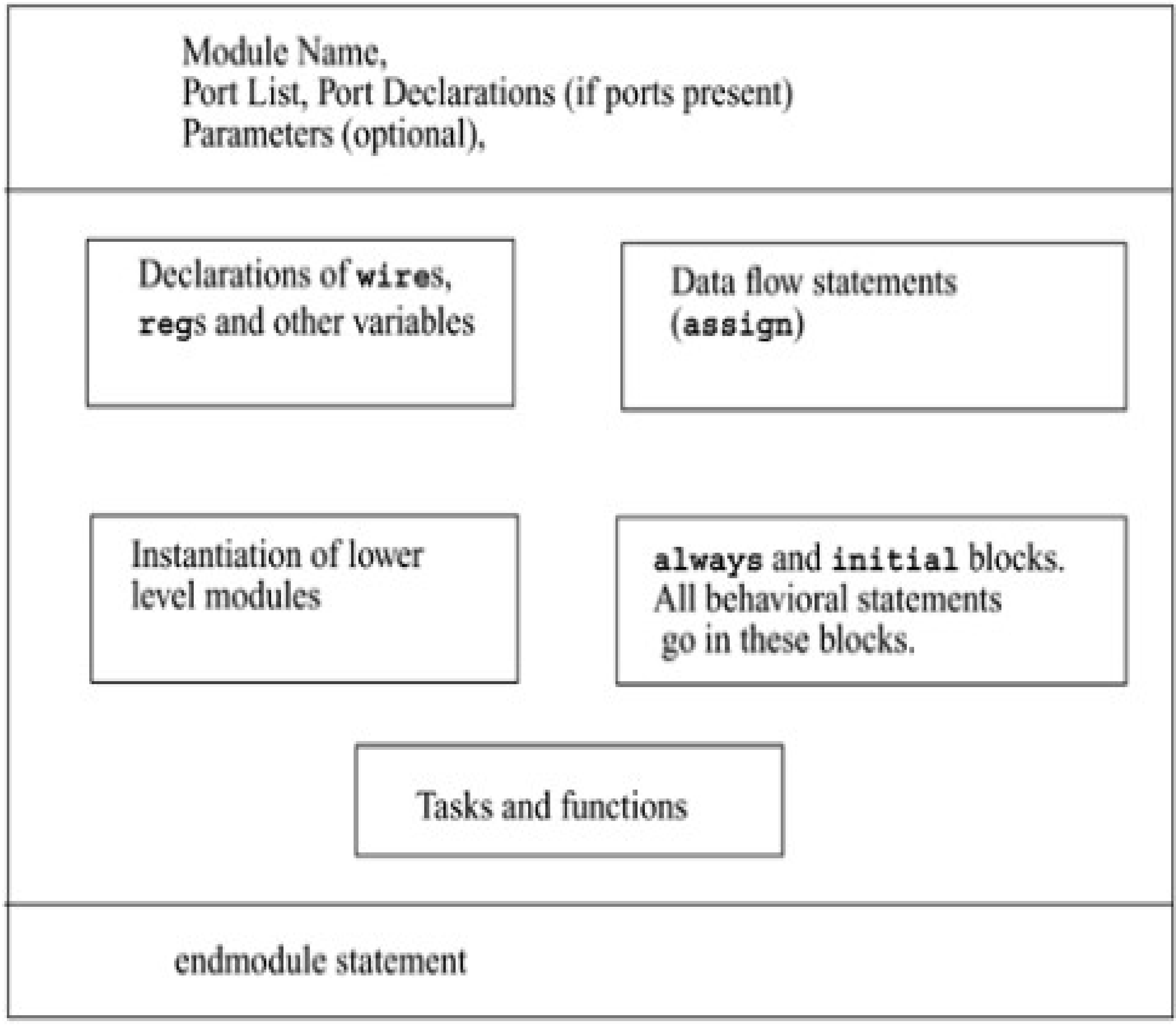
Module in Verilog consists of distinct parts,

Ports provide the interface by which a module can communicate

list of ports module contains list of input / output ports



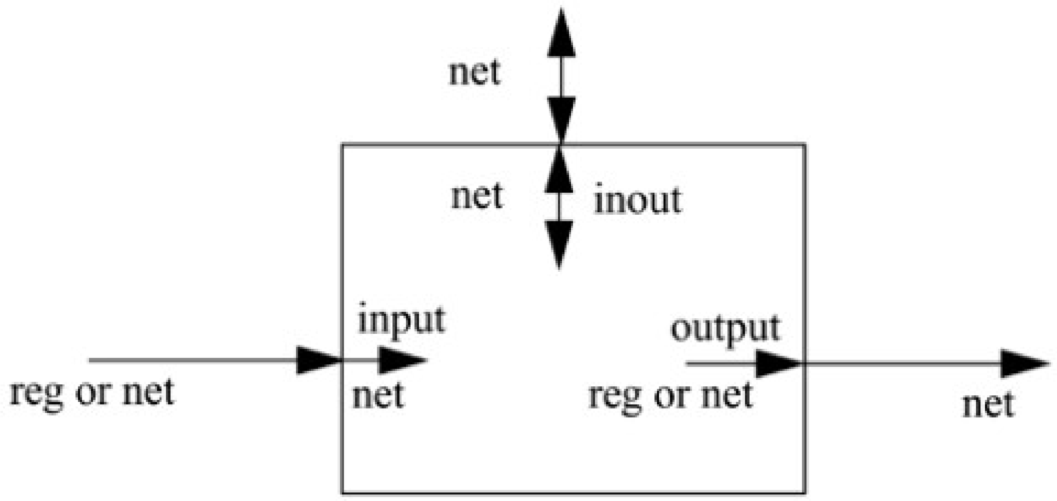
Verilog Keyword	Type of Port
input	Input port
output	Output port
inout	Bidirectional port


Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement

Port connection rule

port are (a) internal to the module (b) external to the module

when modules are instantiated within other follow rules



Connecting by ordered list module instantiation in the same order as the ports in the port list in the module definition

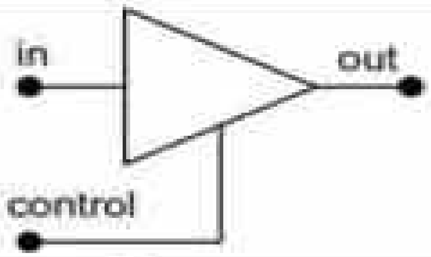
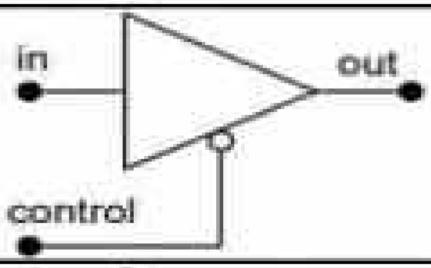
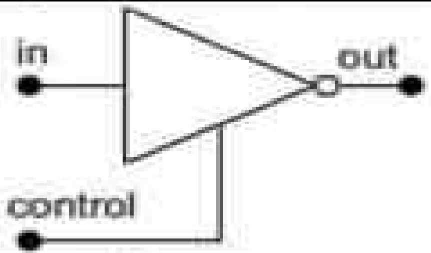
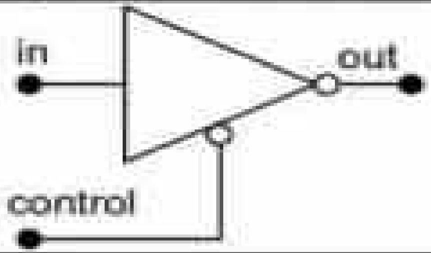
Connecting ports by name capability to connect external signals to ports by the port names, rather than by position.

Gate Level Modeling

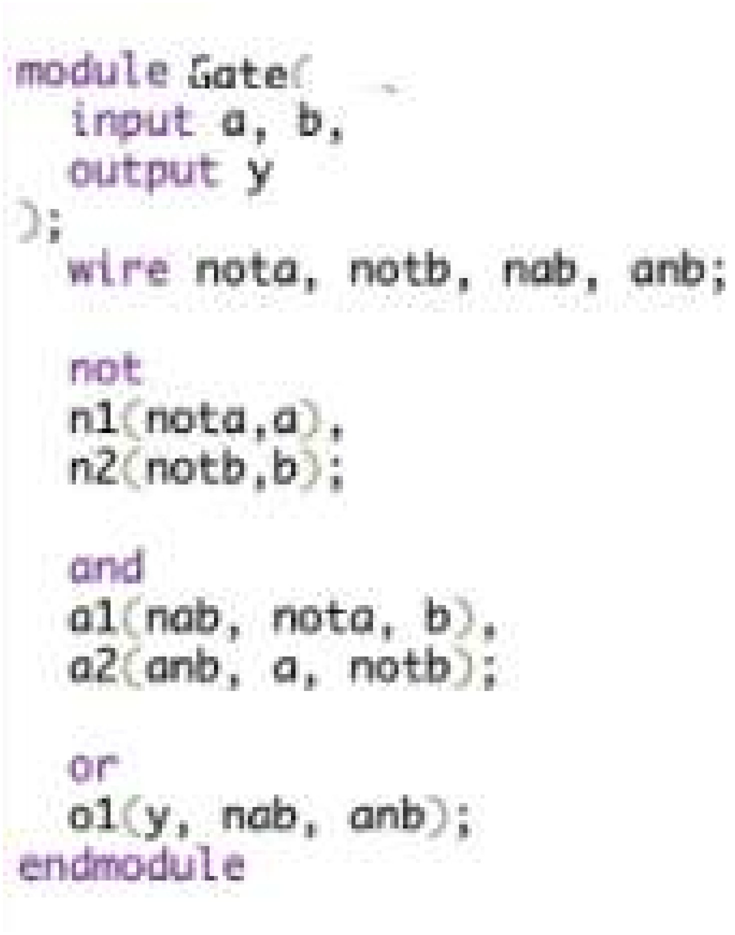
Logic gate and, or, not, nand, nor, xor, xnor are reserved word

Gate instanced by name

Gate port follow output than inputs

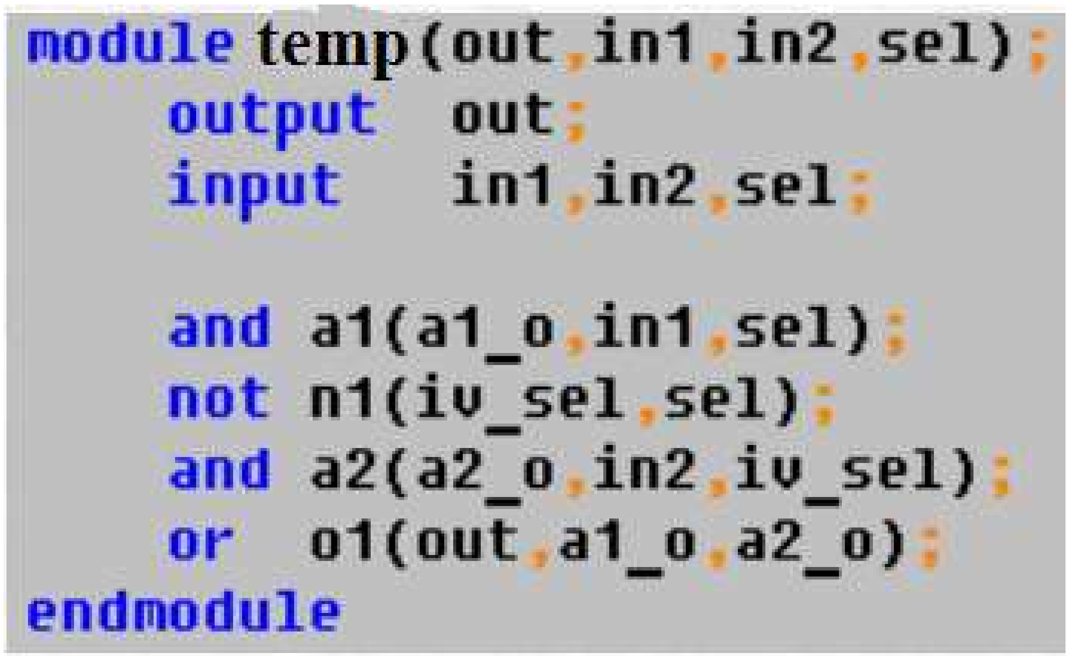
Typical instantiation	Functional representation	Functional description
`bufif1 (out, in, control);`		Out = in if control = 1; else out = z
`bufif0 (out, in, control);`		Out = in if control = 0; else out = z
`notif1 (out, in, control);`		Out = complement of in if control = 1; else out = z
`notif0 (out, in, control);`		Out = complement of in if control = 0; else out = z

Following program presents



```
module Gate(
   input a, b,
   output y
);
   wire nota, notb, nab, anb;

   not
   n1(nota,a),
   n2(notb,b);

   and
   a1(nab, nota, b),
   a2(anb, a, notb);

   or
   o1(y, nab, anb);
endmodule
```

Following program presents



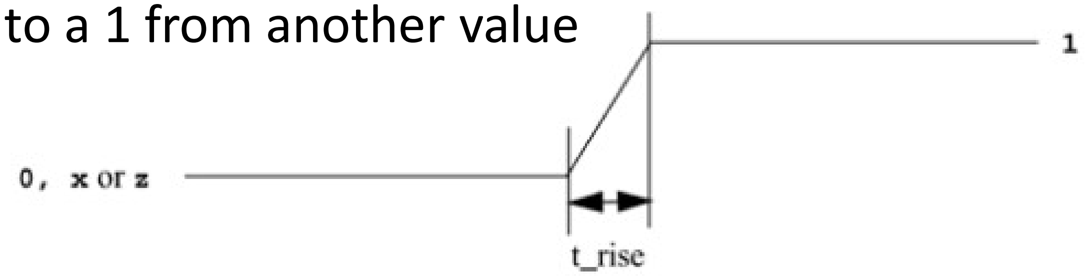
```
module temp(out,in1,in2,sel);
    output  out;
    input   in1,in2,sel;

    and a1(a1_o,in1,sel);
    not n1(iv_sel,sel);
    and a2(a2_o,in2,iv_sel);
    or  o1(out,a1_o,a2_o);
endmodule
```

Gate Delays

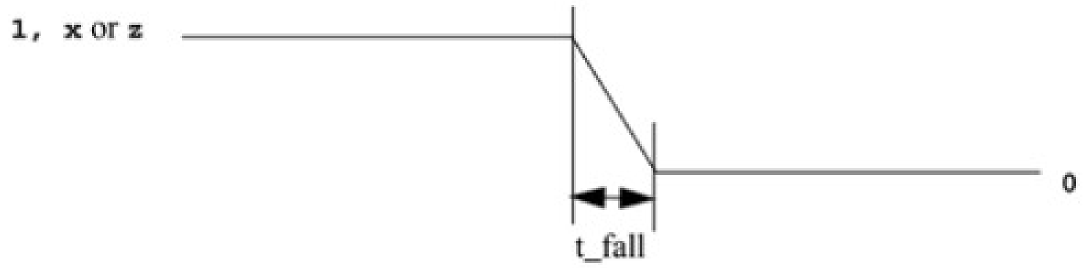
In real circuits, logic gates have delays associated with them.

Gate delays allow the Verilog user to specify delays through the logic circuits

Rise delay is gate output transition to a 1 from another value



Fall delay is gate output transition to a 0 from another value



Turn-off delay is gate output transition to the high impedance value (z) from another value

If the value changes to x, the minimum of the three delays is considered

Rise, Fall, and Turn-off Delay Specification

```
and #(rise_val, fall_val, turnoff_val) b1 (out, in, control);
```

Rise and Fall Delay Specification

```
and #(rise_val, fall_val) a2(out, i1, i2);
```

Delay of delay_time for all transitions

```
and #(delay_time) a1(out, i1, i2);
```

no delays are specified →zero delay

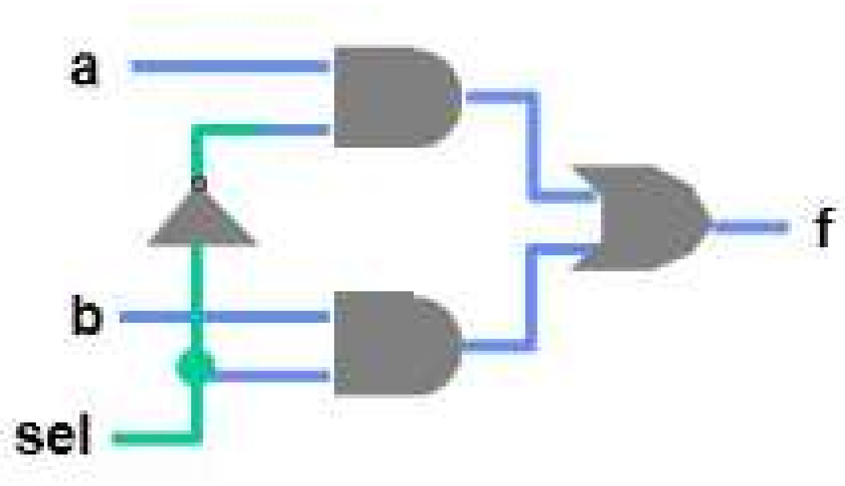
one delay is specified → same for all transition

two delays are specified→ rise and fall delay, turn-off delay is the minimum of the two delays.

three delays are specified → rise, fall, and turn-off delay values.

```
and #(5) a1(out, i1, i2);                  //Delay of 5 for all transitions
and #(4,6) a2(out, i1, i2);                // Rise = 4, Fall = 6
bufif0 #(3,4,5) b1 (out, in, control);     // Rise = 3, Fall = 4, Turn-off = 5
```

```
not # 2 n0(not_cnt, cnt);
and #(2,3) a0(a0_out, a, not_cnt);
 or #(3,2) o0(out, a0_out, a1_out);
```

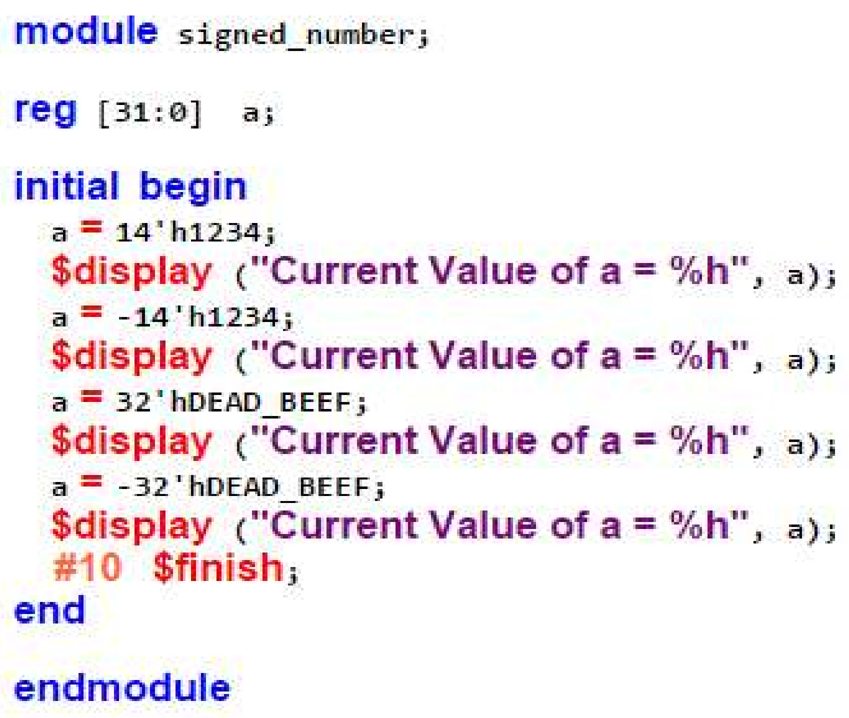


Delay of NOT 1
AND 2
OR 1

Practice

```
module buf_gate ();
reg in;
wire out;
buf #(5) (out,in);
initial begin
 $monitor ("Time = %d in = %b out=%b", $time, in, out);
 in = 0;
 #10 in = 1;
#10 in = 0;
 #10 $finish;
end
endmodule
```

Example



```
module signed_number;

reg [31:0]  a;

initial begin
  a = 14'h1234;
  $display ("Current Value of a = %h", a);
  a = -14'h1234;
  $display ("Current Value of a = %h", a);
  a = 32'hDEAD_BEEF;
  $display ("Current Value of a = %h", a);
  a = -32'hDEAD_BEEF;
  $display ("Current Value of a = %h", a);
  #10  $finish;
end

endmodule
```

Min/Typ/Max Values of Delay

For each type of delay three values, min, typ, and max, can be specified

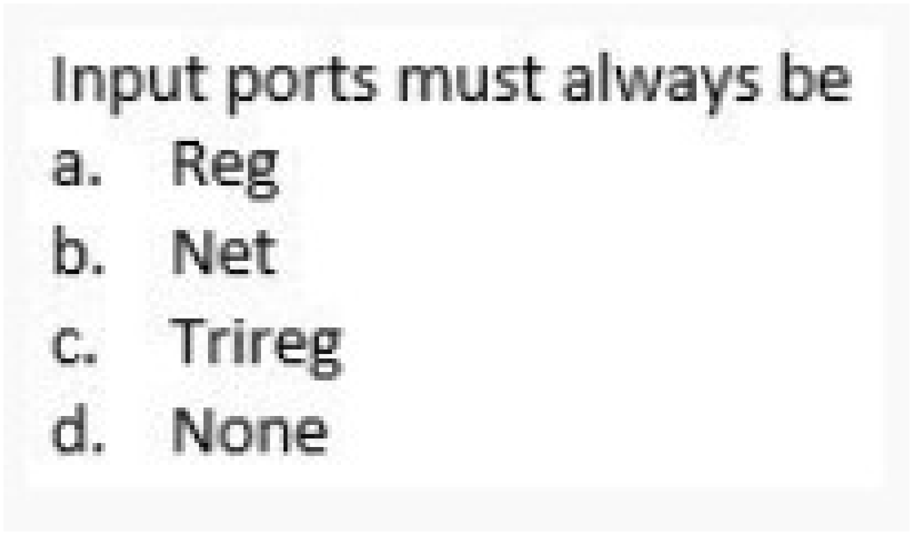
Min → minimum delay value that the designer expects the gate to have

Typ→ typical delay value that the designer expects the gate to have.

