3TB4 – Lab 1 Prelab

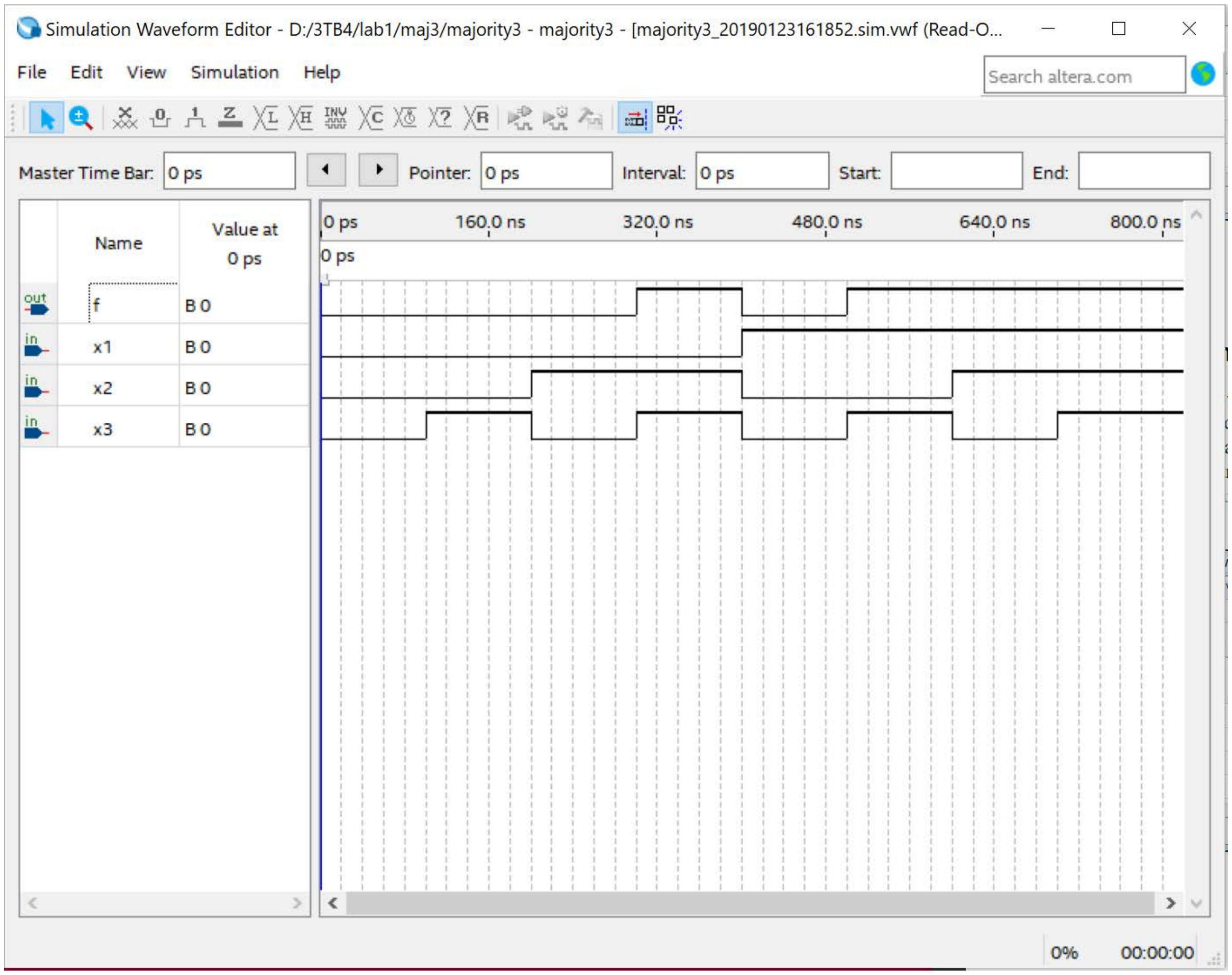
Hank Bae

400062215

Jan 24, 2019

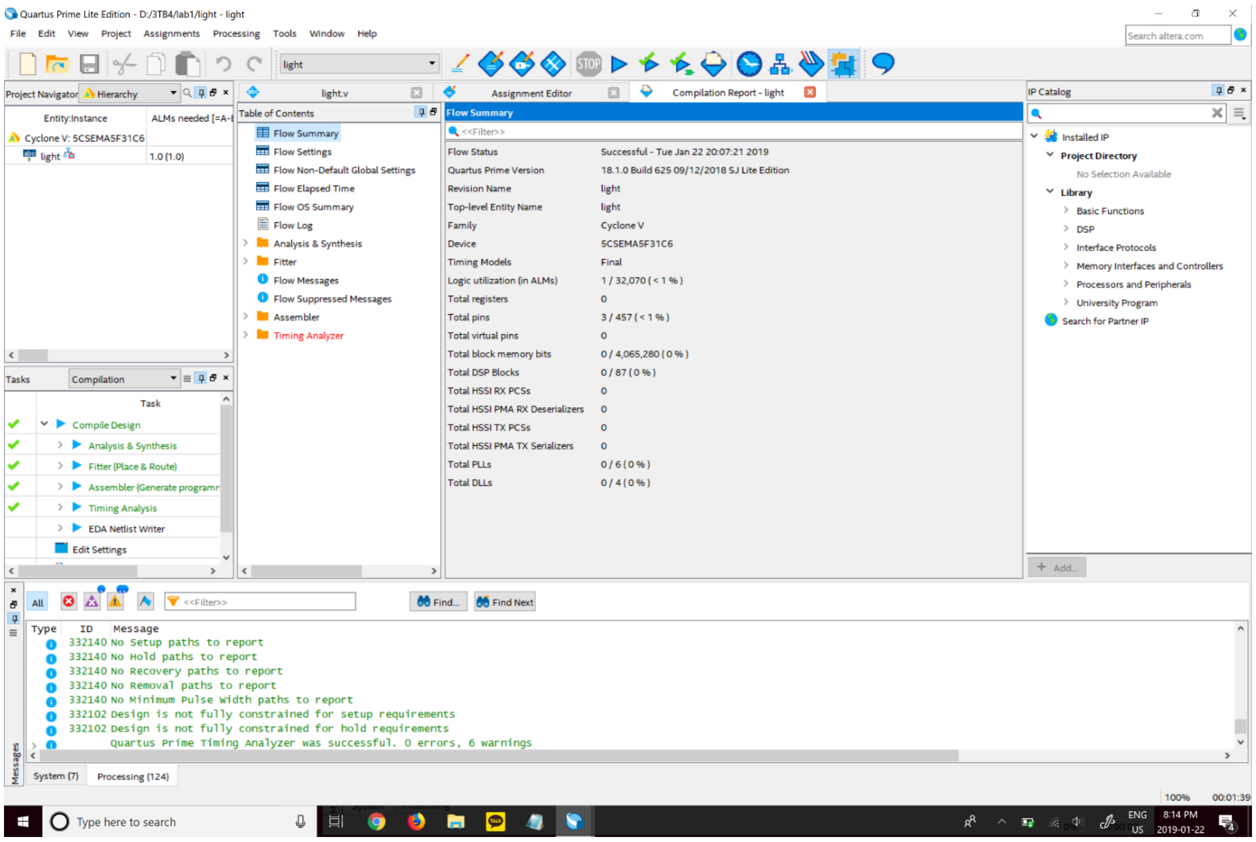
Tutorial Report

In this tutorial, we familiarised ourselves with Intel Quartus programming the De1-SoC so that it was effectively an XOR gate where it would turn an on board LED on and off depending on the state of two on board switches.



Name	Value at 0 ps
f	B 0
x1	B 0
x2	B 0
x3	B 0

Figure 1: Waveform for Majority3



Flow Summary

Flow Status	Successful - Tue Jan 22 20:07:21 2019
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	light
Top-level Entity Name	light
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	1 / 32,070 (< 1 %)
