Name: _____Servando_Olvera________ ID# ________1001909287_______

Date Submitted: _____03-19-2024____ Time Submitted _____4:00_pm______

CSE 3341 Digital Logic Design II

CSE 5357 Advanced Digital Logic Design

Spring Semester 2024

Lab 4 – Registered Eight x Eight Signed Multiplier

200 points

Due Date – March 21, 2024, 11:59 PM

Submit on Canvas Assignments

Note – Late submissions will not be accepted!

USIGNED VERISON

DESIGN REQUIREMENTS

Your assignment is to design a registered eight-by-eight unsigned multiplier by enhancing the four-by-four unsigned shift-and-add multiplier shown and discussed in class.

REQUIREMENTS:

1. Registered 8 x 8 multiplier
2. Unsigned numbers
3. SystemVerilog implementation
4. Design verification (simulation)
5. DE10-Lite realization

DESIGN VERIFICATION (simulation)

1. Simulate your designs to verify their correctness. Use the following values for M and Q in your simulations.
   (a) 01111111 x 00000001
   (b) 00010101 x 00101010
   (b) 01111111 x 11111111
   (c) 10101010 x 00110011
   (d) 10101010 x 11111110
   (e) 11011101 x 11001101
2. Include a screen shot of your simulation waveforms in your report.
3. Record the simulation results in a table for your report (use hexadecimal)
4. How many clock cycles does it take for each case to complete?

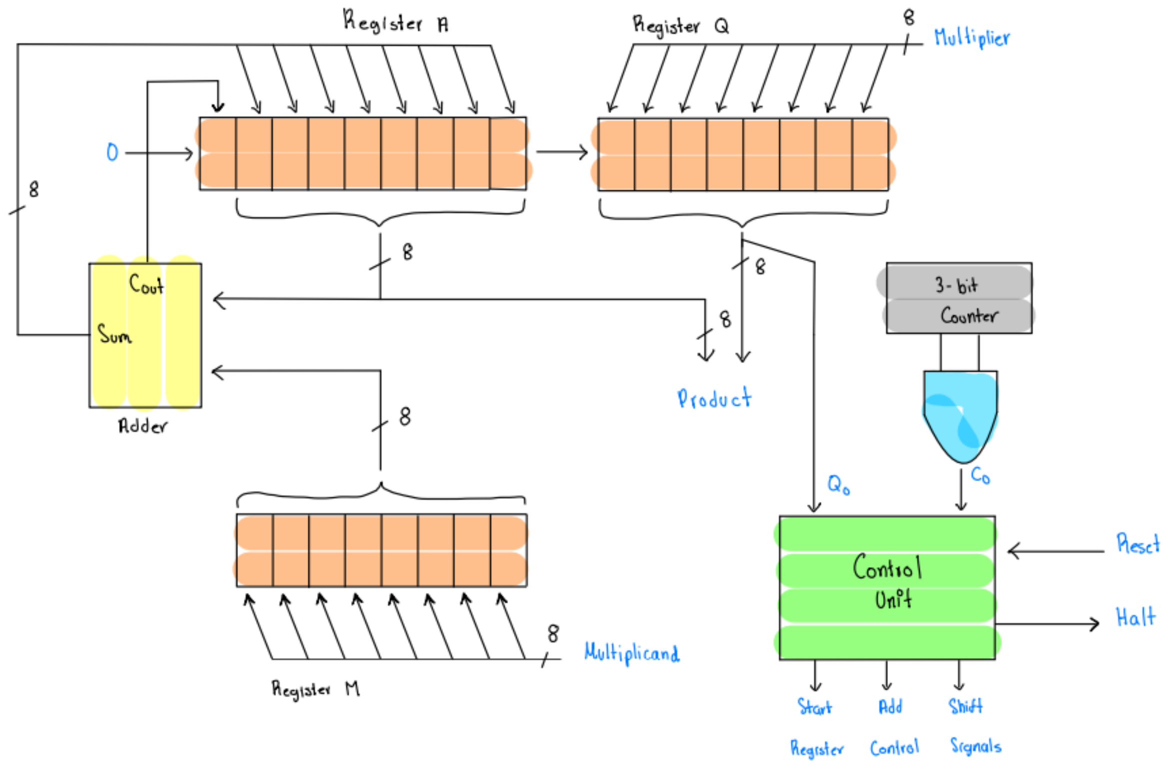
RTL ANALYSIS

1. Generate RTL diagrams using the Quartus Prime Netlist Viewer for each version.
2. Record the compilation summary for your report. How many ALM, registers, and pins does your design require?

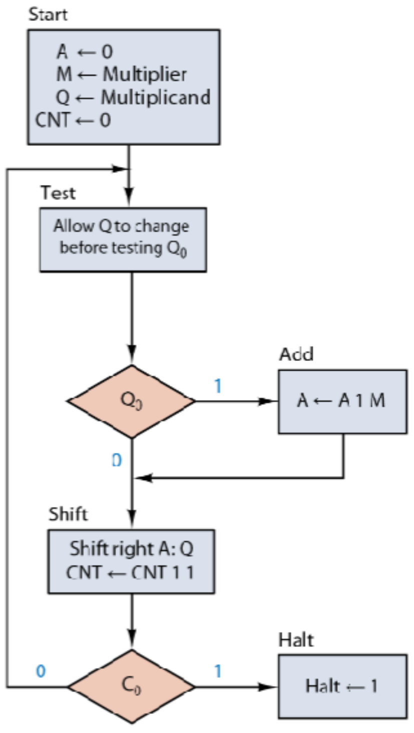
TIMING ANALYSIS

1. Run a timing analysis on your signed multiplier and determine its maximum operating speed in GHz.
2. Capture a screen shot of your timing analysis waveform showing the fastest clock speed your design will accommodate.