Max → maximum delay value that the designer expects the gate to have

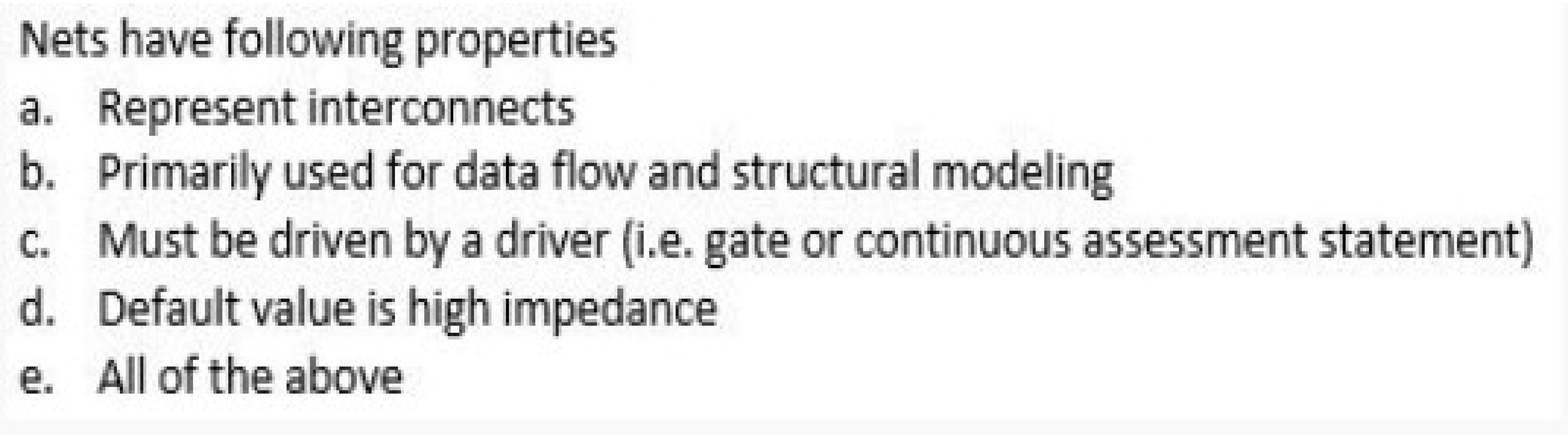
```
and #(4:5:6) a1(out, i1, i2);

and #(3:4:5, 5:6:7) a2(out, i1, i2);

and #(2:3:4, 3:4:5, 4:5:6) a3(out, i1,i2);
```

QUIZ



Input ports must always be

a. Reg
b. Net
c. Trireg
d. None

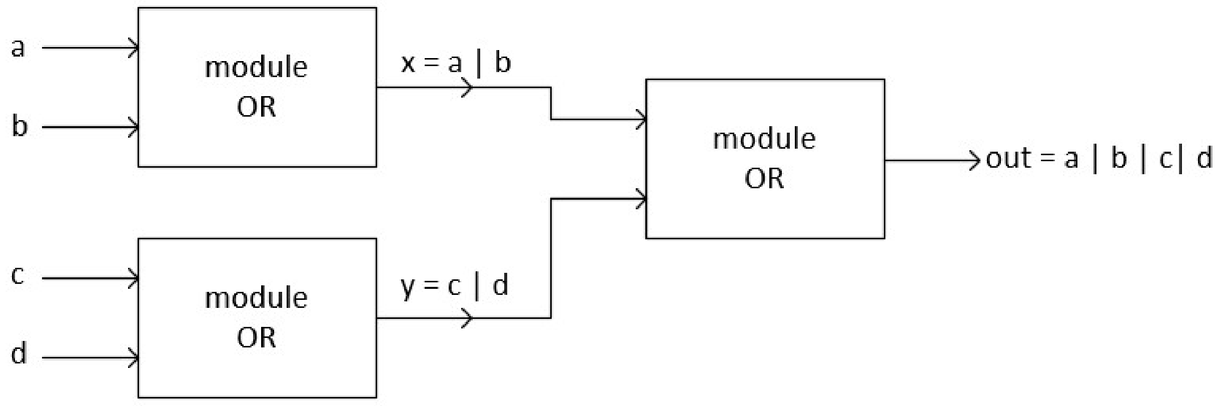


Nets have following properties

a. Represent interconnects
b. Primarily used for data flow and structural modeling
c. Must be driven by a driver (i.e. gate or continuous assessment statement)
d. Default value is high impedance
e. All of the above

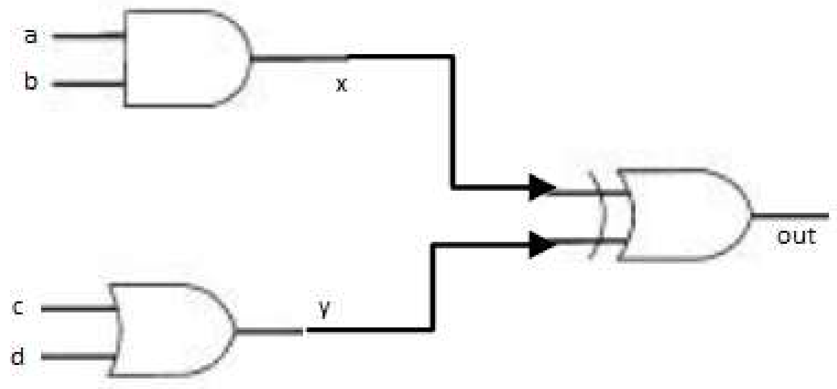
Verilog Program

four input OR gate



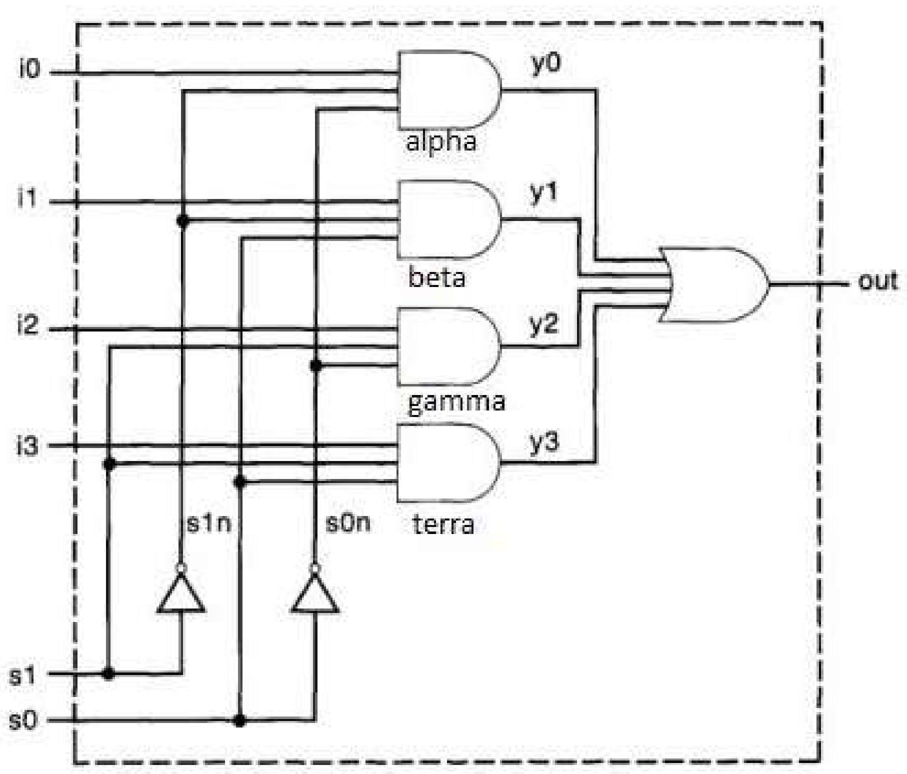
module orgate(out, a, b, c, d);
input a, b, c, d;
wire x, y; **output** out;
or or1(x, a, b);
or or2(y, c, d);
or or3(out, x, y);
endmodule

```
module example_2_bl(out, a, b, c, d);
input a, b, c, d;
output out;
wire x, y;
 and gate_1(x, a, b);
or gate_2(y, c, d);
xor gate_3(out, x, y);
endmodule
```



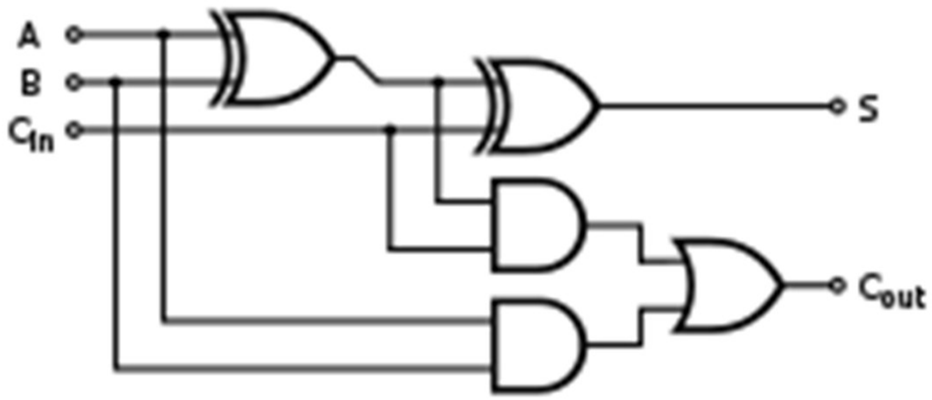
4×1 Multiplexer using gate level Modeling

```
module example_3_bl(out, i0, i1, i2, i3, s1, s0);
input i0, i1, i2, i3, s1, s0;
output out;
wire y0, y1, y2, y3, s1n, s0n;
not n1(s1n, s1);
not n2(s0n, s0);
and alpha(y0, i0, s1n, s0n);
and beta(y1, i1, s1n, s0);
and gamma(y2, i2, s1, s0n);
and terra(y3, i3, s1, s0);
or out2(out, y0, y1, y2, y3);
endmodule
```



Full Adder in Gate Level

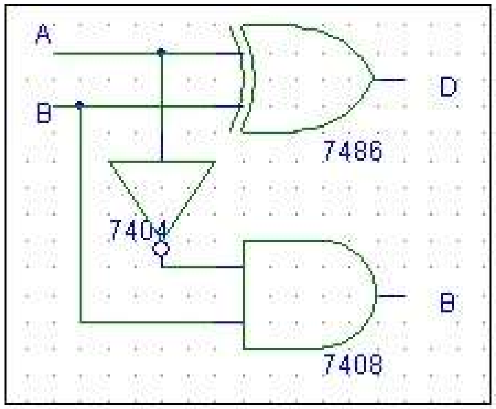
```
module fa(s,co,a,b,ci);
output s,co;
input a,b,ci;
Wire n1,n2,n2;
xor1 u1(s,a,b,ci);
and1 u2(n1,a,b);
and1 u3(n2,b,ci);
and1 u4(n3,a,ci);
or1 u5(co,n1,n2,n3);
endmodule
```

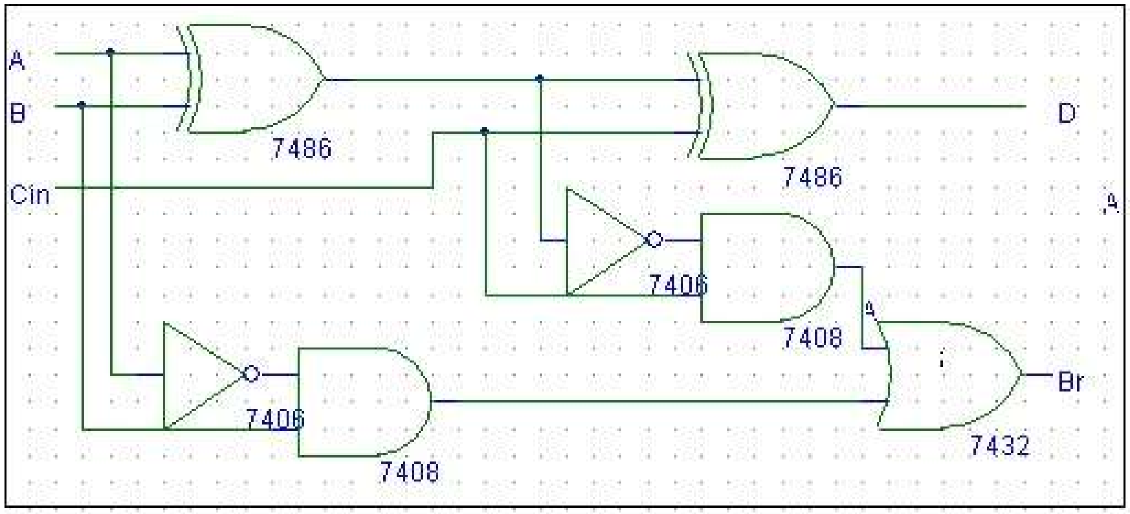


```
module full_adder(x,y,cin,s,cout);
input x,y,cin;
output s,cout;
wire s1,c1,c2;
half_adder ha1(x,y,s1,c1);
half_adder ha2(cin,s1,s,c2);
or(cout,c1,c2); endmodule
```

```
module half_adder(x,y,s,c);
input x,y;
output s,c;
Xor x1(s,x,y);
And a1(c,x,y);
endmodule
```

Subtractor

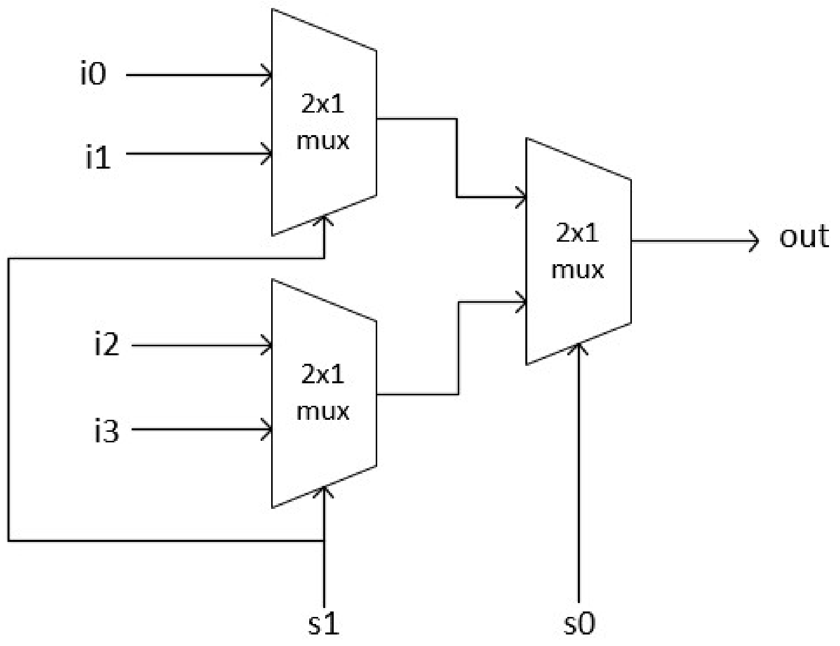


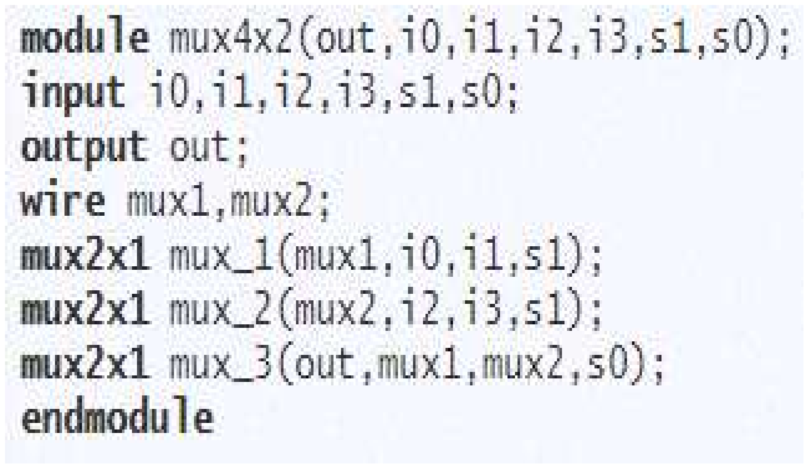


```
module hs ( a, b, d, br)
input a, b;
output d, br;
assign d= a ^ b;
assign br= ~a & b;
end module
```

```
module fs ( a, b, c, d, br)
input a, b, c;
output d, br;
assign d= a ^ b ^ c;
assign br=(( ~a)& (b ^ c)) | (b & c);
end module
```

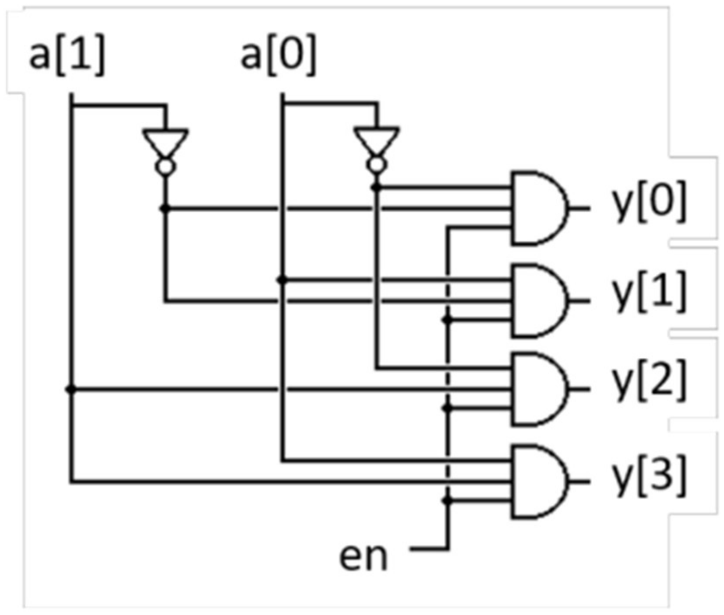
4:1 mux using 2:1 mux





```
module mux4x2(out,i0,i1,i2,i3,s1,s0);
input i0,i1,i2,i3,s1,s0;
output out;
wire mux1,mux2;
mux2x1 mux_1(mux1,i0,i1,s1);
mux2x1 mux_2(mux2,i2,i3,s1);
mux2x1 mux_3(out,mux1,mux2,s0);
endmodule
```

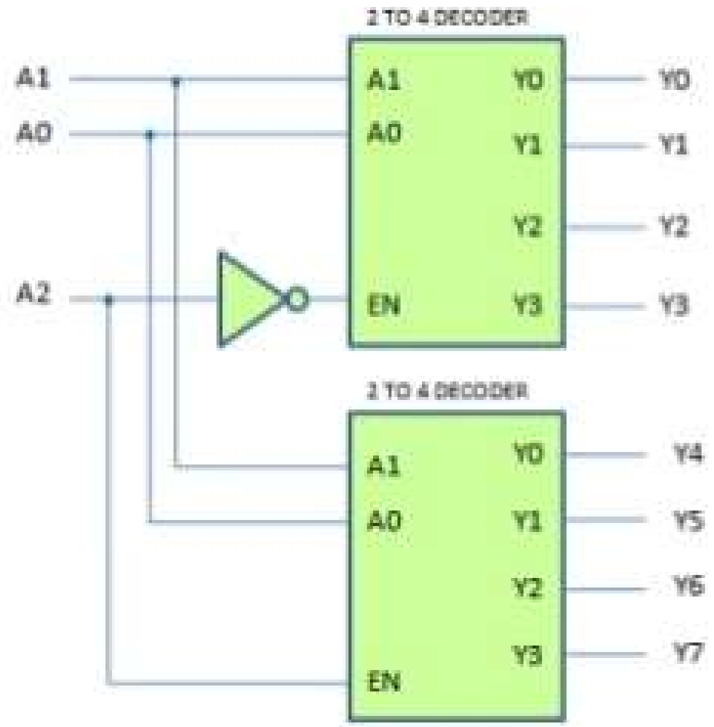
2:4 Decoder



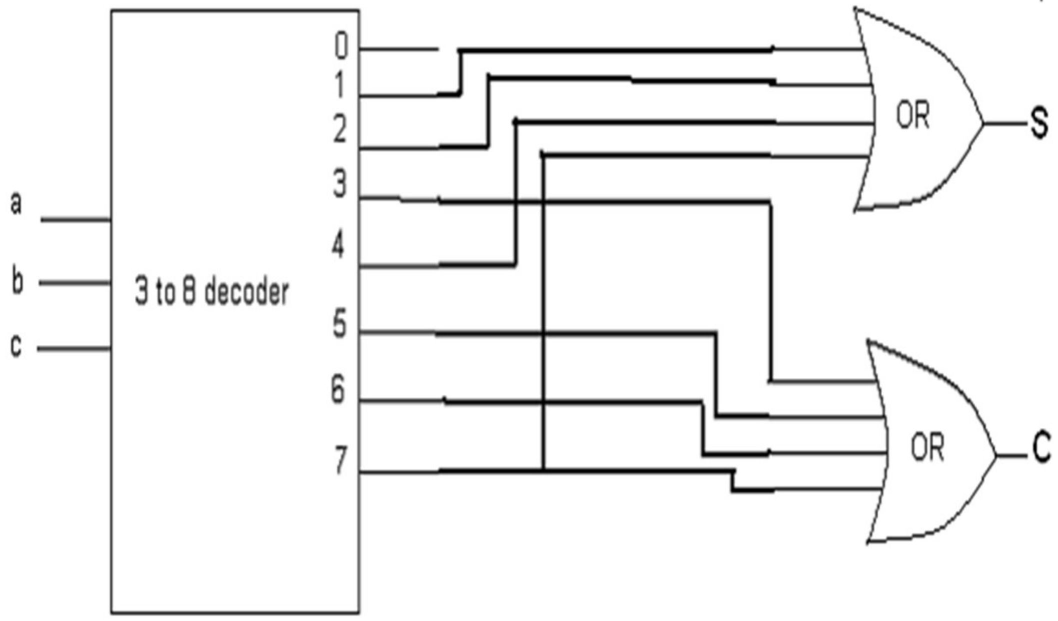
```
module dec24_str(
output [3:0] y,
input [1:0] a,
input en);
and (y[0], ~a[1], ~a[0], en);
and (y[1], ~a[1], a[0], en);
and (y[2], a[1], ~a[0], en);
and (y[3], a[1], a[0], en);
endmodule
```

```
module dec2_4 (a,b,en,y0,y1,y2,y3)
input a, b, en;
output y0,y1,y2,y3;
assign y0= (~a) & (~b) & en;
assign y1= (~a) & b & en;
assign y2= a & (~ b) & en;
assign y3= a & b & en;
end module
```

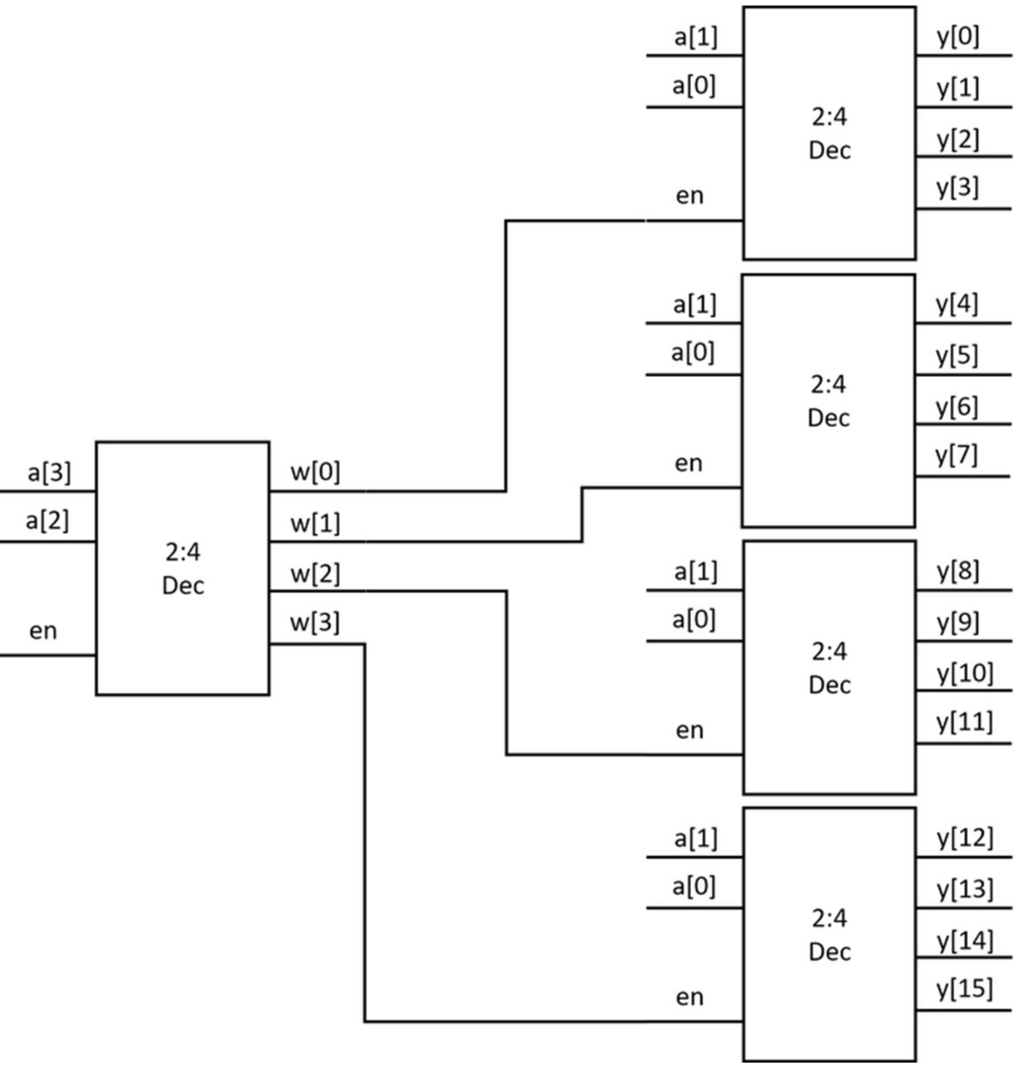
3:8 Decoder using 2:4 Decoder



Full Adder using 3:8 Decoder

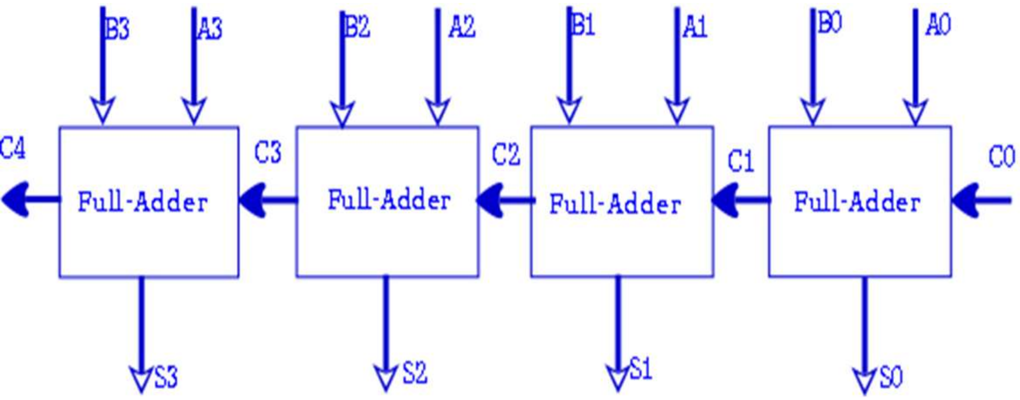


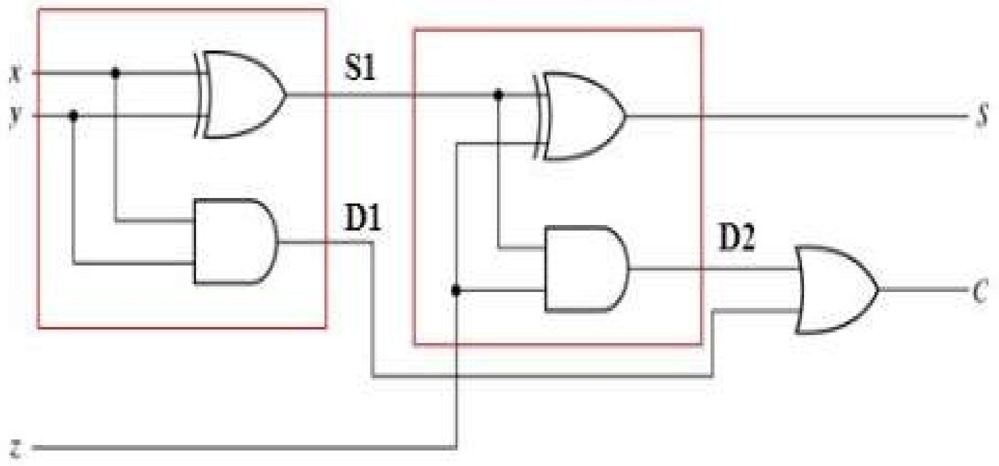
4:16 Decoder using 2:4 Decoder



```
module dec4x16_str(a,en,y);
output [15:0] y,
input [3:0] a,
input en;
wire [3:0] w;
dec2x4_str u0(w, a[3:2], en);
dec2x4_str u1(y[3:0], a[1:0], w[0]);
dec2x4_str u2(y[7:4], a[1:0], w[1]);
dec2x4_str u3(y[11:8], a[1:0], w[2]);
dec2x4_str u4(y[15:12], a[1:0], w[3]);
endmodule
```

