EECE 2322: Fundamentals of Digital Design and Computer Organization

Lecture 3_1: Verilog and FPGA

Xiaolin Xu
Department of ECE
Northeastern University

- TA office hours
    - Tuesday and Wednesday 5-7PM
    - Zoom, please find the links on Canvas

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

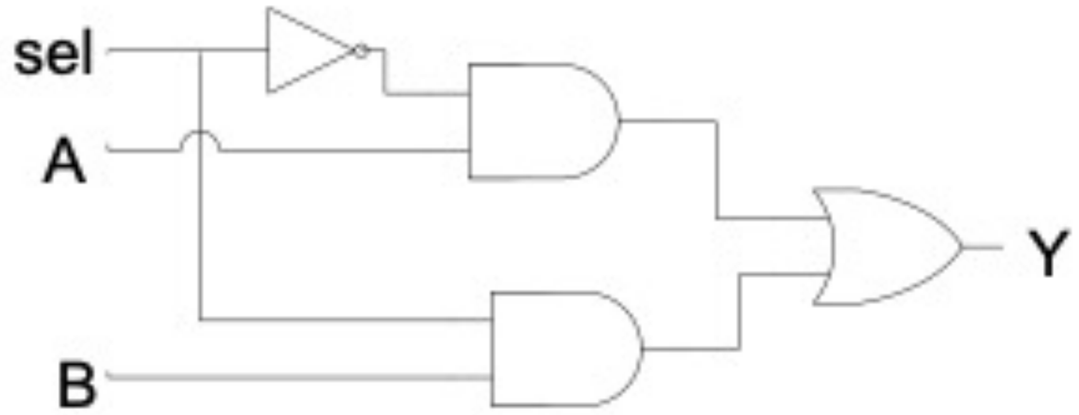
- 2-1 Multiplexer

2-to-1 Multiplexer Behavior

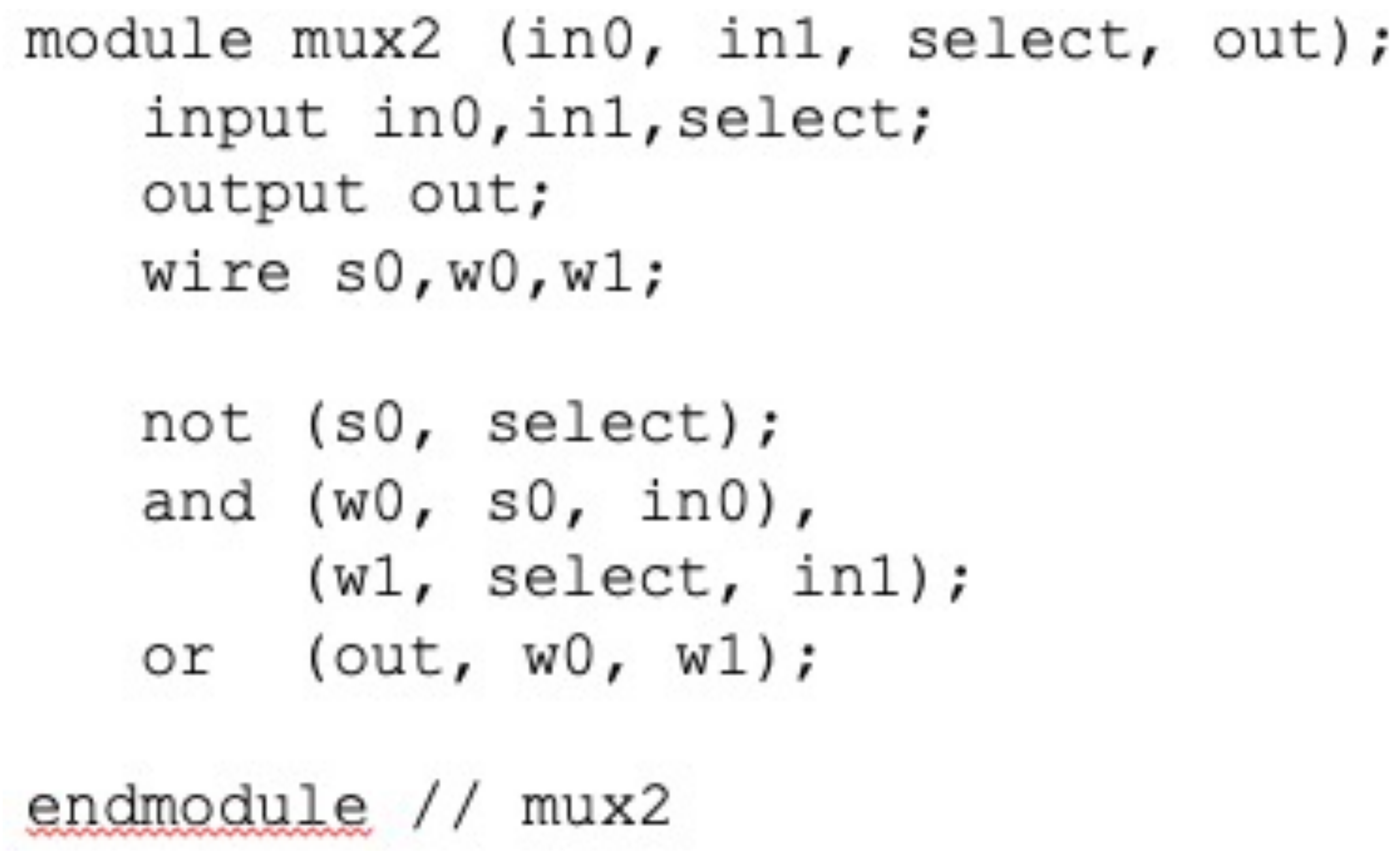
when sel = 0 choose A to send to output Y
sel =1 choose B