DATA PATH DIAGRAM

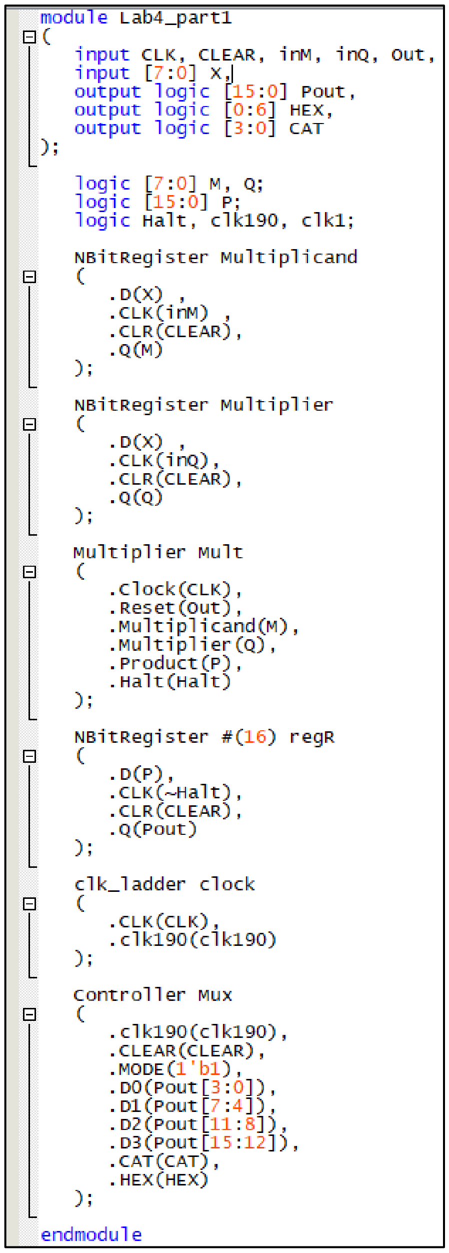


CONTROL PATH DIAGRAM



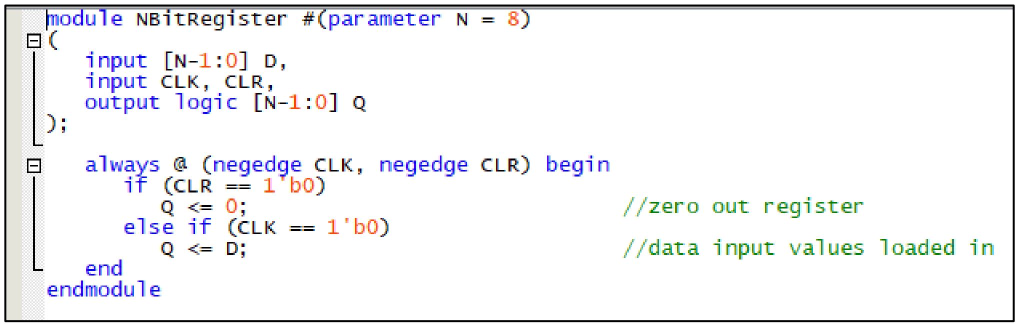
SYSTEM-VERILOG CODE

Top Module



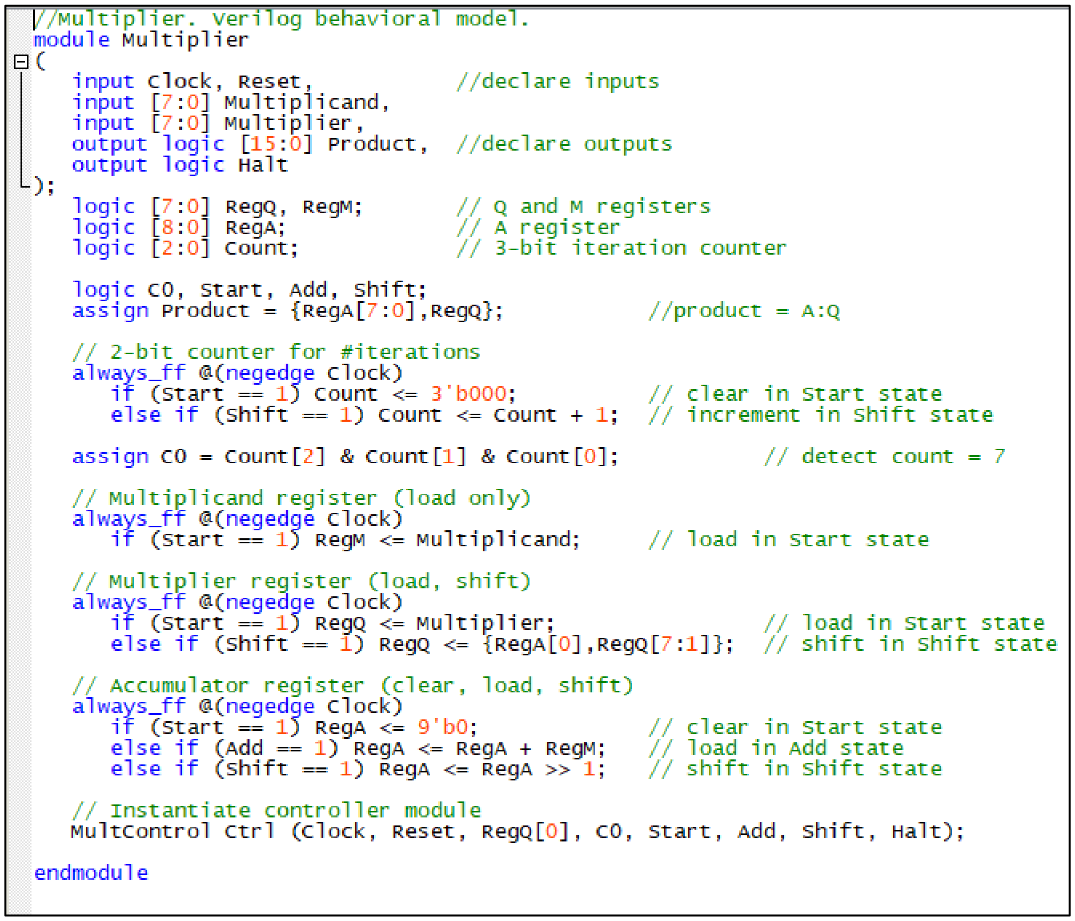
```
module Lab4_part1
(
    input CLK, CLEAR, inM, inQ, Out,
    input [7:0] X,
    output logic [15:0] Pout,
    output logic [0:6] HEX,
    output logic [3:0] CAT
);

    logic [7:0] M, Q;
    logic [15:0] P;
    logic Halt, clk190, clk1;

    NBitRegister Multiplicand
    (
        .D(X) ,
        .CLK(inM) ,
        .CLR(CLEAR),
        .Q(M)
    );

    NBitRegister Multiplier
    (
        .D(X) ,
        .CLK(inQ),
        .CLR(CLEAR),
        .Q(Q)
    );

    Multiplier Mult
    (
        .Clock(CLK),
        .Reset(Out),
        .Multiplicand(M),
        .Multiplier(Q),
        .Product(P),
        .Halt(Halt)
    );

    NBitRegister #(16) regR
    (
        .D(P),
        .CLK(~Halt),
        .CLR(CLEAR),
        .Q(Pout)
    );

    clk_ladder clock
    (
        .CLK(CLK),
        .clk190(clk190)
    );

    Controller Mux
    (
        .clk190(clk190),
        .CLEAR(CLEAR),
        .MODE(1'b1),
        .D0(Pout[3:0]),
        .D1(Pout[7:4]),
        .D2(Pout[11:8]),
        .D3(Pout[15:12]),
        .CAT(CAT),
        .HEX(HEX)
    );
endmodule
```

Register Module



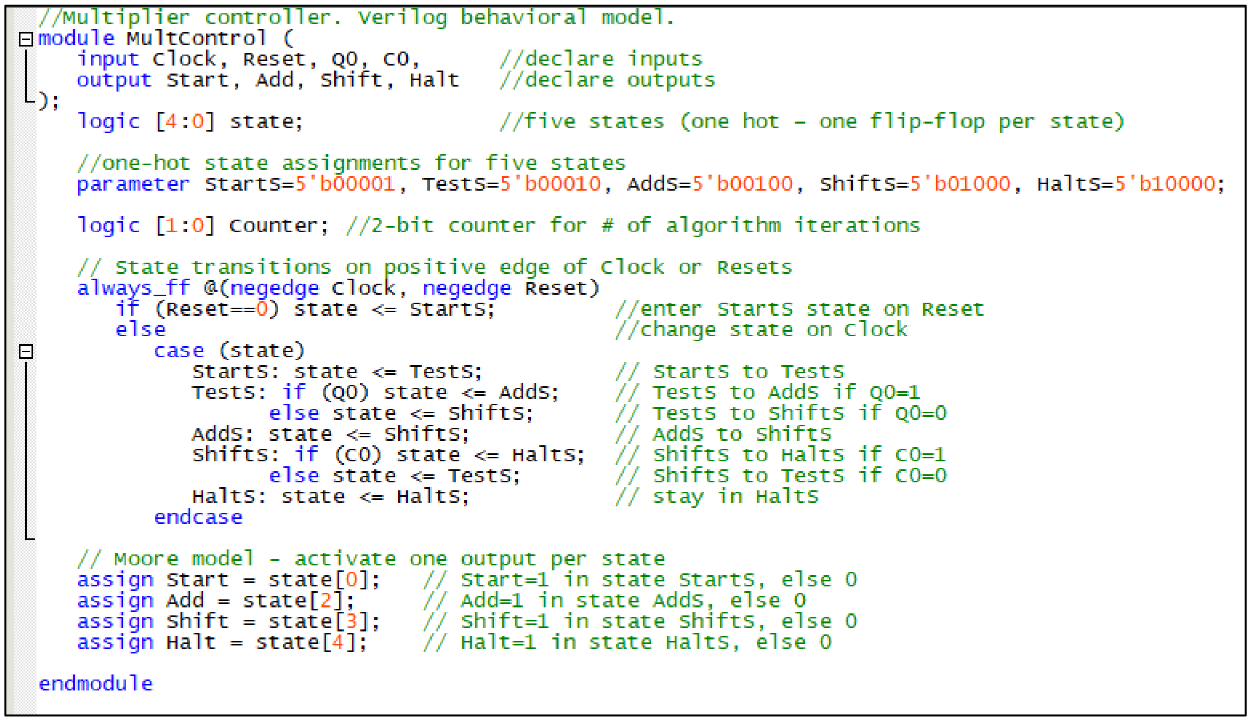
```
module NBitRegister #(parameter N = 8)
(
    input [N-1:0] D,
    input CLK, CLR,
    output logic [N-1:0] Q
);

    always @ (negedge CLK, negedge CLR) begin
        if (CLR == 1'b0)
            Q <= 0;                         //zero out register
        else if (CLK == 1'b0)
            Q <= D;                         //data input values loaded in
    end
endmodule
```

Multiplier Module



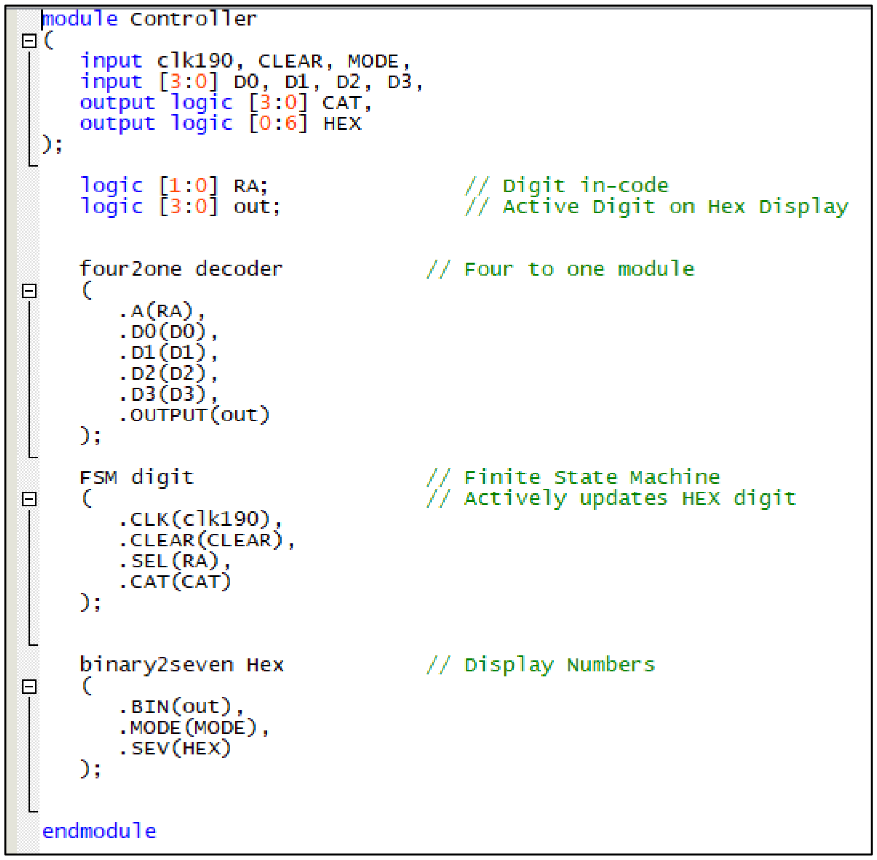
```
//Multiplier. Verilog behavioral model.
module Multiplier
(
   input Clock, Reset,           //declare inputs
   input [7:0] Multiplicand,
   input [7:0] Multiplier,
   output logic [15:0] Product,  //declare outputs
   output logic Halt
);
   logic [7:0] RegQ, RegM;       // Q and M registers
   logic [8:0] RegA;             // A register
   logic [2:0] Count;            // 3-bit iteration counter

   logic C0, Start, Add, Shift;
   assign Product = {RegA[7:0],RegQ};          //product = A:Q

   // 2-bit counter for #iterations
   always_ff @(negedge Clock)
      if (Start == 1) Count <= 3'b000;         // clear in Start state
      else if (Shift == 1) Count <= Count + 1; // increment in Shift state

   assign C0 = Count[2] & Count[1] & Count[0];           // detect count = 7

   // Multiplicand register (load only)
   always_ff @(negedge Clock)
      if (Start == 1) RegM <= Multiplicand;    // load in Start state

   // Multiplier register (load, shift)
   always_ff @(negedge Clock)
      if (Start == 1) RegQ <= Multiplier;                // load in Start state
      else if (Shift == 1) RegQ <= {RegA[0],RegQ[7:1]};  // shift in Shift state

   // Accumulator register (clear, load, shift)
   always_ff @(negedge Clock)
      if (Start == 1) RegA <= 9'b0;            // clear in Start state
      else if (Add == 1) RegA <= RegA + RegM;  // load in Add state
      else if (Shift == 1) RegA <= RegA >> 1;  // shift in Shift state

   // Instantiate controller module
   MultControl Ctrl (Clock, Reset, RegQ[0], C0, Start, Add, Shift, Halt);

endmodule
```