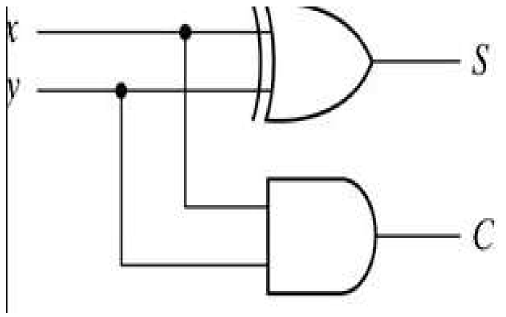
4 bit Ripple Carry Adder





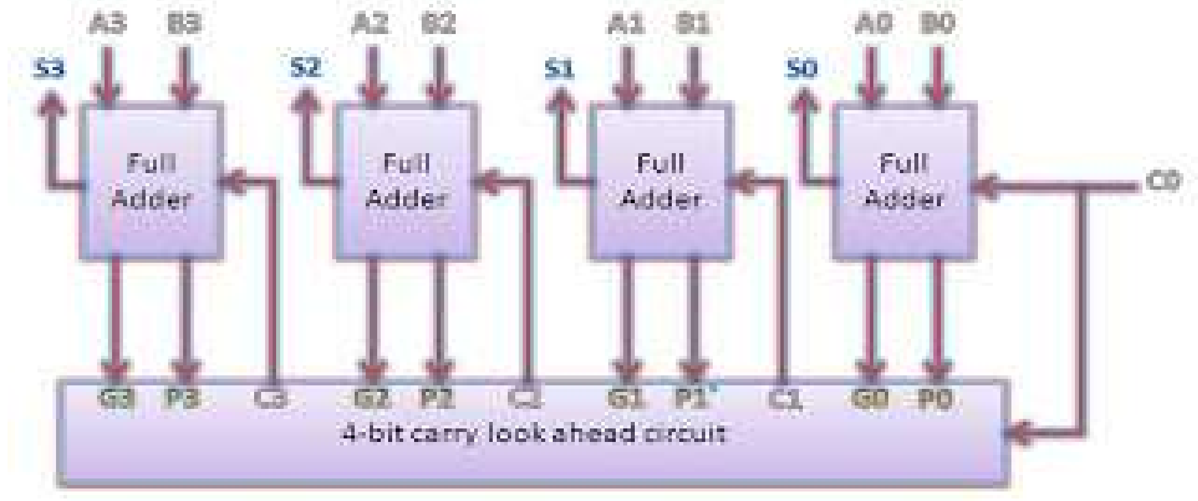
```
module fulladder (S,C,x,y,z);
   input x,y,z;
   output S,C;
   wire S1,D1,D2; //Outputs of
//Instantiate the halfadder
    halfadder HA1 (S1,D1,x,y),
              HA2 (S,D2,S1,z);
    or g1(C,D2,D1);
endmodule
```

```
module four_bit_adder(x,y,cin,sum,cout);
input [3:0] x,y;
input cin;
output[3:0] sum;
output cout;
wire c1,c2,c3;
full_adder fa1(x[0],y[0],cin,sum[0],c1);
full_adder fa2(x[1],y[1],c1,sum[1],c2);
full_adder fa3(x[2],y[2],c2,sum[2],c3);
full_adder fa4(x[3],y[3],c3,sum[3],cout);
endmodule
```



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

Carry Look ahead Adder



4-bit carry look ahead adder

Pi = Ai xor Bi

Gi = Ai.Bi

Si = Pi xor Ci

Ci+1 = Ci.Pi + Gi

C1 = C0P0 + G0
C2 = C1P1 + G1 = (C0P0+G0)P1 + G1 = C0P0P1 + P1G0 + G1
C3 = C2P2 +G2 = (C0P0P1 + P1G0 + G1)P2+G2 = C0P0P1P2 + P2P1G0 + P2G1 + G2
C4 = C3P3 +G3 = (C0P0P1P2 + P2P1G0 + P2G1 + G2)P3 + G3 = C0P0P1P2P3 + P3P2P1G0 + P3P2G1 + G2P3 + G3

Similar to Ripple carry Adder It is used to add together two binary numbers; carry look ahead adders are able to calculate the Carry bit before the Full Adder is done with its operation.

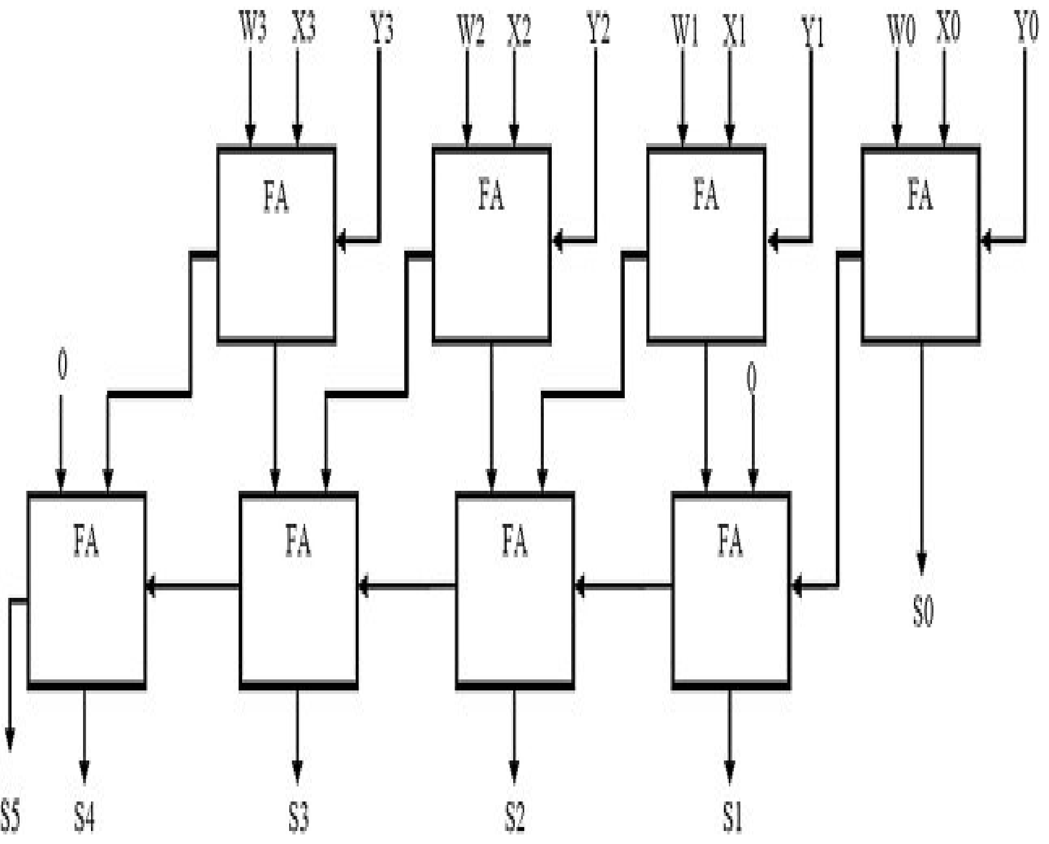
This gives it an advantage over the Ripple Carry Adder because it is able to add two numbers together faster. The drawback is that it takes more logic

CLA

```
module CLA4 (A, B, Ci, S, Co, PG, GG);
   input[3:0] A;
   input[3:0] B;
   input Ci;
   output[3:0] S;
   output Co;
   output PG;
   output GG;

   wire[3:0] G;
   wire[3:0] P;
   wire[3:1] C;
   CLALogic CarryLogic (G, P, Ci, C, Co, PG, GG);
   GPFullAdder FA0 (A[0], B[0], Ci, G[0], P[0], S[0]);
   GPFullAdder FA1 (A[1], B[1], C[1], G[1], P[1], S[1]);
   GPFullAdder FA2 (A[2], B[2], C[2], G[2], P[2], S[2]);
   GPFullAdder FA3 (A[3], B[3], C[3], G[3], P[3], S[3]);
endmodule
```

```
module CLALogic (G, P, Ci, C, Co, PG, GG);
   input[3:0] G;
   input[3:0] P;
   input Ci;
   output[3:1] C;
   output Co;
   output PG;
   output GG;

   wire GG_int;
   wire PG_int;

   assign C[1] = G[0] | (P[0] & Ci) ;
   assign C[2] = G[1] | (P[1] & G[0]) | (P[1] & P[0] & Ci) ;
   assign C[3] = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) | (P[2] & P[1] &
                 P[0] & Ci) ;
   assign PG_int = P[3] & P[2] & P[1] & P[0] ;
   assign GG_int = G[3] | (P[3] & G[2]) | (P[3] & P[2] & G[1]) | (P[3] & P[2] &
                   P[1] & G[0]) ;
   assign Co = GG_int | (PG_int & Ci) ;
   assign PG = PG_int ;
   assign GG = GG_int ;
endmodule

module GPFullAdder (X, Y, Cin, G, P, Sum);
   input X;
   input Y;
   input Cin;
   output G;
   output P;
   output Sum;

   wire P_int;

   assign G = X & Y ;
   assign P = P_int ;
   assign P_int = X ^ Y ;
   assign Sum = P_int ^ Cin ;
endmodule
```

Carry Save Adder



```
module fulladder( a,b,cin,sum,carry);
input a,b,cin;
output sum,carry;
assign sum = a ^ b ^ cin;
assign carry = (a & b) | (cin & b) | (a & cin);
endmodule
module CSA ( x,y,z,s,cout);
  input [3:0] x,y,z;
          output [4:0] s;
          output cout;
wire [3:0] c1,s1,c2;
fulladder fa_inst10(x[0],y[0],z[0],s1[0],c1[0]);
fulladder fa_inst11(x[1],y[1],z[1],s1[1],c1[1]);
fulladder fa_inst12(x[2],y[2],z[2],s1[2],c1[2]);
fulladder fa_inst13(x[3],y[3],z[3],s1[3],c1[3]);

fulladder fa_inst20(s1[1],c1[0],1'b0,s[1],c2[1]);
fulladder fa_inst21(s1[2],c1[1],c2[1],s[2],c2[2]);
fulladder fa_inst22(s1[3],c1[2],c2[2],s[3],c2[3]);
fulladder fa_inst23(1'b0,c1[3],c2[3],s[4],cout);
assign s[0] = s1[0];
endmodule
```

Dataflow Modeling

Preferred for small circuit

circuit design in terms of the data flow between rather than instantiation of individual gates

Continuous Assignments

LHS must scalar or vector **net**

```
assign out = i1 & i2;
```

```
wire out;
assign out = in1 & in2;
```

Implicit Continuous Assignment

Assignment during declaration

```
wire out = in1 & in2;
```

Implicit Net Declaration

inferred for that signal name

```
wire i1, i2;
assign out = i1 & i2;
```

Delay (# value) ns

Control time between change in a RHS operand and new value is assigned to LHS

Regular Assignment Delay (inertial delay) (INTRA Delay)

```
assign #10 out = in1 & in2;
```

change in values of in1 or in2 will result in a delay of 10 time units before computation and the result will be assigned to out

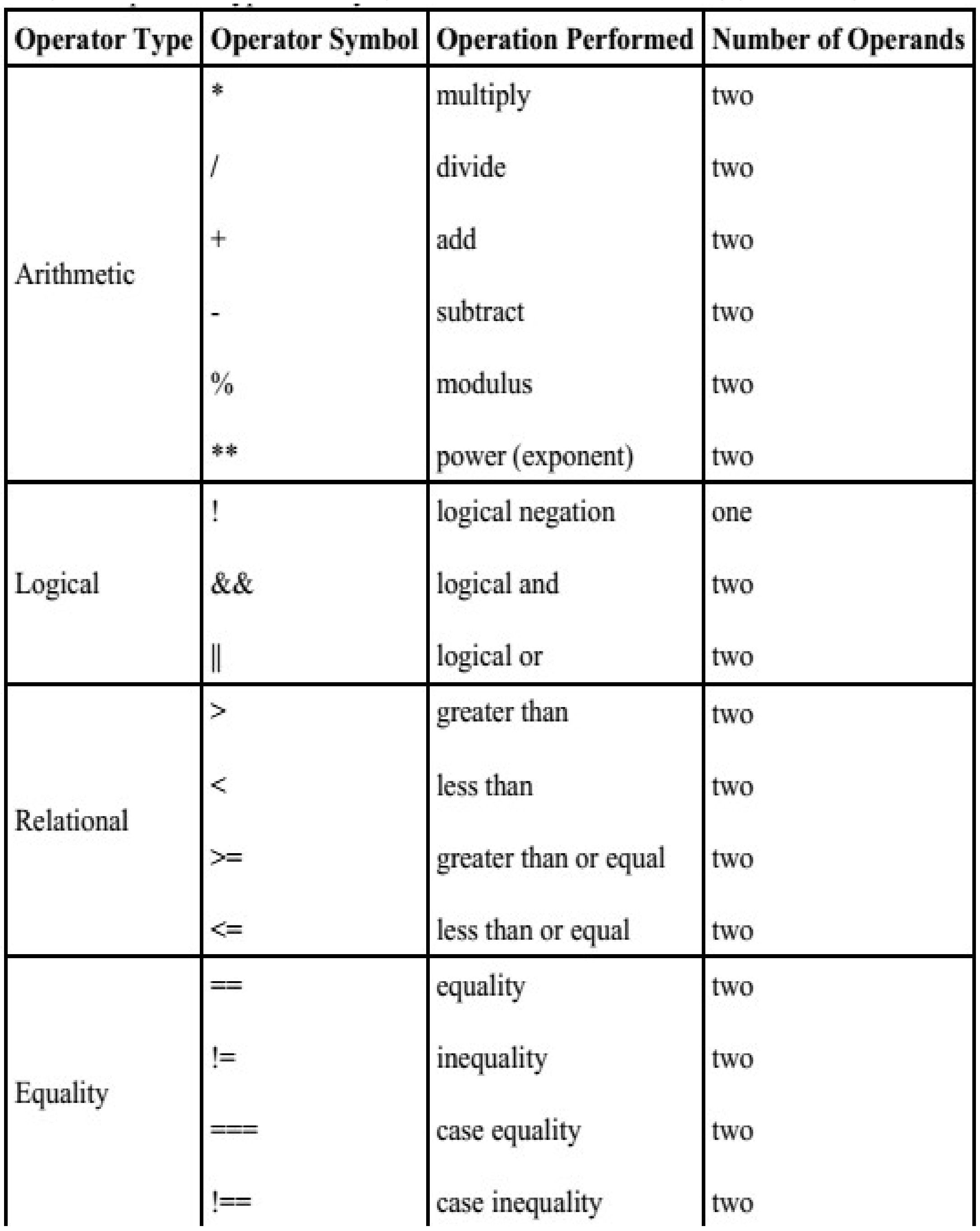
INTER Delay

```
#10 assign out = in1 & in2;
```

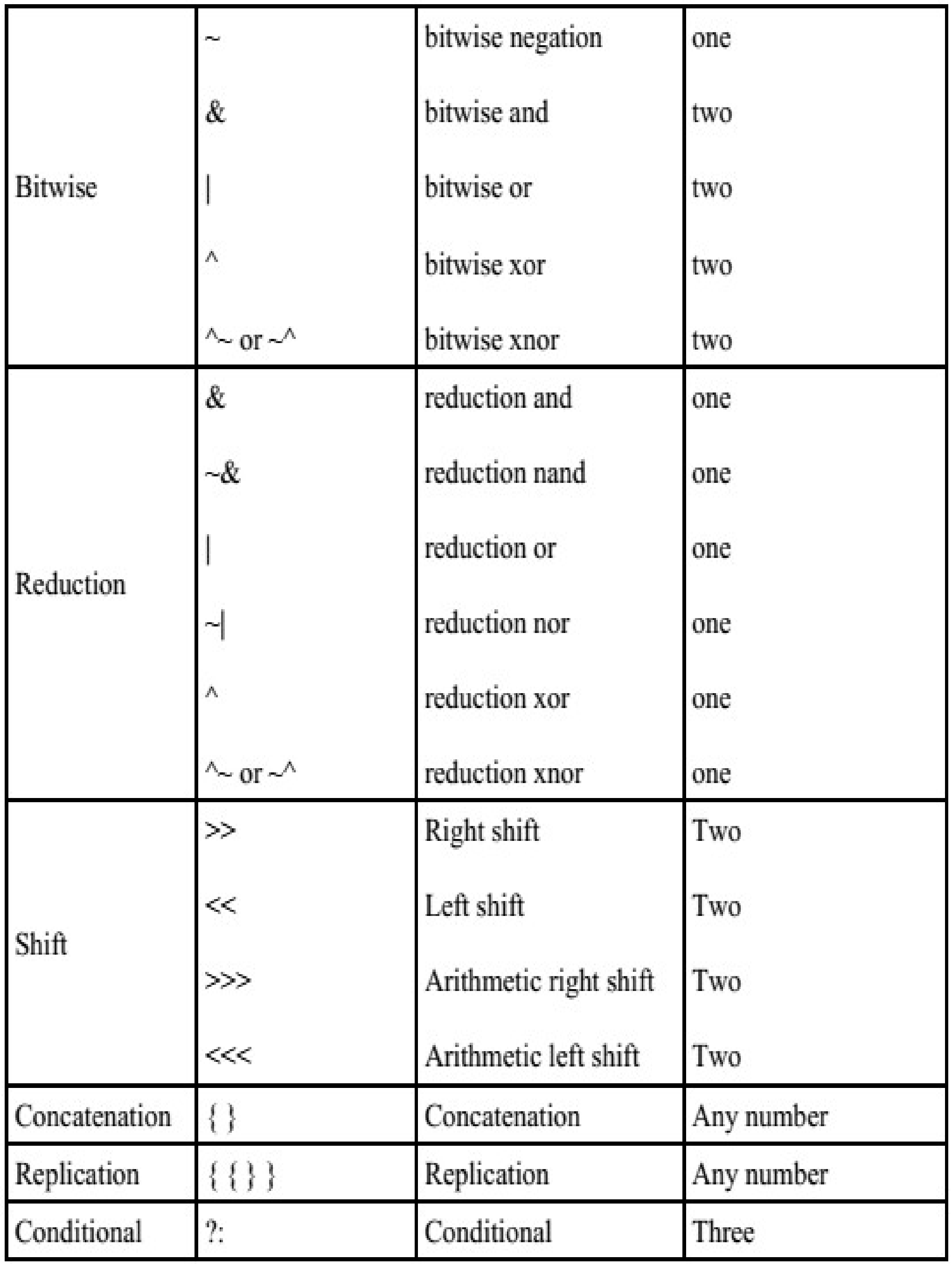
Statement out delay by 10 time units

Operator -- Operand

Operands can be any one of the data types Operators act on the operands to produce desired results



Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two
Logical	!	logical negation	one
	&&	logical and	two
	||	logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two



Bitwise	~	bitwise negation	one
	&	bitwise and	two
	|	bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one
Shift	>>	Right shift	Two
	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	?:	Conditional	Three

Arithmetic Operators multiply (*), divide (/), add (+), subtract (-), power (**),and modulus (%)

```
A = 4'b0011; B = 4'b0100; // A and B are register vectors
D = 6; E = 4; F=2// D and E are integers

A * B // Multiply A and B. Evaluates to 4'b1100
D / E // Divide D by E. Evaluates to 1. Truncates any fractional part.
A + B // Add A and B. Evaluates to 4'b0111
B - A // Subtract A from B. Evaluates to 4'b0001
F = E ** F; //E to the power F, yields 16
```

If any operand bit has a value x, then the result of the entire expression is x

```
in1 = 4'b101x;
in2 = 4'b1010;
sum = in1 + in2; // sum will be evaluated to the value 4'bx
```

Logical Operators logical-and (&&), logical-or (||) and logical-not (!)
always evaluate to a 1-bit value, 0 (false), 1 (true), or x (ambiguous).
If any operand bit is x or z, it is equivalent to x

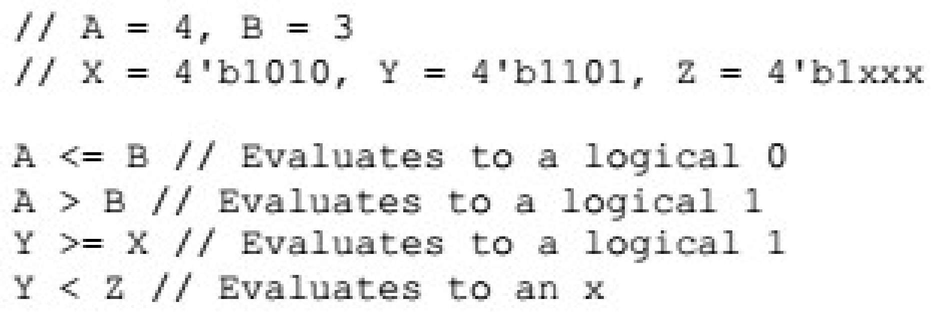
```
A = 3; B = 0;
A && B // Evaluates to 0. Equivalent to (logical-1 && logical-0)
A || B // Evaluates to 1. Equivalent to (logical-1 || logical-0)
!A// Evaluates to 0. Equivalent to not(logical-1)
!B// Evaluates to 1. Equivalent to not(logical-0)
```

```
A = 2'b0x; B = 2'b10;
A && B // Evaluates to x.
```

Relational Operators greater-than (>), less-than (<), greater-than-or-equal-to (>=), and less-than-or-equal-to (<=)

Evaluate to logical value of 1 if expression is true and 0 if expression is false

If any unknown or z bits in the operands evaluates to x

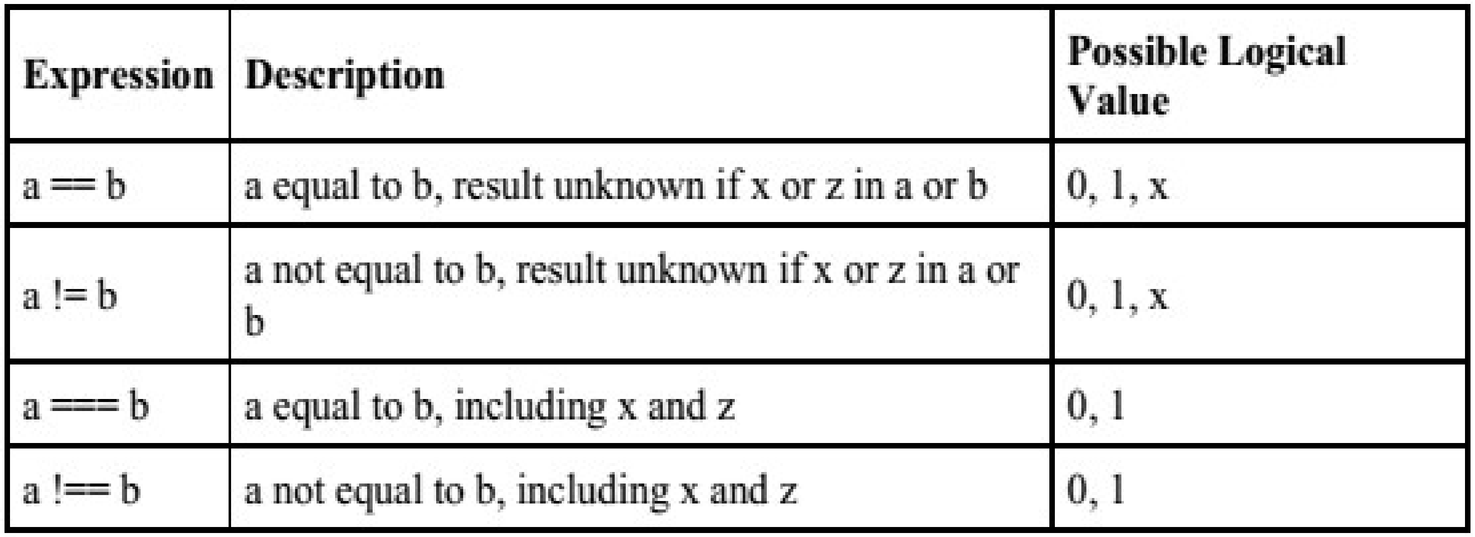


```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx

A <= B // Evaluates to a logical 0
A > B // Evaluates to a logical 1
Y >= X // Evaluates to a logical 1
Y < Z // Evaluates to an x
```

Equality Operators logical equality (==), logical inequality (!=), case equality (===), and case inequality (!==)

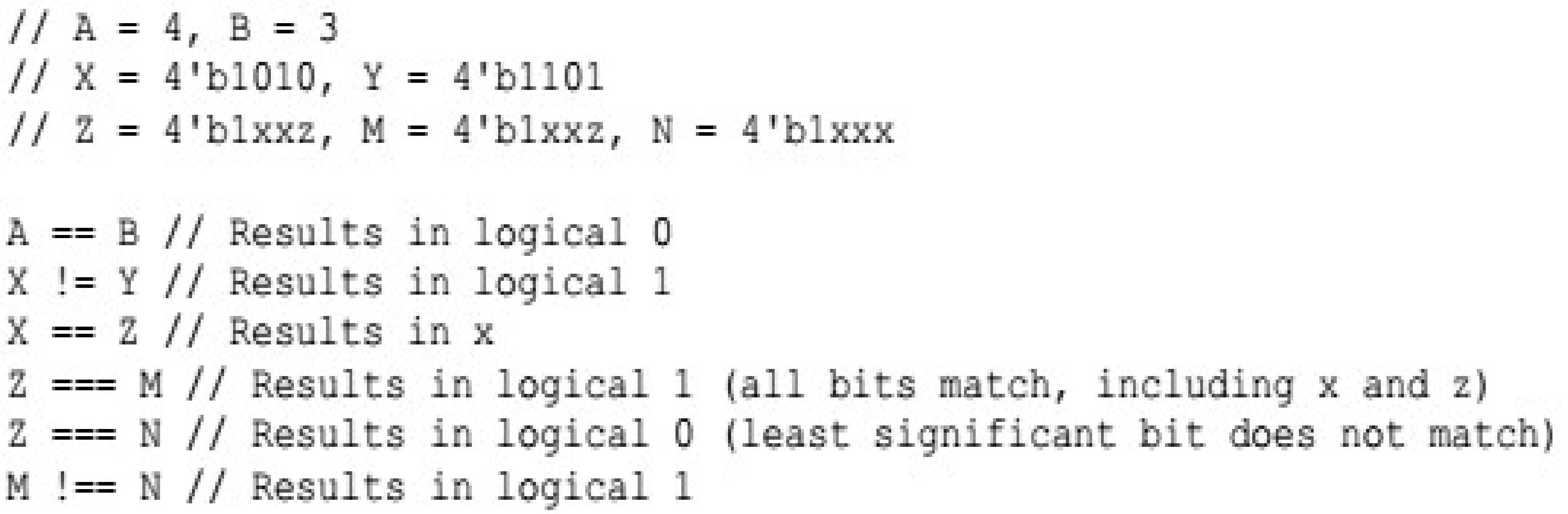
Evaluates to 1 if true, 0 if false.