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

sel	Y
0	A
1	B



Design Code of 2-1 MUX

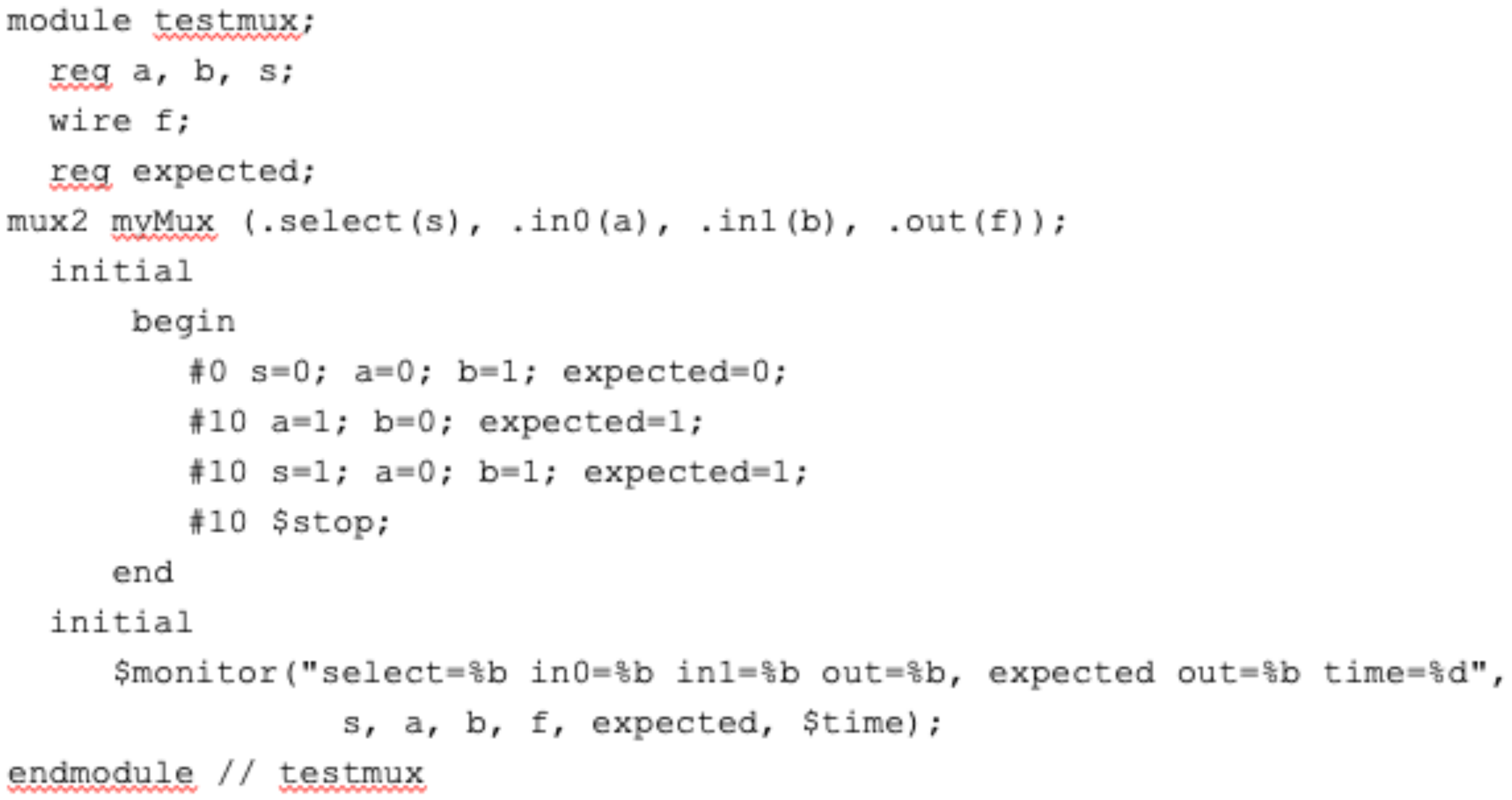
- Structural Model: 2-1 mux
- 

```
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
```