Multiplier Control Module



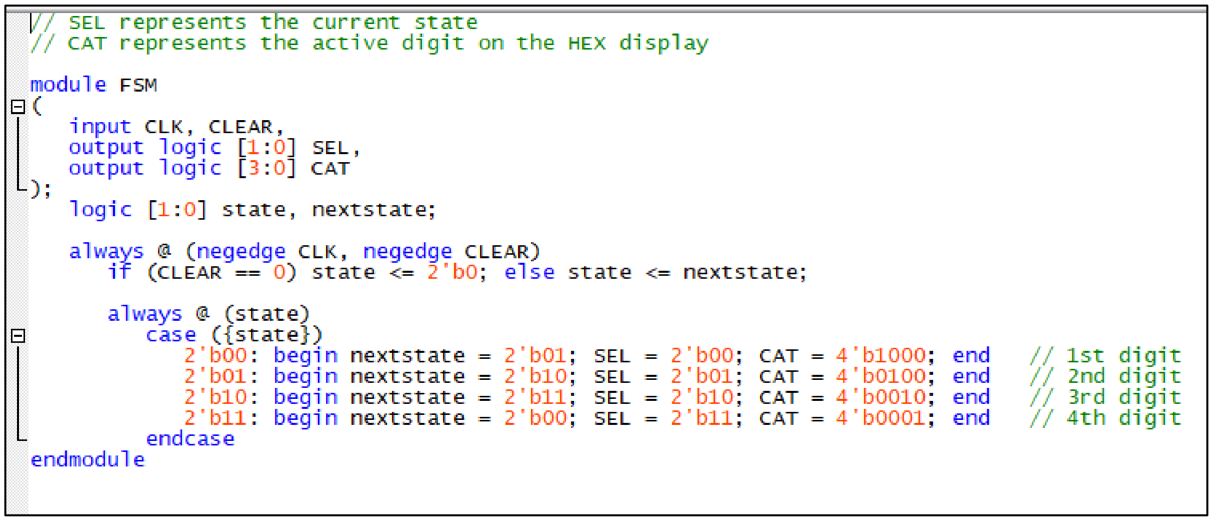
```
//Multiplier controller. Verilog behavioral model.
module MultControl (
   input Clock, Reset, Q0, C0,     //declare inputs
   output Start, Add, Shift, Halt  //declare outputs
);
   logic [4:0] state;              //five states (one hot - one flip-flop per state)

   //one-hot state assignments for five states
   parameter StartS=5'b00001, TestS=5'b00010, AddS=5'b00100, ShiftS=5'b01000, HaltS=5'b10000;

   logic [1:0] Counter; //2-bit counter for # of algorithm iterations

   // State transitions on positive edge of Clock or Resets
   always_ff @(negedge Clock, negedge Reset)
      if (Reset==0) state <= StartS;         //enter StartS state on Reset
      else                                   //change state on Clock
         case (state)
            StartS: state <= TestS;          // StartS to TestS
            TestS: if (Q0) state <= AddS;    // TestS to AddS if Q0=1
                   else state <= ShiftS;     // TestS to ShiftS if Q0=0
            AddS: state <= ShiftS;           // AddS to ShiftS
            ShiftS: if (C0) state <= HaltS;  // ShiftS to HaltS if C0=1
                   else state <= TestS;      // ShiftS to TestS if C0=0
            HaltS: state <= HaltS;           // stay in HaltS
         endcase

   // Moore model - activate one output per state
   assign Start = state[0];   // Start=1 in state StartS, else 0
   assign Add = state[2];     // Add=1 in state AddS, else 0
   assign Shift = state[3];   // Shift=1 in state ShiftS, else 0
   assign Halt = state[4];    // Halt=1 in state HaltS, else 0

endmodule
```

MUX Controller Module



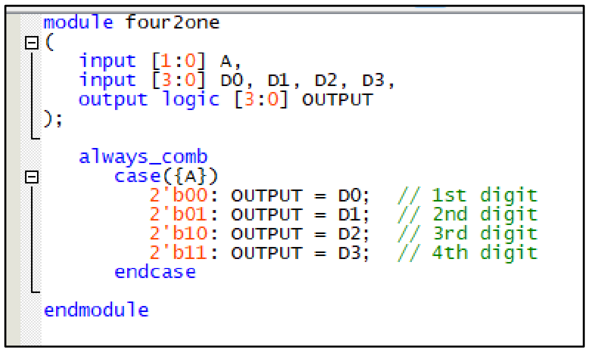
```
module Controller
(
    input clk190, CLEAR, MODE,
    input [3:0] D0, D1, D2, D3,
    output logic [3:0] CAT,
    output logic [0:6] HEX
);

    logic [1:0] RA;               // Digit in-code
    logic [3:0] out;              // Active Digit on Hex Display


    four2one decoder           // Four to one module
    (
        .A(RA),
        .D0(D0),
        .D1(D1),
        .D2(D2),
        .D3(D3),
        .OUTPUT(out)
    );

    FSM digit                  // Finite State Machine
    (                          // Actively updates HEX digit
        .CLK(clk190),
        .CLEAR(CLEAR),
        .SEL(RA),
        .CAT(CAT)
    );


    binary2seven Hex           // Display Numbers
    (
        .BIN(out),
        .MODE(MODE),
        .SEV(HEX)
    );

endmodule
```

Finite State Machine Module



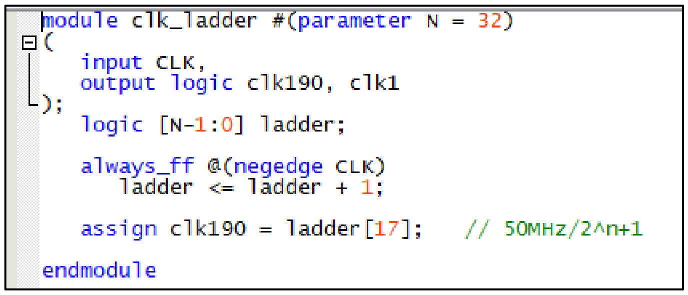
```
// SEL represents the current state
// CAT represents the active digit on the HEX display

module FSM
(
    input CLK, CLEAR,
    output logic [1:0] SEL,
    output logic [3:0] CAT
);
    logic [1:0] state, nextstate;

    always @ (negedge CLK, negedge CLEAR)
        if (CLEAR == 0) state <= 2'b0; else state <= nextstate;

        always @ (state)
            case ({state})
                2'b00: begin nextstate = 2'b01; SEL = 2'b00; CAT = 4'b1000; end   // 1st digit
                2'b01: begin nextstate = 2'b10; SEL = 2'b01; CAT = 4'b0100; end   // 2nd digit
                2'b10: begin nextstate = 2'b11; SEL = 2'b10; CAT = 4'b0010; end   // 3rd digit
                2'b11: begin nextstate = 2'b00; SEL = 2'b11; CAT = 4'b0001; end   // 4th digit
            endcase
endmodule
```

Four to One Decoder Module



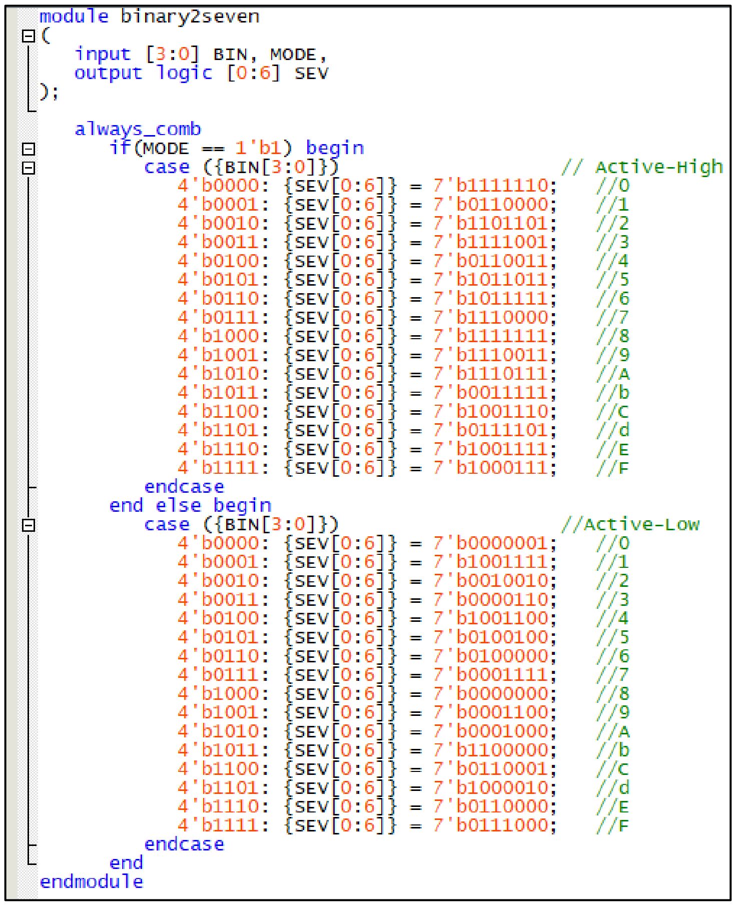
```
module four2one
(
    input [1:0] A,
    input [3:0] D0, D1, D2, D3,
    output logic [3:0] OUTPUT
);

    always_comb
       case({A})
          2'b00: OUTPUT = D0;   // 1st digit
          2'b01: OUTPUT = D1;   // 2nd digit
          2'b10: OUTPUT = D2;   // 3rd digit
          2'b11: OUTPUT = D3;   // 4th digit
       endcase

endmodule
```

Clock Ladder Module



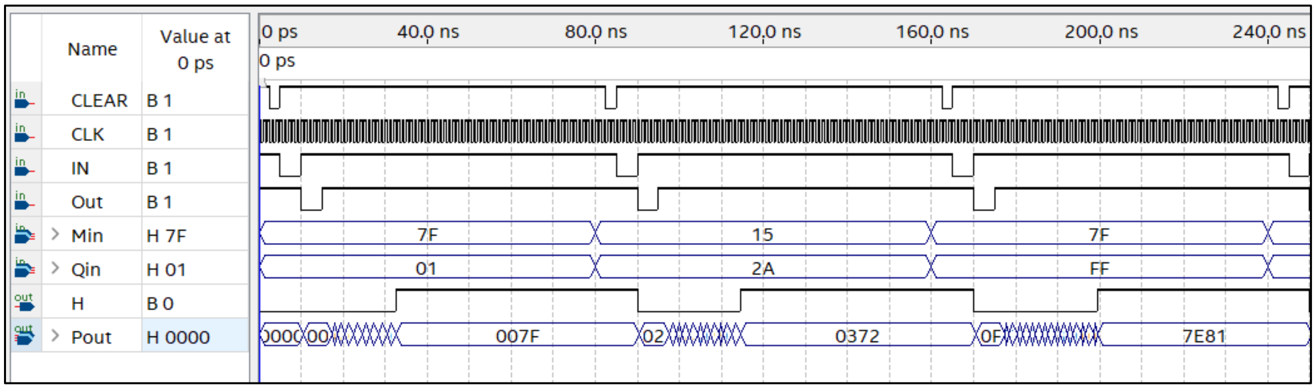
```
module clk_ladder #(parameter N = 32)
(
    input CLK,
    output logic clk190, clk1
);
    logic [N-1:0] ladder;

    always_ff @(negedge CLK)
       ladder <= ladder + 1;

    assign clk190 = ladder[17];    // 50MHz/2^n+1

endmodule
```

