EECE 2322: Fundamentals of Digital Design and Computer Organization

Lecture 2_3: Digital Components and Verilog

Xiaolin Xu
Department of ECE
Northeastern University

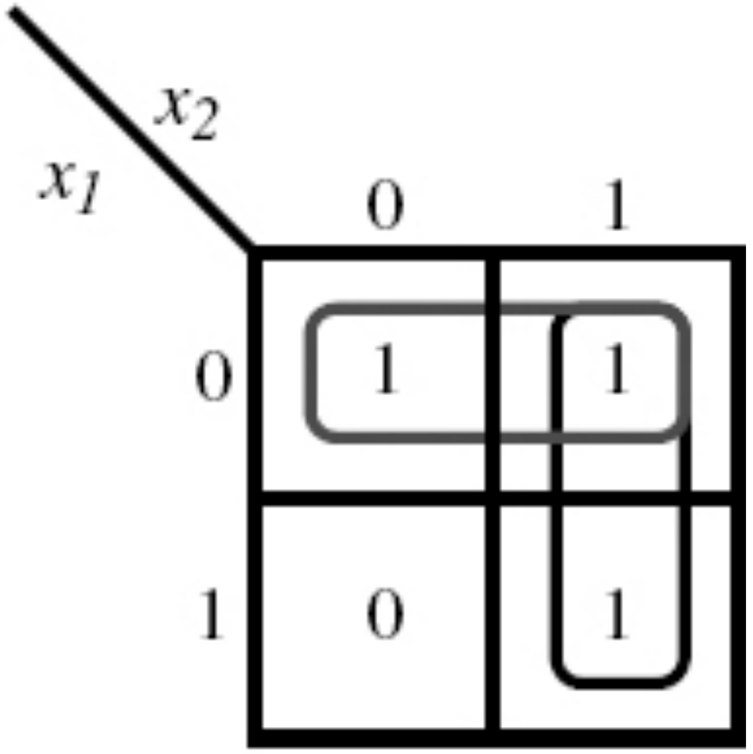
Karnaugh Map

- Simplify a logic function
  - Combine pairs of adjacent 1-cells (minterms) whenever possible
  - Write a sum of product terms that cover all of the 1-cells

Karnaugh Map for 2-input

- What is f()?

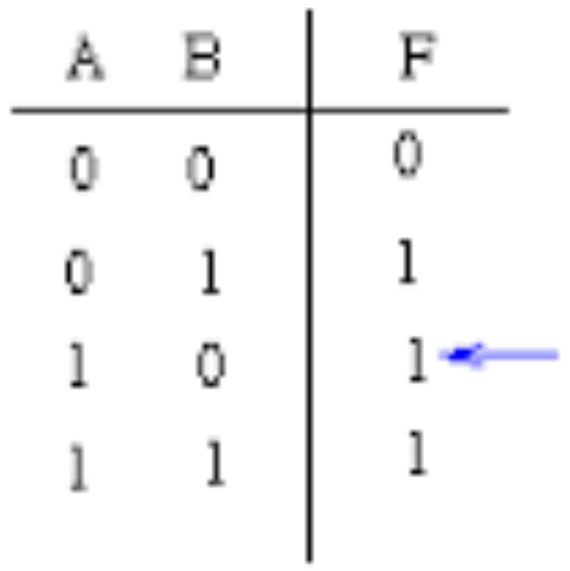
x_1	x_2	f
0	0	1
0	1	1
1	0	0
1	1	1



$f = \overline{x}_1 + x_2$

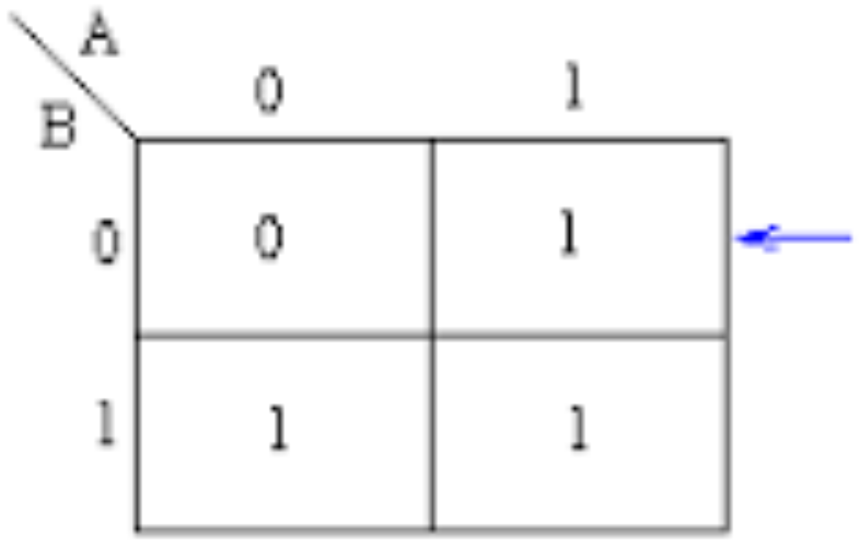
Karnaugh Map for 2-input

- What is F()?



A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Truth Table.

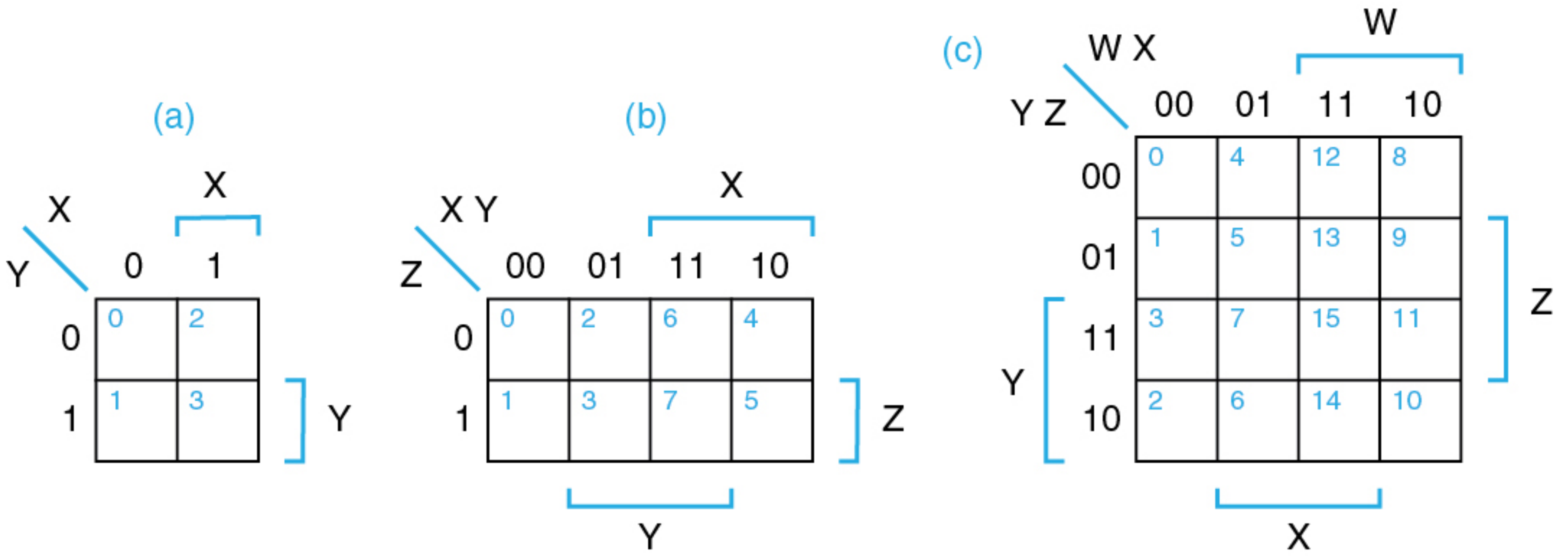


B \ A	0	1
0	0	1
1	1	1

F.

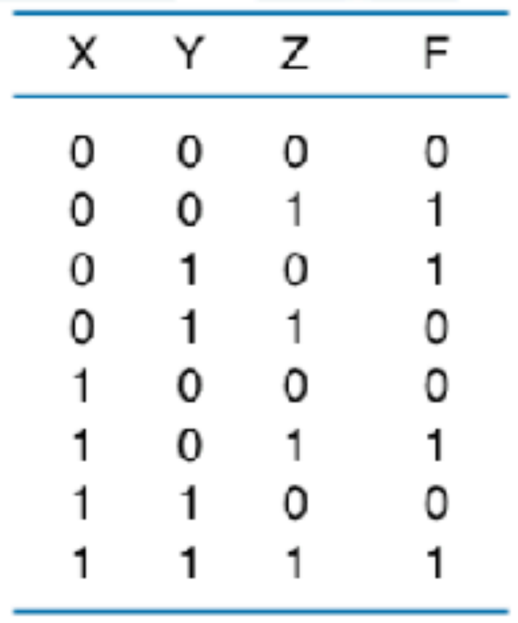
Karnaugh Map

- (A) 2-Variable; (B) 3-Variable; (C) 4-Variable



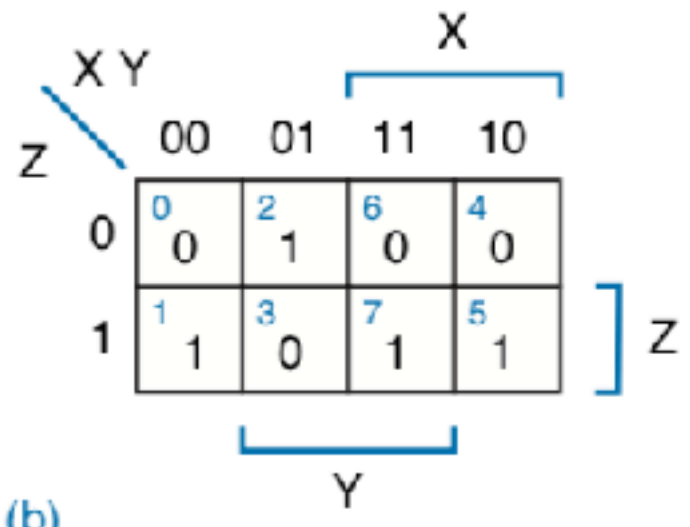
Karnaugh Map

- $F = \Sigma_{X,Y,Z}(1,2,5,7)$:
    - (a) truth table; (b) Karnaugh map; (c) combining adjacent 1-cells.

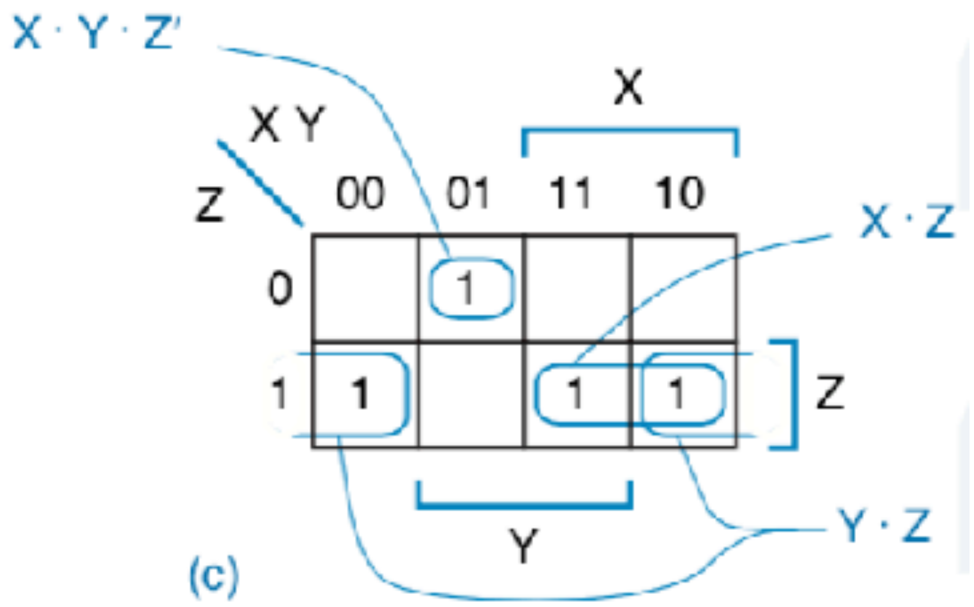


(a)