Compare two operands bit by bit, zero filling if operands are unequal length



Expression	Description	Possible Logical Value
a == b	a equal to b, result unknown if x or z in a or b	0, 1, x
a != b	a not equal to b, result unknown if x or z in a or b	0, 1, x
a === b	a equal to b, including x and z	0, 1
a !== b	a not equal to b, including x and z	0, 1

case equality compare both operands bit by bit including x and z



```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101
// Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx

A == B // Results in logical 0
X != Y // Results in logical 1
X == Z // Results in x
Z === M // Results in logical 1 (all bits match, including x and z)
Z === N // Results in logical 0 (least significant bit does not match)
M !== N // Results in logical 1
```

Bitwise Operators

Bitwise operators are negation (~), and(&), or (|), xor (^), xnor (^~, ~^).
Perform a bit-by-bit operation on two operands. T

```
X = 4'b1010, Y = 4'b0000

X | Y     // bitwise operation. Result is 4'b1010
X || Y    // logical operation. Equivalent to 1 || 0. Result is 1.
```

Reduction Operators

Reduction operators are and (&), nand (~&), or (|), nor (~|), xor (^), and xnor (~^, ^~).

Take only one operand

Perform a bitwise operation on a single vector operand and yield a 1-bit result.

```
X = 4'b1010

&X          //Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
|X          //Equivalent to 1 | 0 | 1 | 0. Results in 1'b1
^X          //Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0
```

A reduction xor or xnor can be used for even or odd parity generation respectively.

Shift Operators

right shift (>>), left shift (<<), arithmetic right shift (>>>), arithmetic left shift (<<<).

Shift a vector operand to the right or the left by a specified number of bits.

```
X = 4'b1100
Y = X >> 1;         //Y is 4'b0110. Shift right 1 bit. 0 filled in MSB position.
Y = X << 1;         //Y is 4'b1000. Shift left 1 bit. 0 filled in LSB position.
Y = X << 2;         //Y is 4'b0000. Shift left 2 bits.
```

Concatenation Operator

{, } provides a mechanism to append multiple operands.

```
A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110

Y = {B , C}                      // Result Y is 4'b0010
Y = {A , B , C , D , 3'b001}     // Result Y is 11'b10010110001
Y = {A , B[0], C[1]}             // Result Y is 3'b101
```

Replication Operator

concatenation of same number by using a replication constant

```
reg A;
reg [1:0] B, C;
reg [2:0] D;
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;

Y = { 4{A} }                     // Result Y is 4'b1111
Y = { 4{A} , 2{B} }              // Result Y is 8'b11110000
Y = { 4{A} , 2{B} , C }          // Result Y is 8'b1111000010
```

Conditional Operator

takes three operands.

condition_expr ? true_expr : false_expr ;

(condition_expr) is true true_expr is evaluated.
(Condition_expr) false false_expr is evaluated.

If the (Condition_expr) is x (ambiguous), both true_expr and false_expr are evaluated and their results are compared, bit by bit, return for each bit position an x if the bits are different and the value of the bits if they are the same

model functionality of a tristate buffer

```
assign addr_bus = drive_enable ? addr_out : 36'bz;
```

model functionality of a 2-to-1 mux

```
assign out = control ? in1 : in0;
```

Practice

For the following Verilog code segment:

```
wire [7:0] A;
wire B;
assign B = ^A;
```

if the value of A is 16’b10110011, what will be the value of {A[4:3], 3{B}}

a. 5’b10111 *
b. 5’b10000
c. 5’b01000
d. None of the above

Correct answer is (a).

FSM (Finite State Machine)

A system models with number of state

Finite number of state

High outputs corresponding to each transition

There are two types of finite state machines that generate output –

- Mealy Machine
- Moore machine

Mealy Machine
output depends on the present state as well as the present input.

Moore Machine
outputs depend on only the present state.

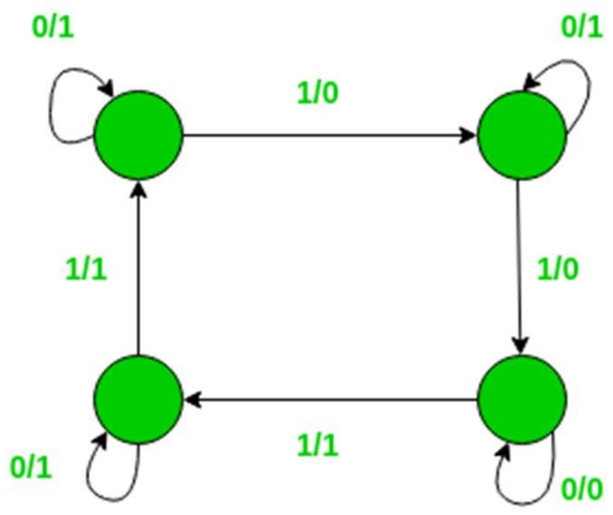


Figure - Mealy machine

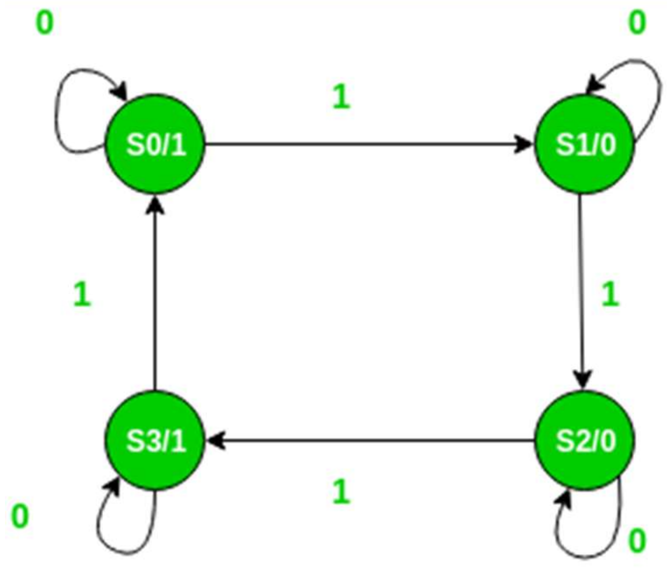


Figure - Moore machine

Sequence Detector

Generate high output when particular sequence detects

It can Mealy (Overlap or non-overlap) Moore (Overlap or non-overlap) type

Overlap FSM – Last bit of one sequence and first bit of next sequence same

QUIZ

An overlapping sequence detector (ab) will produces output for sequence length (aabbaababa)

A. 0010001010
B. 1010001010
C. 0010101010
D. 0010111010

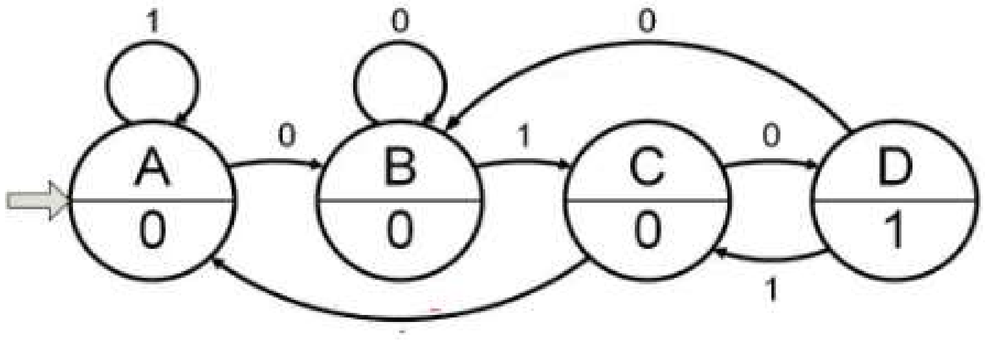
101 sequence detector (Mealy)	101 sequence detector (Moore)

Mealy sequence
Detector 1010

Moore sequence
Detector 1010

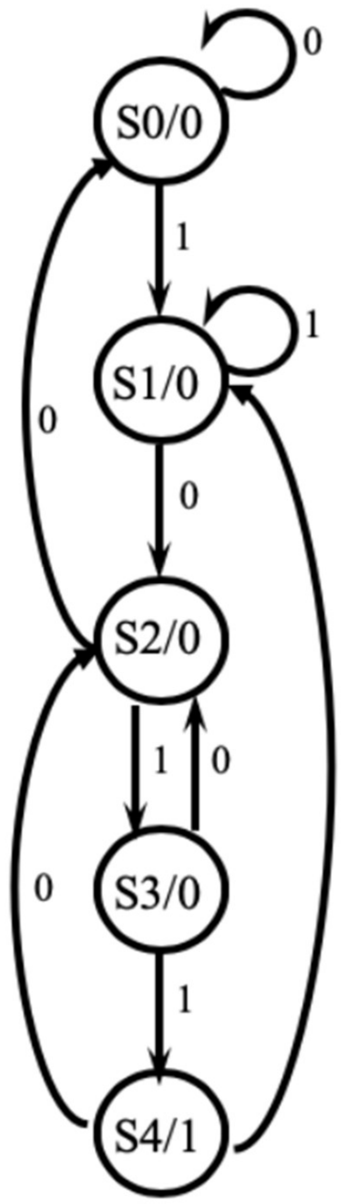
QUIZ

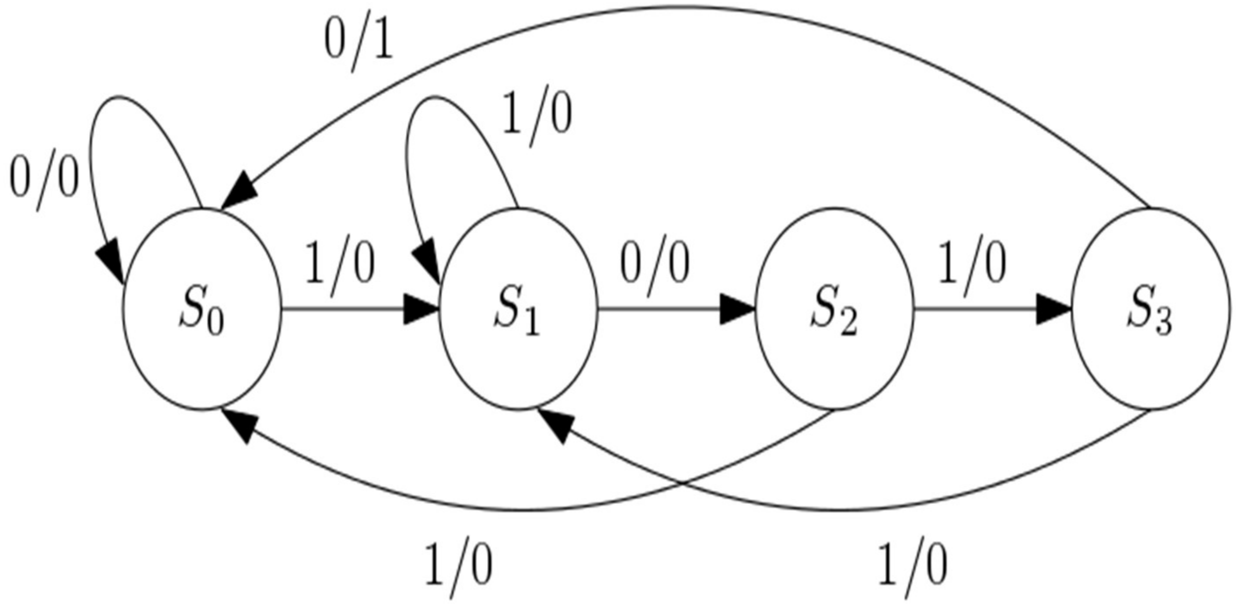
Following machine detects



A. 101 mealy
B. 101 moore
C. 010 mealy
D. 010 moore

Following machine detects





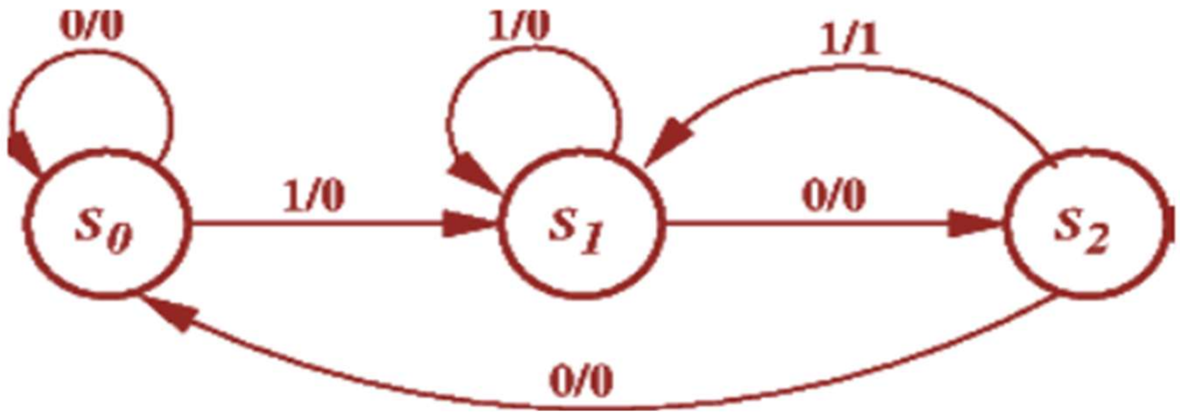
QUIZ

Number of state requires to implement moore machine ’101010’

A. 6
B. 7
C. 8
D. 5

Following sequence detector presents a

A. Mealy 101
B. Moore 101
C. Mealy 1101
D. Moore 1101



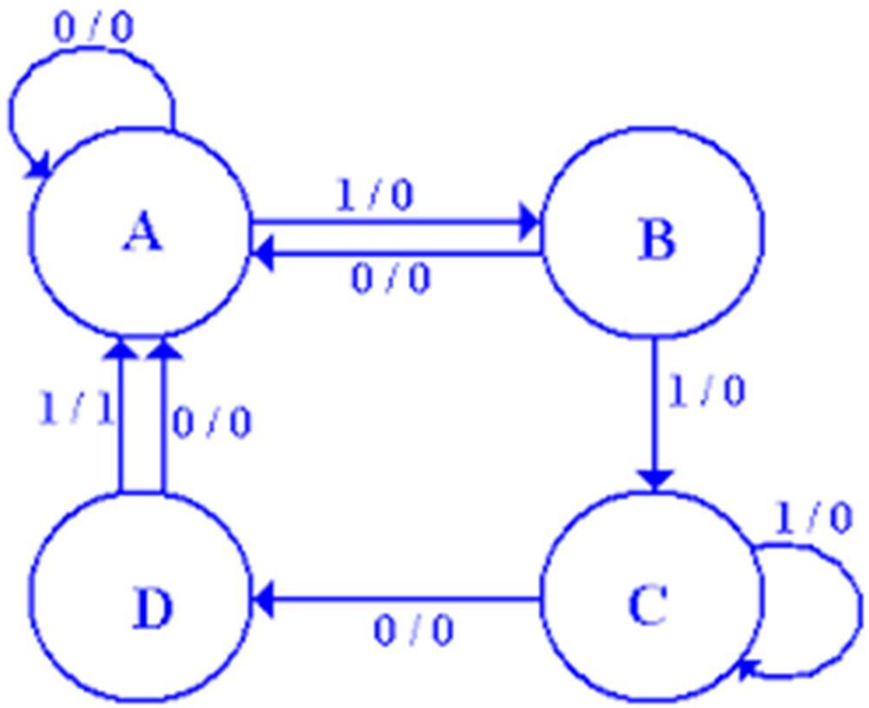
Sequence Detector

Overlap FSM – Last bit of one sequence and first bit of next sequence same

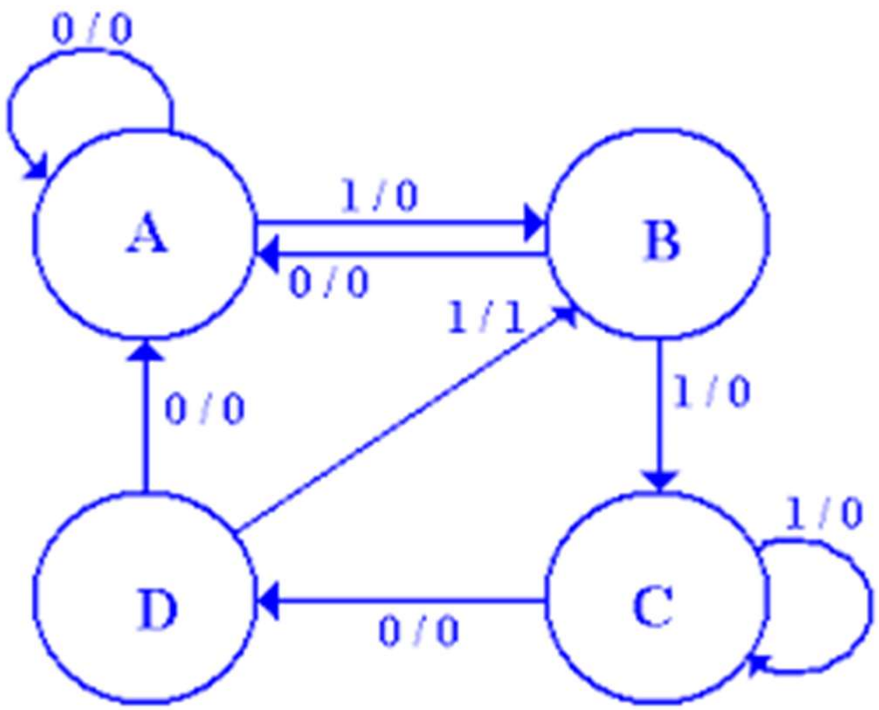
In sequence first and last bit same by n-bit→ n-bit overlap

Next state of main sequence intermediate state

Non-overlap FSM – Next state of main sequence initial state



1101 Sequence Detector
Without Overlap



1101 Sequence Detector
With Overlap

Mealy Machine	Moore Machine
Output depends both upon the present state and the present input	Output depends only upon the present state.
Fewer states	more states
Output presented on transition	Output presented on state
React faster to inputs, in the same clock cycle.	More logic is required, more circuit delays. React one clock cycle later.

Revision

If X=8'b10010100 than assign Y=X[6:3] evaluate to

X= 'd1234; the size of X is

The width of wire addr ?

    assign [3:0]addr = addrr1[20:17] + addr2[10:7]

The possible output value of operator == are

The possible output value of operator === are

If B = 2'b00, C = 2'b10 than Y = {B , C} result into

QUIZ

In continuous assignment left hand side must be

net

reg

scalar or vector net

scalar or vector reg

Behavioral Modeling

Behavioral Modeling

- Highest level of modelling
- describe design functionality in an algorithmic manner.
- resembles C programming
- Verilog constructs are similar to C
- great amount of flexibility
- Describe sequential digital circuit
- Output port are reg data type

Structured Procedures

always and initial

Initial block starts at time 0

executes exactly once during a simulation

does not execute again

Each block finishes execution independently

Always block statement starts at time 0

Executes the statements in the always block continuously in a looping fashion

Model a block of activity that is repeated continuously in a digital circuit

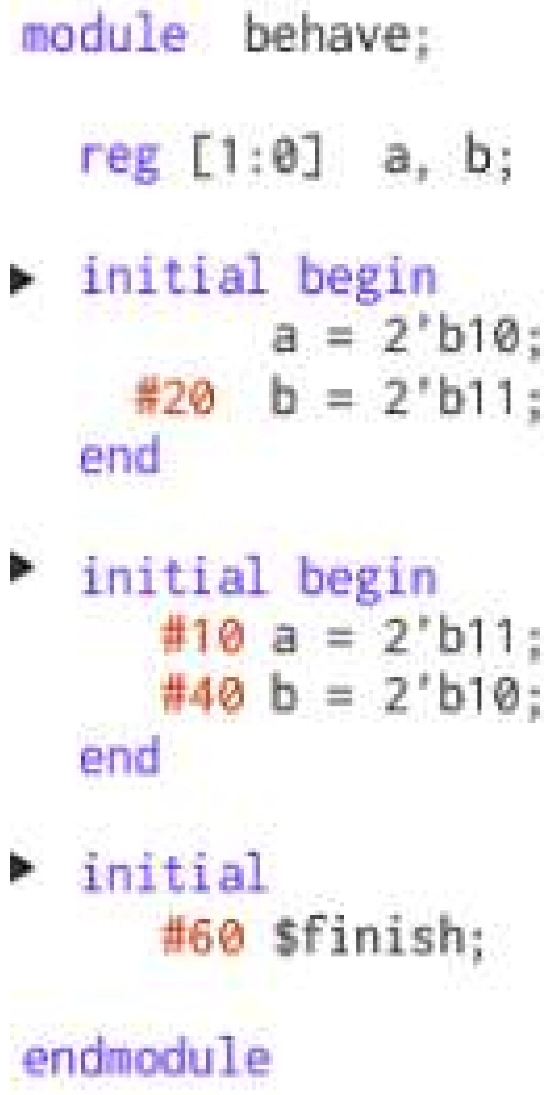
Initial Block Example

```
module stimulus;
    reg x,y, a,b, m;
    initial
          m = 1'b0; //single statement; does not need to be grouped
    initial
          begin
          #5 a = 1'b1; //multiple statements; need to be grouped
          #25 b = 1'b0;
          end
    initial
          begin
          #10 x = 1'b0;
          #25 y = 1'b1;
          end
    initial
          #50 $finish;
    endmodule
```

monitoring, waveforms and processes that must be executed only once

QUIZ

The simulation time is



```
module  behave;

   reg [1:0]  a, b;

   initial begin
              a = 2'b10;
     #20   b = 2'b11;
   end

   initial begin
      #10 a = 2'b11;
      #40 b = 2'b10;
   end

   initial
      #60 $finish;

endmodule
```

A. 20ns

B. 50ns

[✓ C. 60 ns]

D. 80 ns

E. 110 ns

Clock Generator with Always Block

```
module clock_gen (output reg clock);
initial
clock = 1'b0;   //Initialize clock at time zero

always
#10 clock = ~clock;  // //Toggle clock every half-cycle (time period = 20)
 initial
#1000 $finish;
endmodule
```

Clock Generator with Duty Cycle

lock generators with 50% duty cycles.

```
initial
reg clk = 0;

always
begin
#10 clk = 0;
#10 clk = 1;
end
```

20% duty cycle clock in verilog

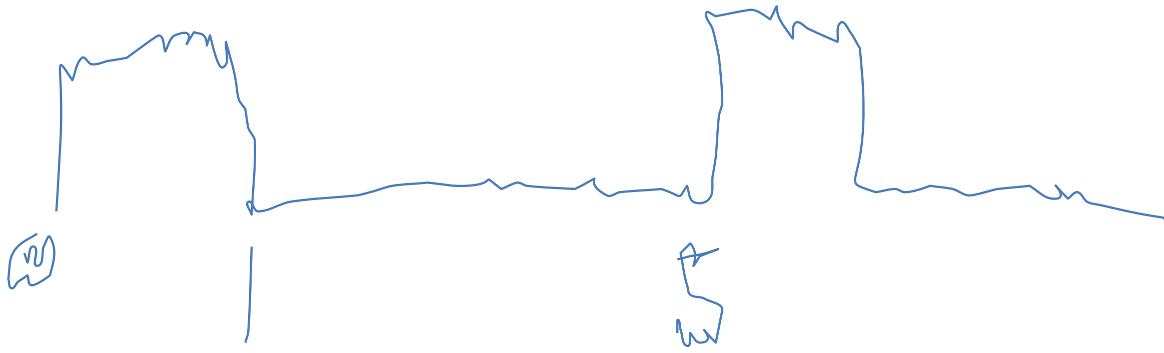
$$\frac{1}{5} = \frac{Ton}{Ton + Toff}$$

$$Toff = 4Ton$$

$$Let\ Ton = 1$$

$$Toff = 4$$

```
initial
reg clk = 0;

always
begin
clk = 1; #1
clk = 0; #4
end
```



Procedural Assignment

Procedural assignments update values of reg, integer, real, or time variables.

Value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value.