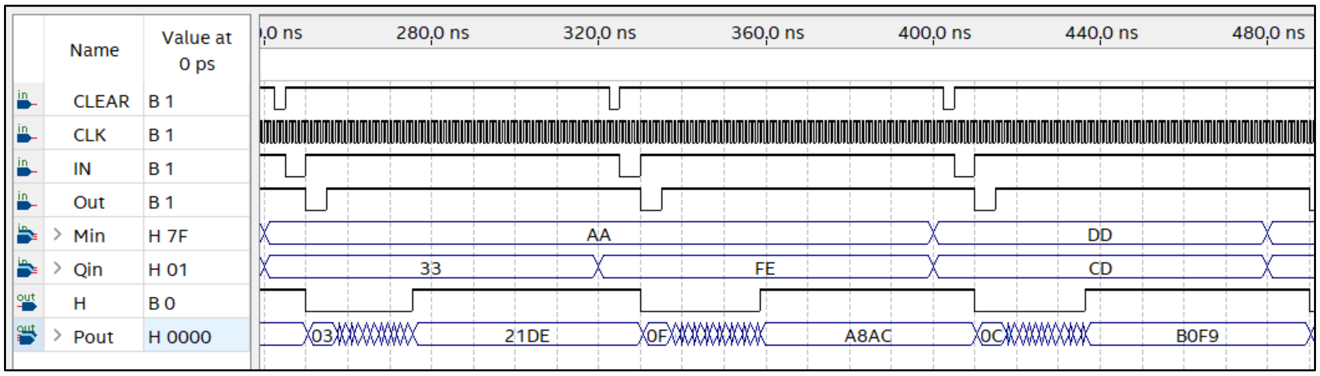
Binary to Seven-Seg Display Decoder Module



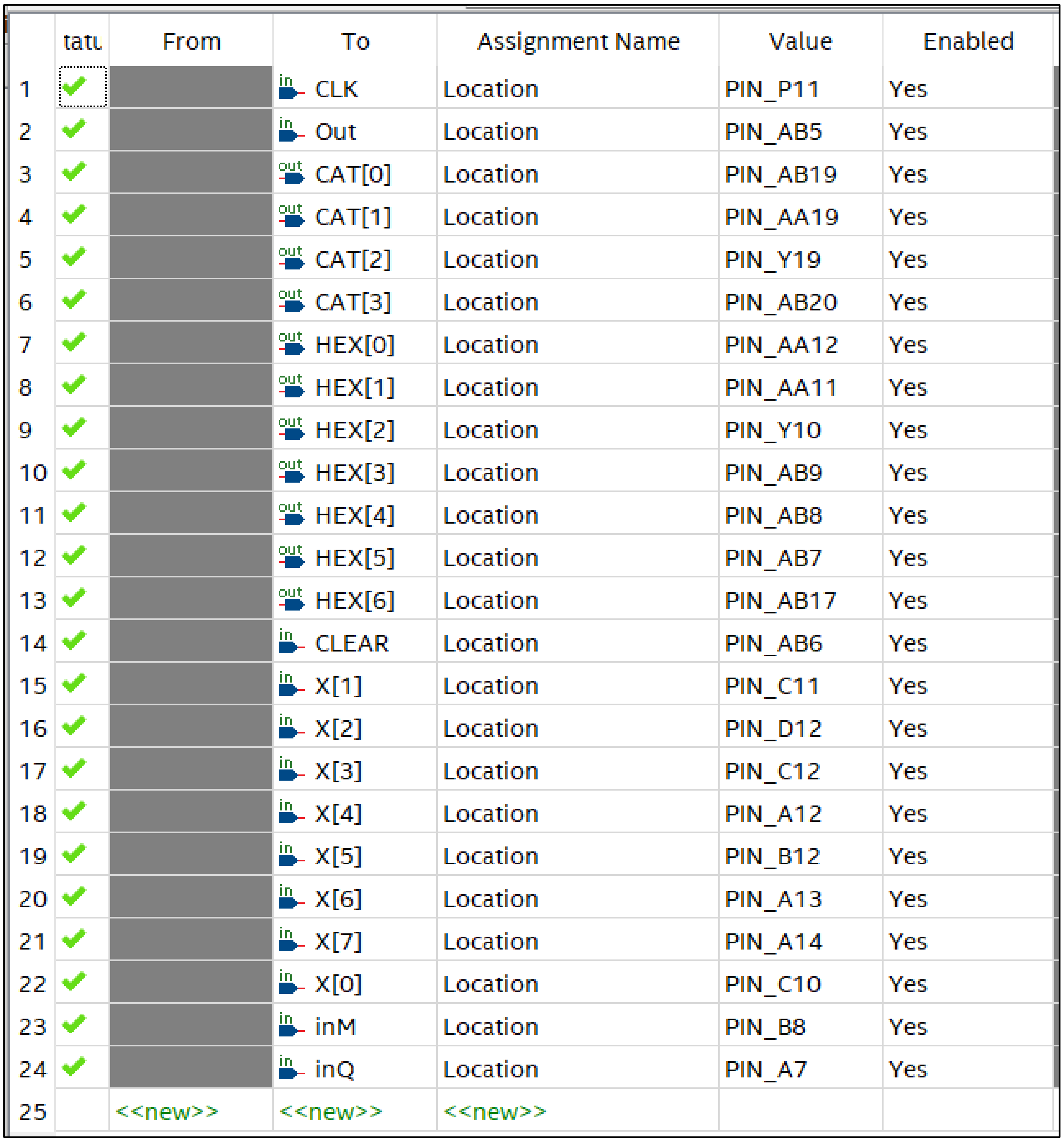
```
module binary2seven
(
    input [3:0] BIN, MODE,
    output logic [0:6] SEV
);

    always_comb
       if(MODE == 1'b1) begin
          case ({BIN[3:0]})                            // Active-High
             4'b0000: {SEV[0:6]} = 7'b1111110;    //0
             4'b0001: {SEV[0:6]} = 7'b0110000;    //1
             4'b0010: {SEV[0:6]} = 7'b1101101;    //2
             4'b0011: {SEV[0:6]} = 7'b1111001;    //3
             4'b0100: {SEV[0:6]} = 7'b0110011;    //4
             4'b0101: {SEV[0:6]} = 7'b1011011;    //5
             4'b0110: {SEV[0:6]} = 7'b1011111;    //6
             4'b0111: {SEV[0:6]} = 7'b1110000;    //7
             4'b1000: {SEV[0:6]} = 7'b1111111;    //8
             4'b1001: {SEV[0:6]} = 7'b1110011;    //9
             4'b1010: {SEV[0:6]} = 7'b1110111;    //A
             4'b1011: {SEV[0:6]} = 7'b0011111;    //b
             4'b1100: {SEV[0:6]} = 7'b1001110;    //C
             4'b1101: {SEV[0:6]} = 7'b0111101;    //d
             4'b1110: {SEV[0:6]} = 7'b1001111;    //E
             4'b1111: {SEV[0:6]} = 7'b1000111;    //F
          endcase
       end else begin
          case ({BIN[3:0]})                            //Active-Low
             4'b0000: {SEV[0:6]} = 7'b0000001;    //0
             4'b0001: {SEV[0:6]} = 7'b1001111;    //1
             4'b0010: {SEV[0:6]} = 7'b0010010;    //2
             4'b0011: {SEV[0:6]} = 7'b0000110;    //3
             4'b0100: {SEV[0:6]} = 7'b1001100;    //4
             4'b0101: {SEV[0:6]} = 7'b0100100;    //5
             4'b0110: {SEV[0:6]} = 7'b0100000;    //6
             4'b0111: {SEV[0:6]} = 7'b0001111;    //7
             4'b1000: {SEV[0:6]} = 7'b0000000;    //8
             4'b1001: {SEV[0:6]} = 7'b0001100;    //9
             4'b1010: {SEV[0:6]} = 7'b0001000;    //A
             4'b1011: {SEV[0:6]} = 7'b1100000;    //b
             4'b1100: {SEV[0:6]} = 7'b0110001;    //C
             4'b1101: {SEV[0:6]} = 7'b1000010;    //d
             4'b1110: {SEV[0:6]} = 7'b0110000;    //E
             4'b1111: {SEV[0:6]} = 7'b0111000;    //F
          endcase
       end
endmodule
```