X	Y	Z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



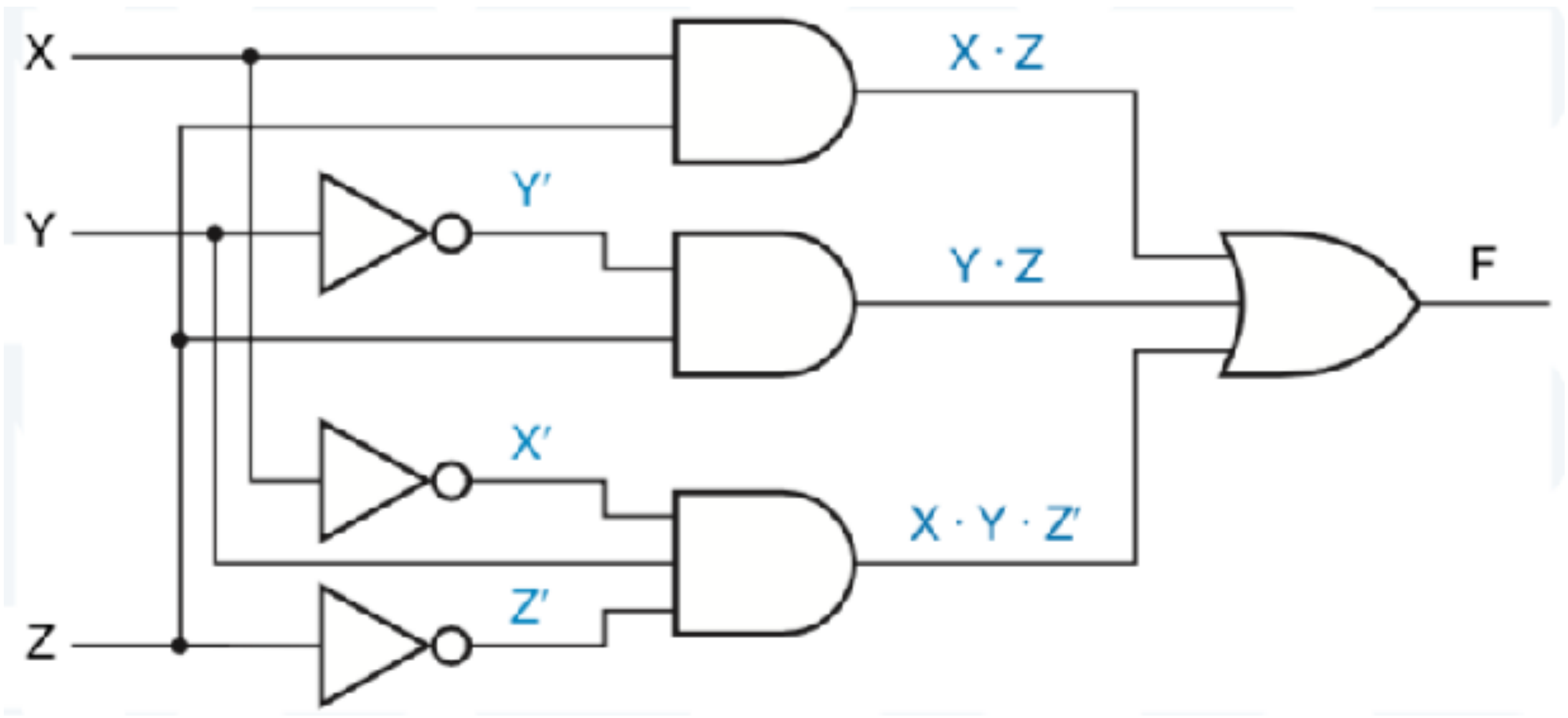
(b)

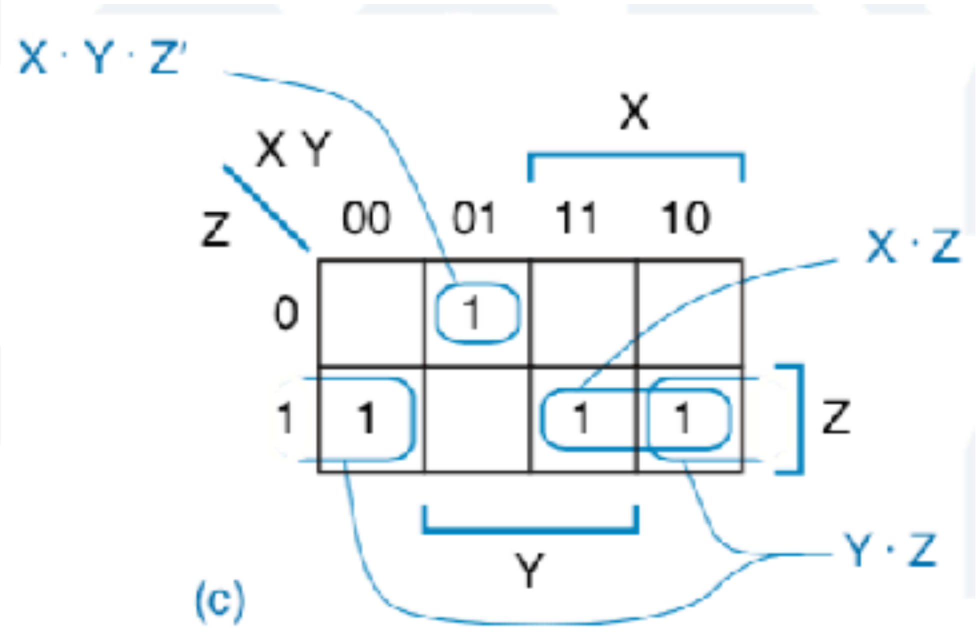


(c)

Karnaugh Map

- $F = \Sigma_{X,Y,Z}(1,2,5,7)$:
  - (a) truth table; (b) Karnaugh map; (c) combining adjacent 1-cells.



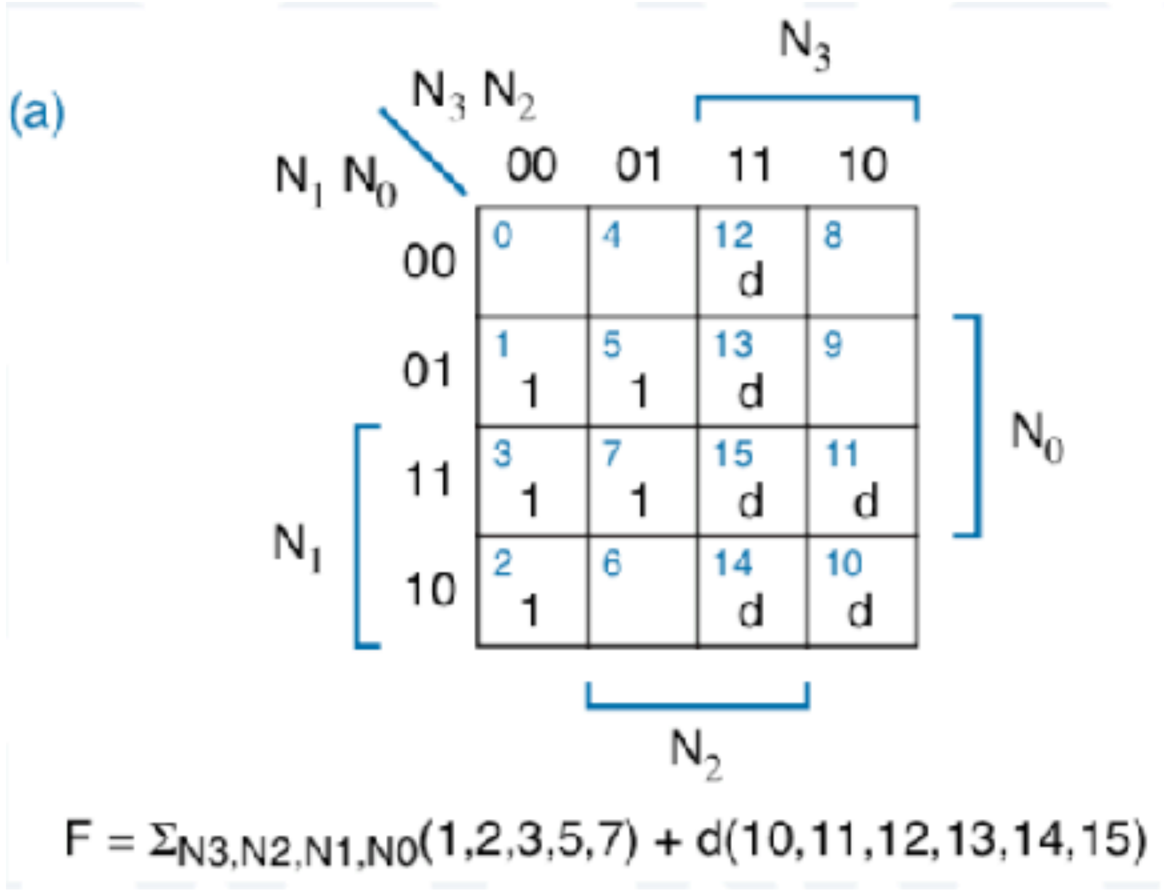


"Don't-Care" Inputs

- Sometimes, the specification of a combinational circuit is such that its output doesn't matter for certain input combinations, called **don't-cares**.
- This may be true because the outputs really don't matter when these input patterns occur, or because these input patterns never occur in practical operation.

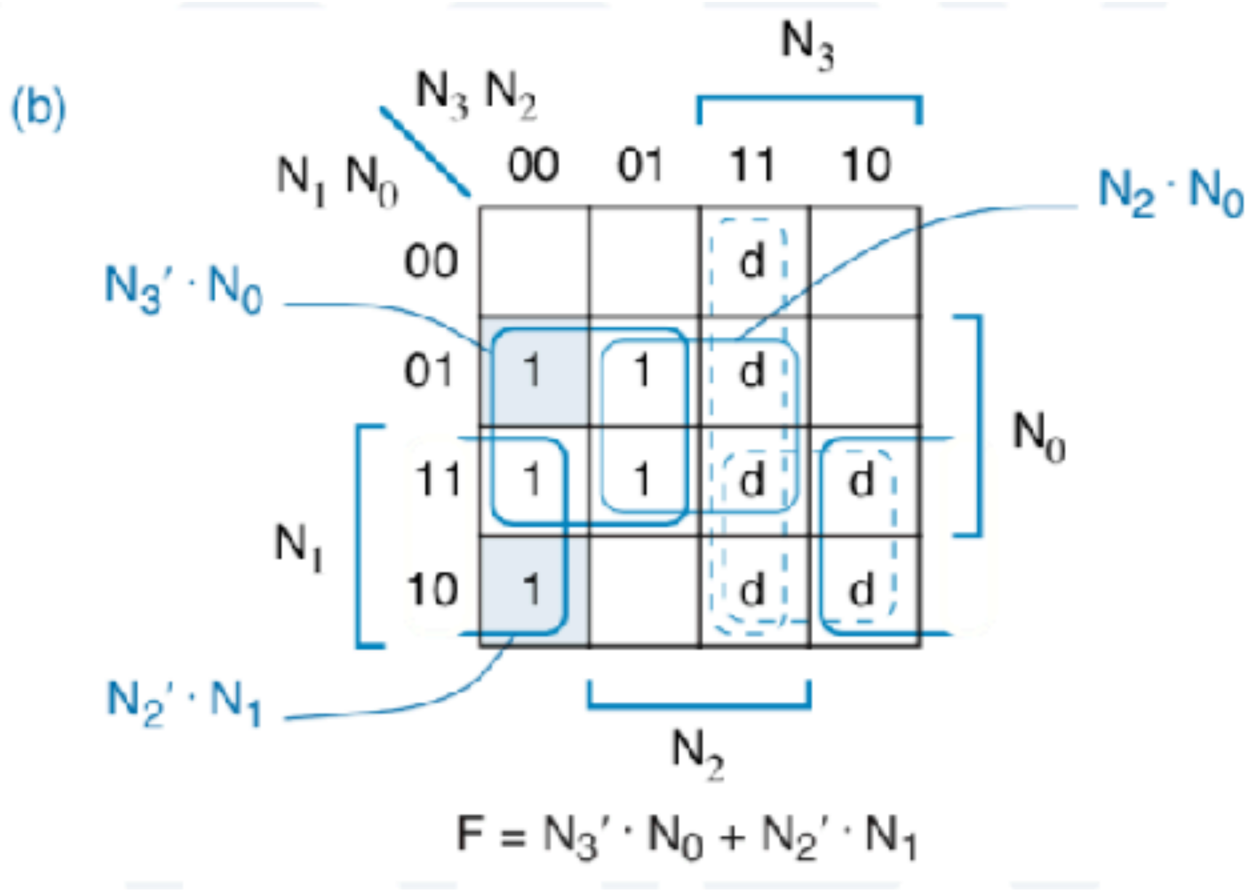
"Don't-Care" Input Combinations

- Input: N0-N3



N1 N0 \ N3 N2	00	01	11	10
00	0	4	12 d	8
01	1 1	5 1	13 d	9
11	3 1	7 1	15 d	11 d
10	2 1	6	14 d	10 d