2 types of procedural assignment statements: blocking and nonblocking

Blocking assignment

- Executed in order specified.
- The = operator specify blocking assignments.
- Update the value instantly

```
reg x, y, z;
    reg [15:0] reg_a, reg_b;
    integer count;
```

```
initial
begin
x = 0; y = 1; z = 1;
count = 0;
reg_a = 16'b0; reg_b = reg_a;
#15 reg_a[2] = 1'b1;
#10 reg_b[15:13] = {x, y, z} ;
count = count + 1;
end
```

Nonblocking assignments

Execute assignments without blocking execution

A <= operator is used to specify nonblocking assignments

Not update value instantly end of simulation

```
initial
    begin
    x = 0; y = 1; z = 1;
    count = 0;
    reg_a = 16'b0; reg_b = reg_a;
    reg_a[2] <= #15 1'b1;
    reg_b[15:13] <= #10 {x, y, z};
    count <= count + 1;
    end
```

Example

```
initial begin
   a = #10 1'b1;// The simulator assigns 1 to a at time 10
  b = #20 1'b0;// The simulator assigns 0 to b at time 30
  c = #40 1'b1;// The simulator assigns 1 to c at time 70
 end
```

```
initial begin
  d <=  #10  1'b1;// The simulator assigns 1 to d at time 10
  e <=  #20  1'b0;// The simulator assigns 0 to e at time 20
  f  <=  #40  1'b1;// The simulator assigns 1 to f at time 40
end
```

QUIZ

Symbol	**meaning**
a) =	(i) blocking
b) <=	(ii) non-blocking

[✓ A. a- (i) b-(ii)]
B. a-(ii) b-(i)

```
assume a=1, b=0
initial
begin
#50 c = a|b;
d = c;
c = #50 a&b;
#20 d = c;
end
```

Time	Statement
50	C=1
50	D=1
100	C=0
120	D=0

```
assume a=1, b=0
initial
begin
#50 c = a&b;
d = a;
c <= #60 a|b;
d <= #80 b|c;
end
```

Time	Statement
50	C=0
50	D=0
110	C=1
130	D=0

```
assume a=1, b=0
initial
begin
c <= 0;
#50 c <= a|b;
d <= c;
c <= #50 a&b;
end
```

Time	Statement
50	C=1
50	D=0
100	C=0

```
always
 begin
   a = 3;
   a = 4;
   a = 5;
 b = a; #5;
  end
```

```
always
 begin
    a = 3;
    a = 4;
    a = 5;
 b <= a; #5;
   end
```

A = 5 and B = 5 in both cases

```
always
 begin
   a = 3;
   a = 4;
   b = a;
 a = 5; #5;
  end
```

```
always
 begin
    a = 3;
    a = 4;
   b <= a;
 a = 5; #5;
  end
```

A = 5 and B = 4 in both cases

```
always begin
        a = 3;
        a = 4;
        b = a;
        a = 5;
        b = a; #5;
end
```

```
always begin
        a = 3;
        a = 4;
        b <= a;
        a = 5;
        b <= a; #5;
end
```

A = 5 and B = 5 in both cases

Swapping Example

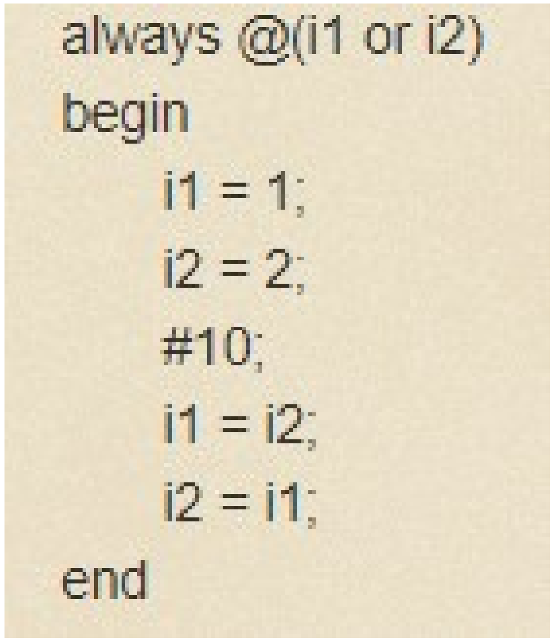
```
always @(posedge clock)
  a = b;
  always @(posedge clock)
  b = a;
```

No Swapping

Racing Error

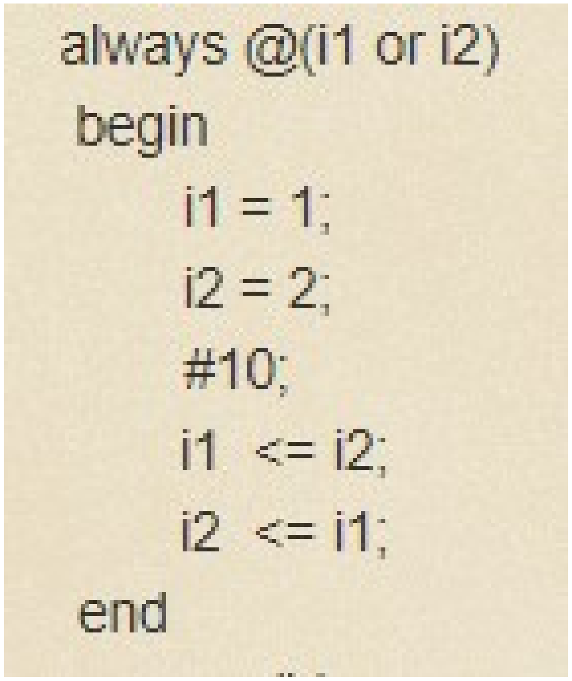
```
always @(posedge clock)
a <= b;
always @(posedge clock)
b <= a;
```

Swapping possible



```
always @(i1 or i2)
begin
        i1 = 1;
        i2 = 2;
        #10;
        i1 = i2;
        i2 = i1;
end
```

I1=2
I2=2



```
always @(i1 or i2)
 begin
        i1 = 1;
        i2 = 2;
        #10;
        i1  <= i2;
        i2  <= i1;
 end
```

I1=2
I2=1

No Swapping

Swapping

QUIZ

Identify true statement

A. Evaluate the RHS of nonblocking statements at the beginning of the time step.

B. Update the LHS of nonblocking statements at the end of the time step.

[✓ C. Both are true]

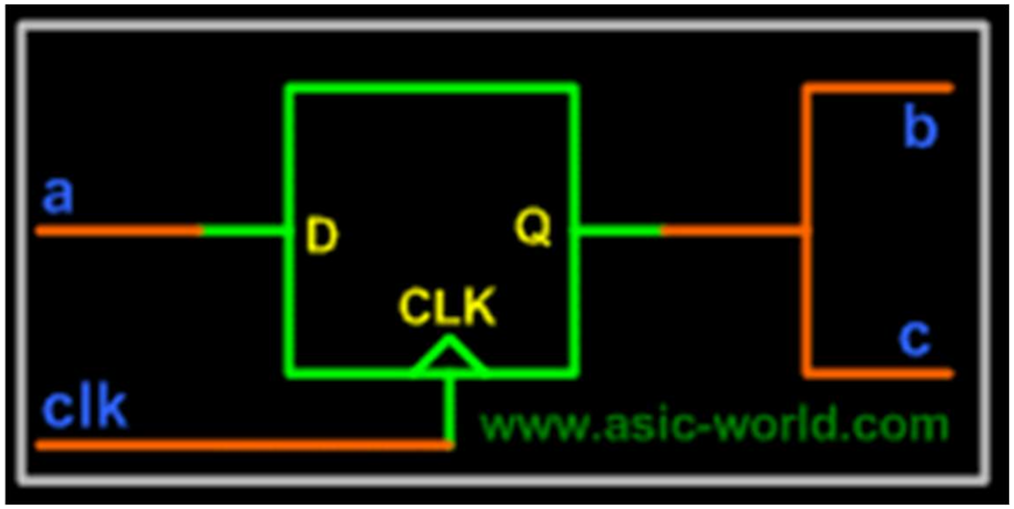
comment on execution

```
module ao4 (y, a, b, c, d);
 output y;
input a, b, c, d;
reg y, tmp1, tmp2;
always @(a or b or c or d)
 begin
tmp1 <= a & b;
tmp2 <= c & d;
 y <= tmp1 | tmp2;
end
 endmodule
```

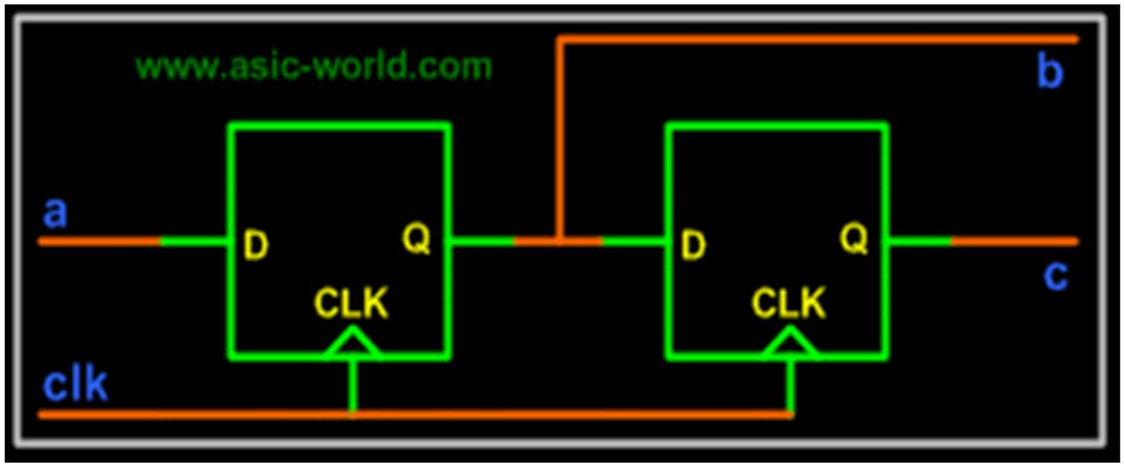
```
module ao2 (y, a, b, c, d);
 output y;
input a, b, c, d;
reg y, tmp1, tmp2;
always @(a or b or c or d)
begin
tmp1 = a & b;
tmp2 = c & d;
y = tmp1 | tmp2;
end
endmodule
```

sequential logic use a clocked always block with Nonblocking assignments.
combinational logic use an always block with Blocking assignments.
Try not to mix the two in the same always block

```
always @ (posedge clk )
begin
b = a;
c = b;
end
```



```
always @ (posedge clk )
begin
b <= a;
c <= b;
end
```



Event-Based Timing Control

Provide a way to specify the simulation time at which procedural statements will execute change in the value on a register or a net
Trigger execution of block

(i) Delay-based (ii) Event based (iii) level-sensitive

Event based control – trigger execution of a statement

(a) Regular event control @ symbol specify an event control

```
@(clock) q = d;
@(posedge clock) q = d;        positive transition ( 0 to 1,x or z, x to 1, z to 1 )
@(negedge clock) q = d;    negative transition ( 1 to 0,x or z,x to 0, z to 0)
q = @(posedge clock) d;
```

(b) Named event control Event trigger by symbol ->

```
always @(posedge clock)
begin
if(last_data_packet)
        ->received_data;
end
```

```
always @(received_data)
data_buf = {data_pkt[0], data_pkt[1]};
```

(c) Event OR Control multiple signals or events can trigger the execution
list of events as a sensitivity list
either ‘or’ of ‘ , ’

```
always @( reset or clock or d)
begin
if (reset)
q = 1'b0;
else if(clock)
q = d;
end
```

@* and @(*) sensitive to a change on any signal

(d) Level-Sensitive Timing Control @ provided edge-sensitive control
wait for triggering of an event
‘wait’ is used for level sensitive constructs.

```
always
wait (count_enable) #20 count = count + 1;
```

Count_enable 0, statement is not entered.
Count_enable 1, statement count = count + 1 is executed after 20 time units

Conditional Statements

```
if (<expression>)
      true_statement ;
```

```
if (<expression>)
      true_statement ;
else
     false_statement ;
```

```
if (<expression1>)
     true_statement1 ;
else if (<expression2>)
     true_statement2 ;
else if (<expression3>)
     true_statement3 ;
else
     default_statement ;
```

Execute statements based on ALU control signal

```
if (alu_control == 0)
y = x + z;
else if(alu_control == 1)
y = x - z;
else if(alu_control == 2)
y = x * z;
else
$display("Invalid ALU control signal")
```

Multiway Branching

One Switch Multiple Branch

Branch number entered in Binary/Octal/Decimal/HexaDecimal Format

```
case (expression)
alternative1: statement1;
alternative2: statement2;
alternative3: statement3;
...
...
default: default_statement;
endcase
```

```
case (alu_control)
2'd0 : y = x + z;
2'd1 : y = x - z;
2'd2 : y = x * z;
default : $display("Invalid ALU control signal");
endcase
```

Example

4:1 Multiplexer using case statement

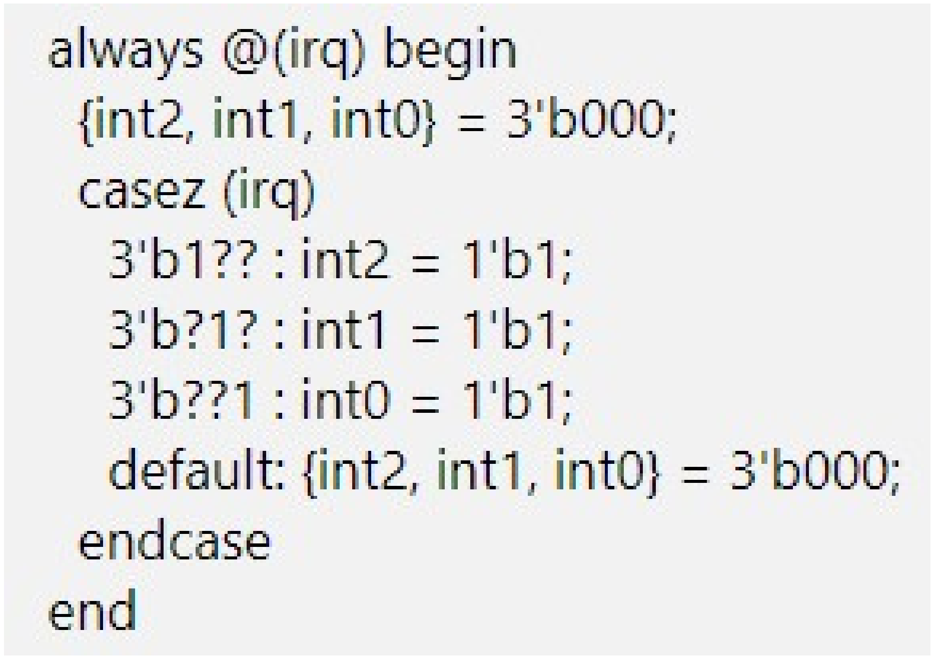
Design a module has 2-bit select signal to route one of three 3-bit signal to output

casez and casex statement

casez allows “Z” and “?” to be treated as don’t care in condition

casex allows “Z”, “?”, and “X” to be treated as don’t care in condition

```
reg [3:0] encoding;
    integer state;
    casex (encoding)
    4'b1xxx : next_state = 3;
    4'bx1xx : next_state = 2;
    4'bxx1x : next_state = 1;
    4'bxxx1 : next_state = 0;
    default : next_state = 0;
    endcase
```



```
always @(irq) begin
  {int2, int1, int0} = 3'b000;
  casez (irq)
    3'b1?? : int2 = 1'b1;
    3'b?1? : int1 = 1'b1;
    3'b??1 : int0 = 1'b1;
    default: {int2, int1, int0} = 3'b000;
  endcase
end
```

input encoding = 4'b10xz would cause next_state = 3

TFF

DFF

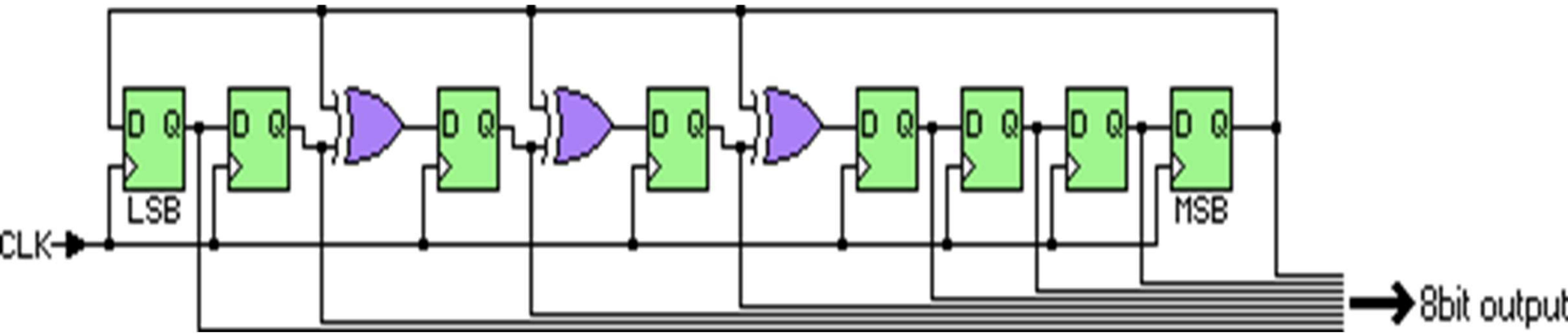
UP Down Counter

Shift Register

LFSR

Which is a Random Number Generator

A. Counter
B. Shifter
C. LFSR
D. Gray Counter


D Q
LSB
MSB
CLK
8bit output

FSM Programming

State transition happens ________ in every clock cycle

a) Once
b) Twice
c) Thrice
d) Four times

LOOP

4 loop statements in Verilog **while, for, repeat, and forever**
Syntax are similar to C-language
A loop statement occur inside initial or always

While Loop- executes until the while expression is not true

Increment count from 0 to 127; Exit at count 128

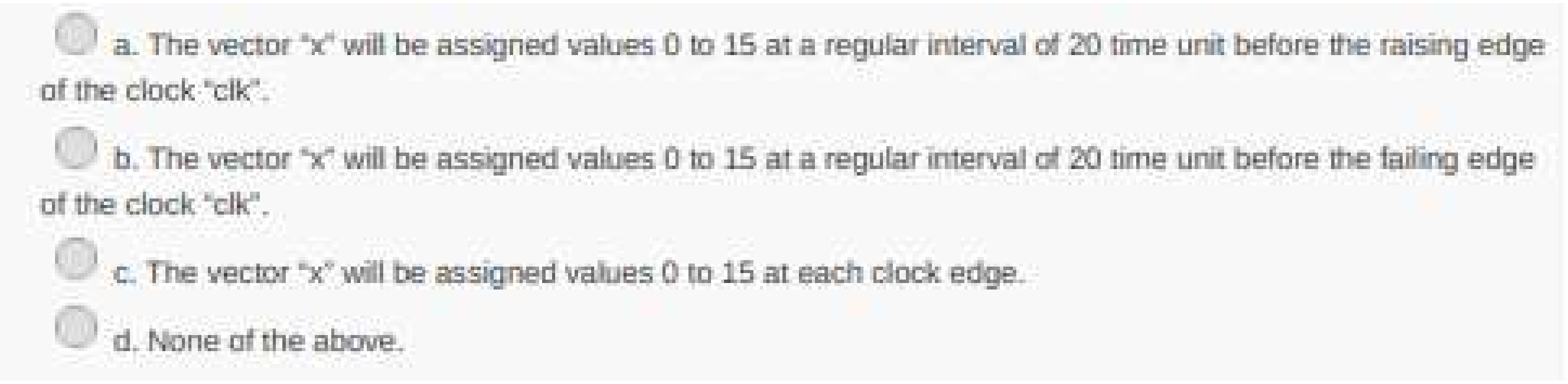
For Loop

- An initial condition
- A check to see if the terminating condition is true
- A procedural assignment to change

Increment count from 0 to 127; Exit at count 128

What does following code do

```
reg [3:0] x;
initial clk =0;
always #10 clk = ~clk;

initial
integer i;
begin
#5 for (i=0; i <16; i=i+1)
#20 x=i;
end
```



a. The vector "x" will be assigned values 0 to 15 at a regular interval of 20 time unit before the raising edge of the clock "clk".

b. The vector "x" will be assigned values 0 to 15 at a regular interval of 20 time unit before the failing edge of the clock "clk".

c. The vector "x" will be assigned values 0 to 15 at each clock edge.

d. None of the above.

Repeat Loop - executes the loop a fixed number of times

Increment count from 0 to 127; Exit at count 128

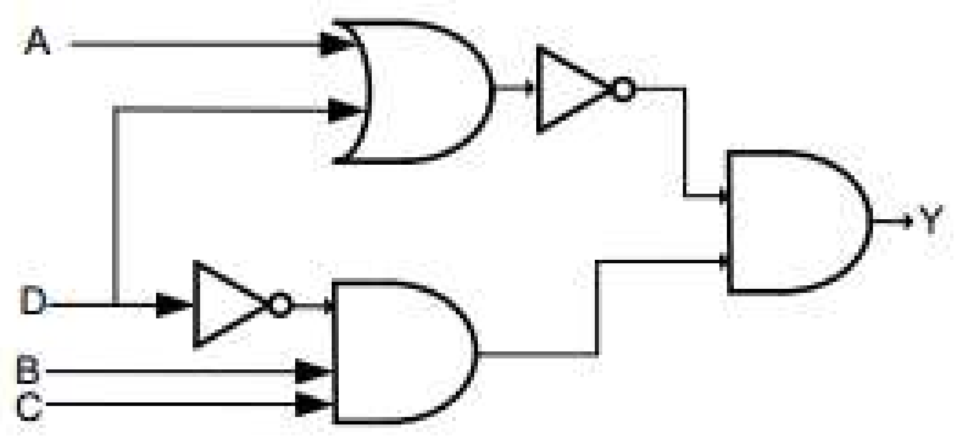
Forever loop does not contain any expression
executes forever until the **$finish** task is encountered

Which is unconditional loop

A. While
B. For
C. Repeat
D. Forever

Loop Example

Write Program and Testbench /Stimulus Program



4-bit Comparator in verilog

```
module test_dut;
initial
clk = 1'b1;
always
forever  begin
    #2.2 clk = 1'b0;
    #1.5 clk = 1'b1;
end
endmodule
```

- ○ a. 40 ns
- ○ b. 3.7 ns
- ○ c. 37 ns
- ○ d. 30 ns

What does the following code segment indicate?

```
initial clk = 1’b1;
 always #10 clk = ~clk;
```

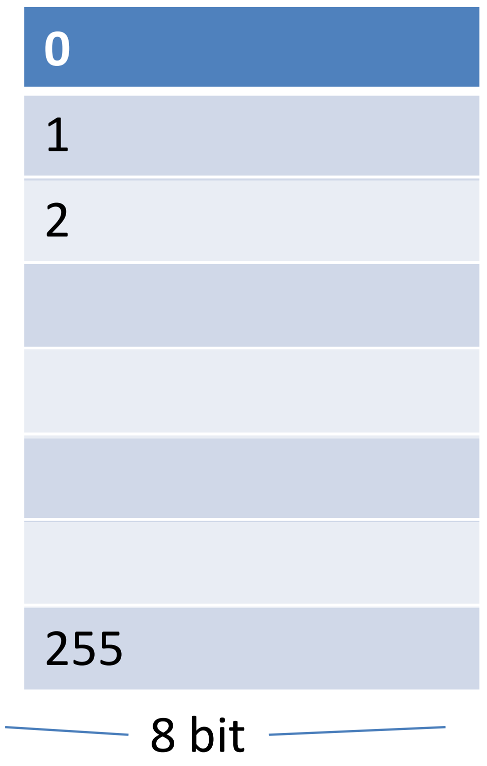
a. Falling edges of the clock will appear at times 10, 30, 50, 70,
b. Falling edges of the clock will appear at times 20, 40, 60, 80,
c. Falling edges of the clock will appear at times 10, 20, 30, 40,
d. None of the above

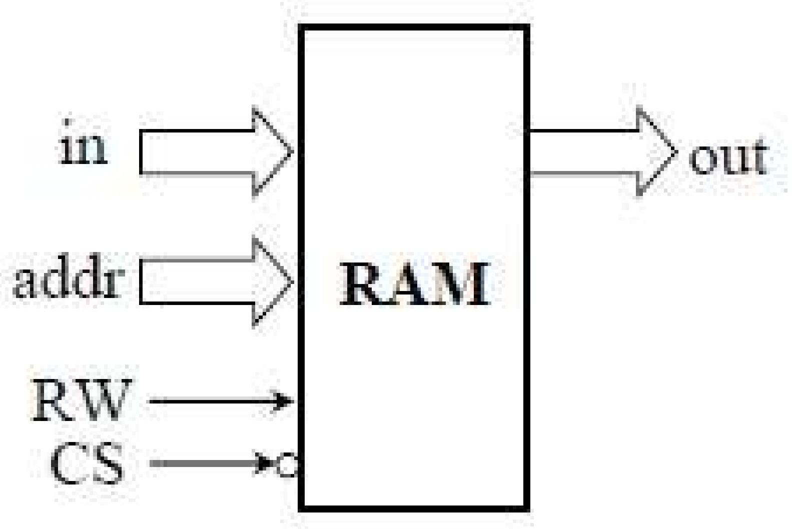
Memories in Verilog

```
reg bit;                            // a single register
reg [31:0] word;                    // a 32-bit register
reg [31:0] array[15:0];             // 16 32-bit regs
reg [31:0] array_2d[31:0][15:0];     // 2 dimensional 32-bit array

assign read_data = array[index];    // combinational (asynch) read
always @(posedge clock)
    array[index] <= write_data;     // clocked (synchronous) write
```

Create RAM of 256 bytes and fill with number 0-255

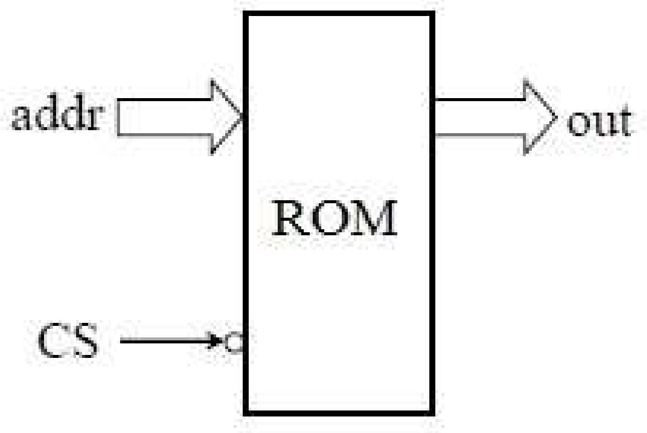
```
module RAM_code(out, in, addr, RW, CS);
output [7:0] out;
input [7:0] in;
input [3:0] addr;
input RW, CS;
reg [7:0] out;
reg [7:0] DATA[15:0];
always @(negedge CS)
begin
if(RW==1'b0) //READ
out=DATA[addr];
else if(RW==1'b1) //WRITE
DATA[addr]=in;
else
out=8'bz;
end
endmodule
```





Verilog ROM Program

```
module ROM_code(out, addr, CS);
output[15:0] out;
input[3:0] addr;
input CS;
reg [15:0] out;
reg [15:0] ROM[15:0];
always @(negedge CS)
begin
ROM[0]=16'h5601; ROM[1]=16'h3401;
ROM[2]=16'h1801; ROM[3]=16'h0ac1;
ROM[4]=16'h0521; ROM[5]=16'h0221;
ROM[6]=16'h5601; ROM[7]=16'h5401;
ROM[8]=16'h4801; ROM[9]=16'h3801;
ROM[10]=16'h3001; ROM[11]=16'h2401;
ROM[12]=16'h1c01; ROM[13]=16'h1601;
ROM[14]=16'h5601; ROM[15]=16'h5401;
out=ROM[addr];
end
endmodule
```



```
reg [31:0] regfile[30:0];
```

A. 31 32-bit words
B. 32 31-bit words

What are the final values of a, b, c, d?