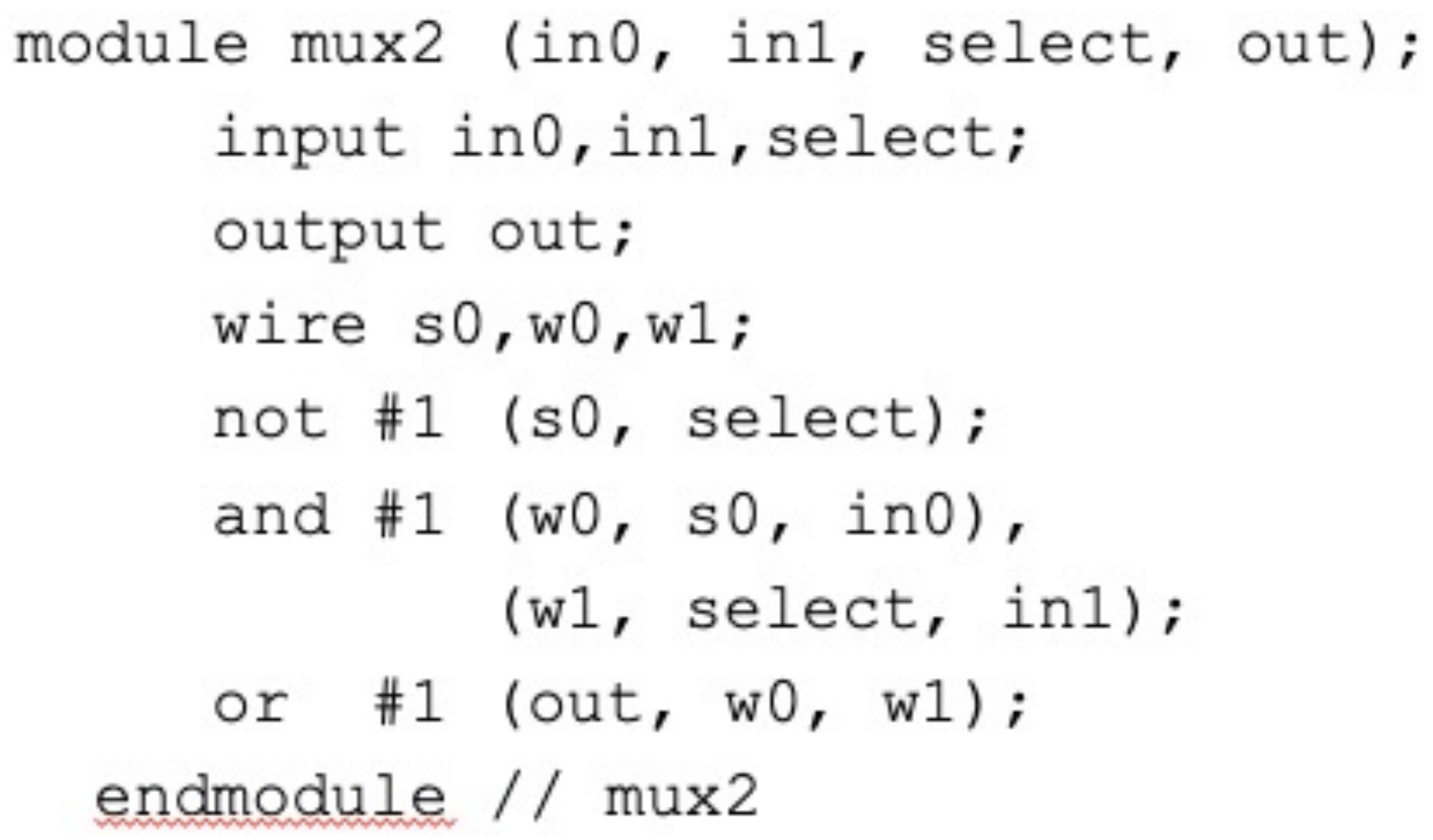
Testing Results

- 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
```

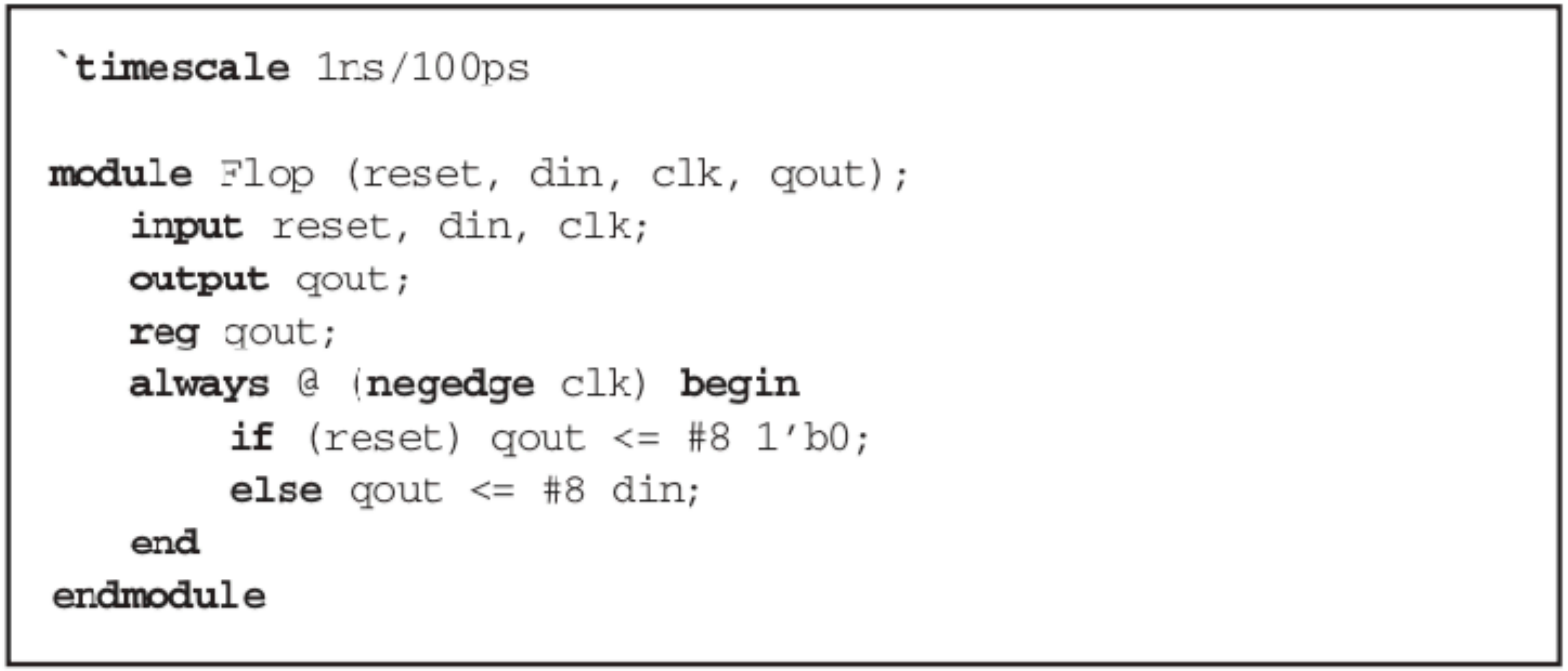
- ***#1 represents delay from inputs to output***

reg and wire

- wire
  - Connects components together
- reg
  - saves a value
  - Does NOT necessarily become a register when you synthesize