$F = \Sigma_{N3,N2,N1,N0}(1,2,3,5,7) + d(10,11,12,13,14,15)$



N1 N0 \ N3 N2	00	01	11	10
00			d	
01	1	1	d	
11	1	1	d	d
10	1		d	d

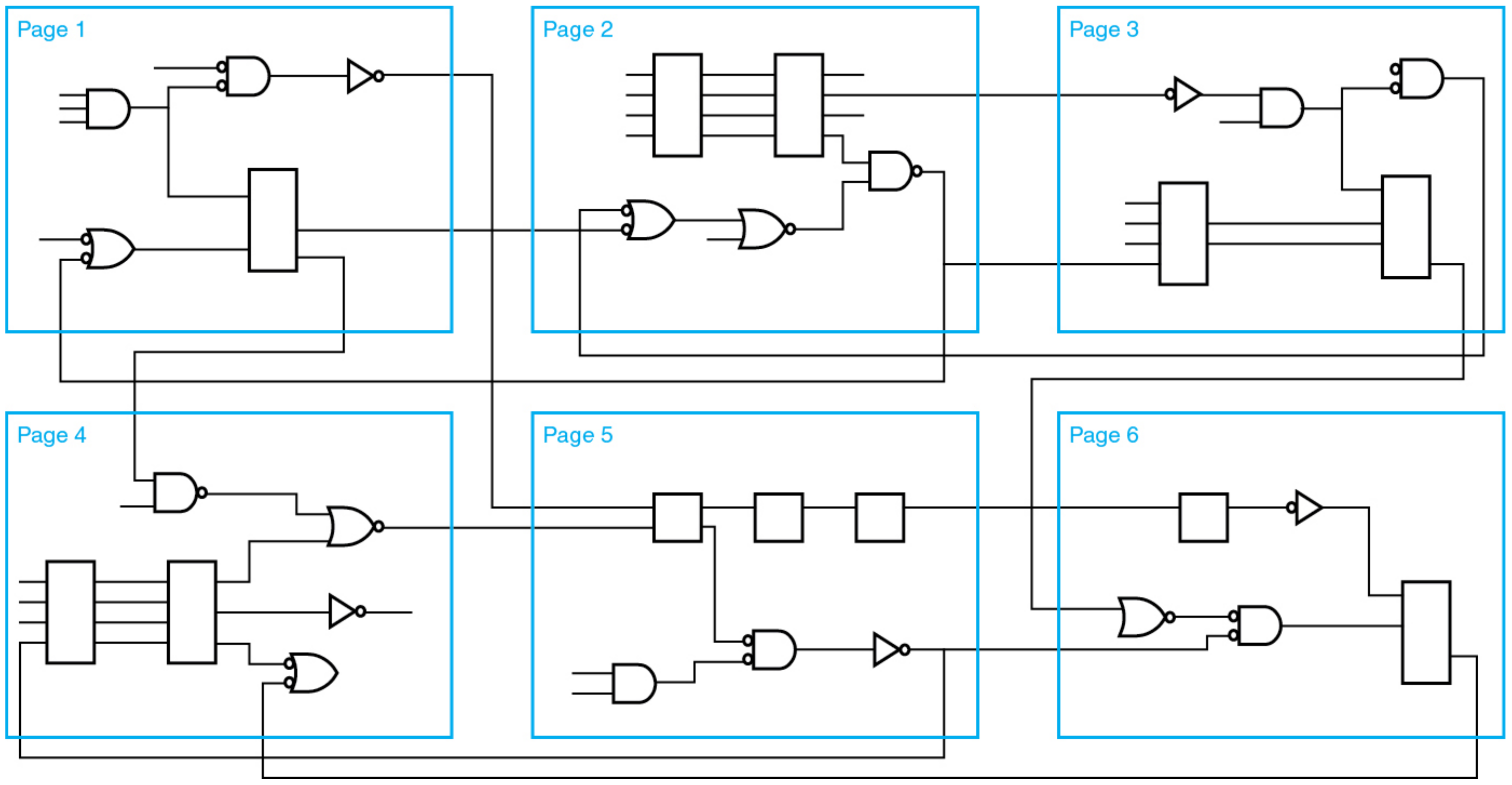
$F = N_3' \cdot N_0 + N_2' \cdot N_1$

Hardware Description Language (HDL)

- Very-High-Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL):
  - Modeling of digital systems
  - Concurrent and sequential statements
  - Machine-readable specification
  - Man- and machine-readable documentation
- International Standards:
  - IEEE Std 1076-1987 - IEEE Std 1076-1993

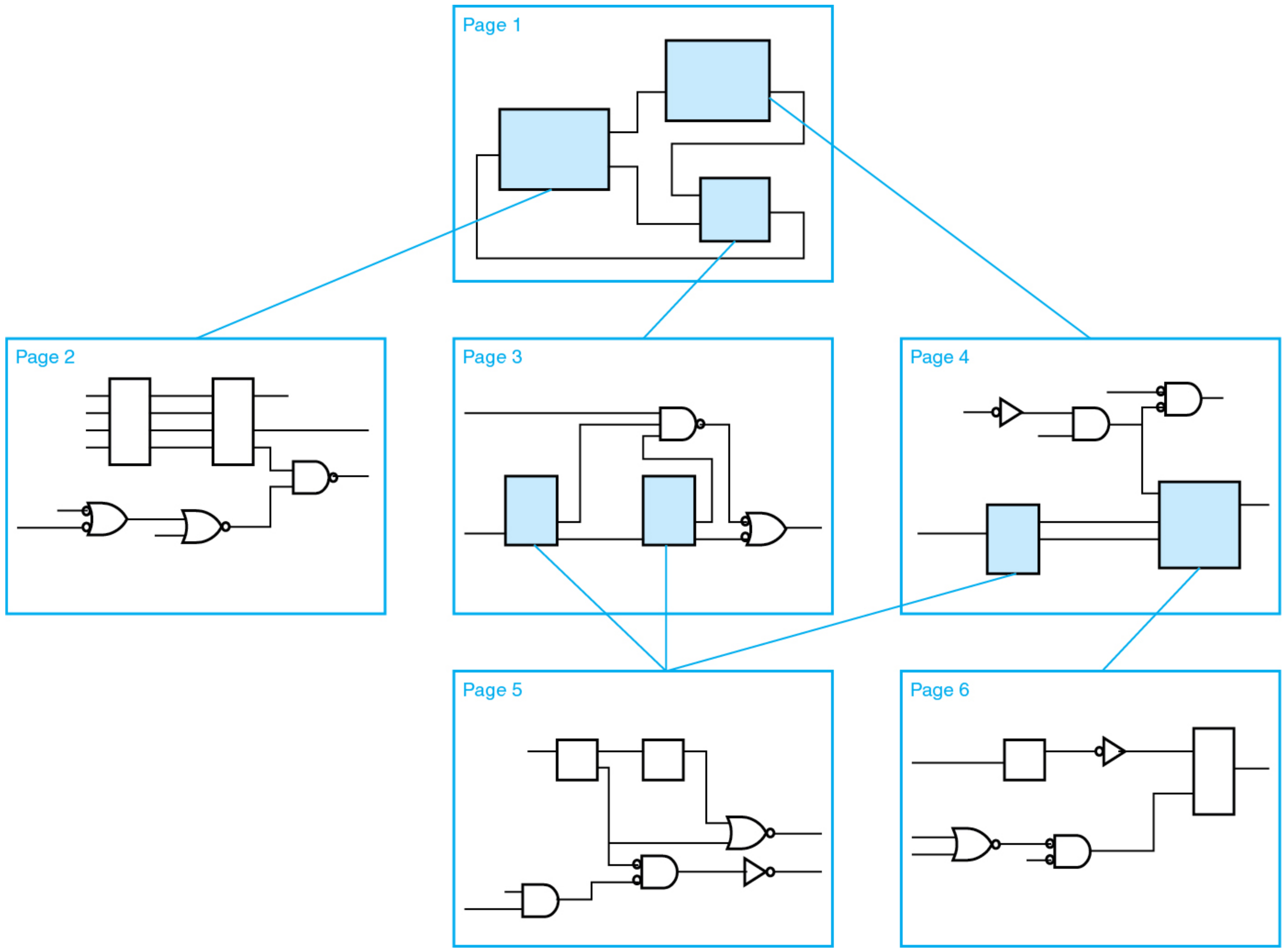
Hardware Description Language (HDL)

- Review the design schematic
  - Flat

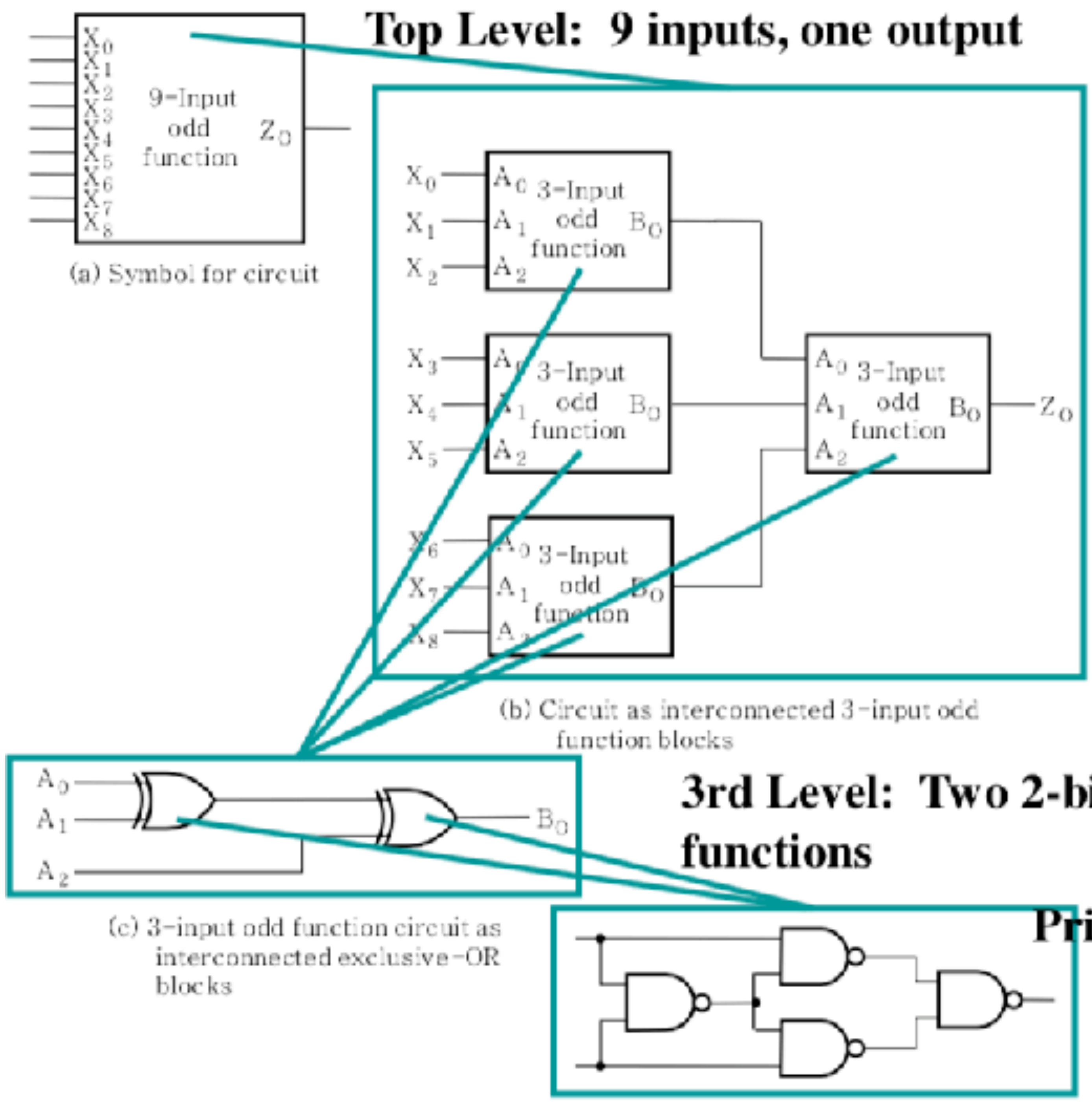


Hardware Description Language (HDL)