Total registers	0
Total pins	3 / 457 (< 1 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

```
Type   ID     Message
       332140 No Setup paths to report
       332140 No Hold paths to report
       332140 No Recovery paths to report
       332140 No Removal paths to report
       332140 No Minimum Pulse Width paths to report
       332102 Design is not fully constrained for setup requirements
       332102 Design is not fully constrained for hold requirements
              Quartus Prime Timing Analyzer was successful. 0 errors, 6 warnings
```

Figure 2: Screenshot of compilation report

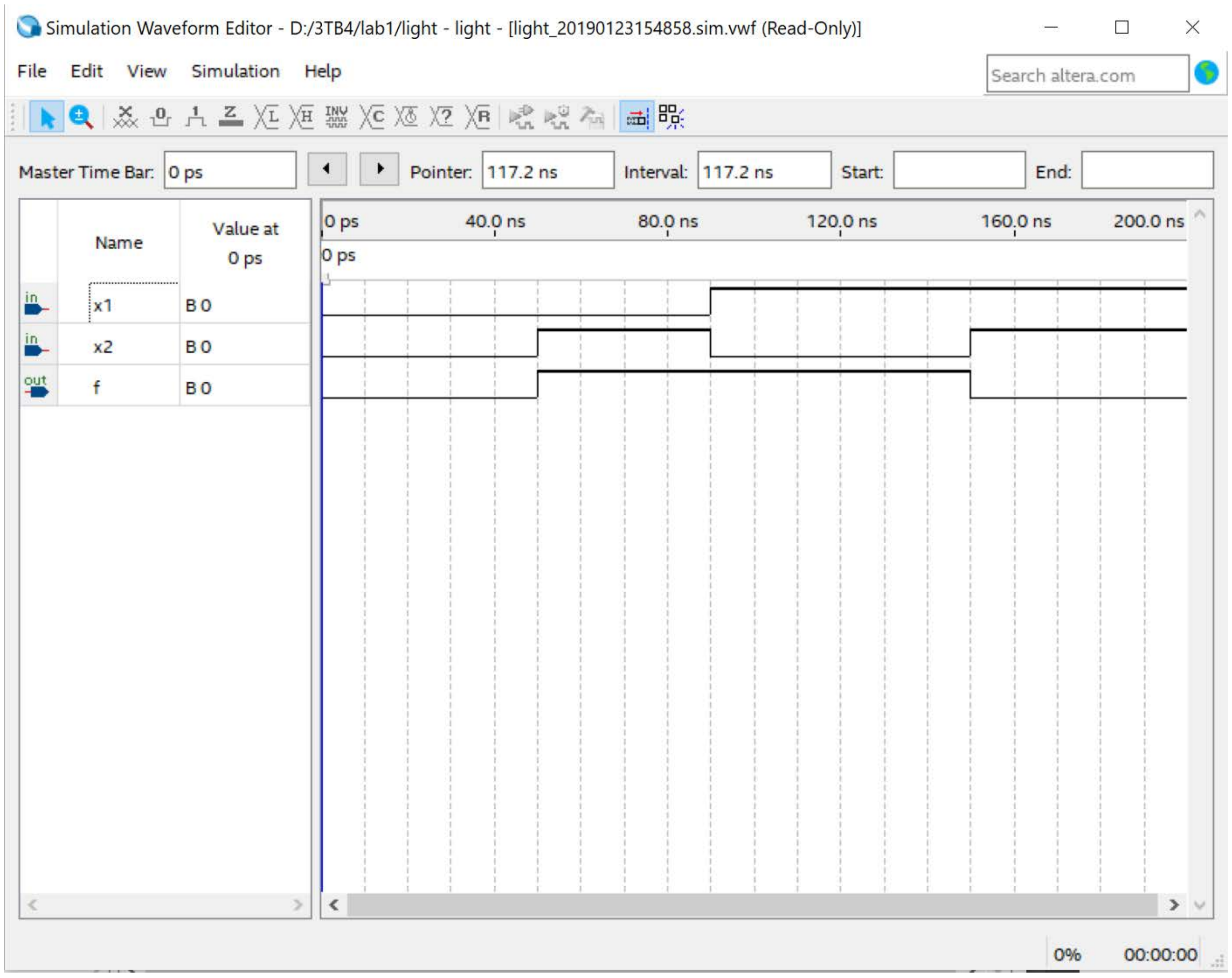
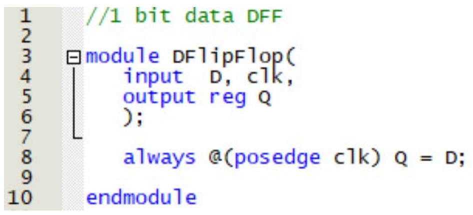


