,

Computer Architecture (Lab). Week 5

Vladislav, Artem, Hamza, Manuel

Innopolis University

vl.ostankovich@innopolis.ru
a.burmyakov@innopolis.ru
m.rodriguez.osuna@innopolis.university
h.salem@innopolis.university

September 19, 2019

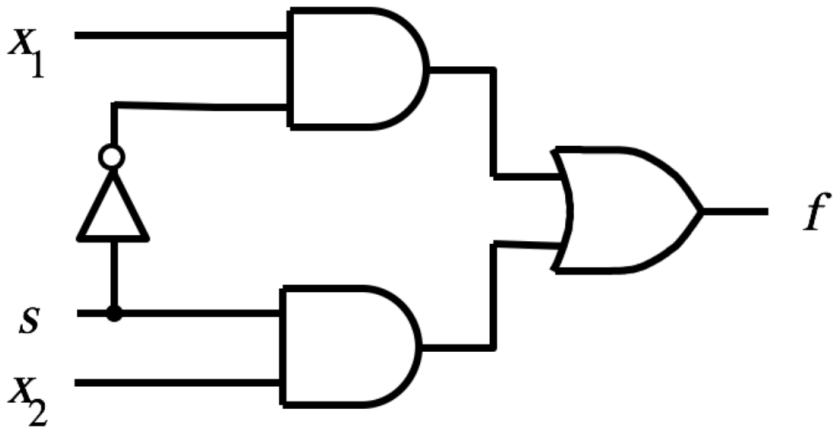
Topic of the lab

- Multiplexor. DeMultiplexor. Verilog Basics

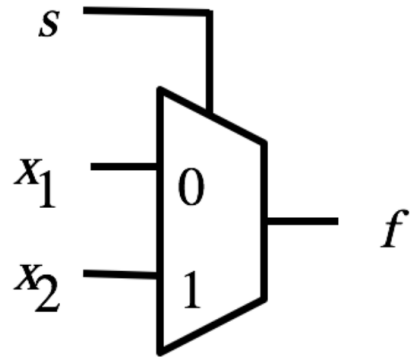
Multiplexor

- Multiplexor (MUX) is a combinational logic device designed to control the transmission of data from several sources to one output channel. According to the definition, the Multiplexor must have one output and two types of inputs: information and address ones. The code, arriving at the address input, determines which of the information inputs is currently connected to the output (being commutated). If the number of address inputs of the Multiplexor is n, then the maximum possible number of its information inputs will be 2^n; if there are less of them, the Multiplexor will be incomplete.
- It should be separately noted that the Multiplexor is a combinational device – not sequential; this means that the change in the signals at the input of the Multiplexor entails a change in the output.

2-1 Multiplexor Schematic



Circuit



Graphical symbol

$$f(s, x1, x2) = \overline{s}x1 + sx2$$

2-1 Multiplexor – Verilog Code

- Solution I

```
module mux_2_1(

    input x1,
    input x2,
    input s,
    output f

);
    assign f = (s & x2) | ((~s) & x1);

endmodule
```

2-1 Multiplexor – Verilog Code

- Solution II: using the ternary operator (? :)

```
module mux_2_1_ternary
(
    input  x1,
    input  x2,
    input  s,
    output f
);

assign f = s==1 ? x2 : x1;

endmodule
```

2-1 Multiplexor – Verilog Code

- Solution III: using conditional operator (if)

```
module b1_mux_2_1_if
(
    input  x1,
    input  x2,
    input  s,
    output reg f
);
    always@(*)
    begin
        if(s==1)
            f = x2;
        else if (s==0)
            f = x1;
    end

endmodule
```

2-1 Multiplexor – Verilog Code

- Solution IV: using conditional operator (case)

```
module b1_mux_2_1_case (
    input x1,
    input x2,
    input s,
    output reg f
);
    always@(*)
    begin
        case (s)
        0: f = x1;
        1: f = x2;
        endcase
    end
endmodule
```

Testbench (1/3)

- Testbench is a module which only task is to test another module
- Testbench is for simulation only, not for synthesis

2-1 Multiplexor – Testbench (2/3)

```
module testbench;
    // input and output test signals
    reg  a;
    reg  b;
    reg  sel;
    wire y_comb;
    wire y_sel;
    wire y_if;
    wire y_case;

   // creating the instance of the module we want to test
    mux_2_1          mux (a, b, sel, y_comb);
    mux_2_1_ternary mux_2_1_ternary  (a, b, sel, y_sel);
    b1_mux_2_1_if    b1_mux_2_1_if  (a, b, sel, y_if);
    b1_mux_2_1_case b1_mux_2_1_case  (a, b, sel, y_case);
```