SIMULATION RESULTS WAVEFORM





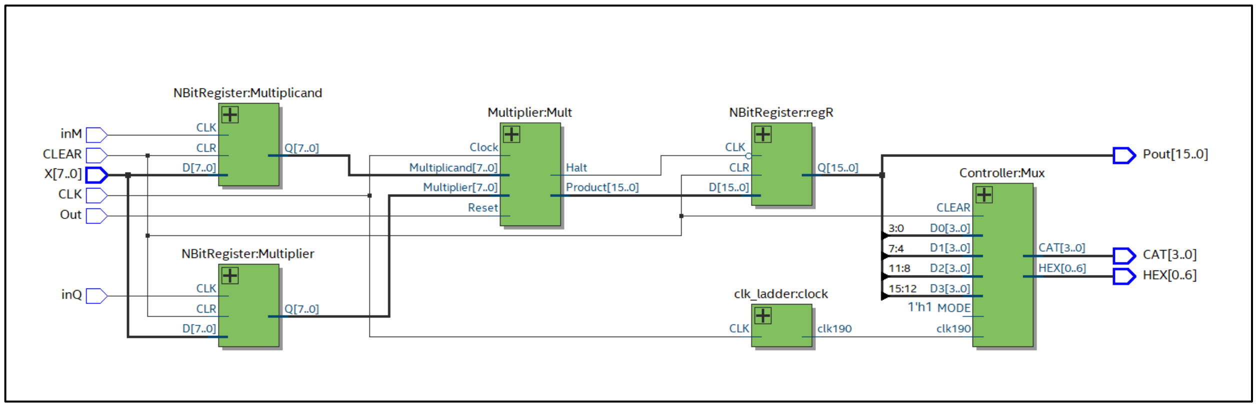
PIN ASSIGMENTS



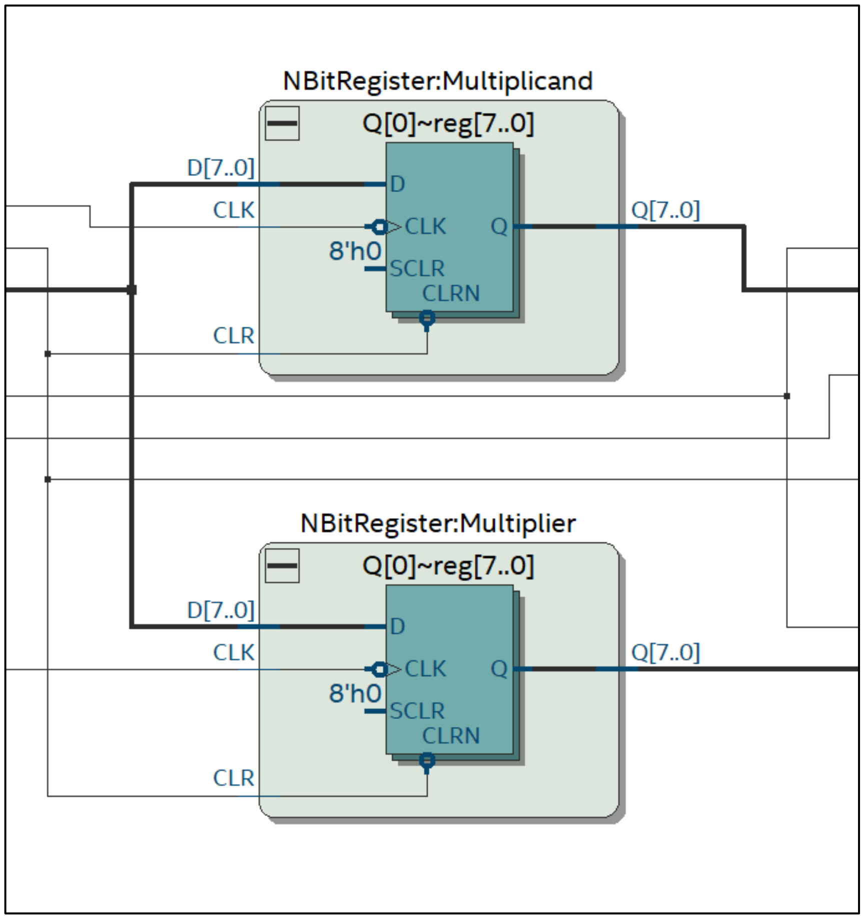
	tatu	From	To	Assignment Name	Value	Enabled
1	✓		in CLK	Location	PIN_P11	Yes
2	✓		in Out	Location	PIN_AB5	Yes
3	✓		out CAT[0]	Location	PIN_AB19	Yes
4	✓		out CAT[1]	Location	PIN_AA19	Yes
5	✓		out CAT[2]	Location	PIN_Y19	Yes
6	✓		out CAT[3]	Location	PIN_AB20	Yes
7	✓		out HEX[0]	Location	PIN_AA12	Yes
8	✓		out HEX[1]	Location	PIN_AA11	Yes
9	✓		out HEX[2]	Location	PIN_Y10	Yes
10	✓		out HEX[3]	Location	PIN_AB9	Yes
11	✓		out HEX[4]	Location	PIN_AB8	Yes
12	✓		out HEX[5]	Location	PIN_AB7	Yes
13	✓		out HEX[6]	Location	PIN_AB17	Yes
14	✓		in CLEAR	Location	PIN_AB6	Yes
15	✓		in X[1]	Location	PIN_C11	Yes
16	✓		in X[2]	Location	PIN_D12	Yes
17	✓		in X[3]	Location	PIN_C12	Yes
18	✓		in X[4]	Location	PIN_A12	Yes
19	✓		in X[5]	Location	PIN_B12	Yes
20	✓		in X[6]	Location	PIN_A13	Yes
21	✓		in X[7]	Location	PIN_A14	Yes
22	✓		in X[0]	Location	PIN_C10	Yes
23	✓		in inM	Location	PIN_B8	Yes
24	✓		in inQ	Location	PIN_A7	Yes
25		<<new>>	<<new>>	<<new>>		