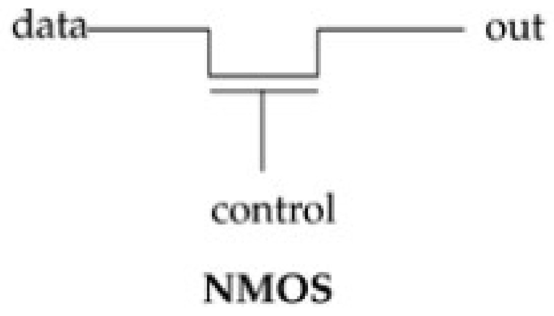
```
initial
    begin
    a = 1'b0;
    #0 c = b;
end

initial
    begin
    b = 1'b1;
    #0 d = a;
end
```

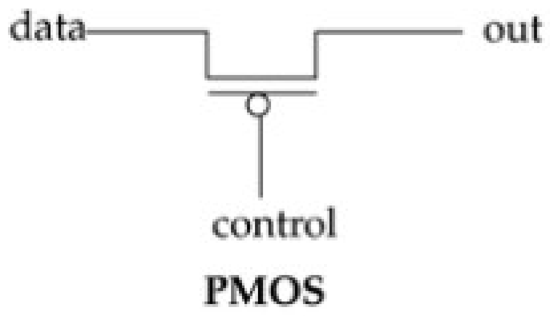
Using the repeat loop, delay the statement a = a + 1 by 20 positive edges of clock.

Switch Level Modelling

Verilog provides the ability to design at a MOS-transistor level.
Lowest level of modeling

MOS Switch





nmos n1(out, data, ncontrol)

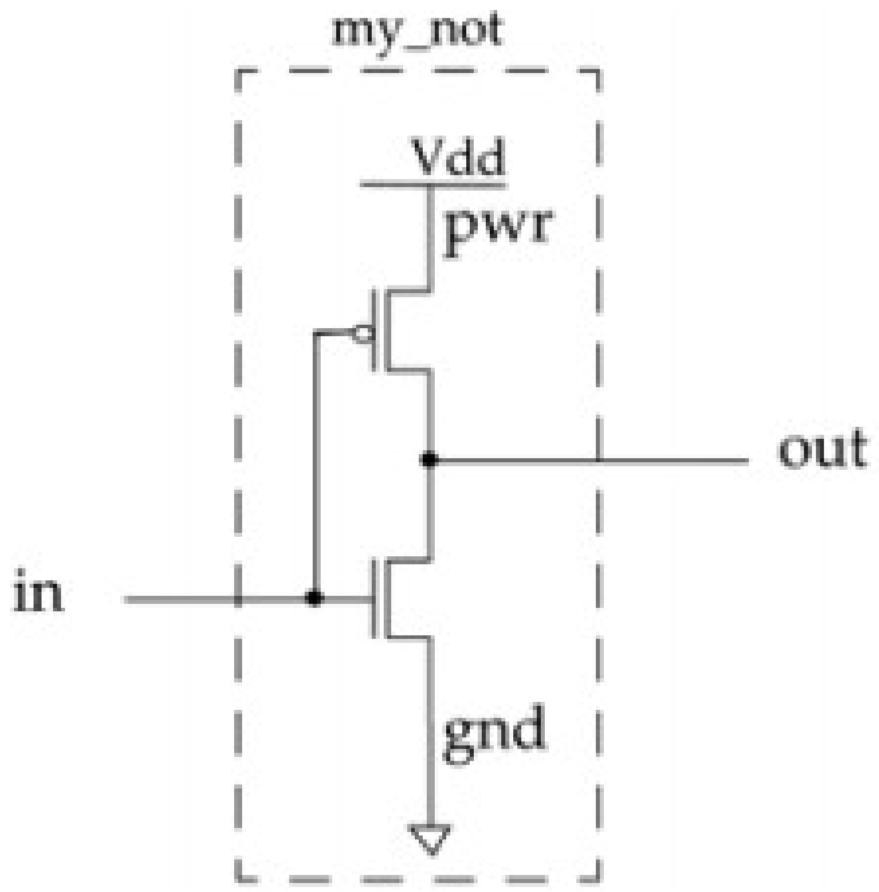
pmos n1(out, data, pcontrol)

Power and Ground → power (Vdd, logic 1) = supply1
ground (Vss, logic 0) == supply 0

```
supply1 vdd;
supply0 gnd;
```

CMOS NOT GATE

```
module my_not(out, in);
output out;
input in;

supply1 pwr;
supply0 gnd;

pmos (out, pwr, in);
nmos (out, gnd, in);
endmodule
```



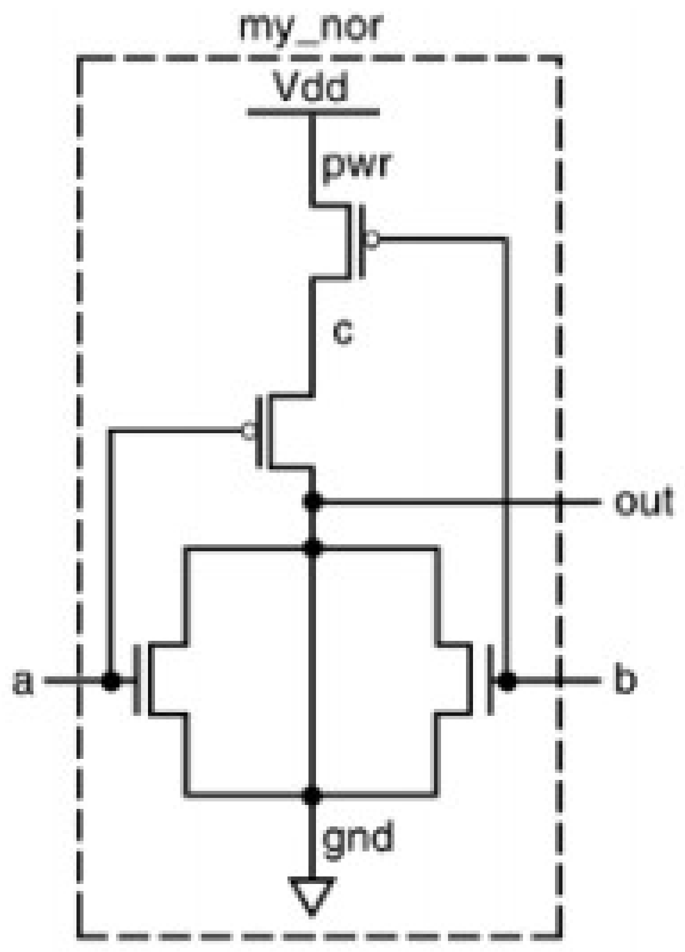
In switch level modeling, a MOS switch have

A. 1 terminal
B. 2 terminal
C. 3 terminal
D. 4 terminal

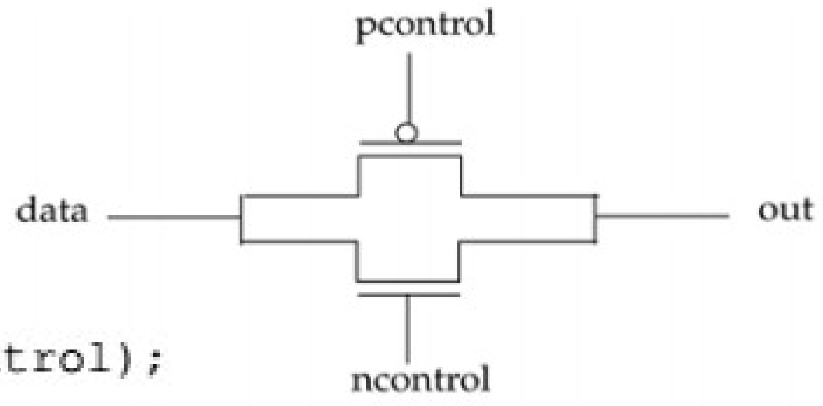
CMOS NOR GATE

```
module my_nor(out, a, b);
output out;
input a, b;

wire c;
supply1 pwr;
supply0 gnd ;

pmos (c, pwr, b);
pmos (out, c, a);

nmos (out, gnd, a);
nmos (out, gnd, b);
endmodule
```

```
module stimulus;
reg A, B;
wire OUT;

my_nor n1(OUT, A, B);
initial
begin
A = 1'b0; B = 1'b0;
#5 A = 1'b0; B = 1'b1;
#5 A = 1'b1; B = 1'b0;
#5 A = 1'b1; B = 1'b1;
end
initial
$monitor($time, " OUT = %b, A = %b, B = %b",
OUT, A, B);
endmodule
```



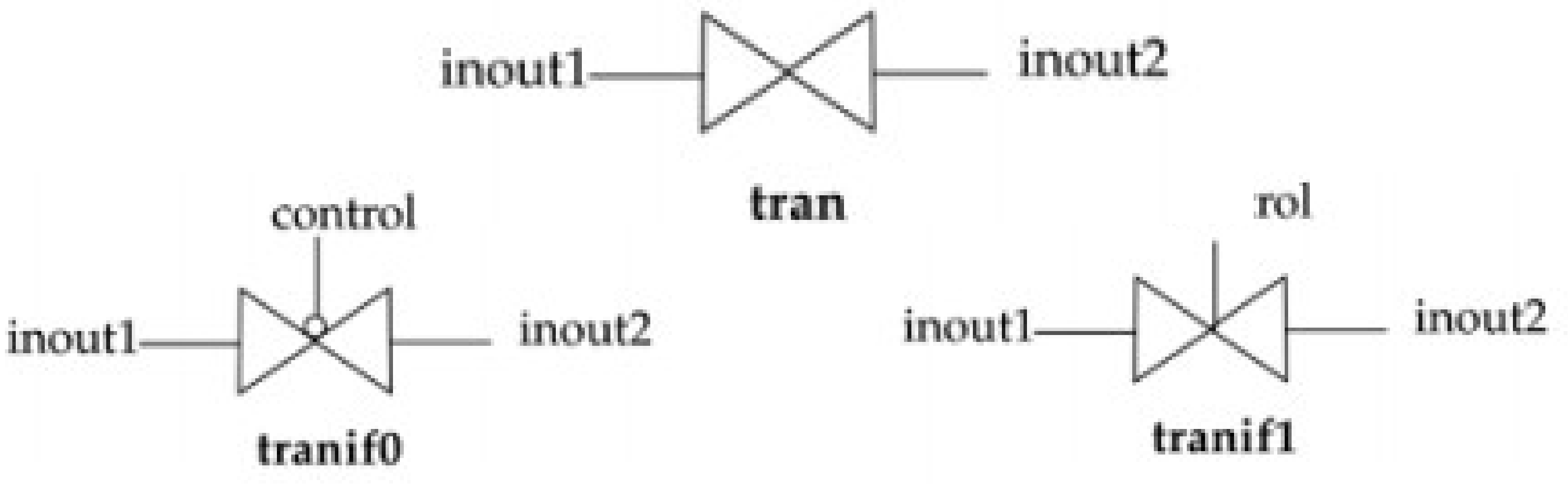
CMOS Switch



```
cmos c1(out, data, ncontrol, pcontrol);
```

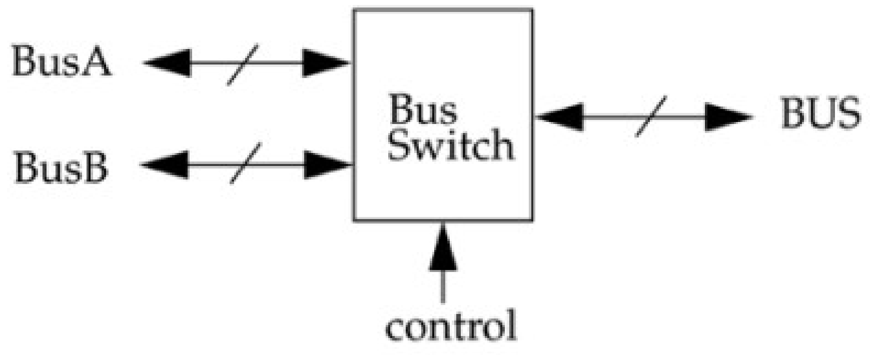
2-to-1 Multiplexer using CMOS switch

Bidirectional Switches



```
tran t1(inout1, inout2);
tranif0 (inout1, inout2, control);
tranif1 (inout1, inout2, control);
```

Design a 4-bit bidirectional bus switch that has two buses, BusA and BusB, on one side and a single bus, BUS, on the other side, 1-bit control signal is used for switching. BusA and BUS are connected if control = 1. BusB and BUS are connected if control = 0.

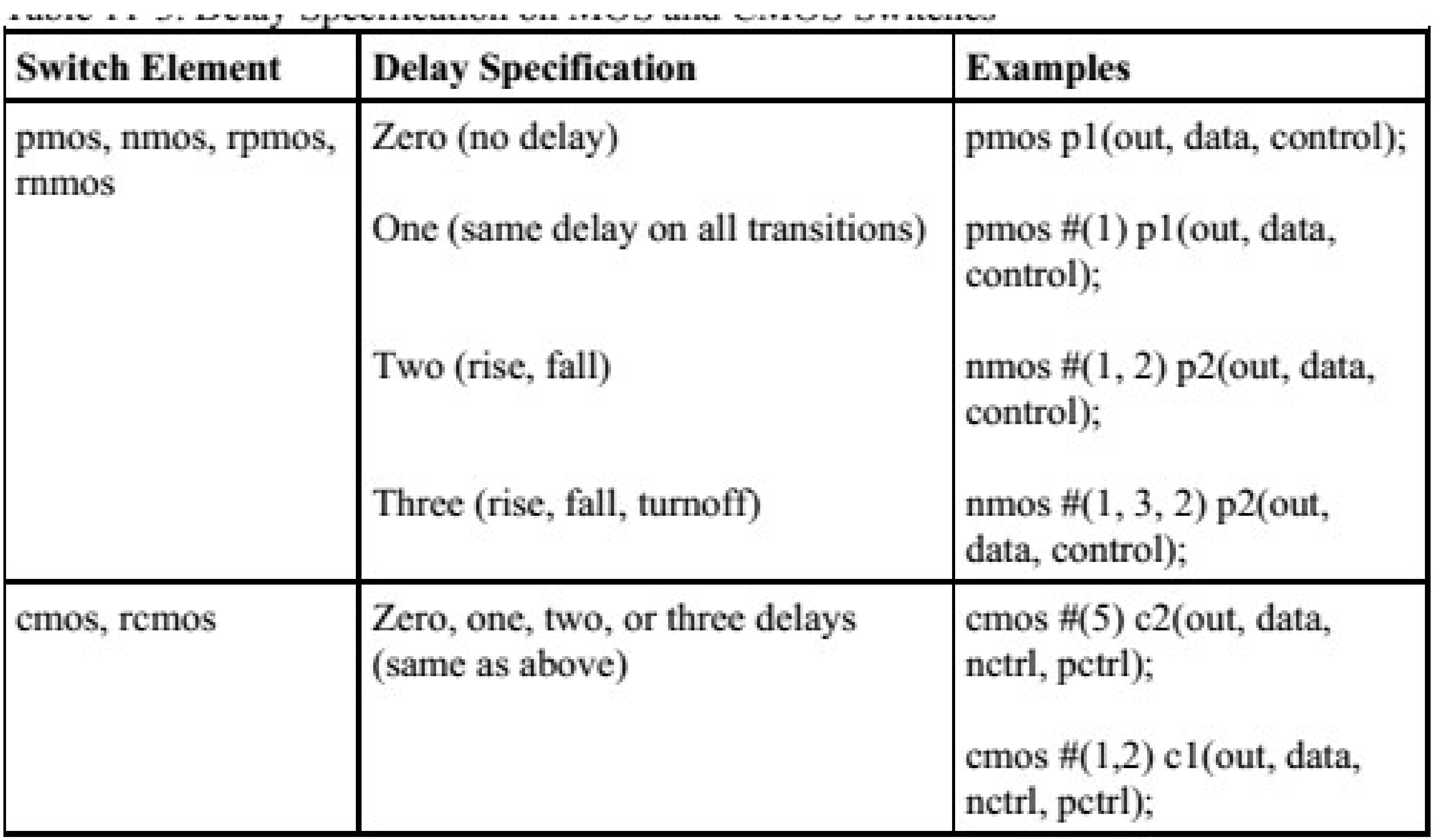


Switch Level Modeling supports module instantion

True

False

Switch with Delay



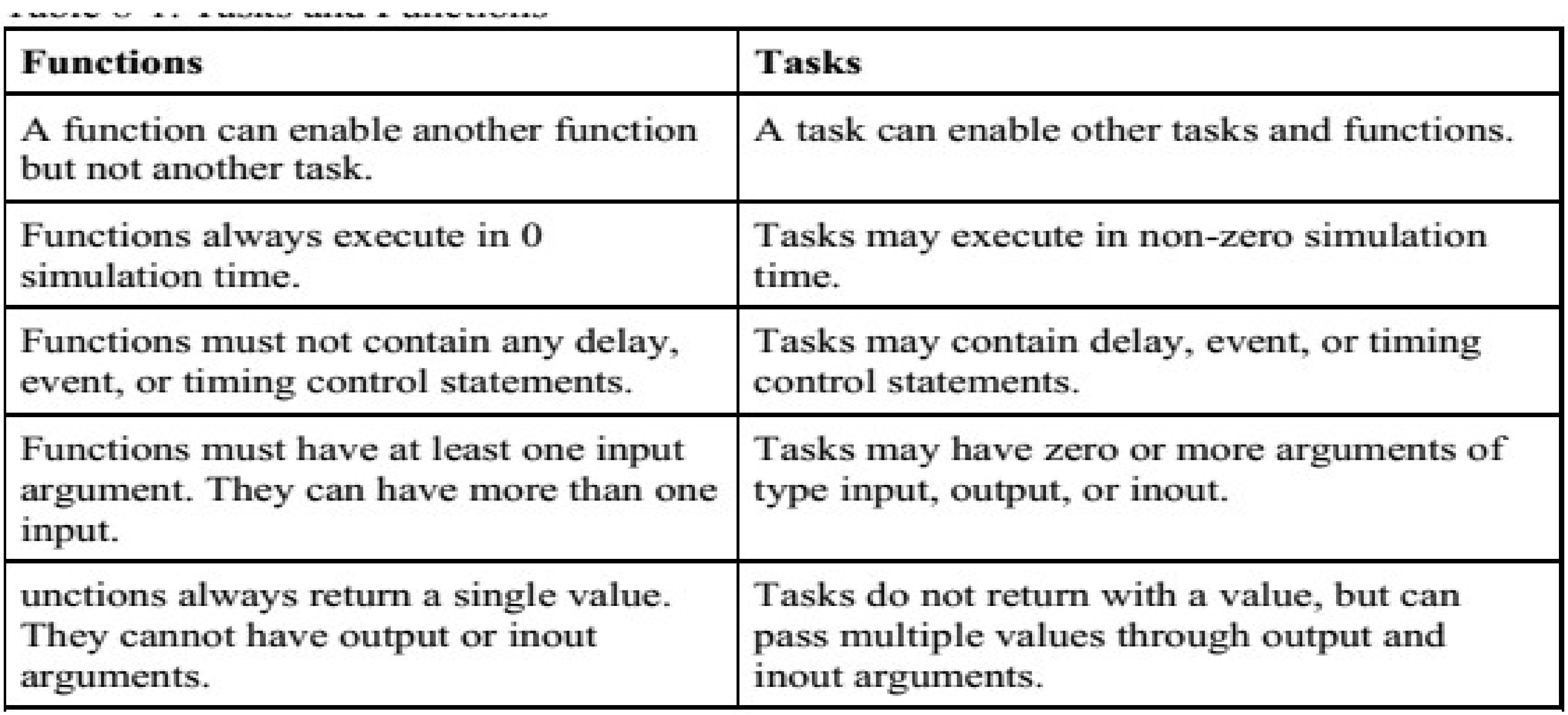
Switch Element	Delay Specification	Examples
pmos, nmos, rpmos, rnmos	Zero (no delay)	pmos p1(out, data, control);
	One (same delay on all transitions)	pmos #(1) p1(out, data, control);
	Two (rise, fall)	nmos #(1, 2) p2(out, data, control);
	Three (rise, fall, turnoff)	nmos #(1, 3, 2) p2(out, data, control);
cmos, rcmos	Zero, one, two, or three delays (same as above)	cmos #(5) c2(out, data, nctrl, pctrl);
		cmos #(1,2) c1(out, data, nctrl, pctrl);

Tasks and Functions

When require same functionality multiple time

Frequently use part abstracted in routine and can be invoked

Similar to subroutine into other programs



Functions	Tasks
A function can enable another function but not another task.	A task can enable other tasks and functions.
Functions always execute in 0 simulation time.	Tasks may execute in non-zero simulation time.
Functions must not contain any delay, event, or timing control statements.	Tasks may contain delay, event, or timing control statements.
Functions must have at least one input argument. They can have more than one input.	Tasks may have zero or more arguments of type input, output, or inout.
unctions always return a single value. They cannot have output or inout arguments.	Tasks do not return with a value, but can pass multiple values through output and inout arguments.

functions can have input arguments. Tasks can have input, output, and inout arguments

- A task begins with keyword **task** and ends with keyword **endtask**
- Inputs and outputs are declared after the keyword task.
- Local variables are declared after input and output declaration

Example—Temperature Celsius to Fahrenheit conversion

```
module task_calling (temp_a, temp_b, temp_c, temp_d);
input [7:0] temp_a, temp_c;
output [7:0] temp_b, temp_d;
 reg [7:0] temp_b, temp_d;

always @ (temp_a)
begin
convert (temp_a, temp_b);
end

always @ (temp_c)
begin
convert (temp_c, temp_d);
```

```
task convert;
input [7:0] temp_in; 5
output [7:0] temp_out; 6

begin
temp_out = (9/5) *( temp_in + 32)
end
endtask

endmodule
```

QUIZ

Which of the following is a difference between a Function and a Task?

A. A Function can call another function; a Task cannot

[✓ B.] A Function cannot call a task; a Task can call another task

C. A Function has one or more inputs; a Task has no inputs

D. A Function argument may be an output; a Task's argument may only be an input

Tasks- keywords task and endtask

Consider a task called bitwise_oper, which computes the bitwise and, bitwise or, and bitwise ex-or of two 16-bit numbers. The two 16-bit numbers a and b are inputs and the three outputs are 16-bit numbers ab_and, ab_or, ab_xor.

```
module operation;
parameter delay = 10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;
always @(A or B)
begin
bitwise_oper(AB_AND, AB_OR, AB_XOR, A, B);
end

task bitwise_oper;
output [15:0] ab_and, ab_or, ab_xor;
input [15:0] a, b;
begin
#delay ab_and = a & b;
ab_or = a | b;
ab_xor = a ^ b;
end
endtask
endmodule
```

Automatic (Re-entrant) Task

Tasks are static in nature.

Task is called concurrently from two places in code, operate on same task variables.

Highly likely that the results of such an operation will be incorrect

To avoid this problem, a keyword **automatic** is added in front of the task

Functions

Declared with the keywords **function** and **endfunction**

- No delay, timing, or event control constructs in the procedure.
- Returns a single value.
- At least one input argument.
- No output or inout arguments
- No nonblocking assignments

Parity calculation with Function

Bidirectional Shifter using Function

```
module shifter;
Input [31:0] addr;
output reg [31:0] left_addr, right_addr;
reg control;
always @(addr)
begin
left_addr = shift(addr, `LEFT_SHIFT);
right_addr = shift(addr, `RIGHT_SHIFT);
end

function [31:0] shift;
input [31:0] address;
input control;
begin
shift = (control ) ?(address << 1) : (address >> 1);
end
endfunction
endmodule
```

Automatic (Recursive) Functions

Functions are normally used non-recursively .