  - So far, used in simulation
- For assign statements use output or wire

reg

- Value assignments to qout take place only on the negative edge of the clock. Therefore, in order for this output to hold its value between clock edges, it has been declared as a reg.

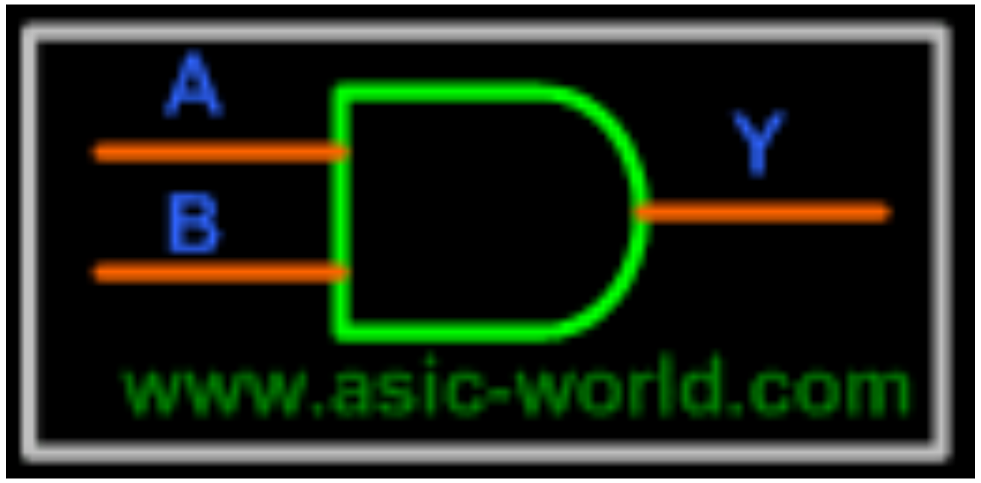