RTL DIAGRAMS

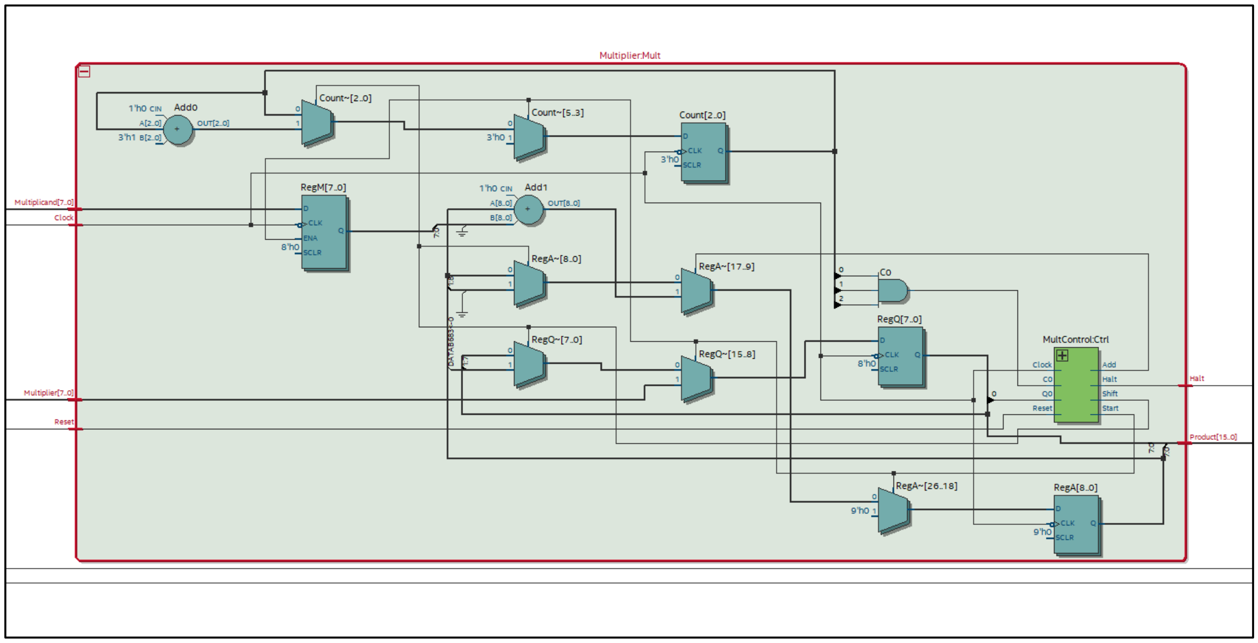
Top Module



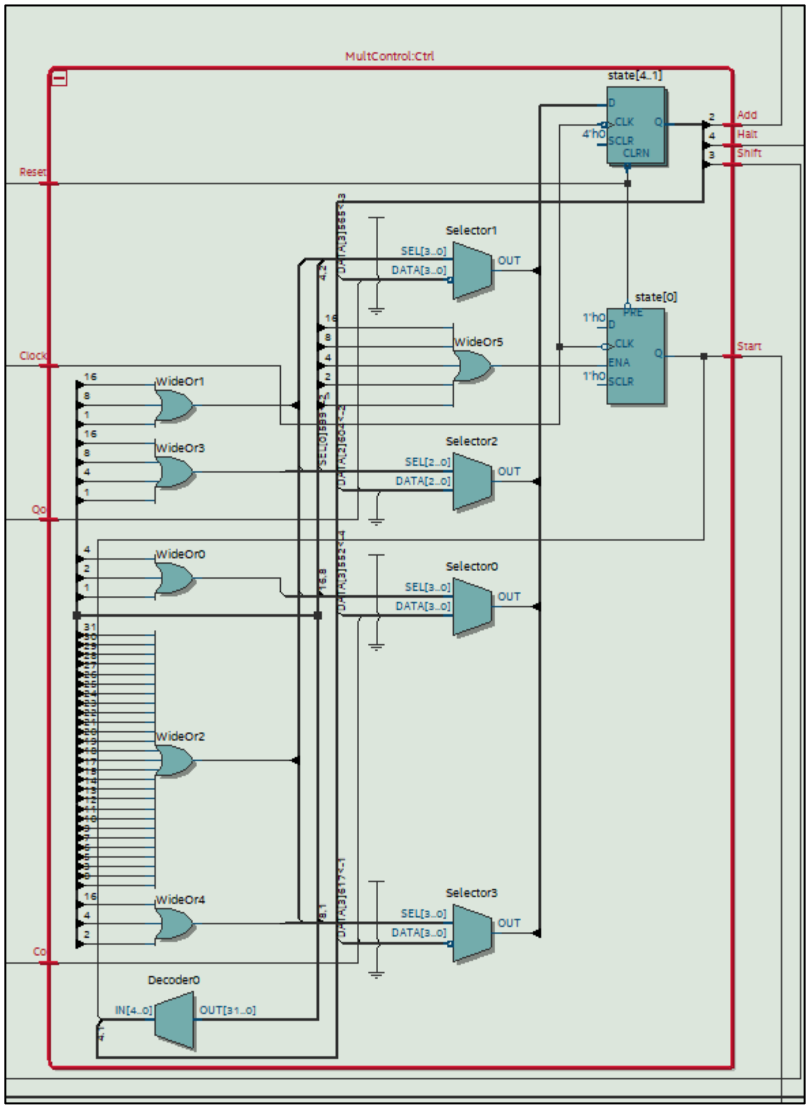
Multiplicand & Multiplier Registers



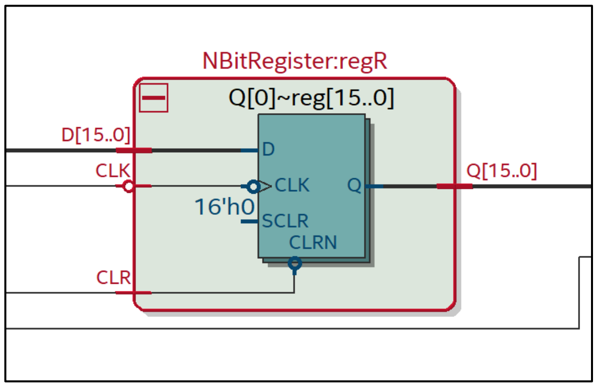
Multiplier



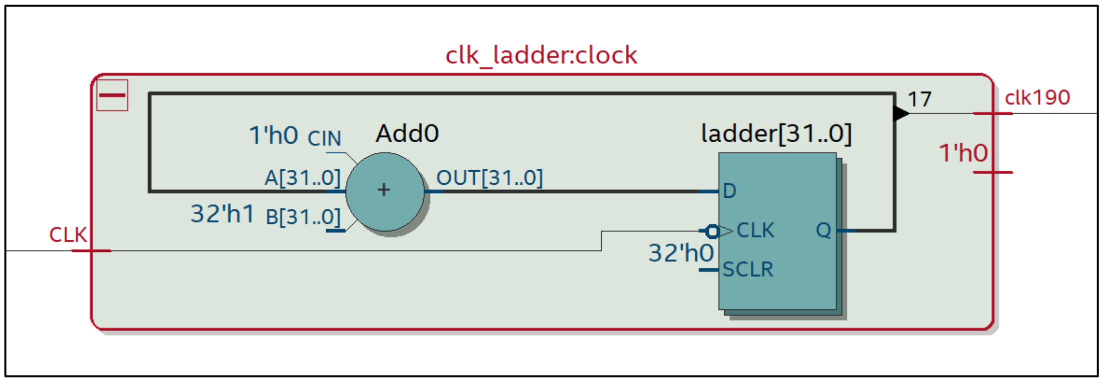
Multiplication Control Module



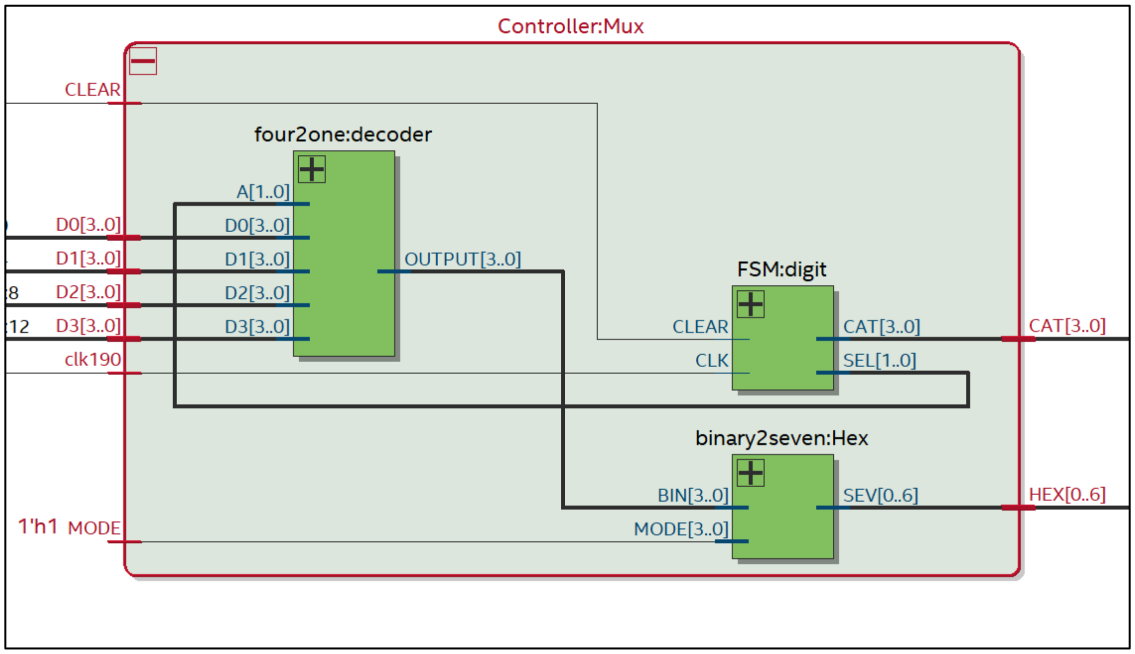
Product Register



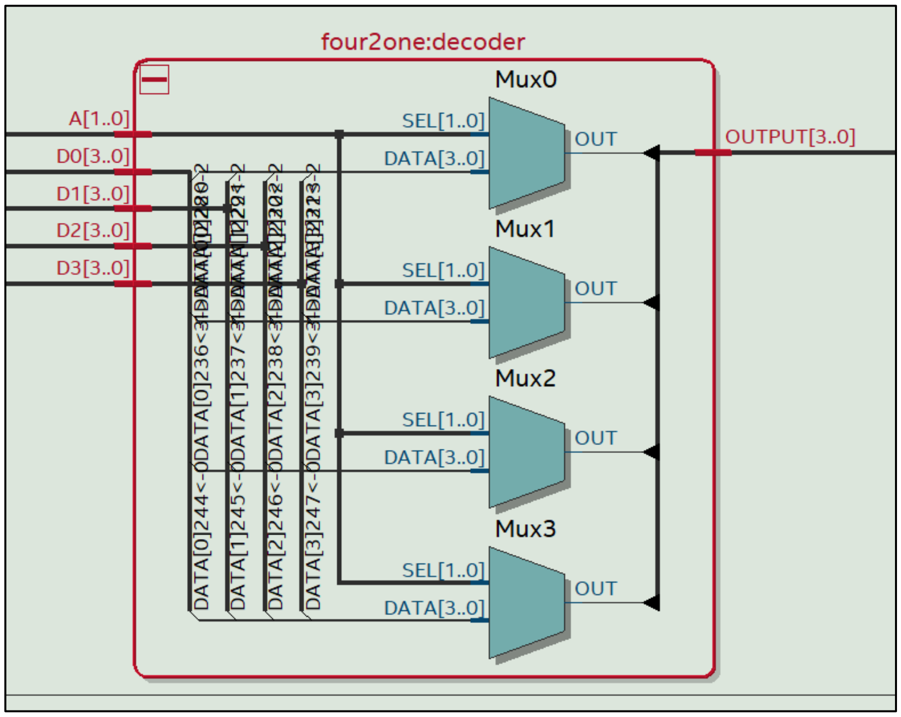
Clock Ladder



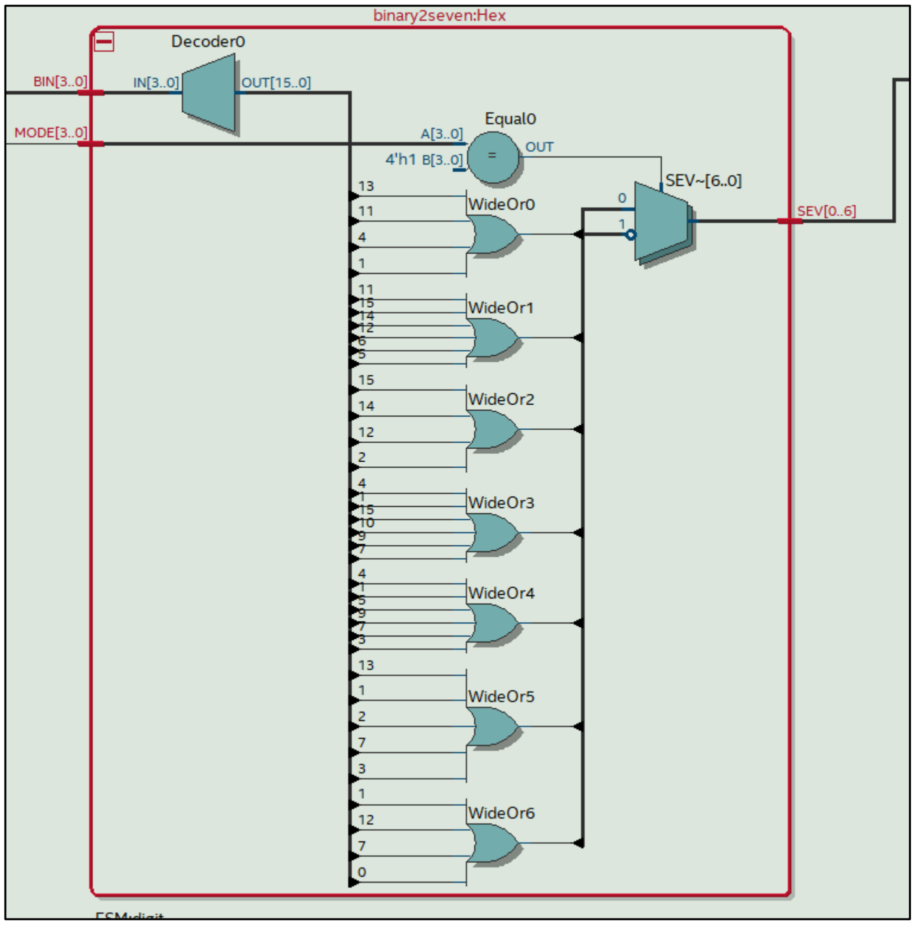
MUX/Controller



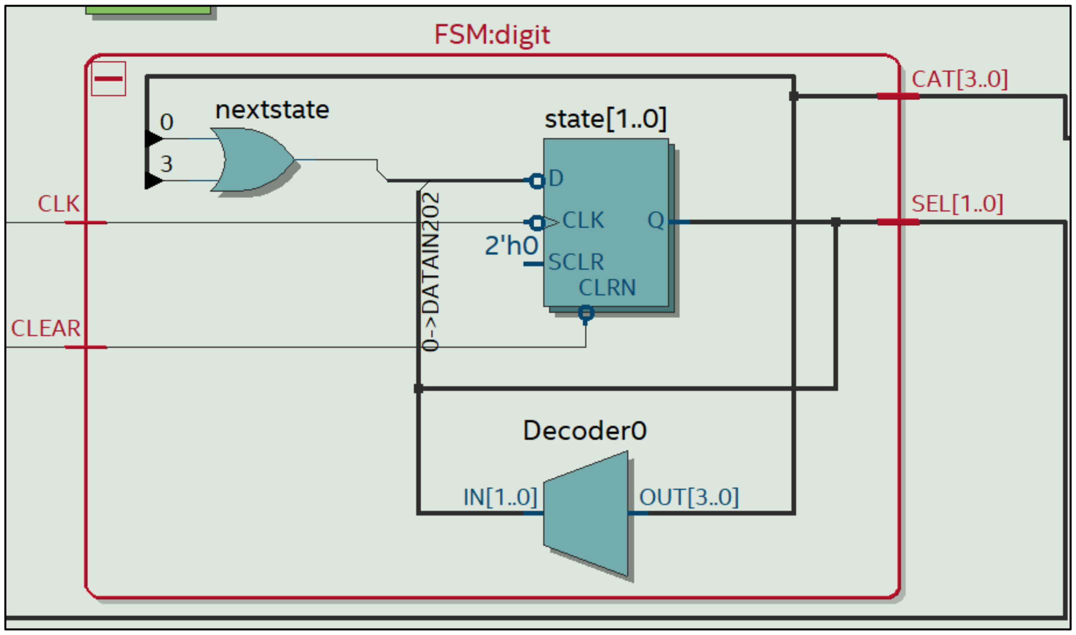
Four to One Decoder



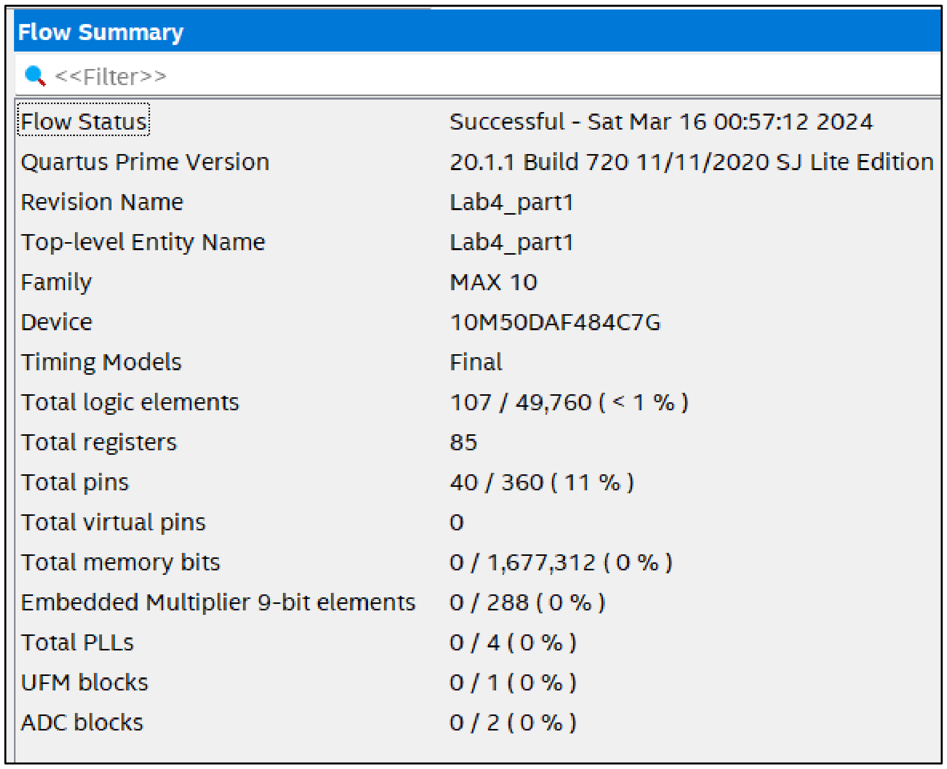
Binary To Seven Segment Display



Finite State Machine



Compilation Summary



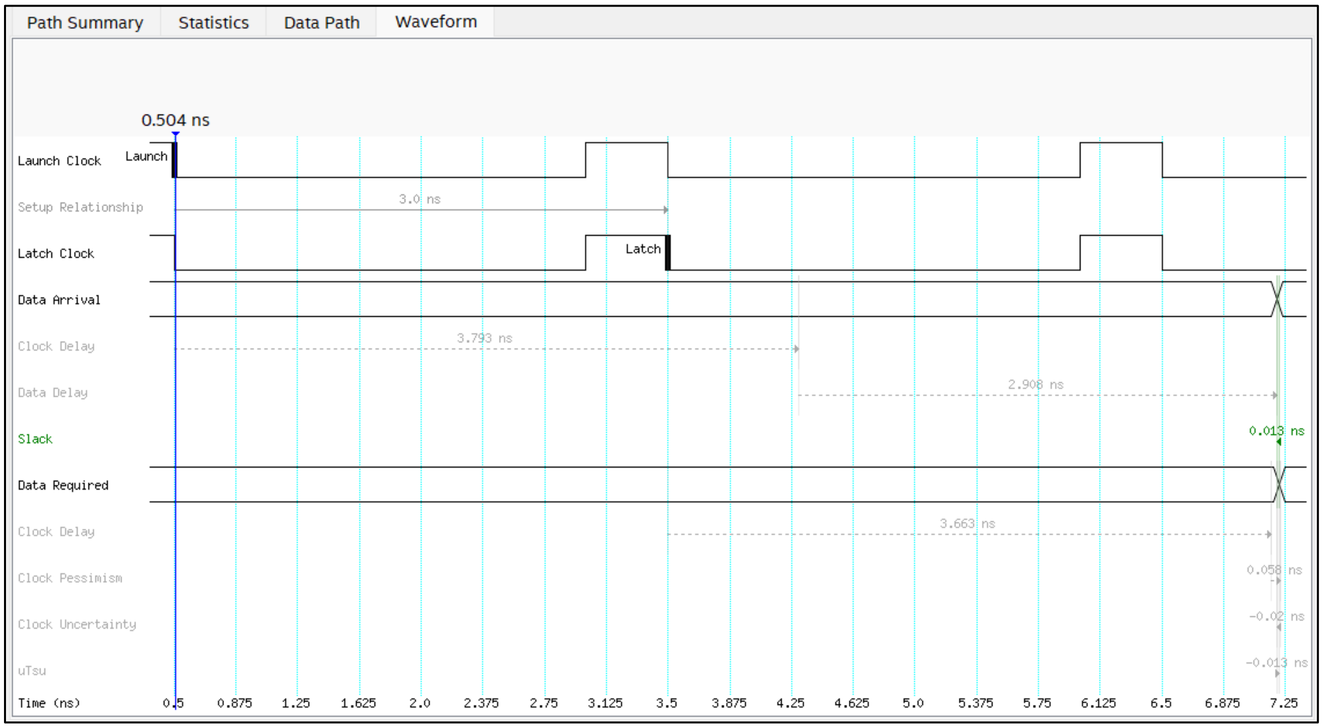
Flow Summary

<<Filter>>

Flow Status	Successful - Sat Mar 16 00:57:12 2024
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	Lab4_part1
Top-level Entity Name	Lab4_part1
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	107 / 49,760 (< 1 %)
Total registers	85
Total pins	40 / 360 (11 %)
Total virtual pins	0