2-1 Multiplexor – Testbench (3/3)

```
    initial
        begin
            a = 1'b0;
            b = 1'b1;
            #5;                 // pause (5 units of delay)
            sel = 1'b0;         // sel change to 0; a -> y
            #10;                // pause (10 units of delay)
            sel = 1'b1;         // sel change to 1; b -> y
            #10;
            b = 1'b0;           // b change; y changes too. sel == 1'b1
            #5;
            b = 1'b1;
            #5;
        end
    //  print signal values on every change
    initial
        $monitor("a=%b b=%b sel=%b y_comb=%b y_sel=%b y_if=%b y_
            case=%b",
                a, b, sel, y_comb, y_sel, y_if, y_case);
    initial
        $dumpvars;  // dump all variables
endmodule
```

Verilog $monitor

- Verilog provides some system functions specifically for generating input and output to help verification.
- Syntax

$$\$monitor(''format\ string'', parameter1, parameter2, ...);$$

- "format string", specifies how the parameters will be displayed:
  - %d will print the variable in decimal
  - %b will print the variable in binary
  - %h will print the variable in hexadecimal

Exercise 1

- Write a Verilog code in Modelsim for 2 bit Multiplexor 4in1 using “case” statement

DeMultiplexor

- The deMultiplexor performs the function of the inverse Multiplexor – it commutes the input signal to the desired output the number of which is set by the selector. The other outputs are set to 0.

1-4 De-Multiplexor – Verilog (1/3)

```
module b1_demux_1_4_case
(
    input       din,
    input       [1:0] sel,
    output reg dout0,
    output reg dout1,
    output reg dout2,
    output reg dout3
);
    always @(*)
        case (sel)
            2'b00:
            begin
                dout0 = din;
                dout1 = 0;
                dout2 = 0;
                dout3 = 0;
            end
```

1-4 De-Multiplexor – Verilog (2/3)

```
2'b01:
begin
     dout0 = 0;
     dout1 = din;
     dout2 = 0;
     dout3 = 0;
end
2'b10:
begin
     dout0 = 0;
     dout1 = 0;
     dout2 = din;
     dout3 = 0;
end
```

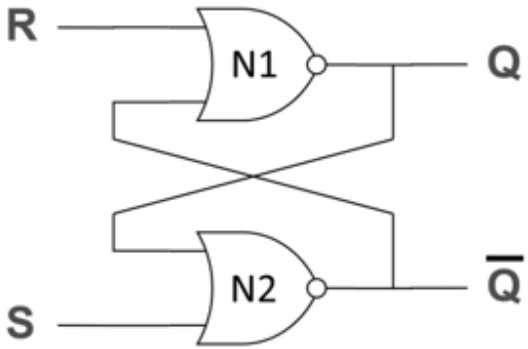
1-4 De-Multiplexor – Verilog (3/3)

```
            2'b11:
            begin
                dout0 = 0;
                dout1 = 0;
                dout2 = 0;
                dout3 = din;
            end
        endcase

endmodule
```