- Review the design schematic
  - Hierarchical



Hierarchy Example



(a) Symbol for circuit

(b) Circuit as interconnected 3-input odd function blocks

(c) 3-input odd function circuit as interconnected exclusive-OR blocks

(d) Exclusive-OR block as interconnected NANDs

Top Level: 9 inputs, one output

2nd Level: Four 3-bit odd parity trees in two levels

Design requires 4 X 2 X 4 = 32 2-input NAND gates

3rd Level: Two 2-bit exclusive-OR functions

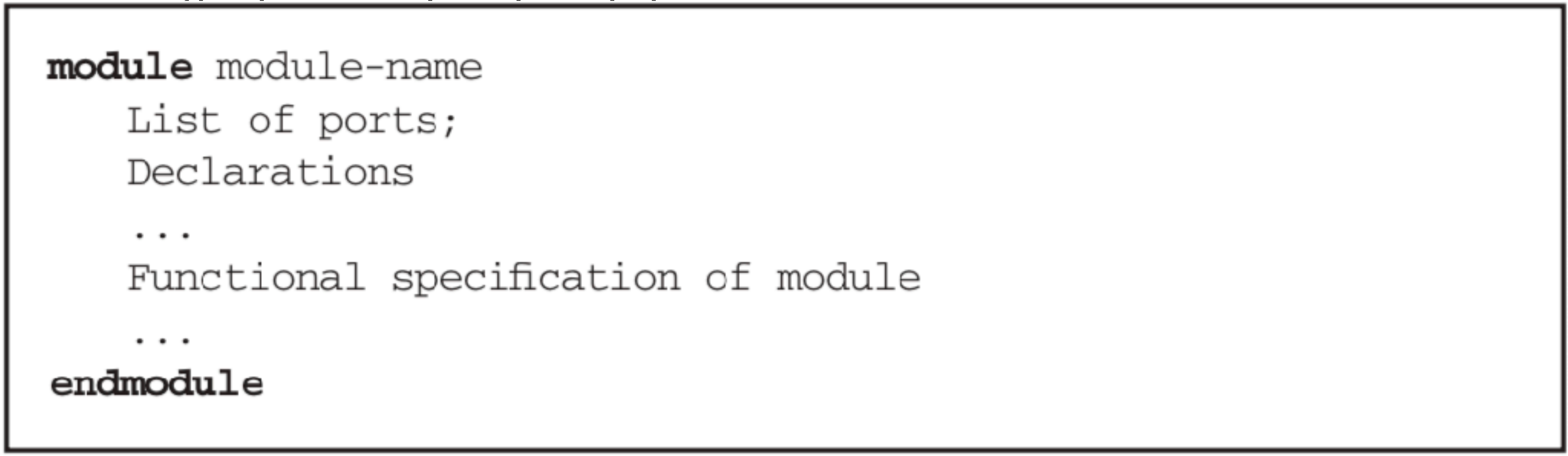
Primitives: Four 2-input NAND gates

Elements of Verilog: Hardware Modules

- Verilog hardware description language (HDL) is used to describe hardware modules of a system and complete systems.
  - keyword module, Name of the module, A list of ports of the hardware module, the module functionality specification, and the keyword endmodule.
  - Following a module name and its list of ports, usually variables, wires, and module parameters are declared.
  - After the declarations, statements in a module specify its functionality. This part defines how output ports react to changes on the input ports.

Elements of Verilog: Hardware Modules

- Verilog hardware description language (HDL) is used to describe hardware modules of a system and complete systems.
  - keyword module, Name of the module, A list of ports of the hardware module, the module functionality specification, and [illegible]



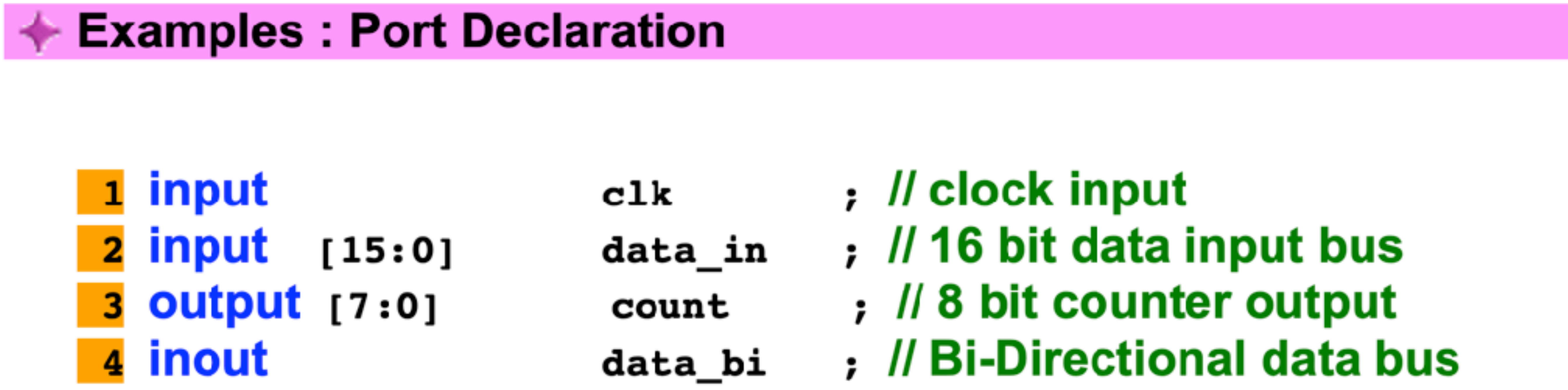
```
module module-name
    List of ports;
    Declarations
    ...
    Functional specification of module
    ...
endmodule
```

Syntax of Verilog Input/Output Declarations

- **Ports**
  - Ports allow communication between a module and its environment.
  - All but the top-level modules in a hierarchy have ports.
  - Ports can be associated by order or by name.
- You declare ports to be input, output or inout. The port declaration syntax is :
  - input [range_val:range_var] list_of_identifiers;
  - output [range_val:range_var] list_of_identifiers;
  - inout [range_val:range_var] list_of_identifiers;

Syntax of Verilog Input/Output Declarations

- **Ports**
  - Ports allow communication between a module and its environment.
  - All but the top-level modules in a hierarchy have ports.
  - Ports can be associated by order or by name.



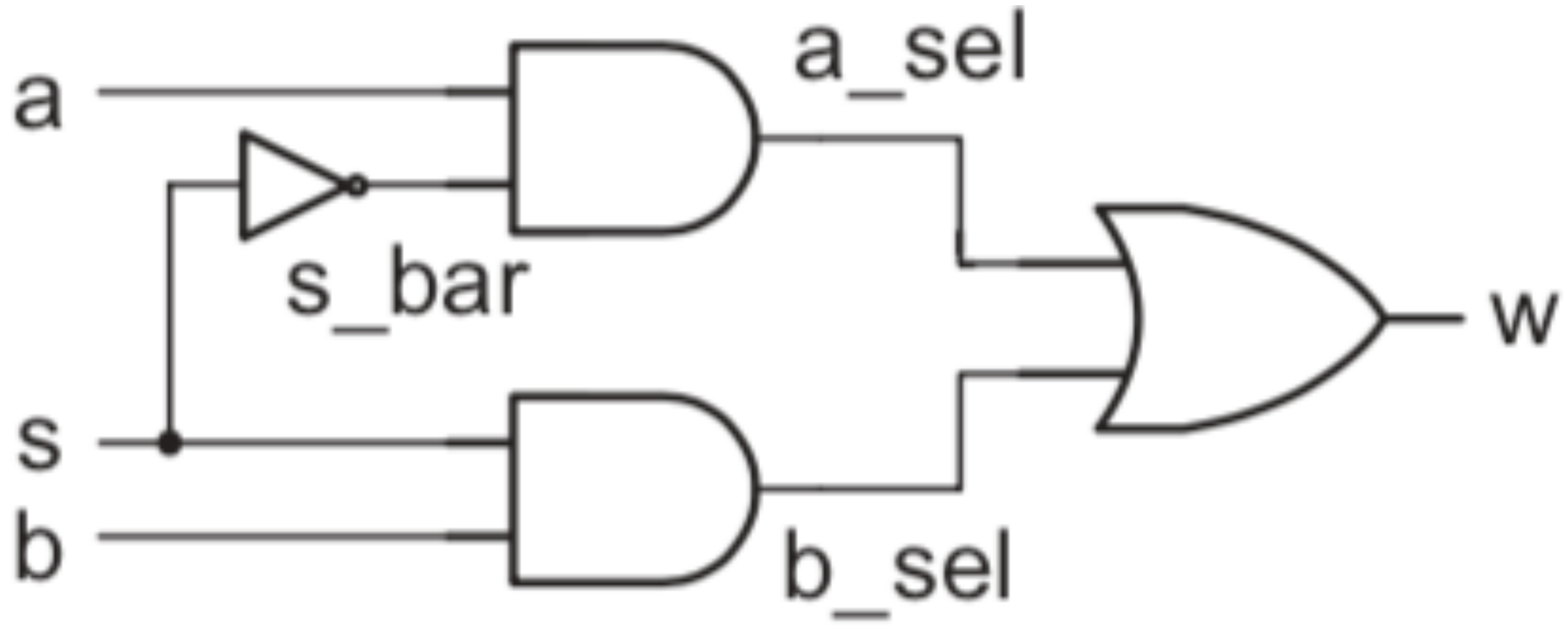
Examples : Port Declaration

```
input                clk       ; // clock input
input   [15:0]       data_in   ; // 16 bit data input bus
output  [7:0]        count     ; // 8 bit counter output
inout                data_bi   ; // Bi-Directional data bus
```

  - inout [range_val:range_var] list_of_identifiers;

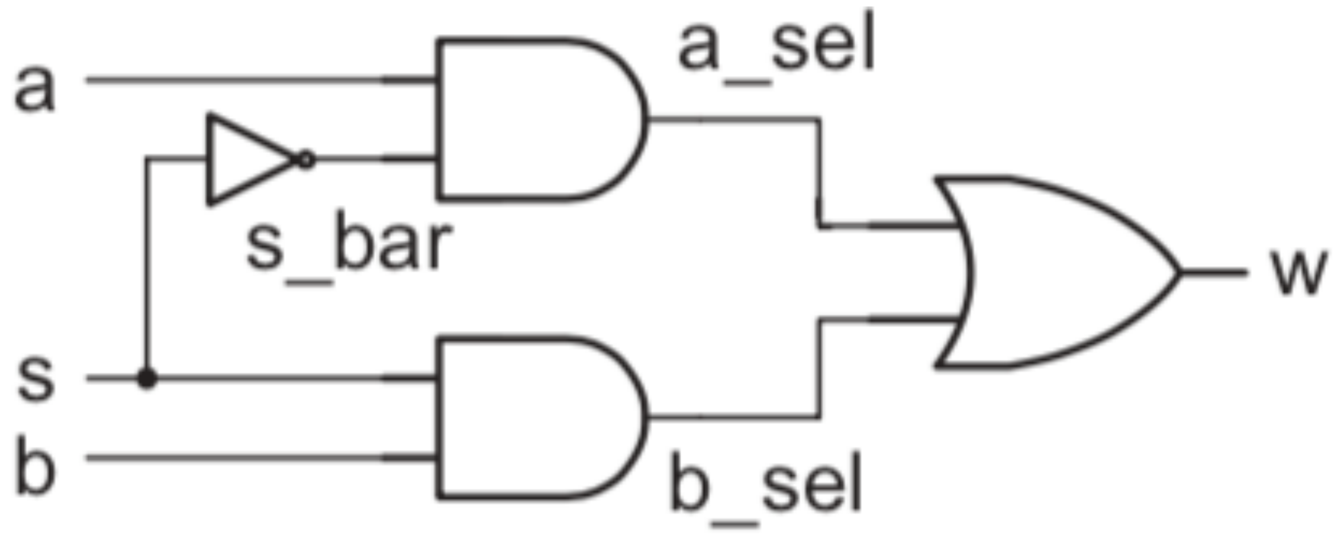
Elements of Verilog: Hardware Modules

- A Multiplexer Using Basic Gates
  - Basic components?