Total memory bits	0 / 1,677,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

# of ALMs	= 107
# of REGISTERS	= 85
# of PINS	= 40

TIMING ANALYSIS DIAGRAM



Max Operating Speed: ~ 0.34 GHz

Fastest Clock Speed: 3.0 ns

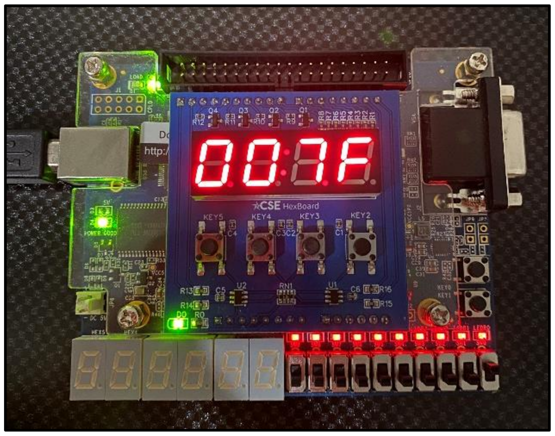
TEST RESULTS

Usigned Tests

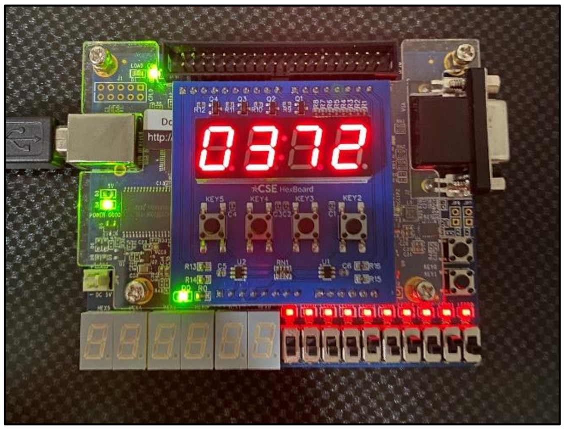
Test	M x Q (Binary)	M x Q (Hex)	Product (Hex)	Clock Cycles
a)	0111 1111 x 0000 0001	7F x 01	007F	22
b)	0001 0101 x 0010 1010	15 x 2A	0372	24
c)	0111 1111 x 1111 1111	7F x FF	7E81	29
d)	1010 1010 x 0011 0011	AA x 33	21DE	25
e)	1010 1010 x 1111 1110	AA x FE	A8AC	28
f)	1101 1101 x 1100 1101	DD x CD	B0F9	26

PHOTOS OF TEST RESULTS

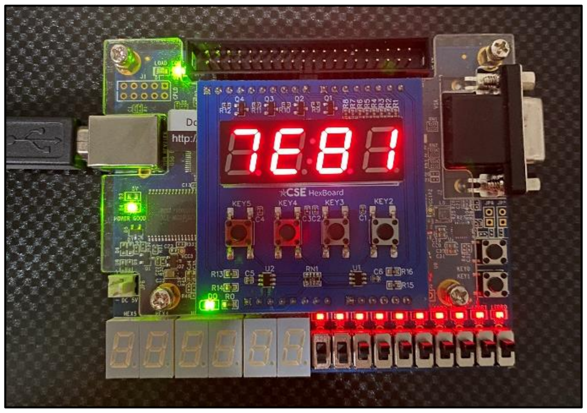
a)



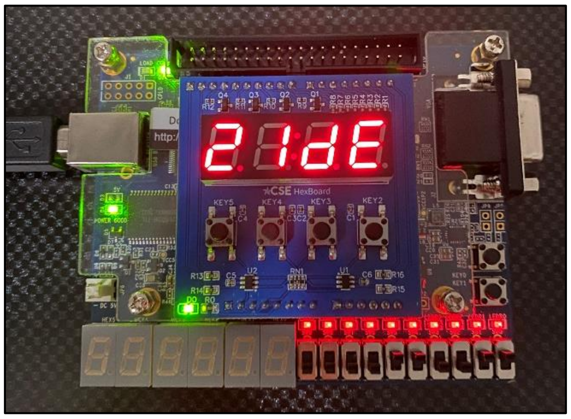
b)



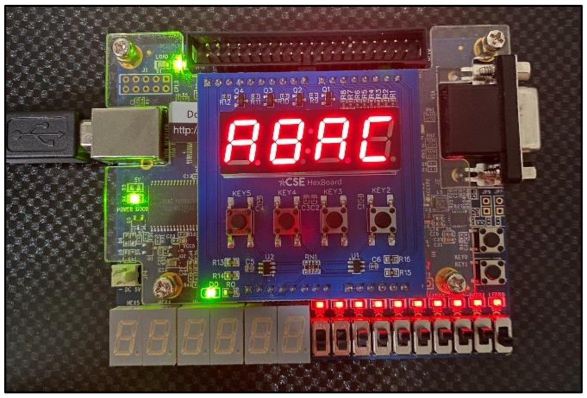
c)



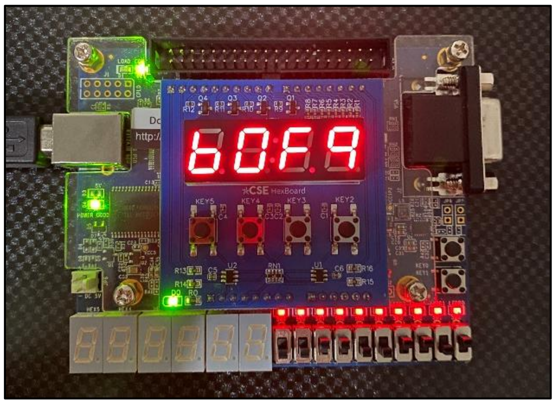
d)



e)



f)



SIGNED VERISON

DESIGN REQUIREMENTS

Your assignment is to design a registered eight-by-eight signed multiplier by enhancing the four- by-four unsigned shift-and-add multiplier shown below and discussed in class.