```
`timescale 1ns/100ps

module Flop (reset, din, clk, qout);
    input reset, din, clk;
    output qout;
    reg qout;
    always @ (negedge clk) begin
        if (reset) qout <= #8 1'b0;
        else qout <= #8 din;
    end
endmodule
```

Difference between Reg and Wire

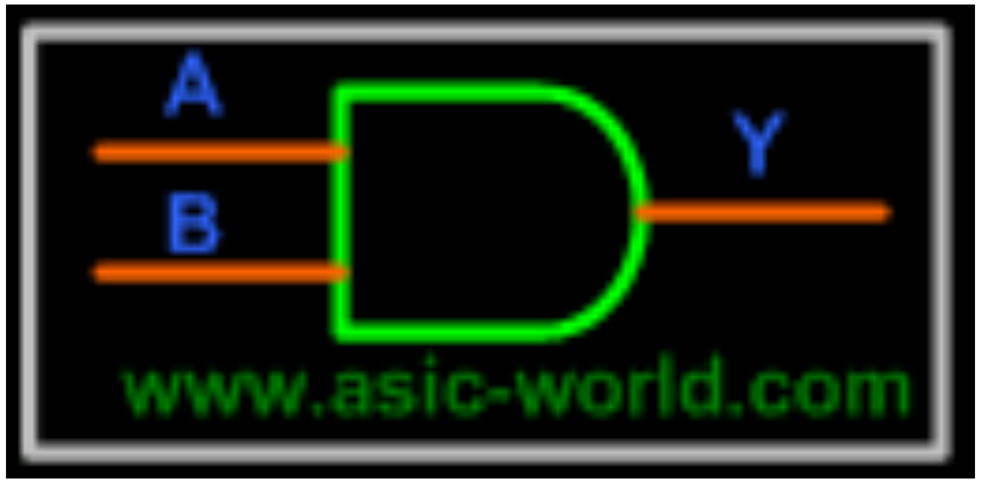
- Examples
- AND gate



- module wire_example(a, b, y);
-   input a, b;
-   output y;
- wire a, b, y;
-   assign y = a & b;
- endmodule

Difference between Reg and Wire

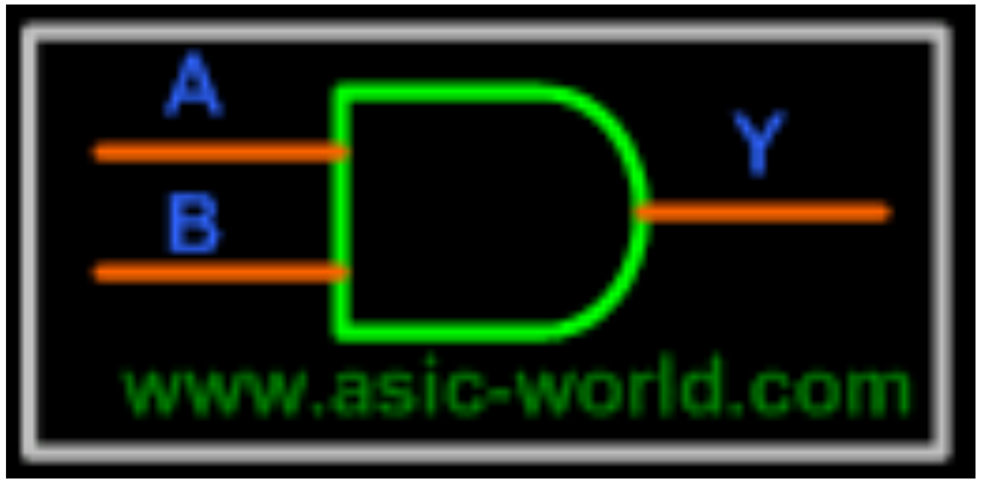
- module wire_example(a, b, y);
- input a, b;
- output y;
- wire a, b, y;
- assign y = a & b;
- endmodule



- Wire is used for designing combinational logic, this kind of logic can not store a value. A wire can be **assigned** a value by an assign statement.
- Default data type is wire: this means that if you declare a variable without specifying reg or wire, it will be a 1-bit wide wire.

Difference between Reg and Wire

- module reg_combo_example(a, b, y);
- input a, b;
- output y;
- reg   y;
- wire a, b;
- always @ (a or b)
- begin
-   y = a & b;
- end
- endmodule



This gives the same output as that of the assign statement, with the only difference that y is declared as reg.

always

always

- Always Blocks

always

- Always Blocks
  - Can contain multiple statements

always

- Always Blocks
  - Can contain multiple statements
- Can contain if, for, case

always

- Always Blocks
  - Can contain multiple statements
- Can contain if, for, case
  - These statements MUST be in an always block

always

- Always Blocks
    - Can contain multiple statements
- Can contain if, for, case
    - These statements MUST be in an always block
- Statements execute sequentially

always

- Always Blocks
  - Can contain multiple statements
- Can contain if, for, case
  - These statements MUST be in an always block
- Statements execute sequentially
  - Continuous assignments execute in parallel

always

- Always Blocks
  - Can contain multiple statements
- Can contain if, for, case
  - These statements MUST be in an always block
- Statements execute sequentially
  - Continuous assignments execute in parallel
- begin/end groups statements

always

- Always Blocks
  - Can contain multiple statements
- Can contain if, for, case
  - These statements MUST be in an always block
- Statements execute sequentially
  - Continuous assignments execute in parallel
- begin/end groups statements
- always @(sensitivity list)

always

- Always Blocks
  - Can contain multiple statements
- Can contain if, for, case
  - These statements MUST be in an always block
- Statements execute sequentially
  - Continuous assignments execute in parallel
- begin/end groups statements
- always @(sensitivity list)
- <begin> <procedural statements> <end>

always Example