Elements of Verilog: Hardware Modules

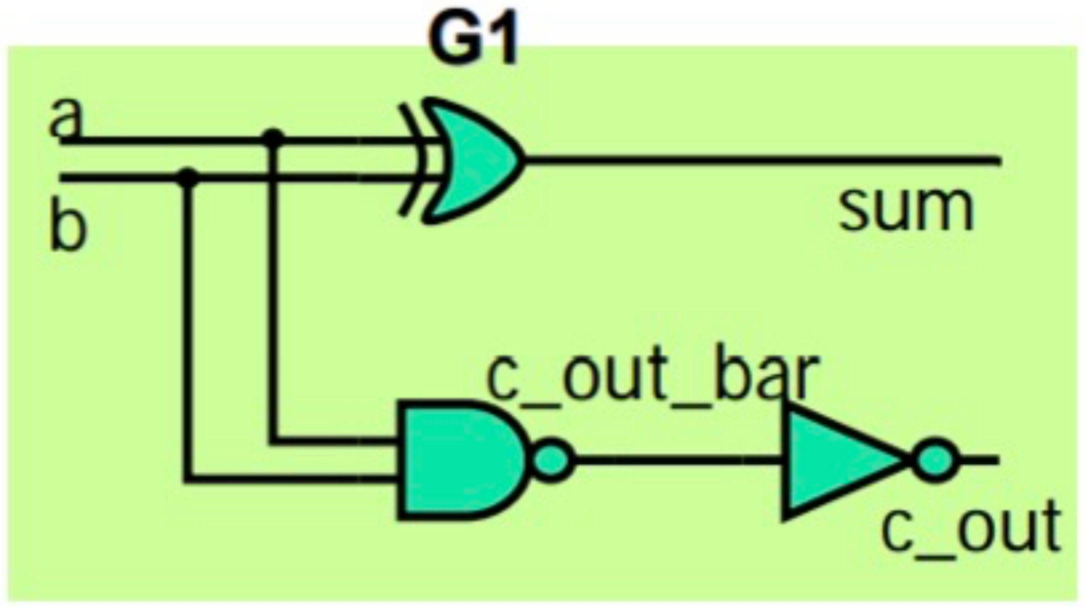
- A Multiplexer Using Basic Gates
  - Basic components?



```
module MultiplexerA (input a, b, s, output w);
   wire a_sel, b_sel, s_bar;
   not U1 (s_bar, s);
   and U2 (a_sel, a, s_bar);
   and U3 (b_sel, b, s);
   or  U4 (w, a_sel, b_sel);
endmodule
```

Write the Code By Yourself

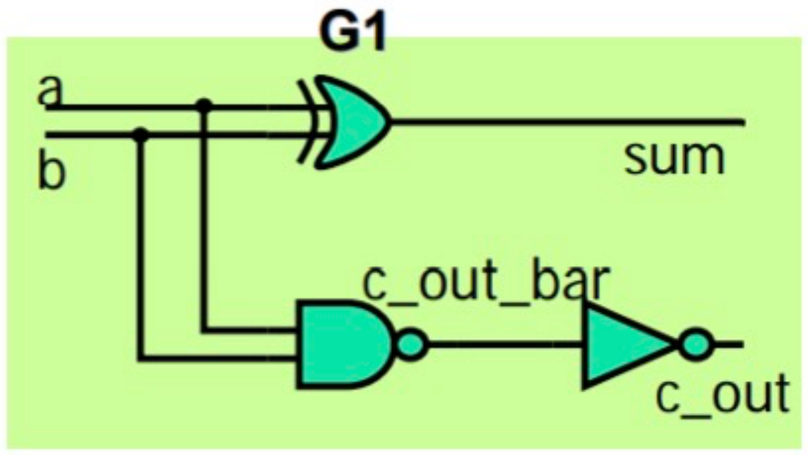
- A Multiplexer Using Basic Gates
  - Basic components?

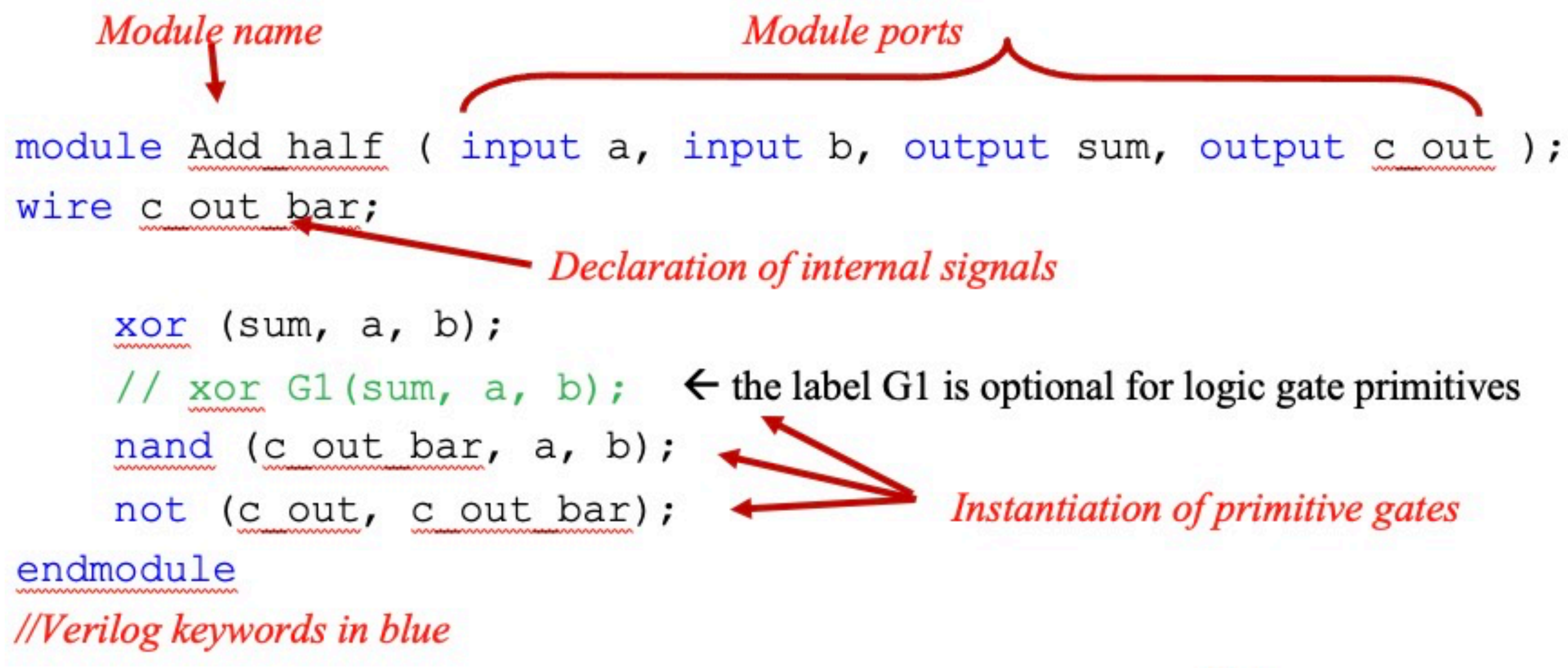


```
module MultiplexerA (input a, b, s, output w);
   wire a_sel, b_sel, s_bar;
   not U1 (s_bar, s);
   and U2 (a_sel, a, s_bar);
   and U3 (b_sel, b, s);
   or  U4 (w, a_sel, b_sel);
endmodule
```

Write the Code By Yourself

- A Multiplexer Using Basic Gates
  - Basic components?





Module name

Module ports

```
module Add_half ( input a, input b, output sum, output c_out );
wire c_out_bar;

    xor (sum, a, b);
    // xor G1(sum, a, b);
    nand (c_out_bar, a, b);
    not (c_out, c_out_bar);
endmodule
//Verilog keywords in blue
```

Declaration of internal signals

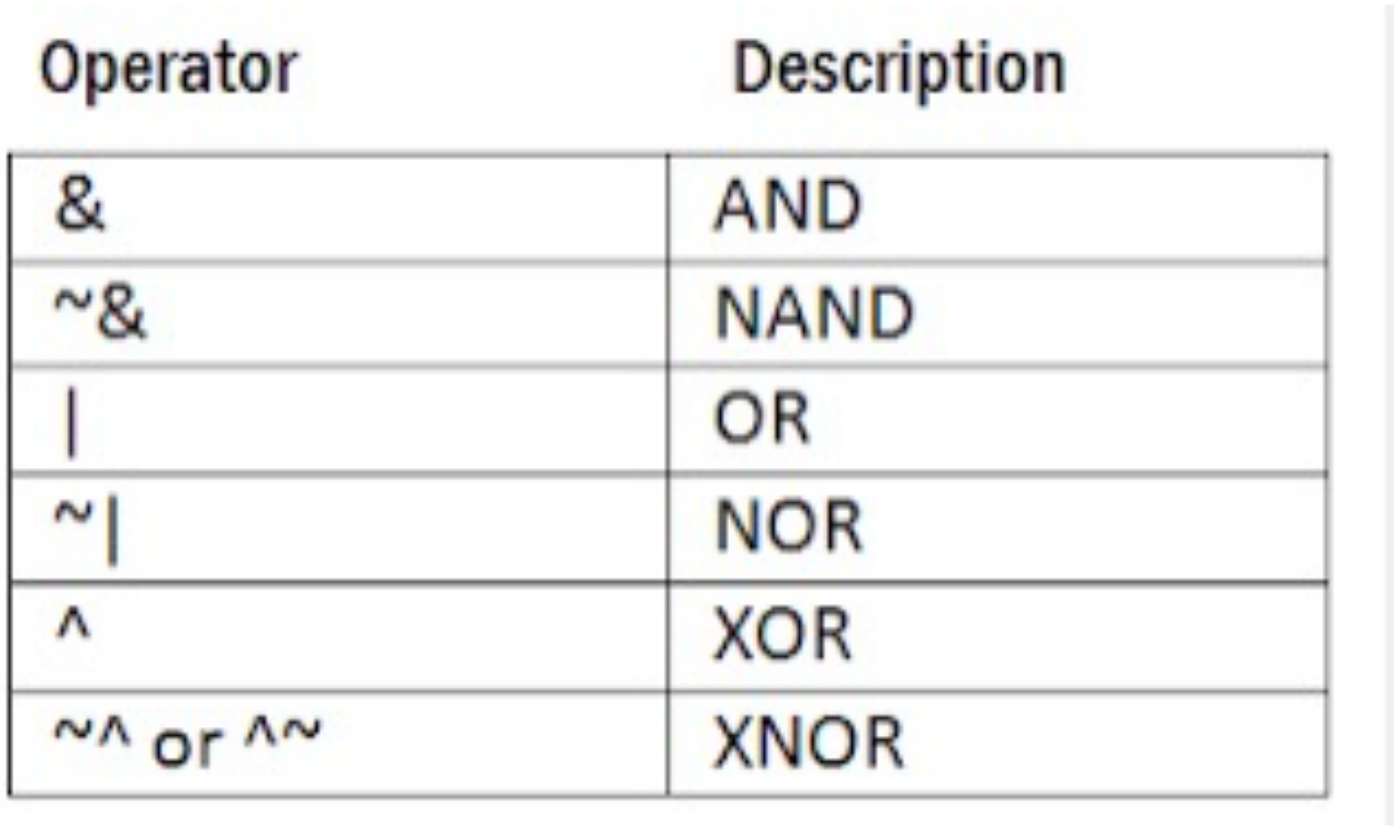
← the label G1 is optional for logic gate primitives

Instantiation of primitive gates

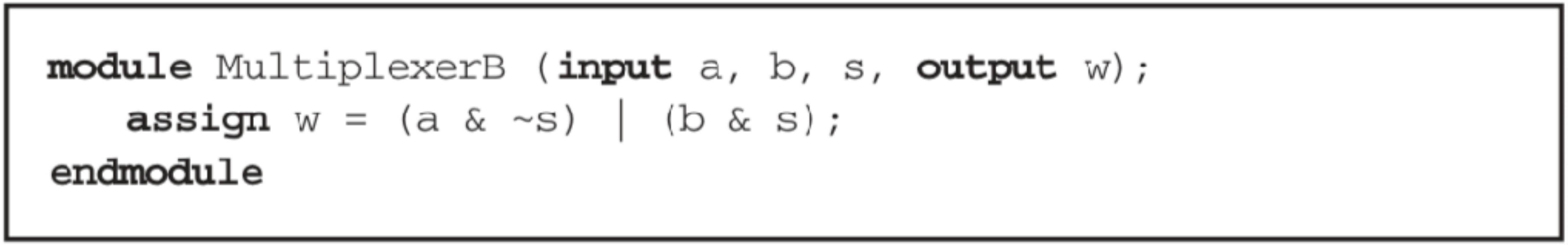
Assign statements

- Instead of describing a component using primitive gates, boolean expressions can be used to describe the logic
- Verilog assign statements can be used for assigning results of these expressions to various outputs.
    - The statement shown in the body of the MultiplexerB module continuously drives w with its right-hand side expression.