Figure 3: Waveform for Light File

Short Answer Questions

1. `reg` can store a value whereas `wire` will only create a node in a circuit.
2. Yes, the `wire` data type can be used on the left side of an expression.
3. The parameters of a module must be specified as either `input` or `output` and their data type must be specified as well unless they will be used are `wire` data types.
4. Continuous assignment are definitions that do not change whereas blocking and nonblocking assignments are executed when only ran. The difference between blocking and nonblocking is that blocking is sequential whereas nonblocking is not meaning if a statement depends on the previous statement, the result will differ depending on a blocking or nonblocking assignment is made.
5. In combinatorial logic, `assign` and `always` may be used but in sequential logic, only `always` may be used.
6. To prevent inferred latches, the code must be complete and disjoint meaning it just cover all cases.
7. `<<` is a binary logical left shift operator whereas `<<<` is a binary arithmetic left shift operator.
8. To declare an array of 6 elements of a 7-bit wire: `wire [6:0] x[5:0];`

Verilog Modules



```
//1 bit data DFF

module DFlipFlop(
    input  D, clk,
    output reg Q
    );

    always @(posedge clk) Q = D;

endmodule
```

Figure 4: One-Bit data width D flip-flop

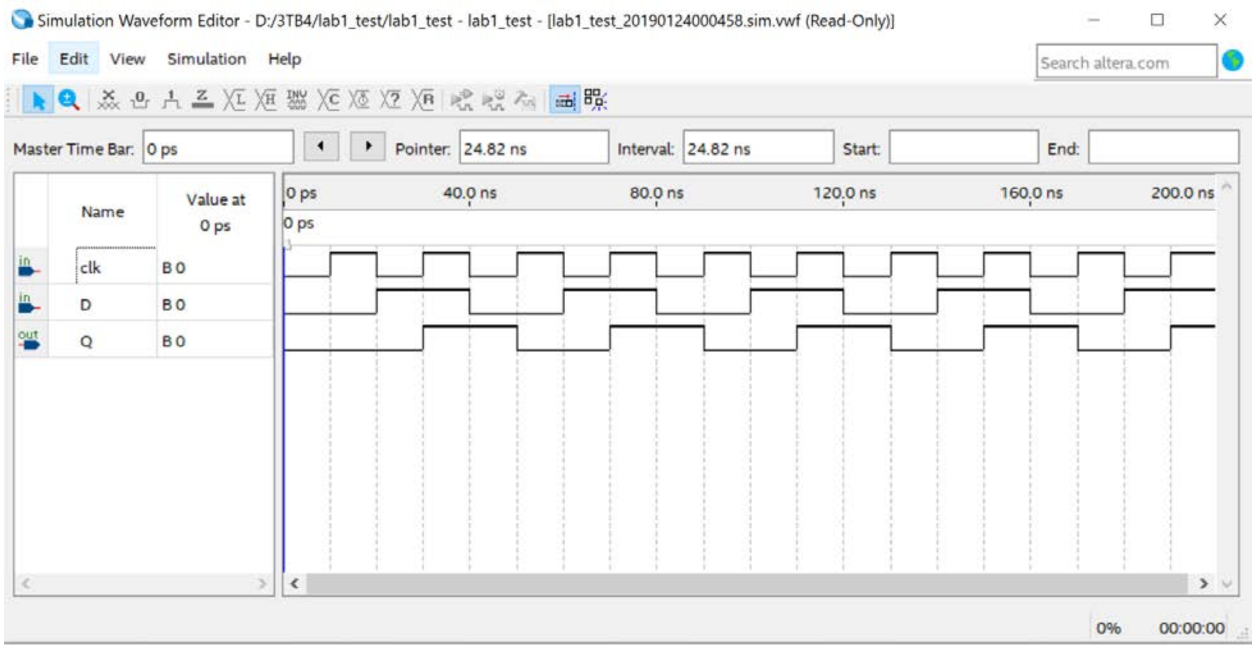
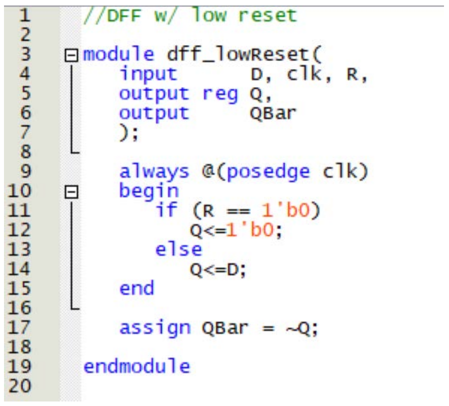
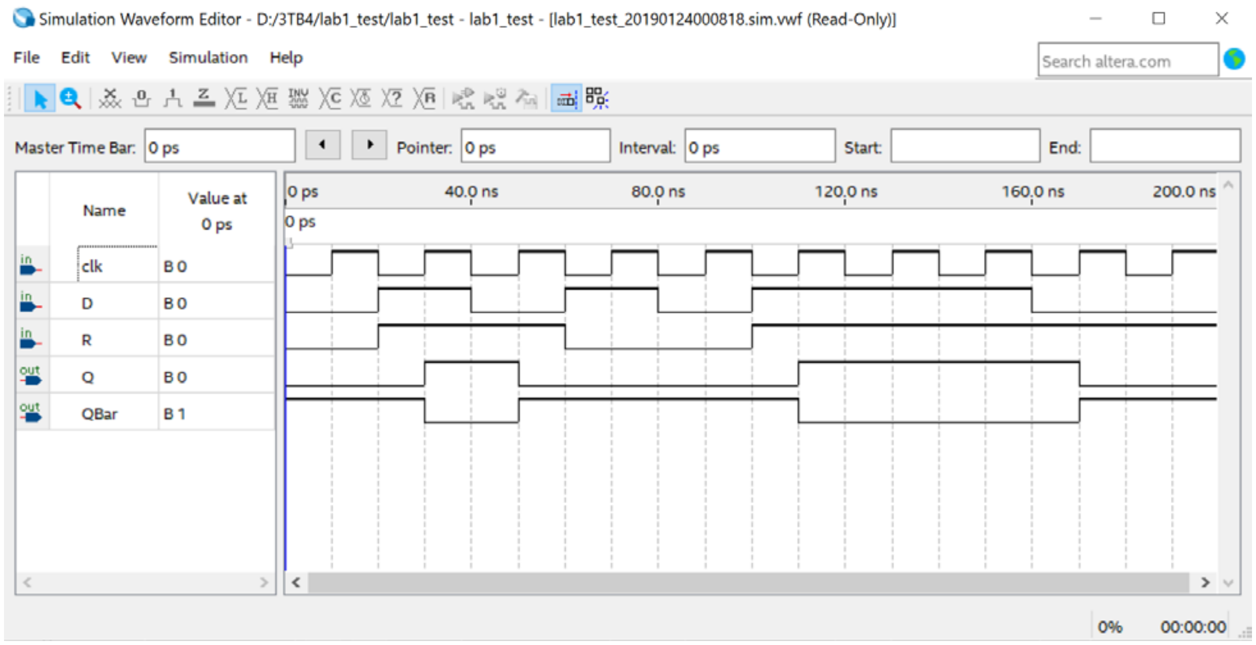