If a function is called concurrently from two locations, the results are non-deterministic because both calls operate on the same variable space.

keyword automatic can be used to declare a recursive (automatic) function where all function declarations are allocated dynamically for each recursive calls

How to avoid error due to recursive call of task or function

A. Automatic

B. Re-entrant

User-Defined Primitives

All the logic gate are in-built primitive

UDP allow own custom built primitive

Combinational UDP- Output defined by logical combination of input

Sequential UDP- Output defined by present input and present state

```
primitive <udp_name> (output_terminal_name,input_terminal_names> );
output <output_terminal_name>;
input <input_terminal_names>;

reg <output_terminal_name>;                    only for sequential UDP
initial <output_terminal_name> = <value>;

table
input _terminal_name : output_terminal_name;
<table entries>
endtable
endprimitive
```

Combinational UDP

Primitive udp_and

UDP Rules

- Only scalar input terminals (1 bit).
- Multiple input terminals are permitted.
- Only one scalar output terminal (1 bit).
- The output terminal must always appear first in the terminal list.
- Multiple output not permitted
- UDPs do not support inout ports

- The state table entries can contain values 0, 1, or x.
- UDPs do not handle z values.
- z values passed to a UDP are treated as x values

- UDP defined with primitive -- endprimitive

- UDPs can instantiated

- Sequential UDP present state declared as reg
- state in a sequential UDP can be initialized with an initial statement

In verilog UDP program first signal in the port list

A. Input
B. Output
C. Reg
D. wire

UDP does **not** handle

A. 0
B. 1
C. Z
D. x

UDP_OR

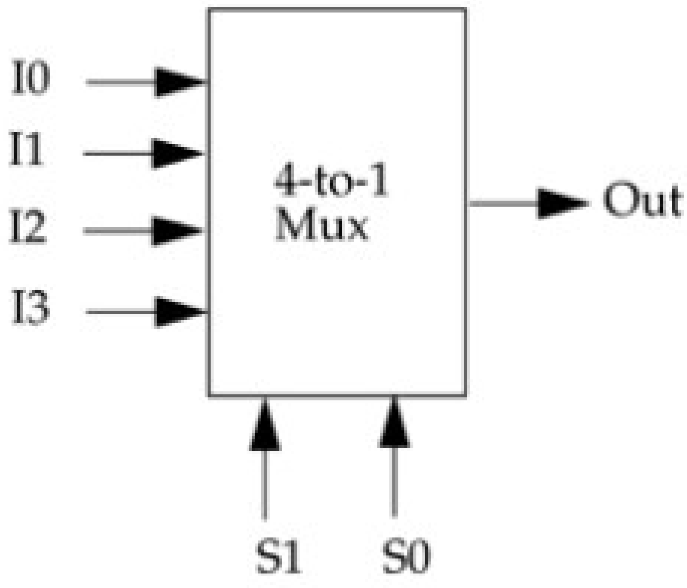
```
primitive udp_or(out, a, b);

output out;
  input a, b;

  table
    //   a    b    :    out;
         0    0    :    0;
         0    1    :    1;
         1    0    :    1;
         1    1    :    1;
         x    1    :    1;
         1    x    :    1;
  endtable

  endprimitive
```

UDP_OR Shortned Table

? symbol is used for a don't care condition.
? symbol is automatically expanded to 0, 1, x

```
table
  //   a    b    :    out;
       0    0    :    0;
       1    ?    :    1; //? expanded to 0, 1, x
       ?    1    :    1; //? expanded to 0, 1, x
       0    x    :    x;
       x    0    :    x;
endtable
```

UDP_MUX4:1



Identify correct

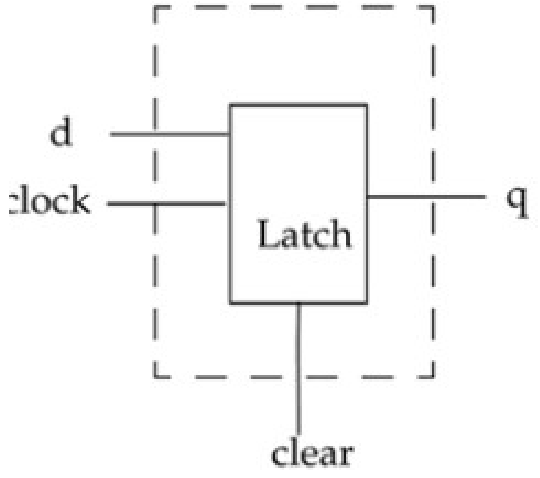
A. state table becomes large very quickly as the number of inputs increases.

B. Memory requirements to simulate UDPs increase exponentially .

[✓ C. Both are true]

Sequential UDP

Level-Sensitive Latch with clear



"-" symbol is used to denote no chang

<input1> <input2> ..... <inputN> : <current_state> : <next_state>;

```
primitive latch(q, d, clock, clear);
output q;
reg q;
input d, clock, clear;

initial
q = 0;
table
d clock clear : q : q+ ;
?   ?         1   : ? : 0 ;
1  1          0   : ? : 1 ;
0  1          0   : ? : 0 ;
?  0          0   : ? : - ;
endtable
endprimitive
```

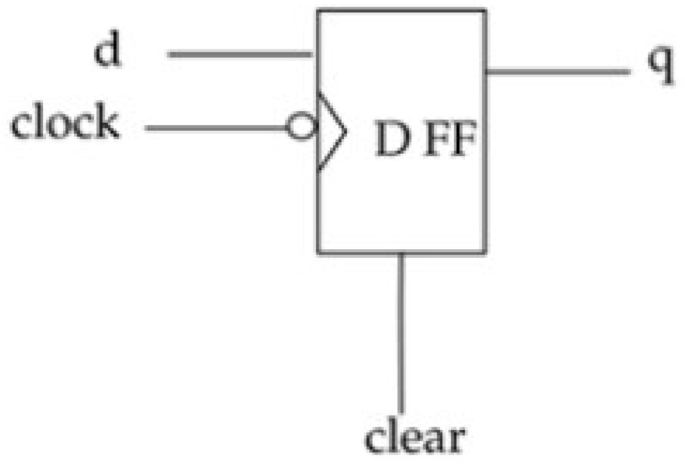
Edge-Sensitive Sequential UDPs

(10) denotes a negative edge transition from logic 1 to logic 0.

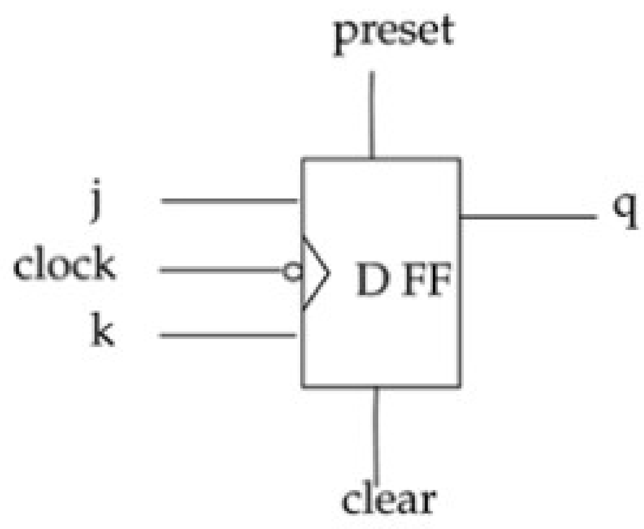
(1x) denotes a transition from logic 1 to unknown x state.

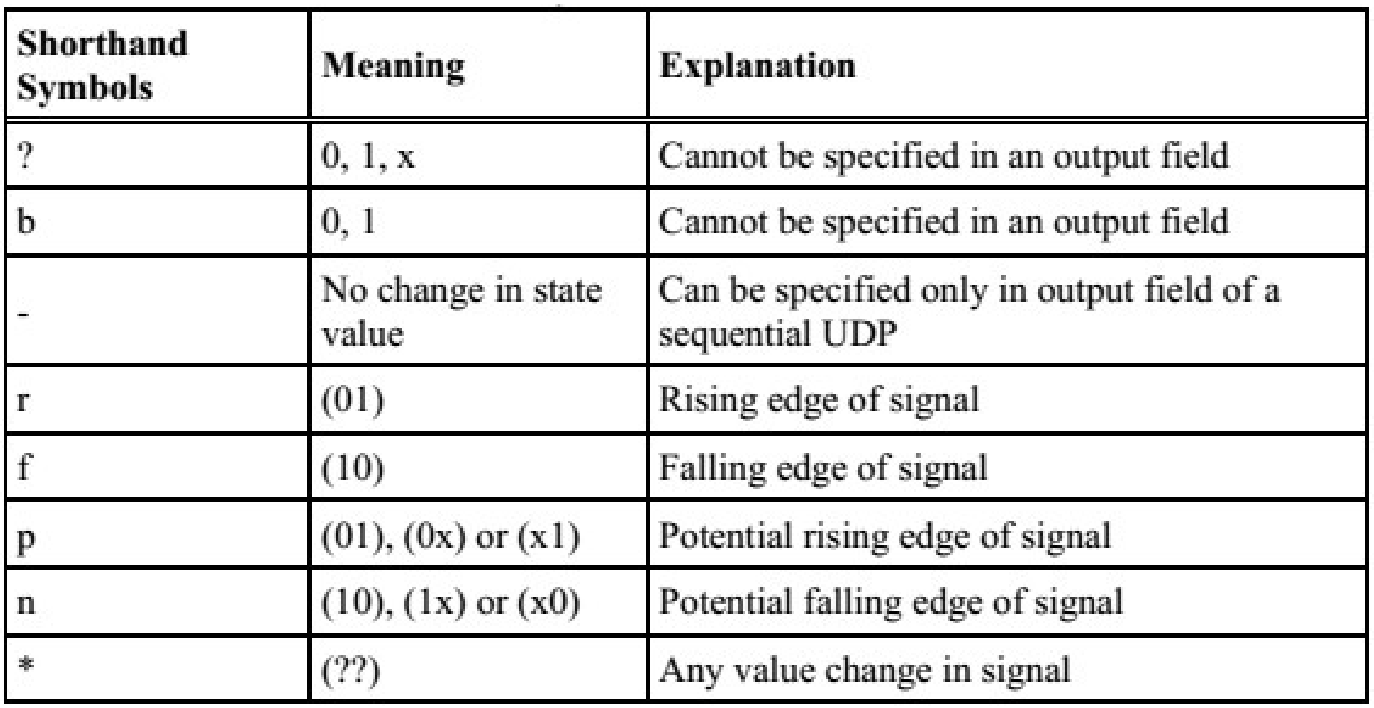
(0?) denotes a transition from 0 to 0, 1, or x. Potential positive-edge transition.

(??) denotes any transition in signal value 0,1, or x to 0, 1, or x.



Define a negative edge-triggered JK flipflop, jk_ff with asynchronous preset and clear as a UDP. q = 1 when preset = 1 and q = 0 when clear = 1.





Shorthand Symbols	Meaning	Explanation
?	0, 1, x	Cannot be specified in an output field
b	0, 1	Cannot be specified in an output field
-	No change in state value	Can be specified only in output field of a sequential UDP
r	(01)	Rising edge of signal
f	(10)	Falling edge of signal
p	(01), (0x) or (x1)	Potential rising edge of signal
n	(10), (1x) or (x0)	Potential falling edge of signal
*	(??)	Any value change in signal

UDPs model functionality only

limit on the maximum number of inputs specific to simulator being → minimum of 9 inputs for sequential UDPs and 10 for combinational UDPs

UDP is typically implemented as a lookup table in memory

Memory requirement for a UDP grows exponentially in relation to the number of inputs

state table should be specified as completely as possible
If a certain combination of inputs is not specified, the default output value x

In verilog UDP state table symbol **'-'** stands for

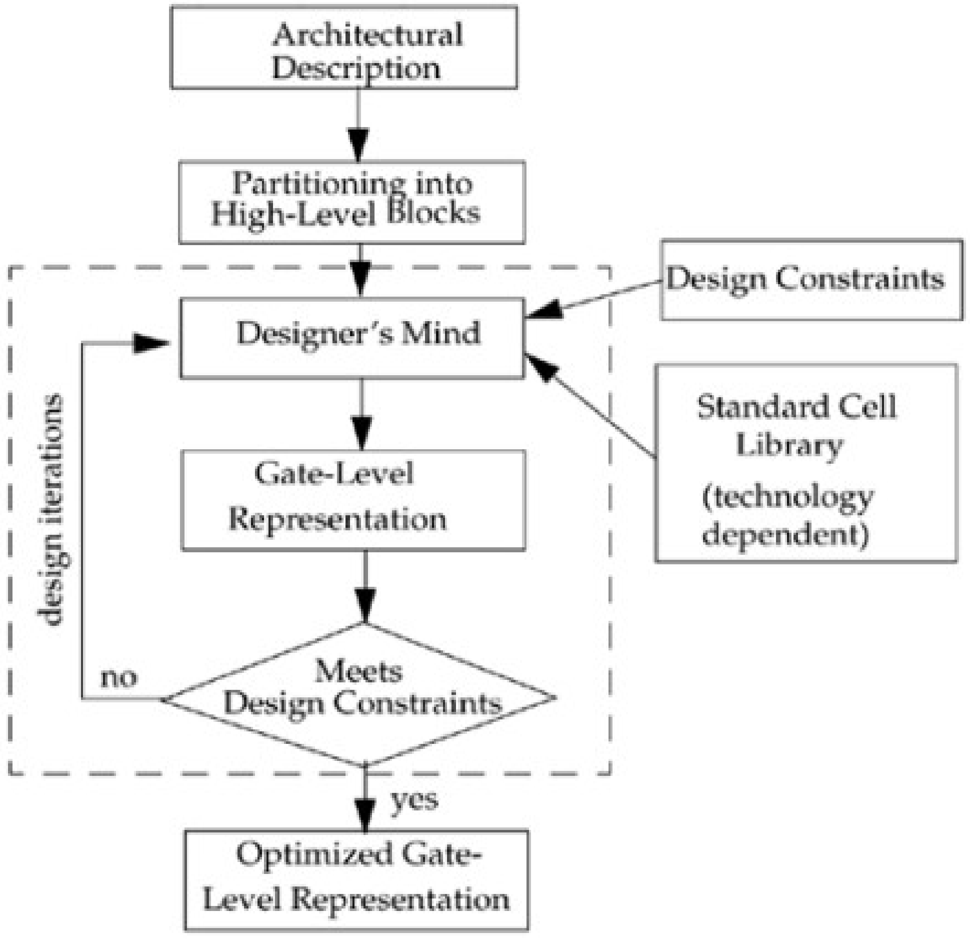
A. No change
B. Set
C. Reset
D. toggle

Synthesis

logic synthesis is the process of converting a high-level description of the design into an optimized gate-level representation, given a standard cell library and certain design constraints.

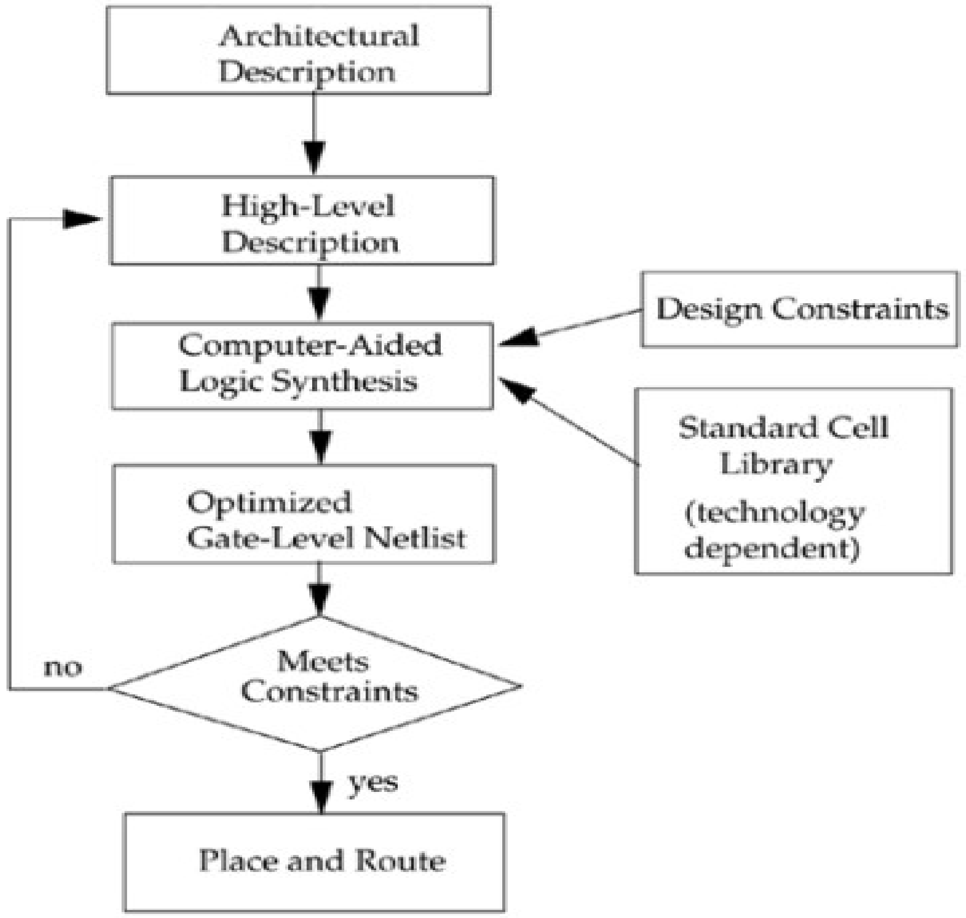
A standard cell library can have simple cells, such as basic logic gates like and, or, and nor, or macro cells, such as adders, muxes, and special flipflops.

Designer's Mind as Logic Synthesis Tool



Architectural Description
Partitioning into High-Level Blocks
Design Constraints
Designer's Mind
Standard Cell Library (technology dependent)
Gate-Level Representation
design iterations
no
Meets Design Constraints
yes
Optimized Gate-Level Representation

Computer-Aided Logic Synthesis Process



Architectural Description
High-Level Description
Design Constraints
Computer-Aided Logic Synthesis
Standard Cell Library (technology dependent)
Optimized Gate-Level Netlist
no
Meets Constraints
yes
Place and Route

CAD Synthesis Tool

Computer-aided logic synthesis tools has automated the process of converting the high-level description to logic gates.

Designers can now concentrate on the architectural trade-offs

High-level description of the design

Accurate design constraints

Impact of synthesis / Limitation of Manual Conversion

- Manual conversion was prone to human error
- Designer could never be sure that the design constraints were going to be met
- Time taken to convert a high-level design into gates.
- If the gate-level design did not meet requirements, the turnaround time for redesign
- Re-design was needed to verify what-if scenarios.
- Each designer would implement design blocks differently
- If a bug was found require redesign
- Timing, area, and power dissipation in library cells are fabrication-technology specific
- Design reuse was not possible

Automated logic synthesis tools Impacts

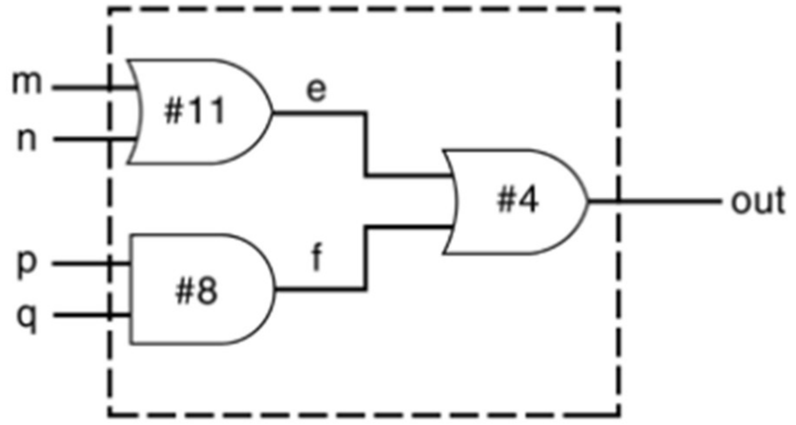
- less prone to human error
- without significant concern about design constraints
- Conversion from high-level design to gates is fast.
- Turnaround time for redesign of blocks is shorter
- What-if scenarios are easy to verify
- optimize the design as a whole
- If a bug is found need to change few code of program
- technology-independent design
- Design reuse is possible

Keyword or Description	**Notes**
input, inout, output	
parameter	
module	
wire, reg, tri	Vectors are allowed
module instances, primitive gate instances	E.g., mymux m1(out, i0, i1, s); E.g., nand (out, a, b);
function, task	Timing constructs ignored
always, if, then, else, case, casex, casez	initial is not supported
begin, end, named blocks, disable	Disabling of named blocks allowed
assign	Delay information is ignored
for, while, forever,	while and forever loops must contain @(posedge clk) or @(negedge clk)

	Synthesizable	Non-Synthesizable
Basic	Identifiers, escaped identifiers, Sized constants (b, o, d, h), Unsized constants (2'b11, 3'07, 32'd123, 8'hff), Signed constants (s) 3'bs101, module, endmodule, macromodule, ANSI-style module, task, and function port lists	system tasks, real constants
Data types	wire, wand, wor, tri, triand, trior, supply0, supply1, trireg (treated as wire), reg, integer, parameter, input, output, inout, memory(reg [7:0] x [3:0];), N-dimensional arrays,	real, time, event, tri0, tri1
Module instances	Connect port by name, order, Override parameter by order, Override parameter by name, Constants connected to ports, Unconnected ports, Expressions connected to ports,	Delay on built-in gates
Generate statements	if,case,for generate, concurrent begin end blocks, genvar,	
Primitives	and, or, nand, nor, xor, xnor,not, notif0, notif1, buf, bufif0, bufif1, tran,	User defined primitives (UDPs), table, pullup, pulldown, pmos, nmos, cmos, rpmos, rnmos, rcmos, tranif0, tranif1, rtran, rtranif0, rtranif1,
and expressions	+, - (binary and unary)	
Bitwise operations	&, |, ^, ~^, ^~	
Reduction operations	&, |, ^, ~&, ~|, ~^, ^~, !, &&, || , ==, !=, <, <=, >, >=, <<, >>, <<< >>>, {}, {n{}}, ?:, function call	===, !==
Event control	event or, @ (partial), event or using comma syntax, posedge, negedge (partial),	Event trigger (->), delay and wait (#)
Bit and part selects	Bit select, Bit select of array element, Constant part select, Variable part select (+:, -:), Variable bit-select on left side of an assignment	
Continuous assignments	net and wire declaration, assign	Using delay
Procedural blocks	always (exactly one @ required),	initial
Procedural statements	;, begin-end, if-else, repeat, case, casex, casez, default, for-while-forever-disable(partial),	fork, join
Procedural assignments	blocking (=), non-blocking (<=)	force, release
Functions and tasks	Functions, tasks	
Compiler directives	`define, `undef, `resetall, `ifndef, `elsif, `line, `ifdef, `else, `endif, `include	

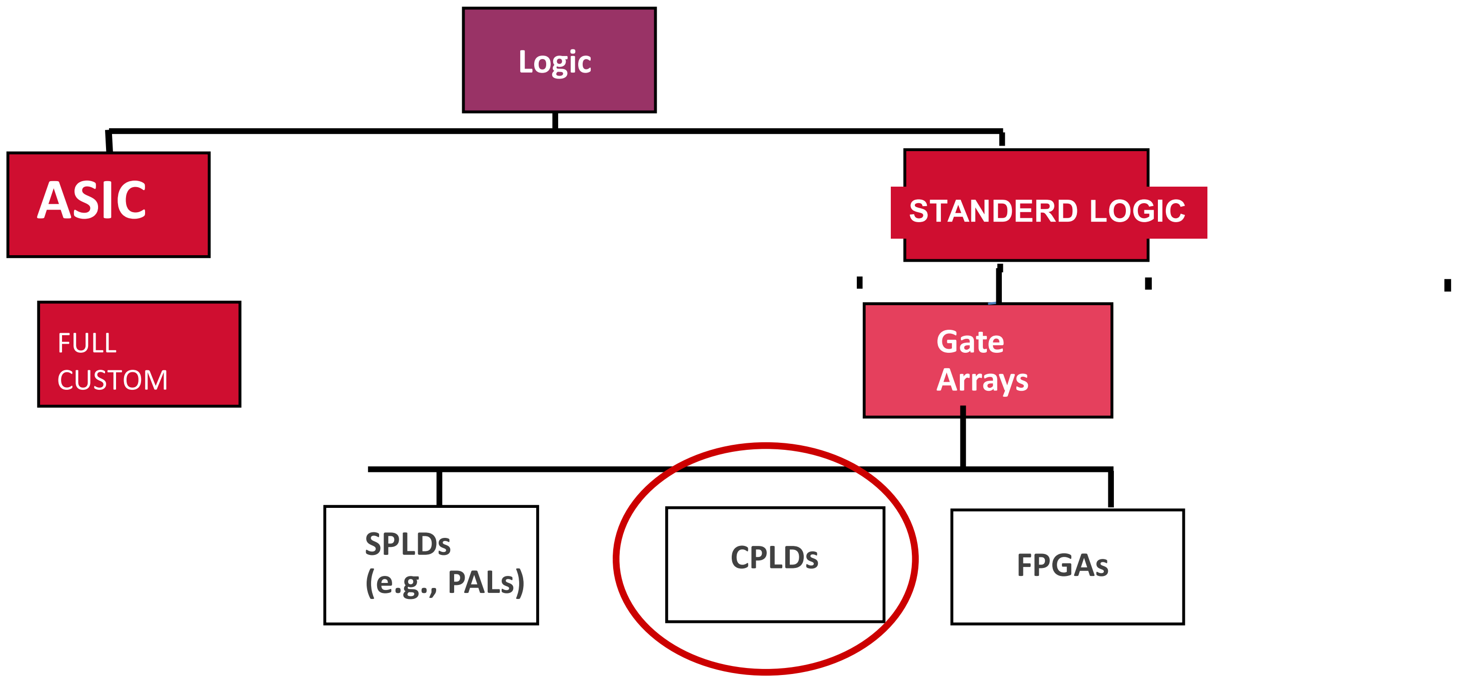
Delay of the circuit shown below is

A. 8
B. 11
C. 12
D. 15



Introduction to FPGA & CPLD Devices

Hierarchy of Logic Implementations



Acronyms

SPLD = Simple Prog. Logic Device
PAL = Prog. Array of Logic
CPLD = Complex PLD
FPGA = Field Prog. Gate Array
ASIC = Application Specific IC

Common Resources

Configurable Logic Blocks (CLB)
- Memory Look-Up Table (LUT)
- AND-OR planes
- Simple gates

Input / Output Blocks (IOB)
- Bidirectional, latches, inverters, pullup/pulldowns