```
module and_gate(out, in1, in2);
   input in1, in2;
   output out;
   reg out;

   always @(in1 or in2) begin
       out = in1 & in2;
     end
endmodule
```

Not really a register. Holds assignment in always block.

Specifies when block is executed. Triggered by changes in in1 or in2.

- The compiler will not use a register when it synthesizes this gate, because *out* changes whenever the inputs change.

always Example

```
module and_gate(out, in1, in2);
   input in1, in2;
   output out;
   reg out;

   always @(in1 or in2) begin
      out = in1 & in2;
    end
endmodule
```

Not really a register. Holds assignment in always block.

Specifies when block is executed. Triggered by changes in in1 or in2.

- The compiler will not use a register when it synthesizes this gate, because *out* changes whenever the inputs change.

always Example

```
always  @ (posedge clk )
if (reset == 0) begin
   y <= 0;
end else if (sel == 0) begin
   y <= a;
end else begin
   y <= b;
end
```

always Example

- Always Blocks

```
always  @ (posedge clk )
if (reset == 0) begin
   y <= 0;
end else if (sel == 0) begin
   y <= a;
end else begin
   y <= b;
end
```

Be Careful in Using always

- Any issue with this design?

```
module and_gate (out, in1, in2);
  input in1, in2;
  output out;
  reg out;
  always @(in1) begin
     out = in1 & in2;
  end
endmodule
```

Be Careful in Using always

- Any issue with this design?
  - Where is in2?

```
module and_gate (out, in1, in2);
   input in1, in2;
   output out;
   reg out;
   always @(in1) begin
      out = in1 & in2;
   end
endmodule
```

Be Careful in Using always

- Any issue with this design?
  - Where is in2?

```
module and_gate (out, in1, in2);
  input in1, in2;
  output out;
  reg out;
  always @(in1) begin
     out = in1 & in2;
  end
endmodule
```

Always include all inputs in sensitivity list for combinational logic

@

- Specified by the statement following the @ sign, the body of this always statement is executed at the **negative** edge of the clk signal.

```
`timescale 1ns/100ps

module Flop (reset, din, clk, qout);
    input reset, din, clk;
    output qout;
    reg qout;
    always @ (negedge clk) begin
        if (reset) qout <= #8 1'b0;
        else qout <= #8 din;
    end
endmodule
```

`timescale <time unit>/<time precision>

- time unit : the time to be used as one unit for all the delays used in the design
- time precision : This represents the minimum delay which needs to be considered during simulation or it decides that how many decimal point would be used with the time unit.
- Range of Timescale :
    - Range for time unit can be from seconds to femto-seconds, which includes all the time units including s (second), ms(mili-second), us(micro-second), ns(nano-second), ps(pico-second) and fs(femto-second).

`timescale <time unit>/<time precision>

- Example :
    - `timescale 1ns/1ps
    - 1ps = 0.001 ns
    - #1; // = 1ns delay
    - #1.003; // = will be considered as a valid delay
    - #1.0009; // = ???

`timescale <time unit>/<time precision>

- Example :
  - `timescale 1ns/1ps
  - 1ps = 0.001 ns
  - #1; // = 1ns delay
  - #1.003; // = will be considered as a valid delay
  - #1.0009; // = will be taken as 1 ns only since it is out of the precision value.

Differences Between Two Designs

```
always  @ (a or b or sel)
begin
  y = 0;
  if (sel == 0) begin
    y = a;
  end else begin
    y = b;
  end
end
```

```
always  @ (posedge clk )
if (reset == 0) begin
  y <= 0;
end else if (sel == 0) begin
  y <= a;
end else begin
  y <= b;
end
```

Difference1: Level-Sensitive and Edge-Sensitive

- Left: level-sensitive
- Right: edge-sensitive

```
always  @ (a or b or sel)
begin
   y = 0;
   if (sel == 0) begin
      y = a;
   end else begin
      y = b;
   end
end
```

```
always  @ (posedge clk )
if (reset == 0) begin
   y <= 0;
end else if (sel == 0) begin
   y <= a;
end else begin
   y <= b;
end
```

Difference1: Level-Sensitive and Edge-Sensitive

- Left: will block until there is a change in the value of a or b
- Right: will block until clk transitions from 0 to 1.

```
always  @ (a or b or sel)
begin
   y = 0;
   if (sel == 0) begin
      y = a;
   end else begin
      y = b;
   end
end
```

```
always  @ (posedge clk )
if (reset == 0) begin
   y <= 0;
end else if (sel == 0) begin
   y <= a;
end else begin
   y <= b;
end
```

Difference2: Blocking and Nonblocking Statements

- Left: Blocking statement =
- Right: Nonblocking statement <=

```
always  @ (a or b or sel)
begin
   y = 0;
   if (sel == 0) begin
      y = a;
   end else begin
      y = b;
   end
end
```