Figure 5: Waveform for One-Bit data width D flip-flop



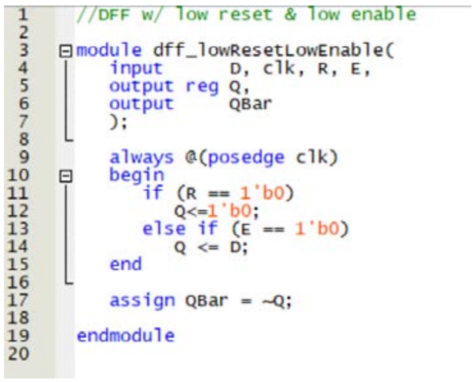
```
//DFF w/ low reset

module dff_lowReset(
    input       D, clk, R,
    output reg Q,
    output      QBar
    );

    always @(posedge clk)
    begin
       if (R == 1'b0)
          Q<=1'b0;
       else
          Q<=D;
    end

    assign QBar = ~Q;

endmodule

```

Figure 6: One-Bit data width D flip-flop with active low synchronous reset



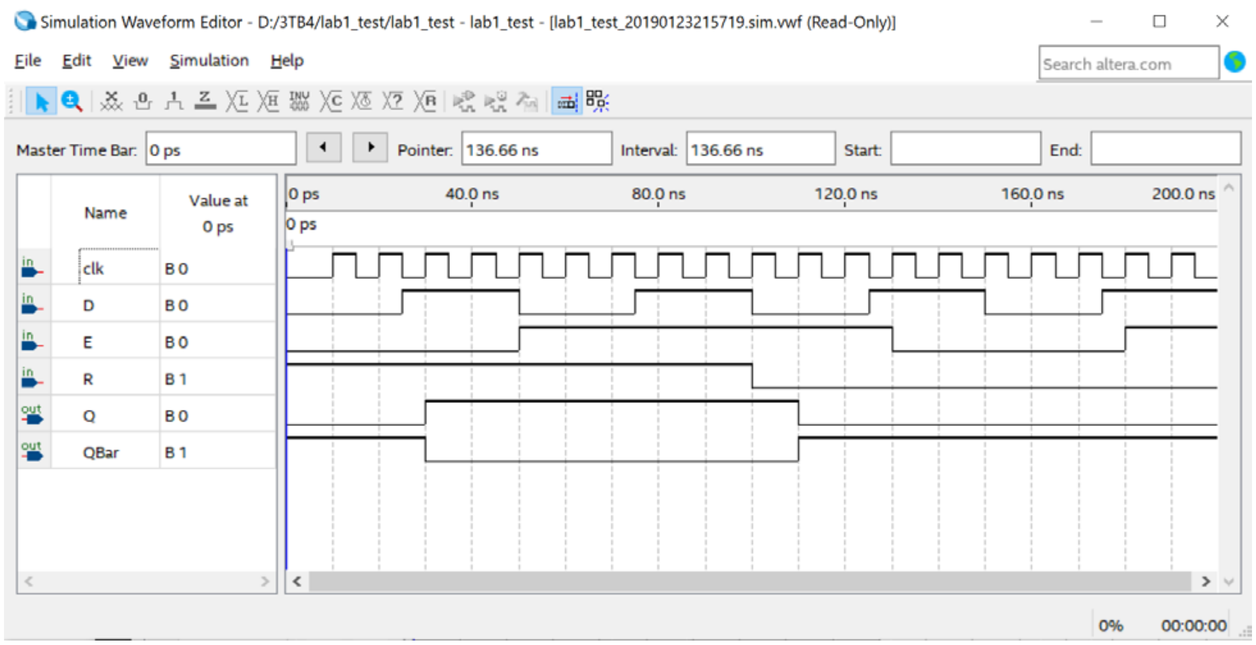
Name	Value at 0 ps
clk	B 0
D	B 0
R	B 0
Q	B 0
QBar	B 1

Figure 7: Waveform for One-Bit data width D flip-flop with active low synchronous reset



```verilog
//DFF w/ low reset & low enable

module dff_lowResetLowEnable(
    input       D, clk, R, E,
    output reg Q,
    output      QBar
    );

    always @(posedge clk)
    begin
        if (R == 1'b0)
            Q<=1'b0;
        else if (E == 1'b0)
            Q <= D;
    end

    assign QBar = ~Q;

endmodule
```

Figure 8: One-Bit data width D flip-flop with active low synchronous reset and active low enable