Verilog: Operator



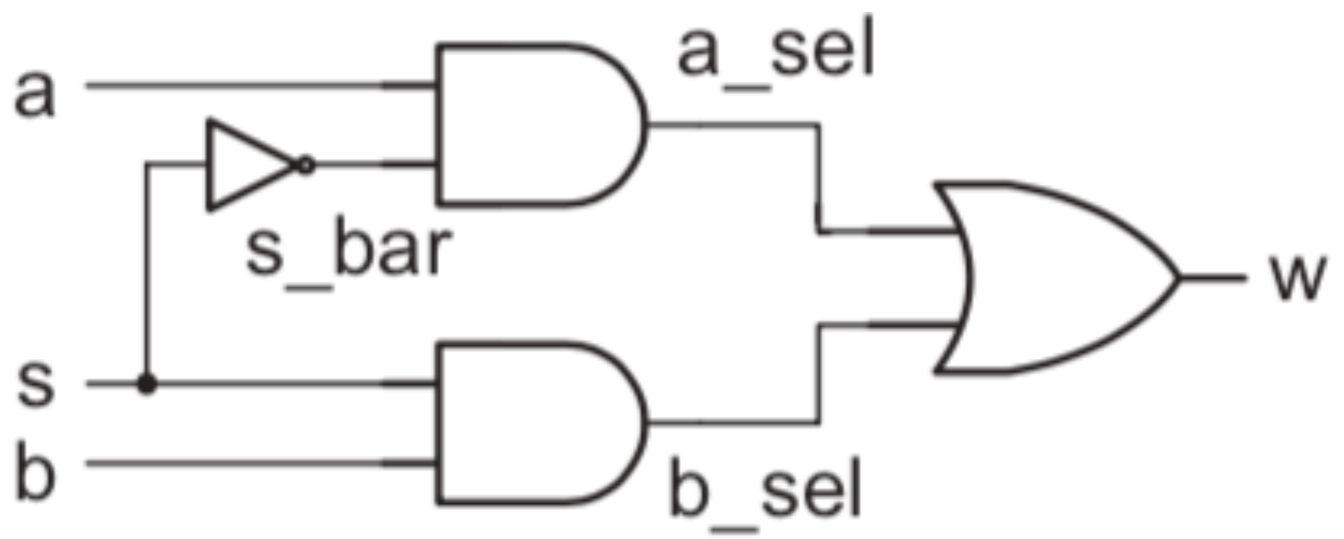
Operator	Description
&	AND
~&	NAND
|	OR
~|	NOR
^	XOR
~^ or ^~	XNOR



```
module MultiplexerB (input a, b, s, output w);
   assign w = (a & ~s) | (b & s);
endmodule
```

Elements of Verilog: assign

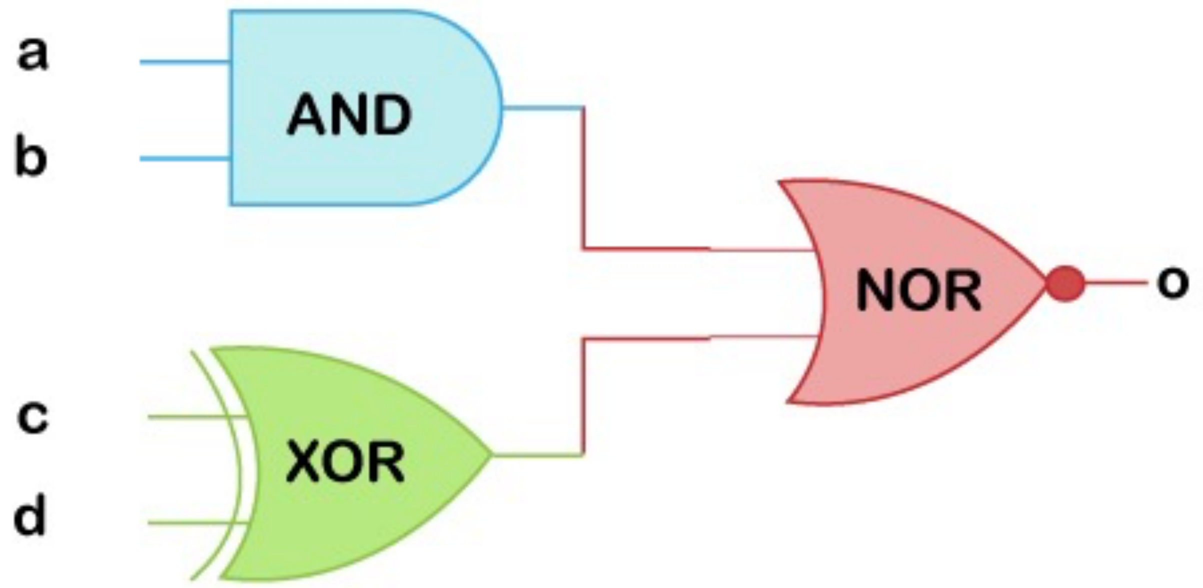
- Assign statement and Boolean expression



```
module MultiplexerB (input a, b, s, output w);
    assign w = (a & ~s) | (b & s);
endmodule
```

Elements of Verilog: assign

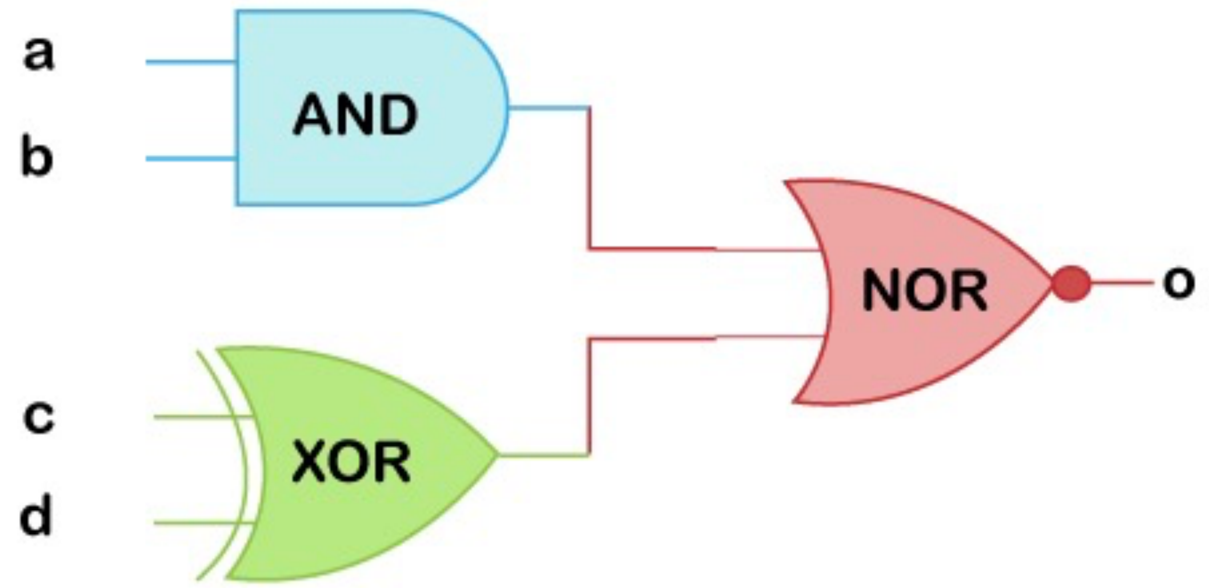
- Assign statement and Boolean expression
- Write the code by yourself

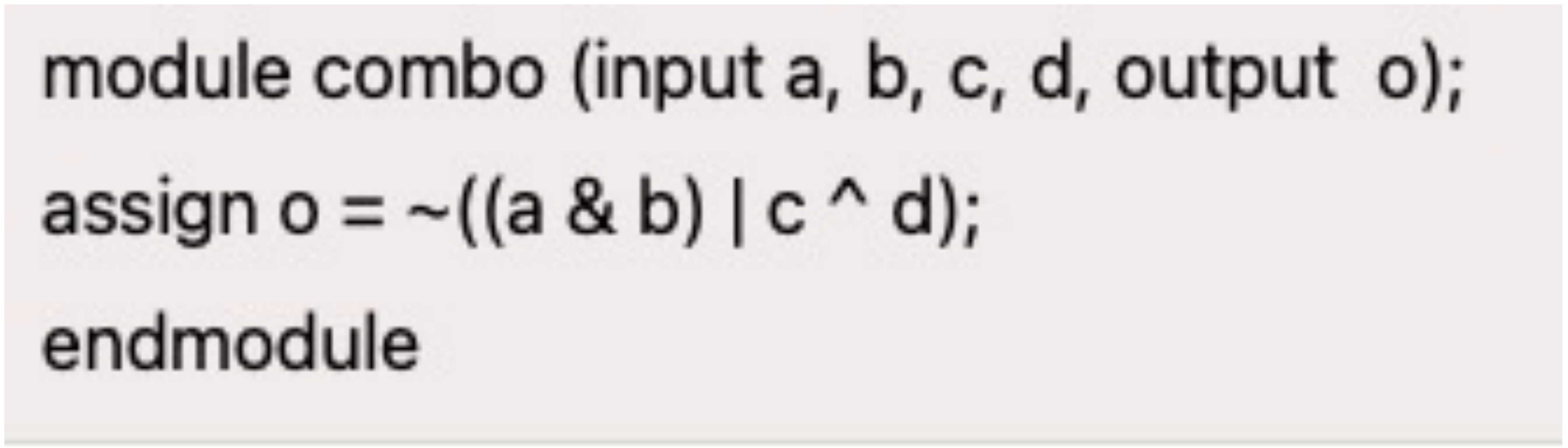


```
module MultiplexerB (input a, b, s, output w);
    assign w = (a & ~s) | (b & s);
endmodule
```

Elements of Verilog: assign

- Assign statement and Boolean expression
- Write the code by yourself





```
module combo (input a, b, c, d, output  o);
assign o = ~((a & b) | c ^ d);
endmodule
```

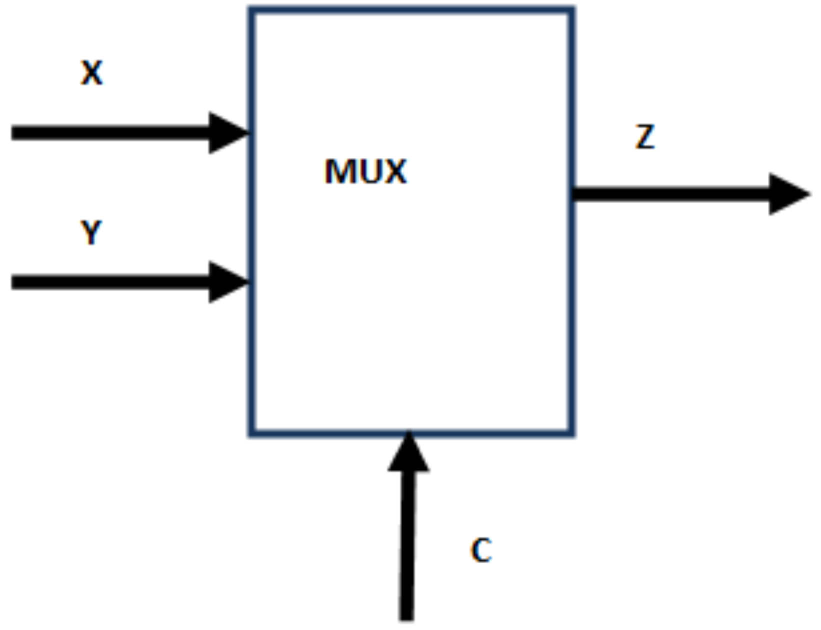
Conditional Operator in Verilog

- The conditional operator has the following C-like format:
- cond_expr ? true_expr : false_expr
- The true_expr or the false_expr is evaluated and used as a result depending on what cond_expr evaluates to (true or false).