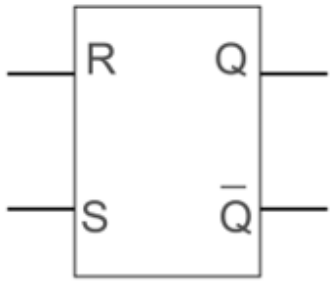
Exercise 2

- Copy and paste the code of DeMultiplexor and run

SR Latch



SR Latch Schematic



SR-Latch Symbol

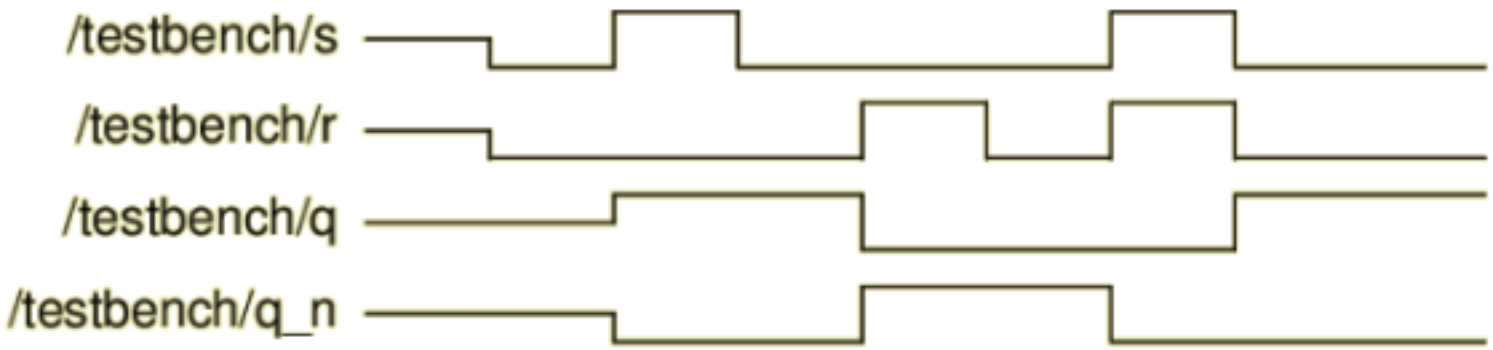
SR Latch – Verilog HDL

```
module sr_latch
 (
      input   s,
      input   r,
      output  q,
      output  q_n
 );
      assign q   = ~ ( r | q_n );
      assign q_n = ~ ( s | q   );

 endmodule
```

SR Latch – Test Bench

```
module testbench;

    reg  s, r;
    wire q, q_n;
    sr_latch sr_latch (s, r, q, q_n);
    initial $dumpvars;
    initial
    begin
       $monitor ("%0d s %b r %b q %b q_n %b", $time, s, r, q, q_n);
        # 10;    s = 0; r = 0;
        # 10;    s = 1; r = 0;
        # 10;    s = 0; r = 0;
        # 10;    s = 0; r = 1;
        # 10;    s = 0; r = 0;
        # 10;    s = 1; r = 1;
        # 10;    s = 0; r = 0;
        # 10;    s = 0; r = 0;
        # 10;
        $finish;
    end
endmodule
```

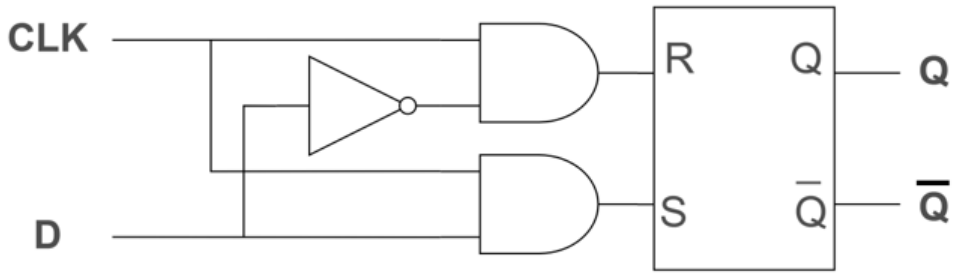
SR Latch Waveform



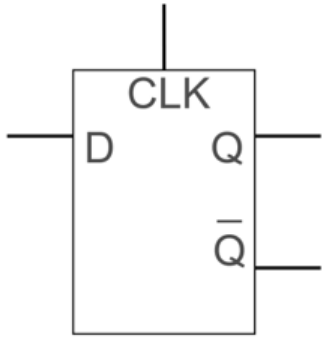
Exercise 3

- Copy and paste the code of SR Latch and run

D Latch



D-latch schematic

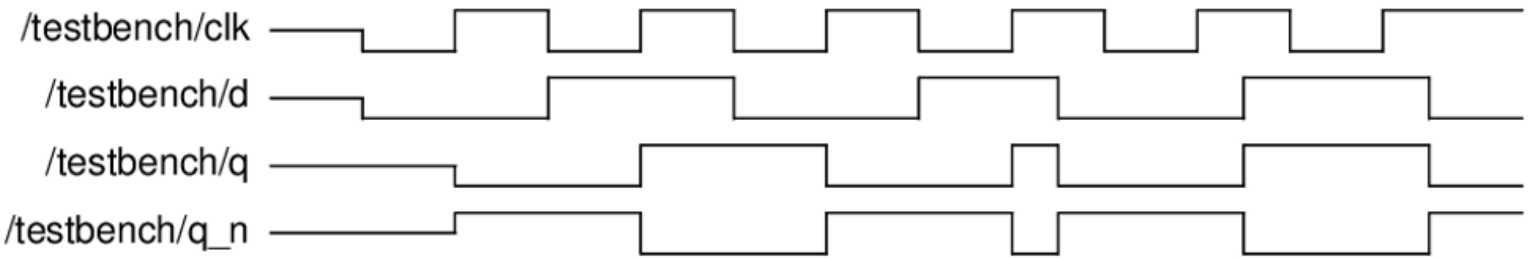


D-latch symbol

D Latch – Verilog HDL

```
module d_latch
 (
      input   clk,
      input   d,
      output  q,
      output  q_n
 );
      wire r = ~d & clk;
      wire s = d & clk;

      sr_latch sr_latch (s, r, q, q_n);
 endmodule
```