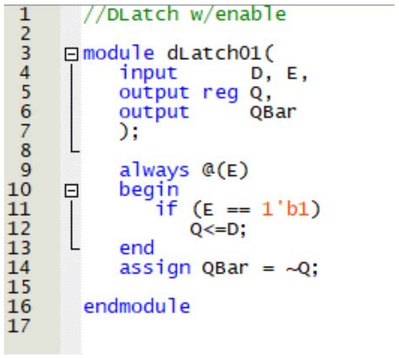


Simulation Waveform Editor - D:/3TB4/lab1_test/lab1_test - lab1_test - [lab1_test_20190123215719.sim.vwf (Read-Only)]

Master Time Bar: 0 ps   Pointer: 136.66 ns   Interval: 136.66 ns   Start: End:

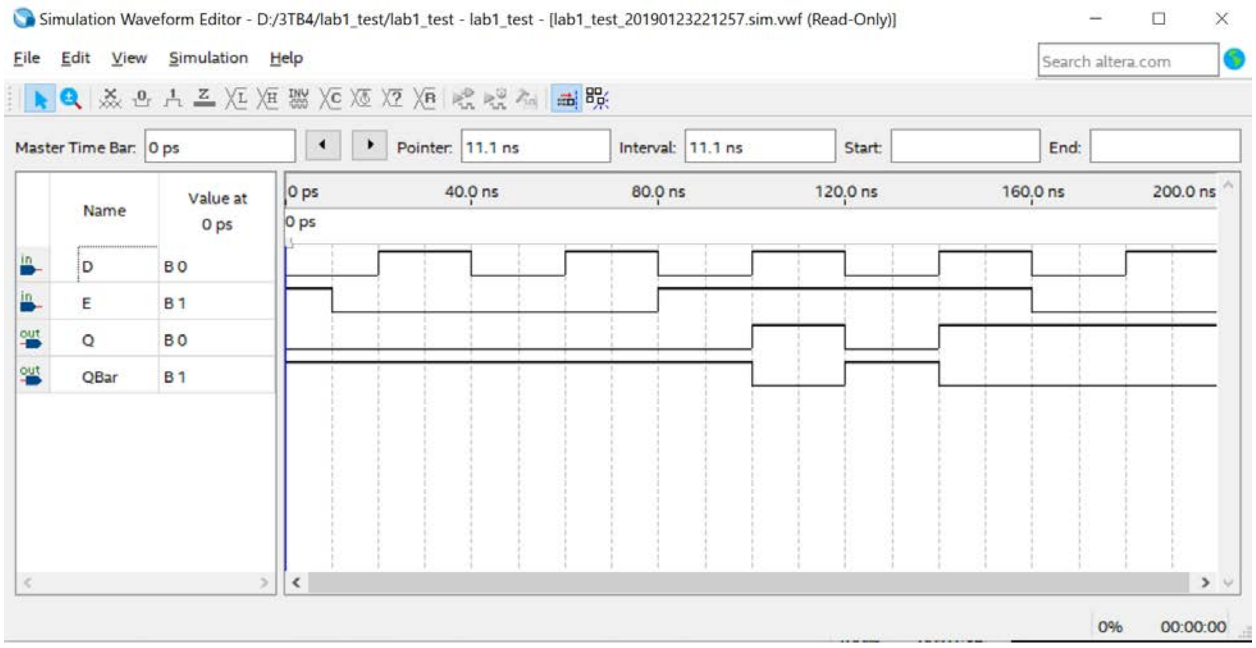
Name	Value at 0 ps
clk	B 0
D	B 0
E	B 0
R	B 1
Q	B 0
QBar	B 1

Figure 9: Waveform of One-Bit data width D flip-flop with active low synchronous reset and active low enable



```
//DLatch w/enable

module dLatch01(
    input       D, E,
    output reg Q,
    output      QBar
    );

    always @(E)
    begin
        if (E == 1'b1)
            Q<=D;
    end
    assign QBar = ~Q;

endmodule
```

Figure 10: D-Latch with synchronous enable control

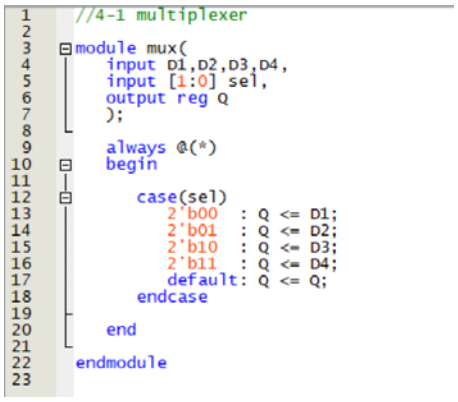


Simulation Waveform Editor - D:/3TB4/lab1_test/lab1_test - lab1_test - [lab1_test_20190123221257.sim.vwf (Read-Only)]

Master Time Bar: 0 ps | Pointer: 11.1 ns | Interval: 11.1 ns | Start: | End:

Name	Value at 0 ps
D	B 0
E	B 1
Q	B 0
QBar	B 1

Figure 11: Waveform of D-Latch with synchronous enable control



```
//4-1 multiplexer

module mux(
    input D1,D2,D3,D4,
    input [1:0] sel,
    output reg Q
    );

    always @(*)
    begin

        case(sel)
            2'b00   : Q <= D1;
            2'b01   : Q <= D2;
            2'b10   : Q <= D3;
            2'b11   : Q <= D4;
            default: Q <= Q;
        endcase

    end

endmodule
```