Conditional Operator in Verilog

- The conditional operator has the following C-like format:
- cond_expr ? true_expr : false_expr

- module Mux2_1(z, c, x, y);
    - input c, x, y;
    - output z;
    - assign z = c ? x : y;
- endmodule



Procedural Blocks

- What if the operation of a unit is too complex to be described? by assignment of boolean or conditional expressions?

```
module MultiplexerD (input a, b, s, output w);
   reg w;
   always @ (a, b, s) begin
         if (s) w = b;
         else w = a;
   end
endmodule
```

Procedural Blocks

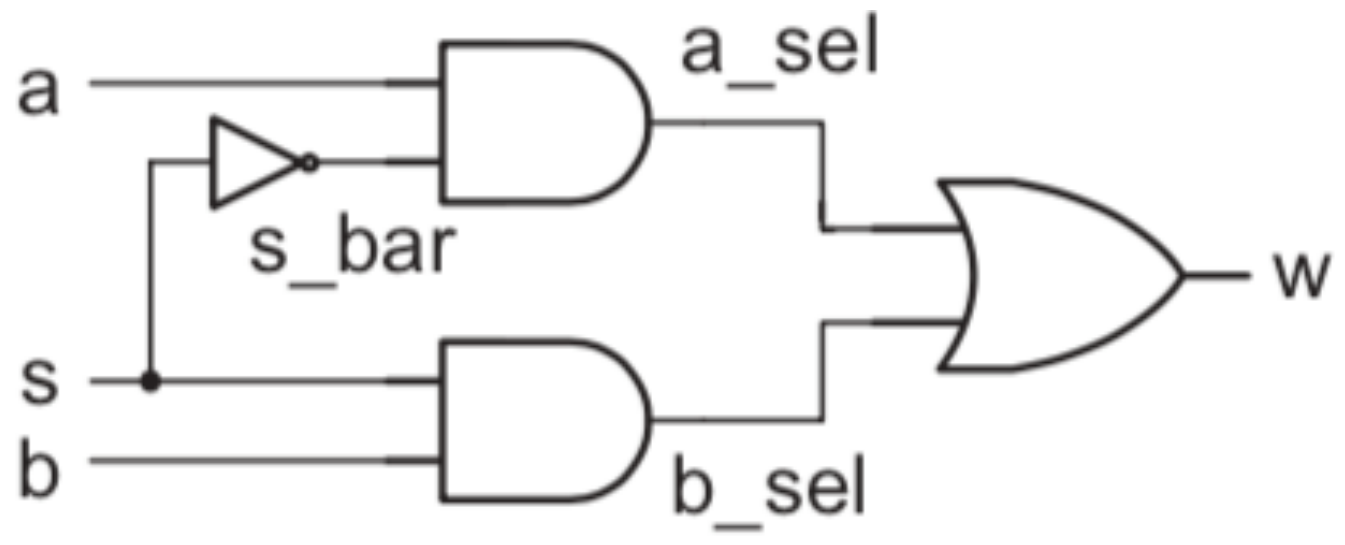
- What if the operation of a unit is too complex to be described? by assignment of boolean or conditional expressions?
    - higher-level procedural constructs should be used.

```
module MultiplexerD (input a, b, s, output w);
   reg w;
   always @ (a, b, s) begin
        if (s) w = b;
        else w = a;
   end
endmodule
```

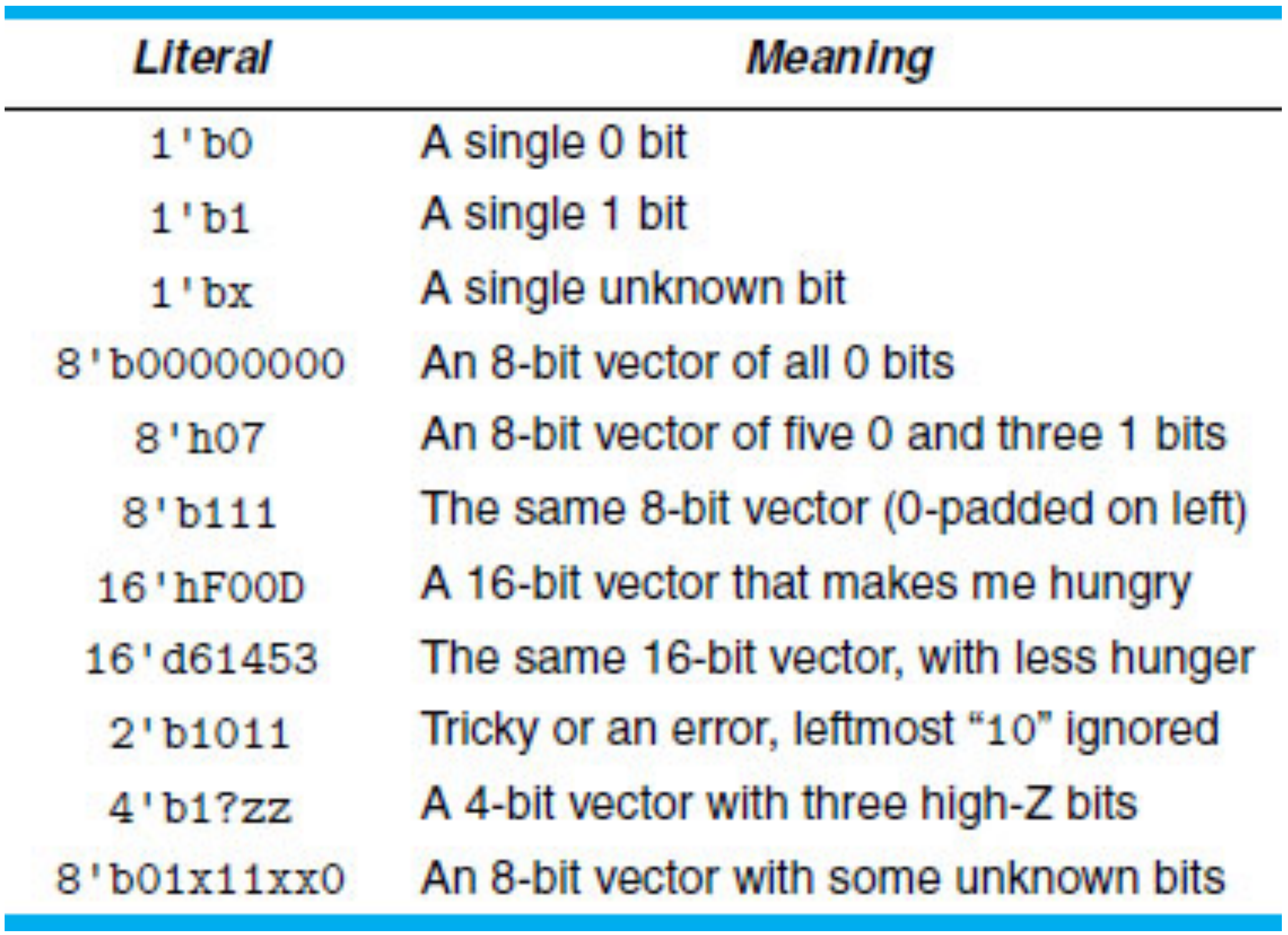
Procedural Blocks

- What if the operation of a unit is too complex to be described? by assignment of boolean or conditional expressions?
    - higher-level procedural constructs should be used.

```
module MultiplexerD (input a, b, s, output w);
   reg w;
   always @ (a, b, s) begin
        if (s) w = b;
        else w = a;
   end
endmodule
```



Data Representation in Verilog



Literal	*Meaning*
`1'b0`	A single 0 bit
`1'b1`	A single 1 bit
`1'bx`	A single unknown bit
`8'b00000000`	An 8-bit vector of all 0 bits
`8'h07`	An 8-bit vector of five 0 and three 1 bits
`8'b111`	The same 8-bit vector (0-padded on left)
`16'hF00D`	A 16-bit vector that makes me hungry
`16'd61453`	The same 16-bit vector, with less hunger
`2'b1011`	Tricky or an error, leftmost "10" ignored
`4'b1?zz`	A 4-bit vector with three high-Z bits
`8'b01x11xx0`	An 8-bit vector with some unknown bits

Testbench and Simulation

- **A NAND gate**

```
// Data-flow modeling of a nand gate
        module nand_gate(c,a,b);
              input a,b;
              output c;
              assign c =~(a&b);
    endmodule
```

Xilinx Vivado or Online Simulator

- Use the Xilinx tools to simulate your Verilog
- OR you can use online tools:
- Icarus Verilog http://iverilog.icarus.com
-

Testbench and Simulation: NAND Gate

```
// high level module to test nand, test_nand1.v
   module nand_test;
          reg a,b;
          wire c;
           nand_gate  nand_test(c,a,b);
      initial begin          // apply the stimulus, test data
          a=0; b=0;      // initial value
          #1 a=1;          // delay one simulation cycle, then change a=1.
          #1 b=1;
          #1 a=0;
          #1;
      end
  initial begin
           $monitor($time," a=%b, b=%b, c=%b",a,b,c);
       end
endmodule
```