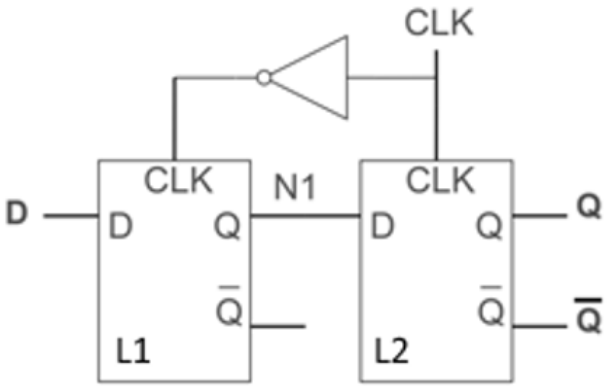
D Latch Waveform



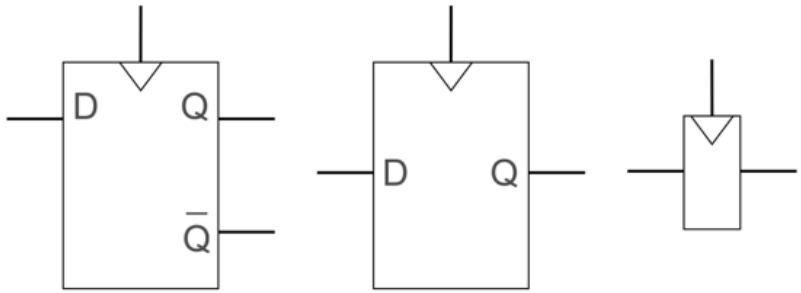
Exercise 4

- Copy and paste the code of D Latch and run

D Flip-flop Schematic



D flip-flop schematic

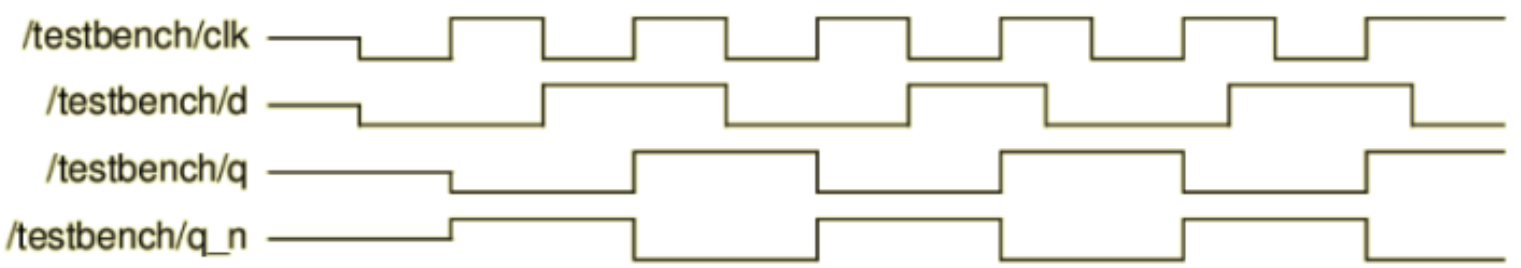


D flip-flop symbol

D Flip-flop – Verilog HDL

```
module d_flip_flop (
     input   clk,
     input   d,
     output  q,
     output  q_n
 );
     wire n1;

     d_latch master (
          .clk ( ~clk ),
          .d   (  d   ),
          .q   (  n1  )
     );

     d_latch slave (
          .clk ( clk ),
          .d   ( n1  ),
          .q   ( q   ),
          .q_n ( q_n )
     );
 endmodule
```

D Flip Flop Waveform



Exercise 5

- Copy and paste the code of D Flip Flop and run

Acknowledgements

- This lab was created and maintained by Vitaly Romanov, Aidar Gabdullin, Munir Makhmutov, Ruzilya Mirgalimova, Muhammad Fahim, Vladislav Ostankovich, Alena Yuryeva and Artem Burmyakov