Figure 12: 4-to-1 multiplexer

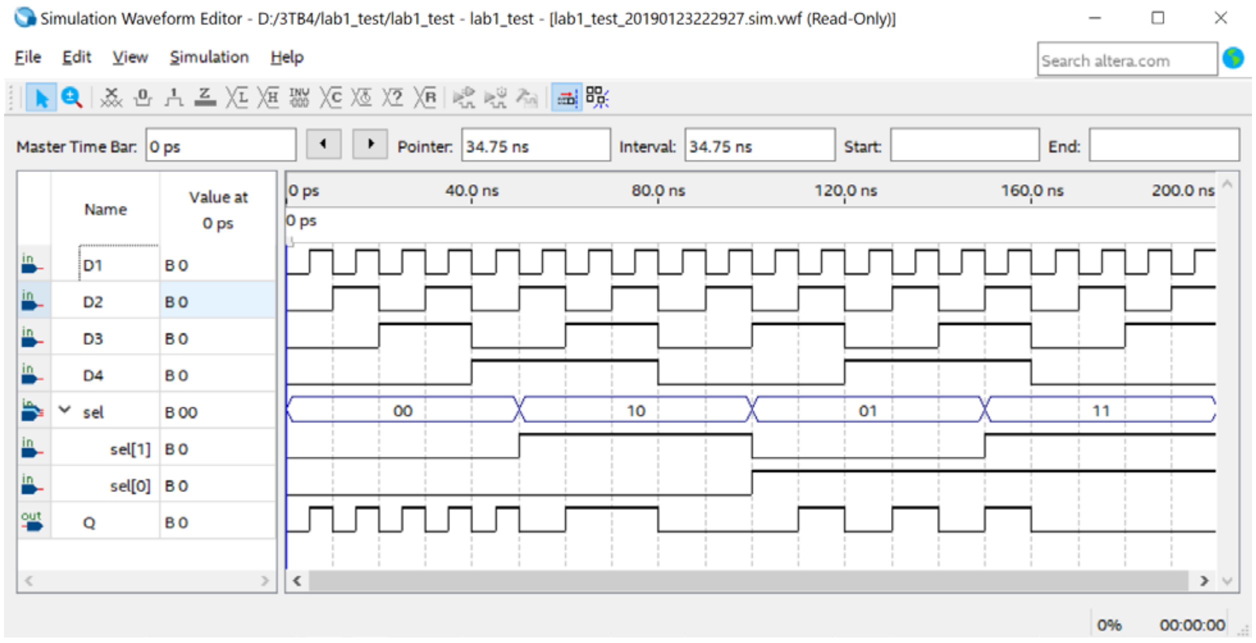
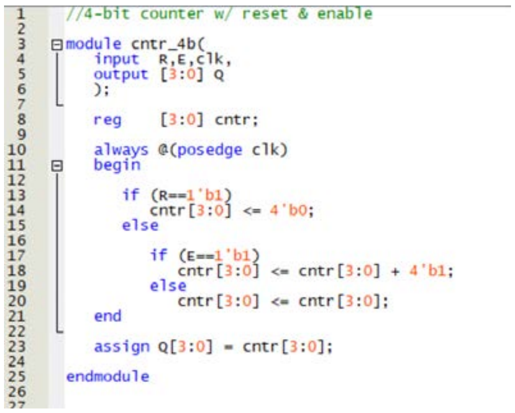
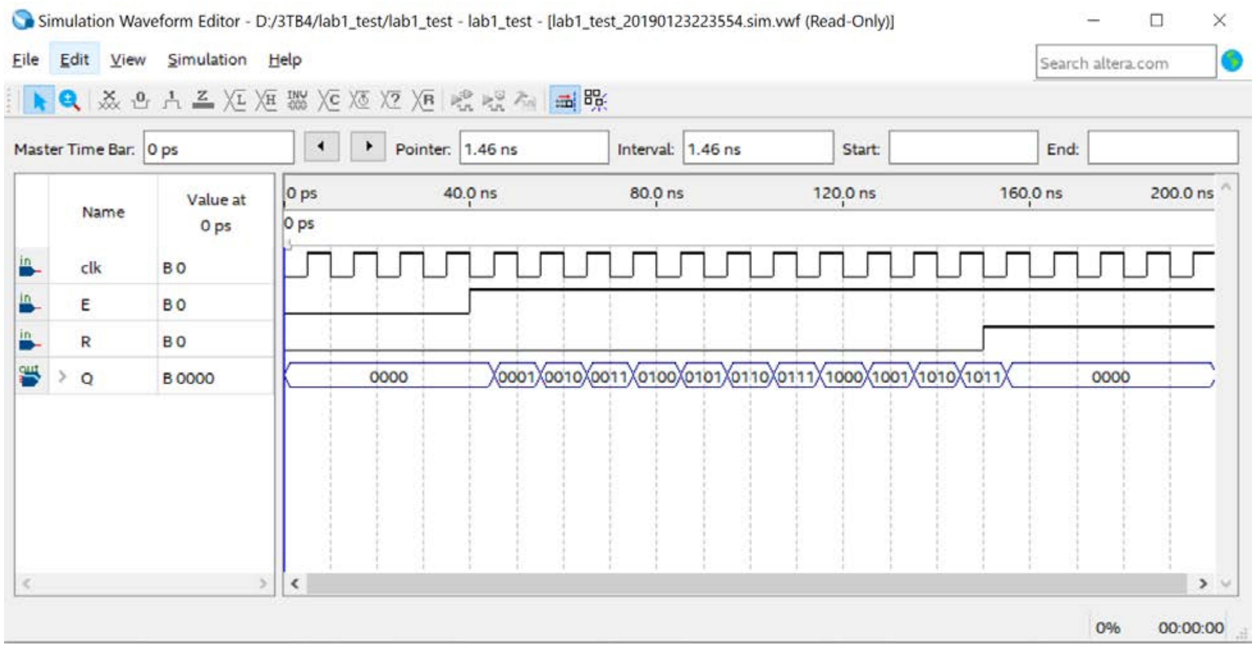


Figure 13: Waveform of 4-to-1 multiplexer



```
//4-bit counter w/ reset & enable

module cntr_4b(
    input  R,E,clk,
    output [3:0] Q
    );

    reg    [3:0] cntr;

    always @(posedge clk)
    begin

       if (R==1'b1)
          cntr[3:0] <= 4'b0;
       else

          if (E==1'b1)
             cntr[3:0] <= cntr[3:0] + 4'b1;
          else
             cntr[3:0] <= cntr[3:0];
    end

    assign Q[3:0] = cntr[3:0];

endmodule
```

Figure 14: 4-bit counter with reset and enable controls



Simulation Waveform Editor - D:/3TB4/lab1_test/lab1_test - lab1_test - [lab1_test_20190123223554.sim.vwf (Read-Only)]

Master Time Bar: 0 ps   Pointer: 1.46 ns   Interval: 1.46 ns   Start: End:

Name	Value at 0 ps
clk	B 0
E	B 0
R	B 0
Q	B 0000

Figure 15: Waveform of 4-bit counter with reset and enable controls

Truth Table for Seven-Segment Decoder

Table 1: Truth table for 7 segment display

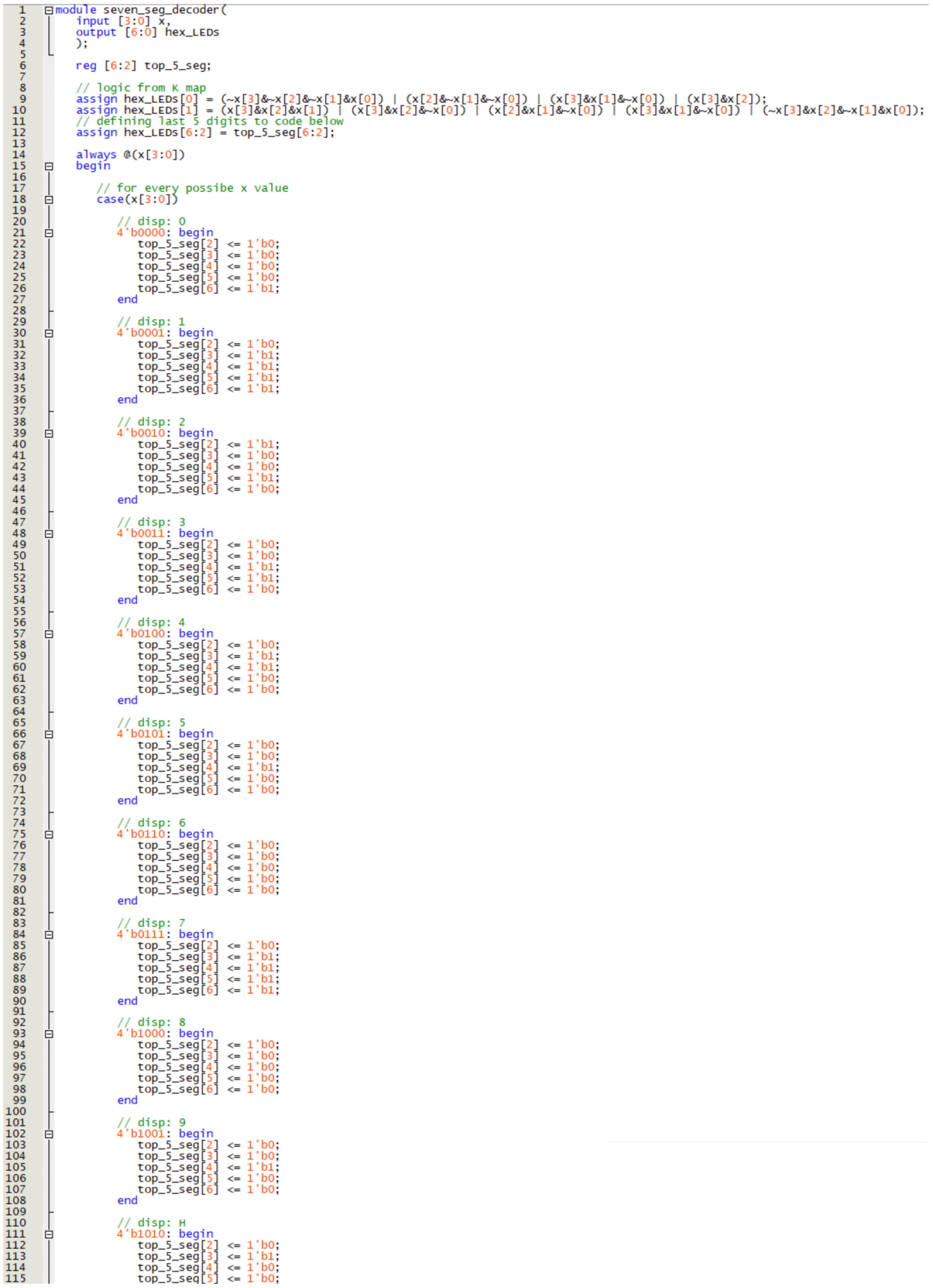
Char	Input				Segment						
	A_3	A_2	A_1	A_0	S_0	S_1	S_2	S_3	S_4	S_5	S_6
0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	1	1	1	1
2	0	0	1	0	0	0	1	0	0	1	0
3	0	0	1	1	0	0	0	0	1	1	0
4	0	1	0	0	1	0	0	1	1	0	0
5	0	1	0	1	0	1	0	0	1	0	0
6	0	1	1	0	0	1	0	0	0	0	0
7	0	1	1	1	0	0	0	1	1	1	1
8	1	0	0	0	0	0	0	0	0	0	0
9	1	0	0	1	0	0	0	0	1	0	0
H	1	0	1	0	1	1	0	1	0	0	0
A	1	0	1	1	0	0	0	1	0	0	0
N	1	1	0	0	1	1	0	1	0	1	0
K	1	1	0	1	1	0	0	1	0	0	0
B	1	1	1	0	1	1	0	0	0	0	0
NULL	1	1	1	1	1	1	1	1	1	1	1

Logic Expressions

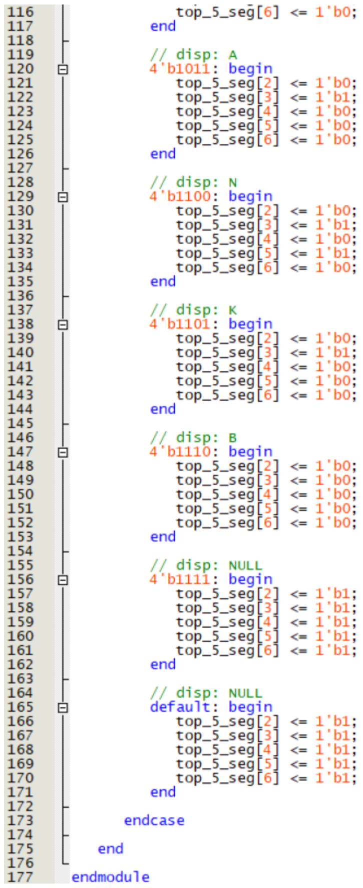
$$S_0 = \bar{A}_3\bar{A}_2\bar{A}_1A_0 + A_2\bar{A}_1\bar{A}_0 + A_3A_1\bar{A}_0 + A_3A_2$$

$$S_1 = A_3A_2A_1 + A_3A_2\bar{A}_0 + A_2A_1\bar{A}_0 + A_3A_1\bar{A}_0 + \bar{A}_3A_2\bar{A}_1A_0$$