More Example: 2-1 MUX

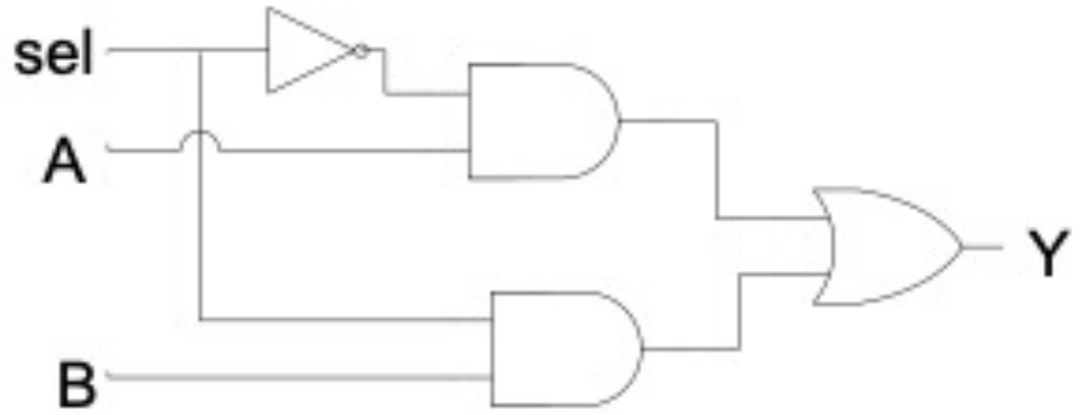
- 2-1 Multiplexer

2-to-1 Multiplexer Behavior

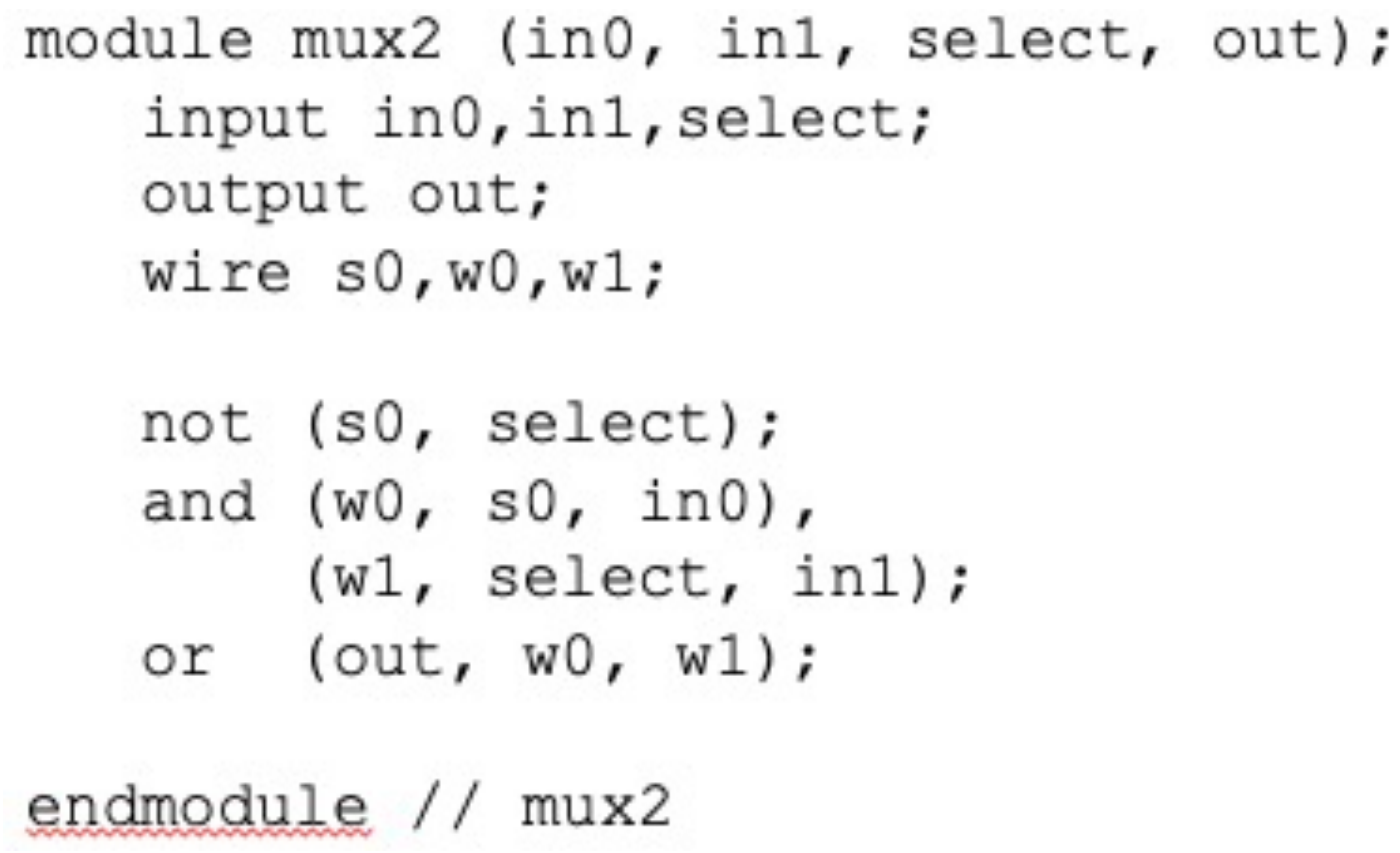
when sel = 0 choose A to send to output Y
sel =1 choose B

$Y = \overline{Sel} \bullet A + Sel \bullet B$

sel	Y
0	A
1	B



Design Code of 2-1 MUX

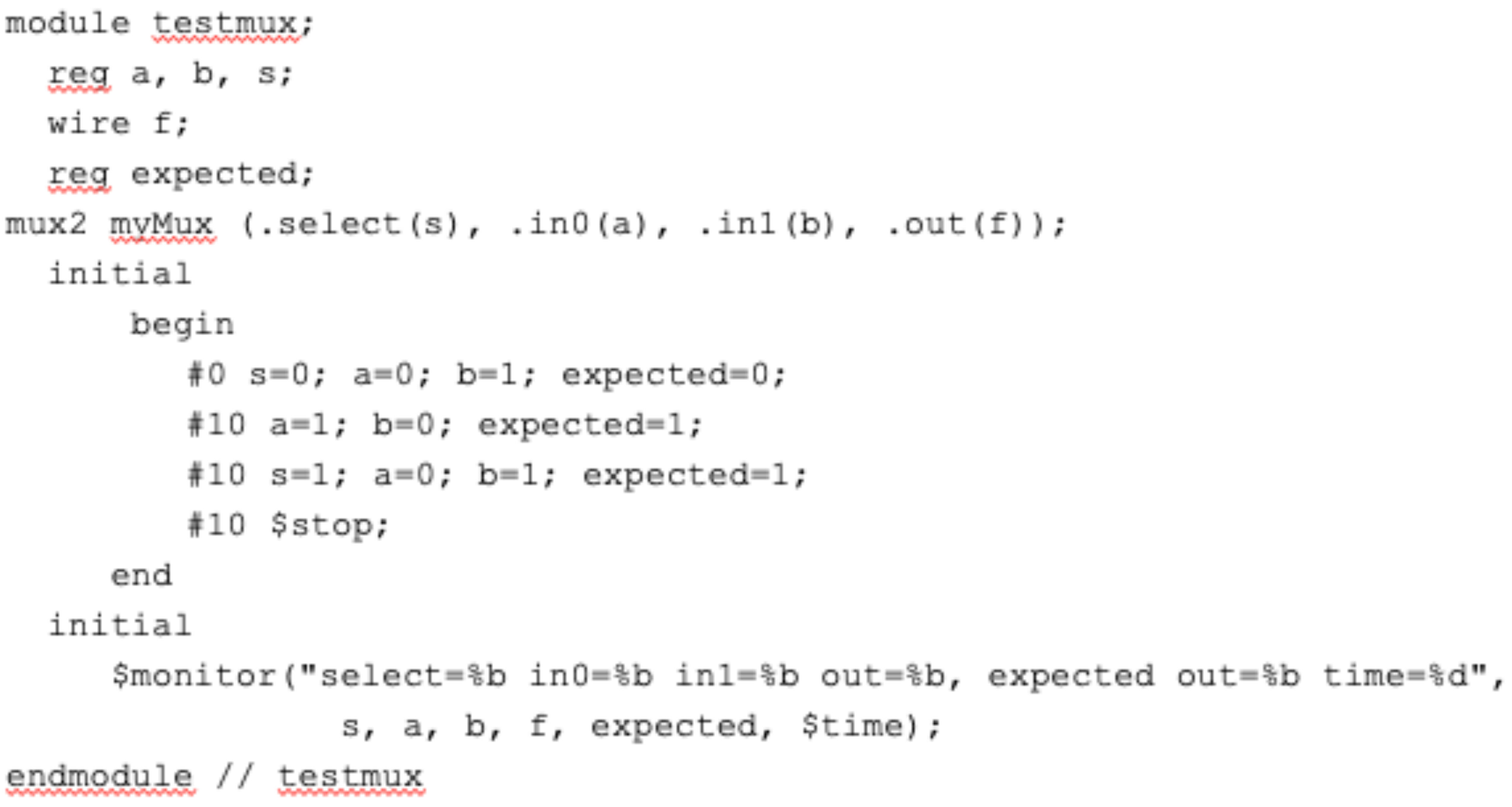
- Structural Model: 2-1 mux
- 

```
module mux2 (in0, in1, select, out);
    input in0,in1,select;
    output out;
    wire s0,w0,w1;

    not (s0, select);
    and (w0, s0, in0),
        (w1, select, in1);
    or  (out, w0, w1);

endmodule // mux2
```

Testbench of 2-1 MUX



```
module testmux;
   reg a, b, s;
   wire f;
   reg expected;
mux2 myMux (.select(s), .in0(a), .in1(b), .out(f));
   initial
        begin
           #0 s=0; a=0; b=1; expected=0;
           #10 a=1; b=0; expected=1;
           #10 s=1; a=0; b=1; expected=1;
           #10 $stop;
      end
   initial
      $monitor("select=%b in0=%b in1=%b out=%b, expected out=%b time=%d",
                   s, a, b, f, expected, $time);
endmodule // testmux
```

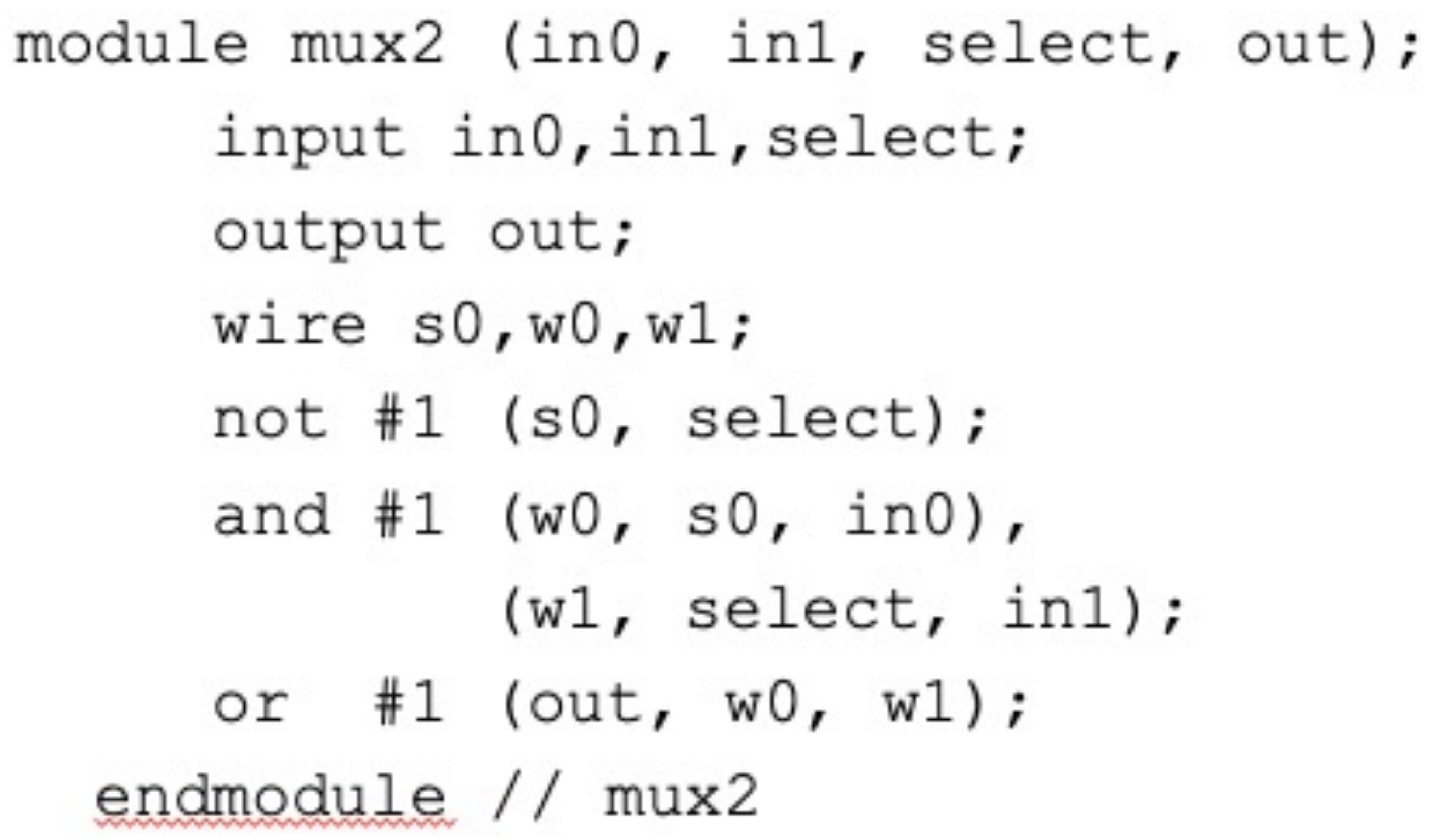
Testing Results

- Result of running testbench:
  - select=0 in0=0 in1=1 out=0, expected out=0 time=0
  - select=0 in0=1 in1=0 out=1, expected out=1 time=10
  - select=1 in0=0 in1=1 out=1, expected out=1 time=20

Testing Results

- Testbench:
  - Provides inputs, checks on outputs
  - Design is instantiated inside testbench
  - **Verilog code in testbench does not describe circuits**
    - **Behaves like an ordinary programming language**
- ***Does this testbench cover all possible input combinations?***

Modeling Circuit Delay



```
module mux2 (in0, in1, select, out);
     input in0,in1,select;
     output out;
     wire s0,w0,w1;
     not #1 (s0, select);
     and #1 (w0, s0, in0),
            (w1, select, in1);
     or  #1 (out, w0, w1);
  endmodule // mux2
```

- ***#1 represents delay from inputs to output***