```
always  @ (posedge clk )
if (reset == 0) begin
   y <= 0;
end else if (sel == 0) begin
   y <= a;
end else begin
   y <= b;
end
```

Difference2: Blocking and Nonblocking Statements

- Left: Blocking statement =
  - A blocking statement must be executed before the execution of the statements that follow it in a sequential block.
- Right: Nonblocking statement <=
  - Use the nonblocking procedural statement whenever you want to make several register assignments within the same time step
  - **It means that nonblocking statements resemble actual hardware more than blocking assignments**

Blocking Assignment Example

- Used in testbenches

```
initial
    begin
        B = A;
        C = B;
    end
```

- Result is that new contents of B are in C, so all have contents of A

Non-blocking Assignment

- initial
- begin
- B <= A;
- C <= B;
- end
- All results evaluated first, then assigned
  - Result is that old contents of B are in C
  - Won't need for testing, but for synthesis

Reconfigurable Hardware

- By far the most dominant implementation platform in terms of the number of designs per year.
- FPGA : filed programmable gate array
  - Revenues grew by more than one third to surpass the $4 B level,
  - The number of new designs based on FPGA was 110,000. The colossal size of this number can be best seen from the fact that only 2,500 ASIC designs were initiated
  - At the same time, FPGA has been recognized as an exceptionally efficient platform due to its flexibility compared to ASICs , and due to its efficiency compared to implementations based on the general purpose microprocessors

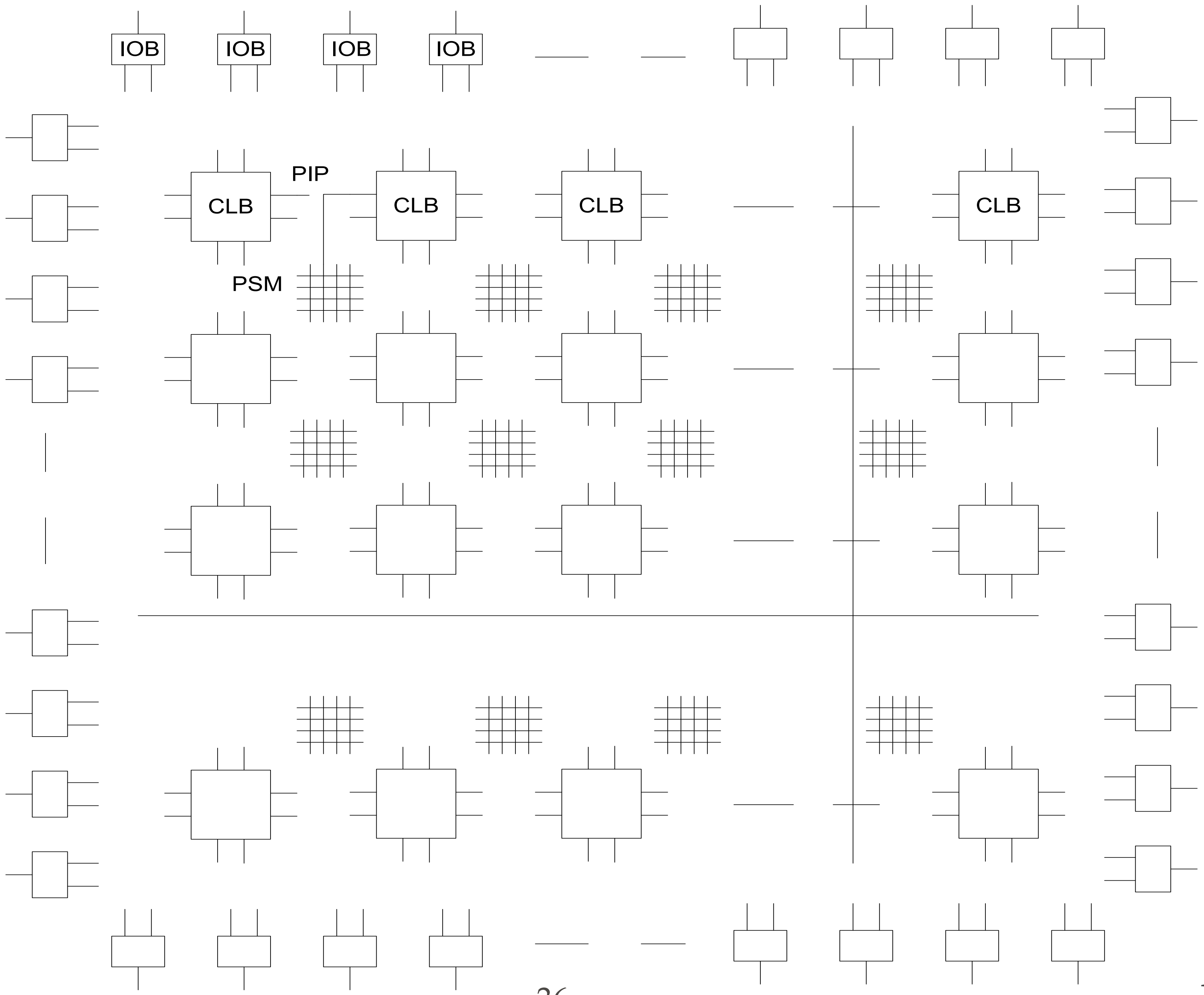
Inside an FPGA

- An FPGA has three main elements:
  - Look-Up Tables (LUT)
  - Flip-flops
  - Routing matrix
- All work together to create a very flexible device

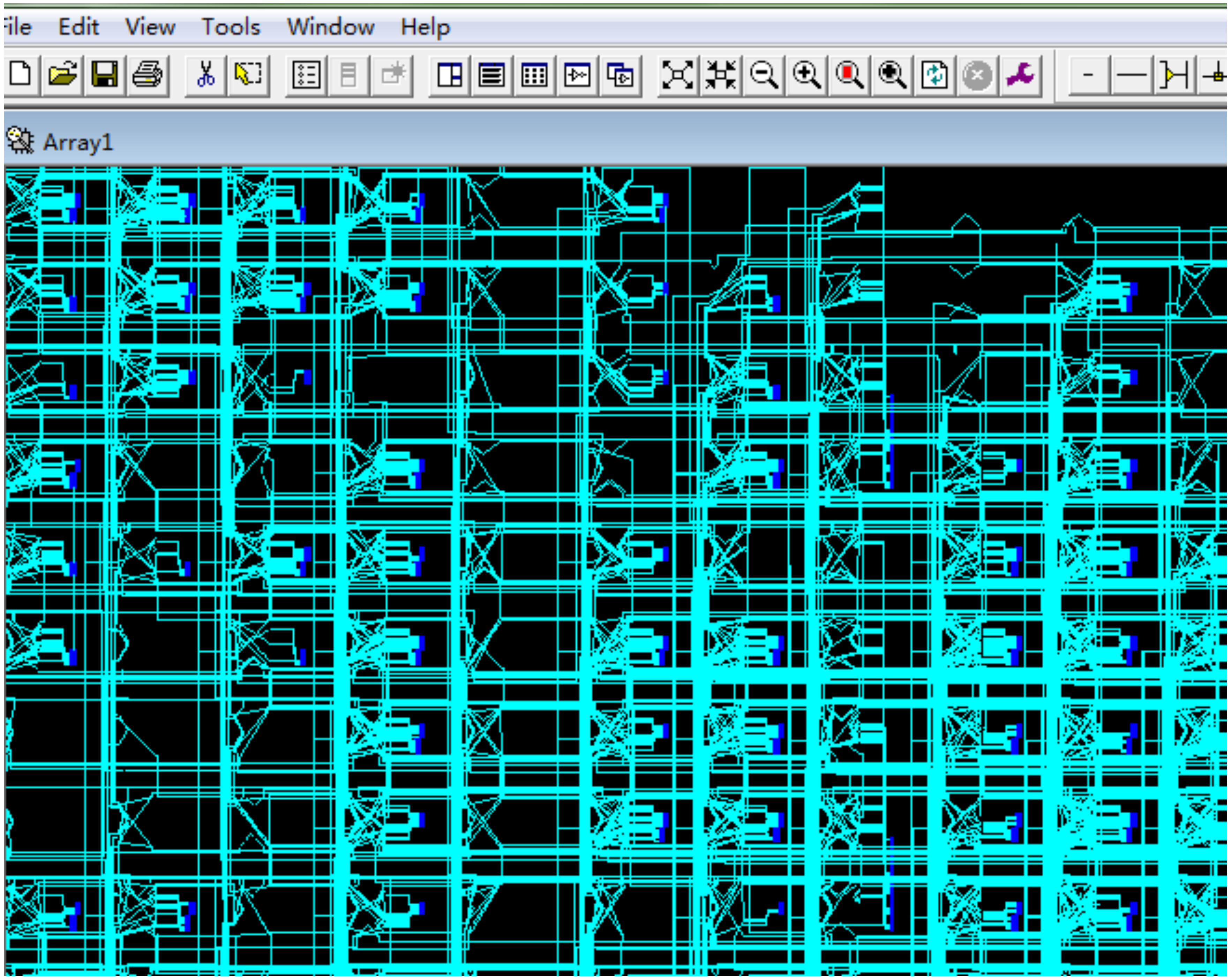
Inside an FPGA

- An FPGA has the following major elements:
  - Look-Up Tables (LUT)
  - Flip-flops
  - CLB: Configurable Logic Block
  - IOB: Input/Output Block
  - PSM: Programmable Switch Matrix
  - PIP: Programmable Interconnect Point
- All work together to create a very flexible device

Hardware Floor-plan of an FPGA



After Mapping Implementations on an FPGA



How to Make a Device Reconfigurable to Any Logic?

In[1:0]	Out
00	?
01	?
10	?
11	?

How to Make a Device Reconfigurable to Any Logic?

In[1:0]	Out
00	?
01	?
10	?
11	?

- **Reconfigure the memory!**
- **A memory cell can be written with a 1 or a 0**

Addressable Memory

- SRAM: Static Radom Accessible Memory
- A0-A3: address
- F: output
- How many memory cells?