Verilog Circuit



```
module seven_seg_decoder(
    input [3:0] x,
    output [6:0] hex_LEDs
    );

    reg [6:2] top_5_seg;

    // logic from K map
    assign hex_LEDs[0] = (~x[3]&~x[2]&~x[1]&x[0]) | (x[2]&~x[1]&~x[0]) | (x[3]&x[1]&~x[0]) | (x[3]&x[2]);
    assign hex_LEDs[1] = (x[3]&x[2]&x[1]) | (x[3]&x[2]&~x[0]) | (x[2]&x[1]&~x[0]) | (x[3]&x[1]&~x[0]) | (~x[3]&x[2]&~x[1]&x[0]);
    // defining last 5 digits to code below
    assign hex_LEDs[6:2] = top_5_seg[6:2];

    always @(x[3:0])
    begin

       // for every possibe x value
       case(x[3:0])

          // disp: 0
          4'b0000: begin
             top_5_seg[2] <= 1'b0;
             top_5_seg[3] <= 1'b0;
             top_5_seg[4] <= 1'b0;
             top_5_seg[5] <= 1'b0;
             top_5_seg[6] <= 1'b1;
          end

          // disp: 1
          4'b0001: begin
             top_5_seg[2] <= 1'b0;
             top_5_seg[3] <= 1'b1;
             top_5_seg[4] <= 1'b1;
             top_5_seg[5] <= 1'b1;
             top_5_seg[6] <= 1'b1;
          end

          // disp: 2
          4'b0010: begin
             top_5_seg[2] <= 1'b1;
             top_5_seg[3] <= 1'b0;
             top_5_seg[4] <= 1'b0;
             top_5_seg[5] <= 1'b1;
             top_5_seg[6] <= 1'b0;
          end

          // disp: 3
          4'b0011: begin
             top_5_seg[2] <= 1'b0;
             top_5_seg[3] <= 1'b0;
             top_5_seg[4] <= 1'b1;
             top_5_seg[5] <= 1'b1;
             top_5_seg[6] <= 1'b0;
          end

          // disp: 4
          4'b0100: begin
             top_5_seg[2] <= 1'b0;
             top_5_seg[3] <= 1'b1;
             top_5_seg[4] <= 1'b1;
             top_5_seg[5] <= 1'b0;
             top_5_seg[6] <= 1'b0;
          end

          // disp: 5
          4'b0101: begin
             top_5_seg[2] <= 1'b0;
             top_5_seg[3] <= 1'b0;
             top_5_seg[4] <= 1'b1;
             top_5_seg[5] <= 1'b0;
             top_5_seg[6] <= 1'b0;
          end

          // disp: 6
          4'b0110: begin
             top_5_seg[2] <= 1'b0;
             top_5_seg[3] <= 1'b0;
             top_5_seg[4] <= 1'b0;
             top_5_seg[5] <= 1'b0;
             top_5_seg[6] <= 1'b0;
          end

          // disp: 7
          4'b0111: begin
             top_5_seg[2] <= 1'b0;
             top_5_seg[3] <= 1'b1;
             top_5_seg[4] <= 1'b1;
             top_5_seg[5] <= 1'b1;
             top_5_seg[6] <= 1'b1;
          end

          // disp: 8
          4'b1000: begin
             top_5_seg[2] <= 1'b0;
             top_5_seg[3] <= 1'b0;
             top_5_seg[4] <= 1'b0;
             top_5_seg[5] <= 1'b0;
             top_5_seg[6] <= 1'b0;
          end

          // disp: 9
          4'b1001: begin
             top_5_seg[2] <= 1'b0;
             top_5_seg[3] <= 1'b0;
             top_5_seg[4] <= 1'b1;
             top_5_seg[5] <= 1'b0;
             top_5_seg[6] <= 1'b0;
          end

          // disp: H
          4'b1010: begin
             top_5_seg[2] <= 1'b0;
             top_5_seg[3] <= 1'b1;
             top_5_seg[4] <= 1'b0;
             top_5_seg[5] <= 1'b0;
```



```verilog
            top_5_seg[6] <= 1'b0;
         end

         // disp: A
         4'b1011: begin
            top_5_seg[2] <= 1'b0;
            top_5_seg[3] <= 1'b1;
            top_5_seg[4] <= 1'b0;
            top_5_seg[5] <= 1'b0;
            top_5_seg[6] <= 1'b0;
         end

         // disp: N
         4'b1100: begin
            top_5_seg[2] <= 1'b0;
            top_5_seg[3] <= 1'b1;
            top_5_seg[4] <= 1'b0;
            top_5_seg[5] <= 1'b1;
            top_5_seg[6] <= 1'b0;
         end

         // disp: K
         4'b1101: begin
            top_5_seg[2] <= 1'b0;
            top_5_seg[3] <= 1'b1;
            top_5_seg[4] <= 1'b0;
            top_5_seg[5] <= 1'b0;
            top_5_seg[6] <= 1'b0;
         end

         // disp: B
         4'b1110: begin
            top_5_seg[2] <= 1'b0;
            top_5_seg[3] <= 1'b0;
            top_5_seg[4] <= 1'b0;
            top_5_seg[5] <= 1'b0;
            top_5_seg[6] <= 1'b0;
         end

         // disp: NULL
         4'b1111: begin
            top_5_seg[2] <= 1'b1;
            top_5_seg[3] <= 1'b1;
            top_5_seg[4] <= 1'b1;
            top_5_seg[5] <= 1'b1;
            top_5_seg[6] <= 1'b1;
         end

         // disp: NULL
         default: begin
            top_5_seg[2] <= 1'b1;
            top_5_seg[3] <= 1'b1;
            top_5_seg[4] <= 1'b1;
            top_5_seg[5] <= 1'b1;
            top_5_seg[6] <= 1'b1;
         end

      endcase

   end

endmodule
```