Interconnect or Routing
- Local, internal feedback, and global

Two competing implementation approaches

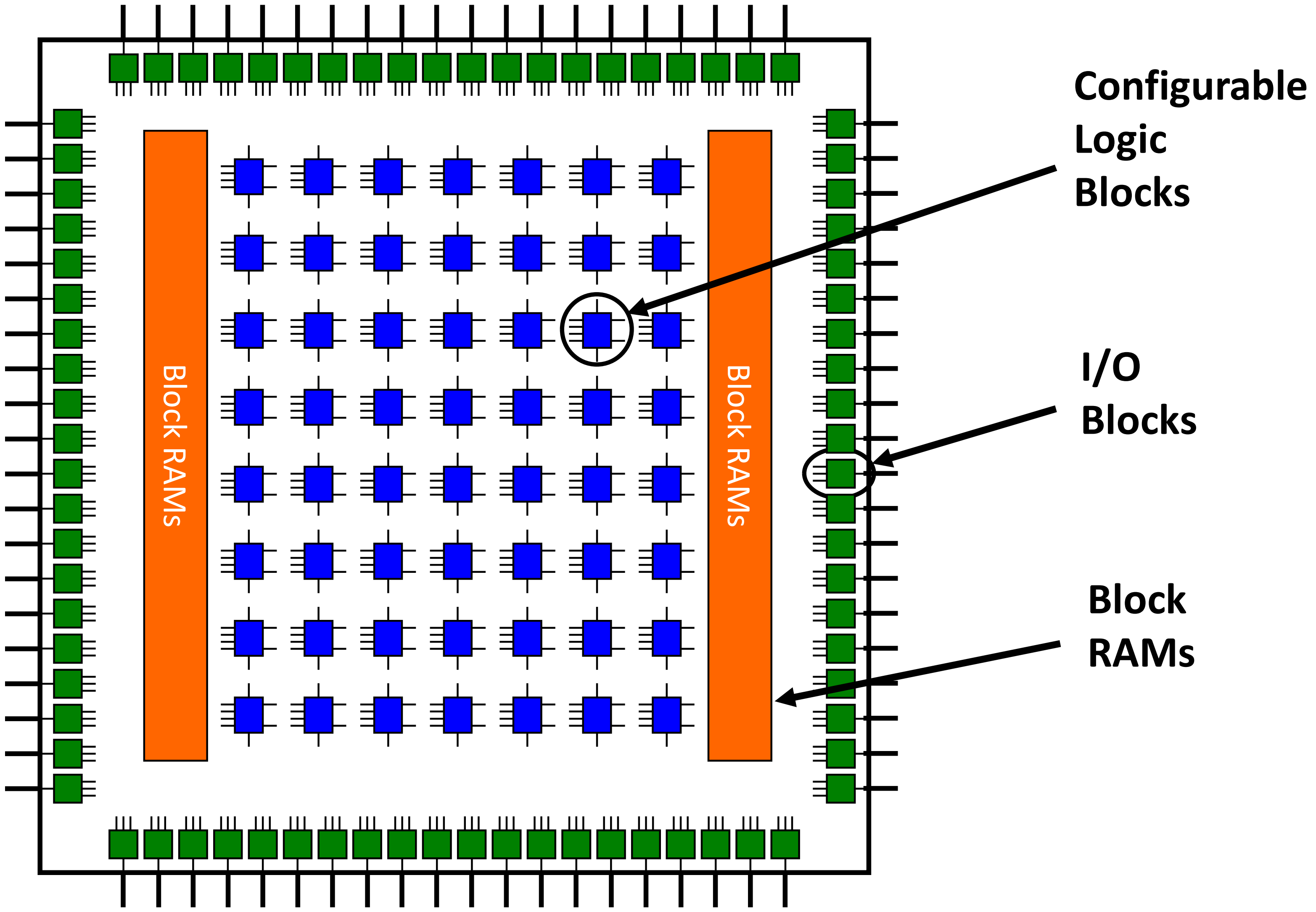
ASIC
Application Specific Integrated Circuit

- designs must be sent for expensive and time consuming fabrication in semiconductor foundry
- designed all the way from behavioral description to physical layout
- High performance
- Low power
- Low cost in high volume

FPGA
Field Programmable Gate Array

- bought off the shelf (ready to use) and reconfigured by designers themselves
- no physical layout design; design ends with a bitstream used to configure a device
- Low development cost
- Short time of market
- Reconfigurability

What is an FPGA?



Other FPGA Advantages

- Manufacturing cycle for ASIC is very costly, lengthy and engages lots of manpower
  - Mistakes not detected at design time have large impact on development time and cost
  - FPGAs are perfect for rapid prototyping of digital circuits
- Easy upgrades like in case of software
- Unique applications
  - reconfigurable computing

Major FPGA Vendors

SRAM-based FPGAs

- **Xilinx, Inc.**
- **Altera Corp.**

$\left.\begin{array}{l}\\ \\ \end{array}\right\}$ **Share over 60% of the market**

- Atmel
- Lattice Semiconductor

Flash & antifuse FPGAs

- Actel Corp.
- Quick Logic Corp.

Xilinx FPGA Families

- Old families
  - XC3000, XC4000, XC5200
  - Old 0.5µm, 0.35µm and 0.25µm technology. Not recommended for modern designs.
- **High-performance families**
  - Virtex (0.22µm)
  - Virtex-E, Virtex-EM (0.18µm)
  - Virtex-II, Virtex-II PRO (0.13µm)
  - Virtex-4 (0.09µm)
- **Low Cost Family**
  - Spartan/XL – derived from XC4000
  - Spartan-II – derived from Virtex
  - Spartan-IIE – derived from Virtex-E
  - **Spartan-3**

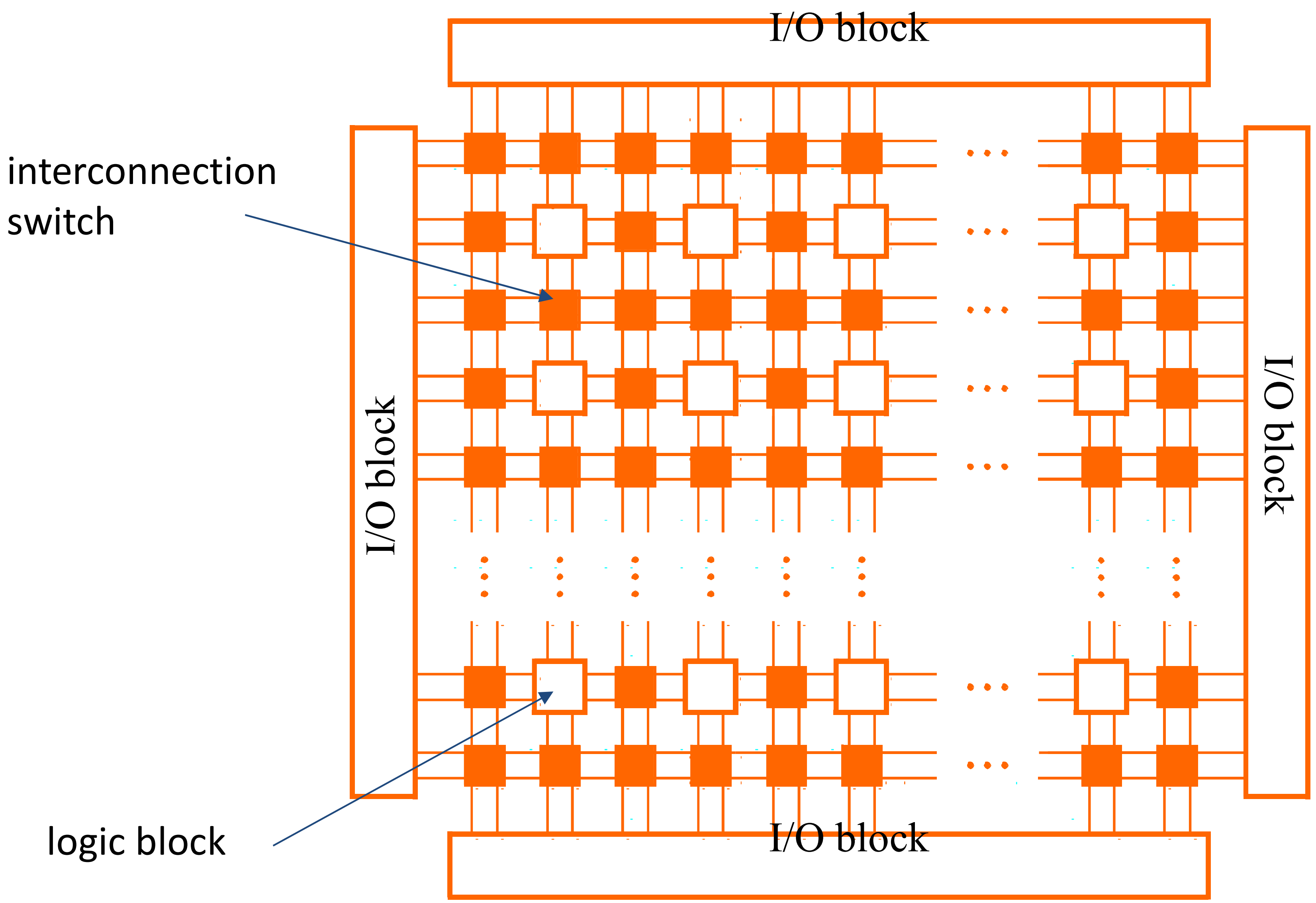
FPGA

- SPLDs and CPLDs are relatively small and useful for simple logic devices
  - Up to about 20000 gates

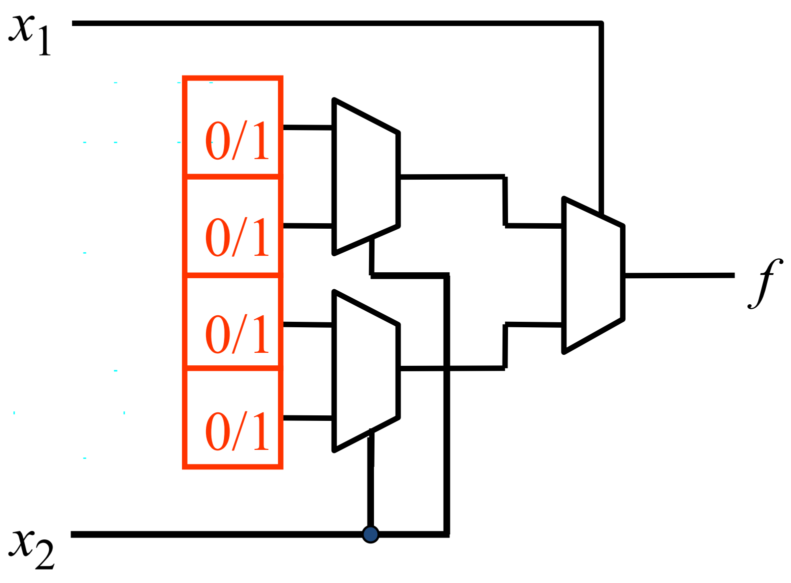
- Field Programmable Gate Arrays (FPGA) can handle larger circuits
  - No AND/OR planes
  - Provide logic blocks, I/O blocks, and interconnection wires and switches

  - Logic blocks provide functionality
  - Interconnection switches allow logic blocks to be connected to each other and to the I/O pins

- Structure of an FPGA



- LUTs
  - Logic blocks are implemented using a lookup table (LUT)
    - Small number of inputs, one output
    - Contains storage cells that can be loaded with the desired values
    - A 2 input LUT uses 3 MUXes to implement any desired function of 2 variables
      - Shannon's expansion at work!

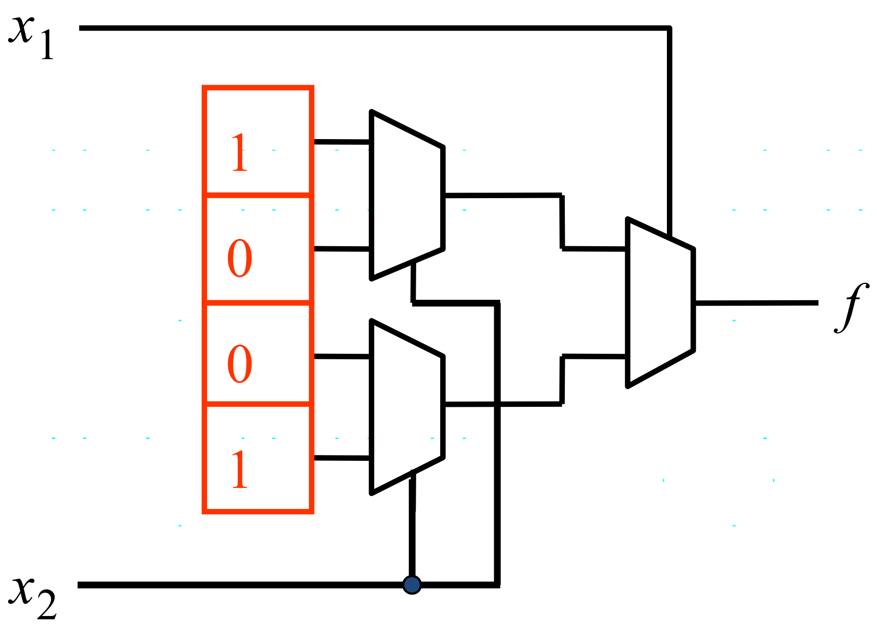


- Example 2 Input LUT

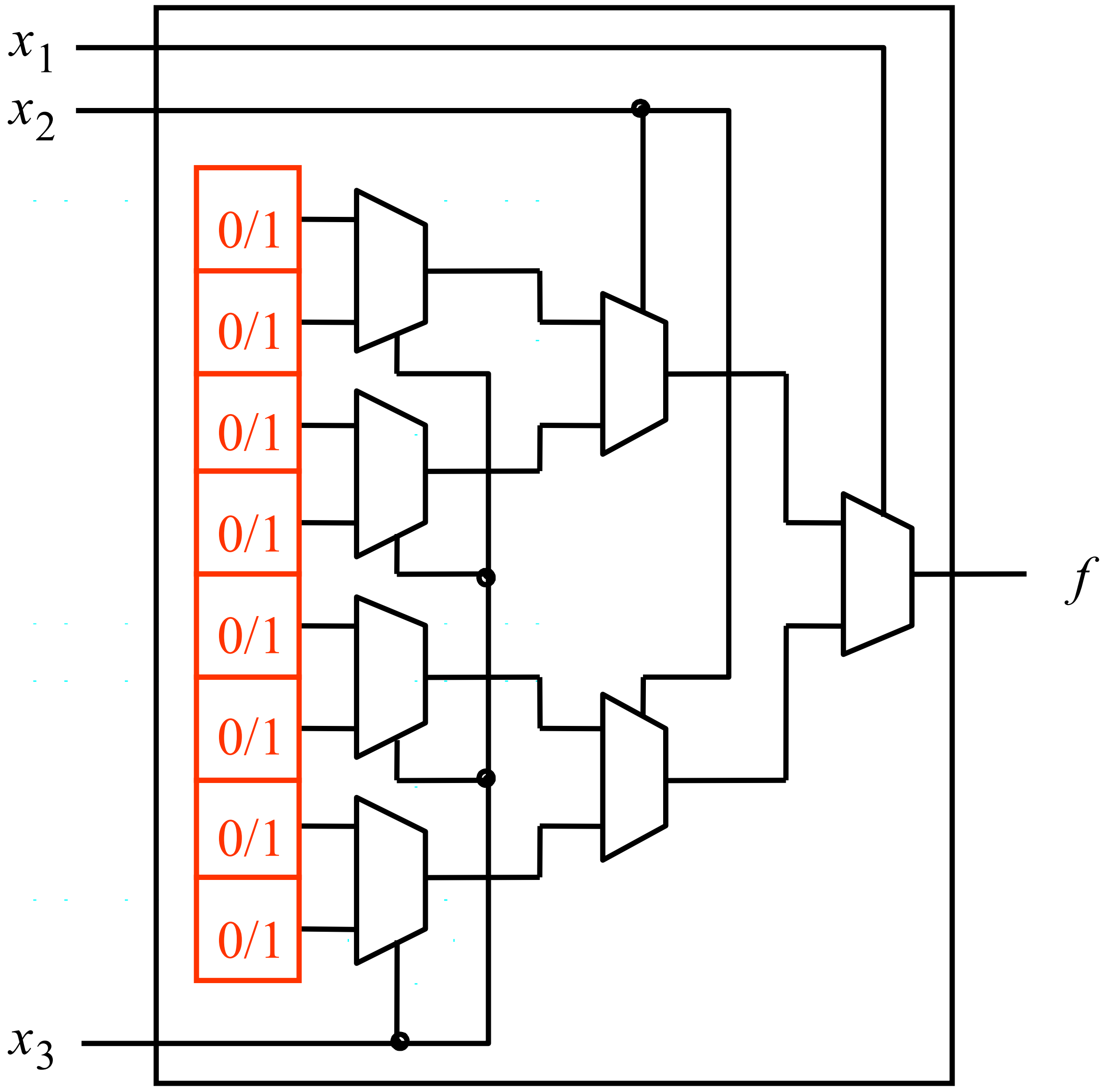
x_1	x_2	f
0	0	1
0	1	0
1	0	0
1	1	1

f = $x_1'x_2' + x_1x_2$, or using Shannon's expansion:

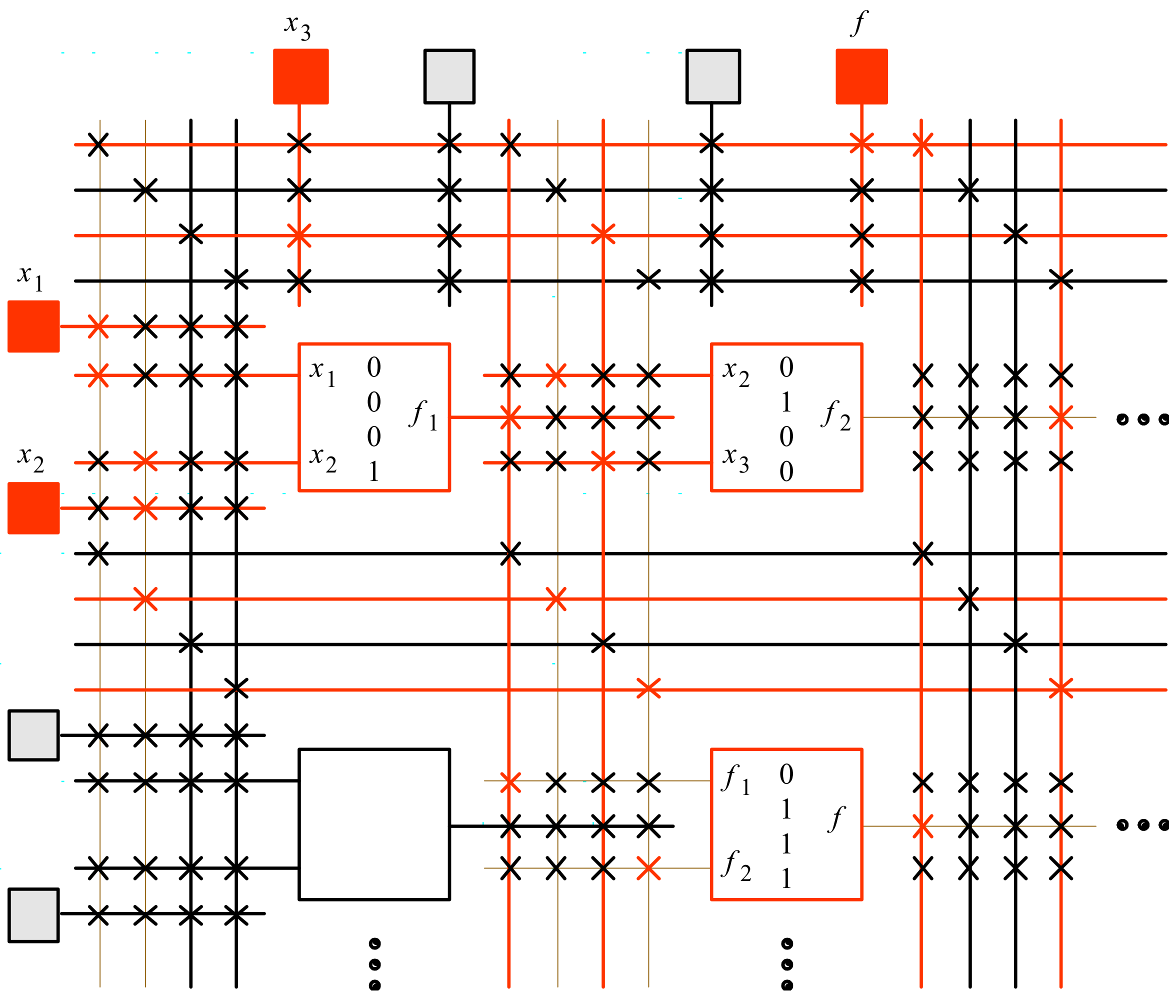
$$f = x_1'(x_2') + x_1(x_2)$$
$$= x_1'(x_2'(1) + x_2(0)) + x_1(x_2'(0) + x_2(1))$$



- 3 Input LUT
  - 7 2x1 MUXes and 8 storage cells are required
  - Commercial LUTs have 4-5 inputs, and 16-32 storage cells

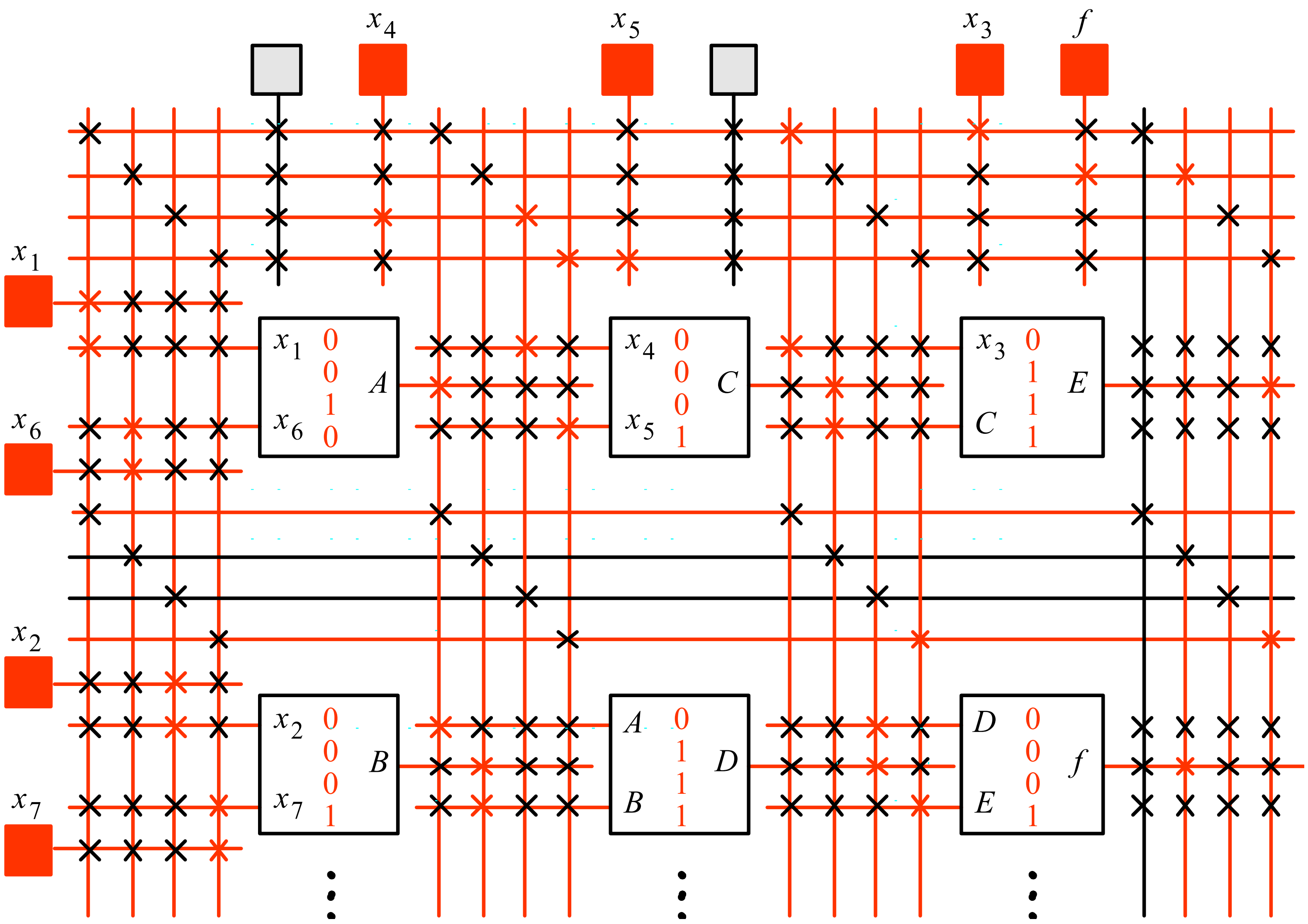


- Example FPGA
  - Use an FPGA with 2 input LUTS to implement the function f = $x_1x_2 + x_2'x_3$
    - $f_1 = x_1x_2$
    - $f_2 = x_2'x_3$
    - $f = f_1 + f_2$



- FPGA Implementation

  $f = (x_1 x_6' + x_2 x_7)(x_3 + x_4 x_5)$



________________ based designs must be sent for expensive and time consuming fabrication in semiconductor foundry

A. ASIC
B. FPGA
C. Both
D. None

________________ based designs bought off the shelf (ready to use) and reconfigured by designers themselves

A. ASIC
B. FPGA
C. Both
D. None

Internal component of FPGA are

A. CLB
B. IOB
C. BRAM
D. All of the above

Look-up-table (LUT) are composed of

A. multiplexer
B. decoder
C. demultiplexer
D. encoder