REQUIREMENTS:

1. Registered 8 x 8 multiplier
2. Signed numbers (use 2's complement for negative numbers)
3. SystemVerilog implementation
4. Design verification (simulation)
5. DE10-Lite realization

DESIGN VERIFICATION (simulation)

1. Simulate your designs to verify their correctness. Use the following values for M and Q in your simulations.
   (a) 01111111 x 00000001
   (b) 00010101 x 00101010
   (b) 01111111 x 11111111
   (c) 10101010 x 00110011
   (d) 10101010 x 11111110
   (e) 11011101 x 11001101
2. Include a screen shot of your simulation waveforms in your report.
3. Record the simulation results in a table for your report (use hexadecimal)
4. How many clock cycles does it take for each case to complete?

RTL ANALYSIS

1. Generate RTL diagrams using the Quartus Prime Netlist Viewer for each version.
2. Record the compilation summary for your report. How many ALM, registers, and pins does your design require?

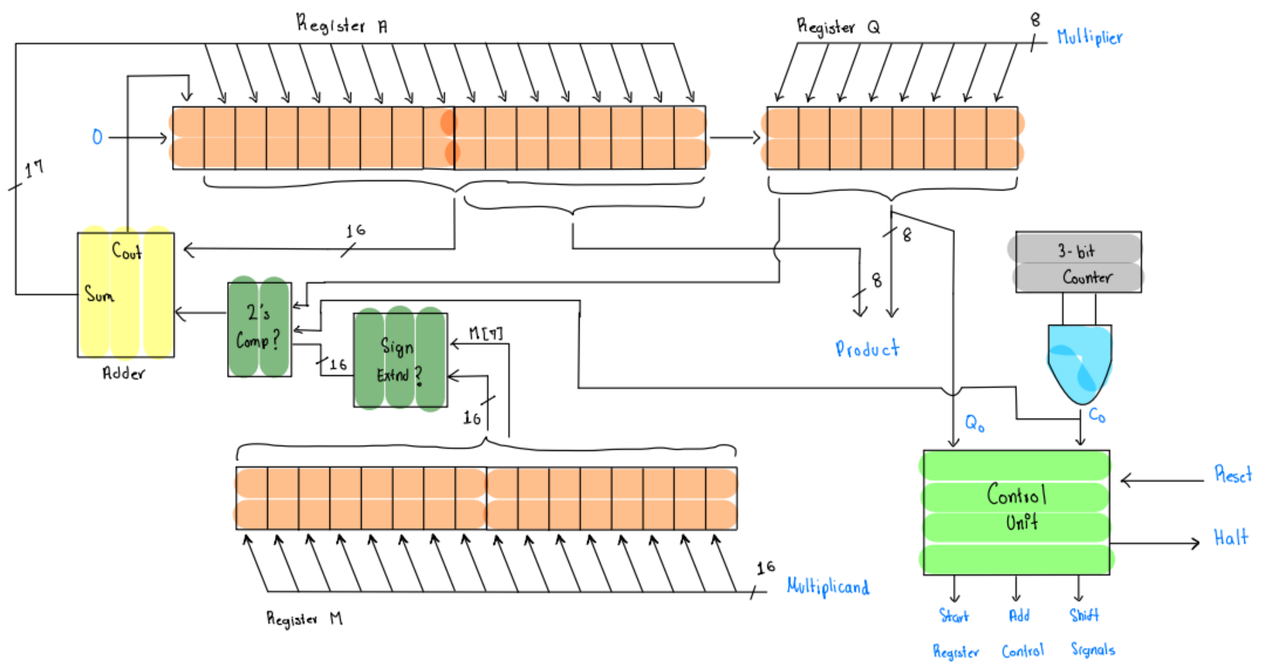
TIMING ANALYSIS

1. Run a timing analysis on your signed multiplier and determine its maximum operating speed in GHz.
2. Capture a screen shot of your timing analysis waveform showing the fastest clock speed your design will accommodate.

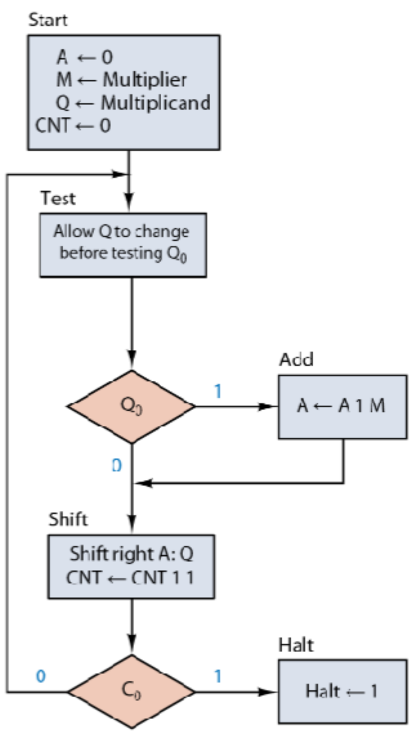
DE10-Lite IMPLEMENTATION (signed version only)

1. Implement your signed multiplier on the DE10-Lite using the following inputs/outputs. Use
   pin assignments of your choice
   Inputs M, Q, InM, InQ,
   Outputs Mout, Qout, Pout. Display in hexadecimal on the HEX displays.
2. Include a table of your pin assignments in your report.
3. Program the DE10-Lite with your design.

DATA PATH DIAGRAM

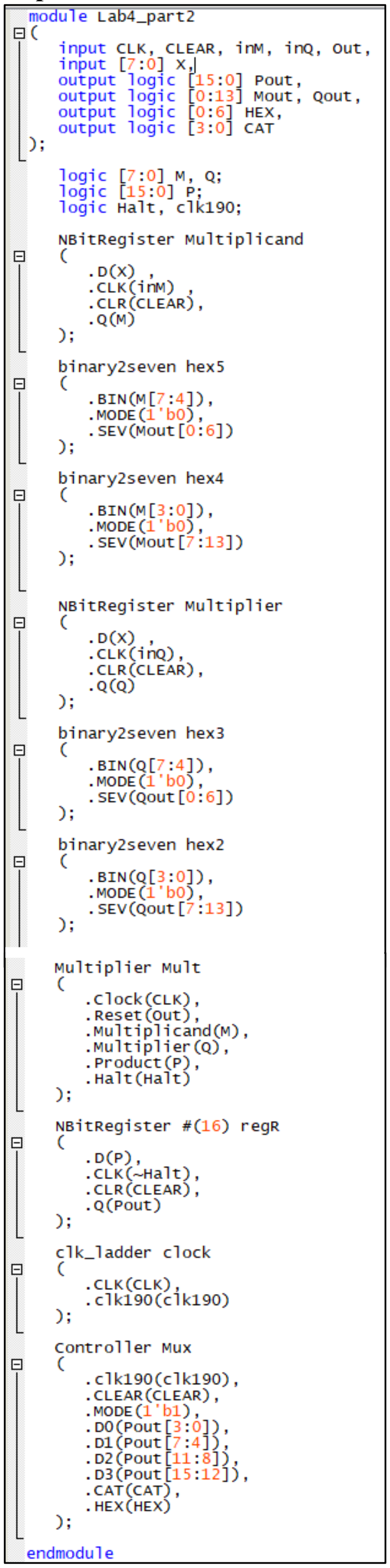


CONTROL PATH DIAGRAM



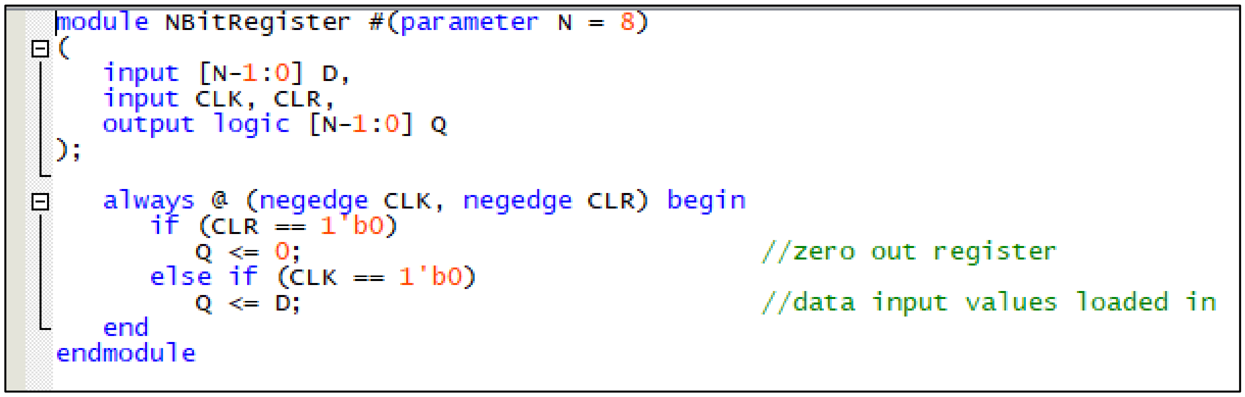
SYSTEM-VERILOG CODE

Top Module



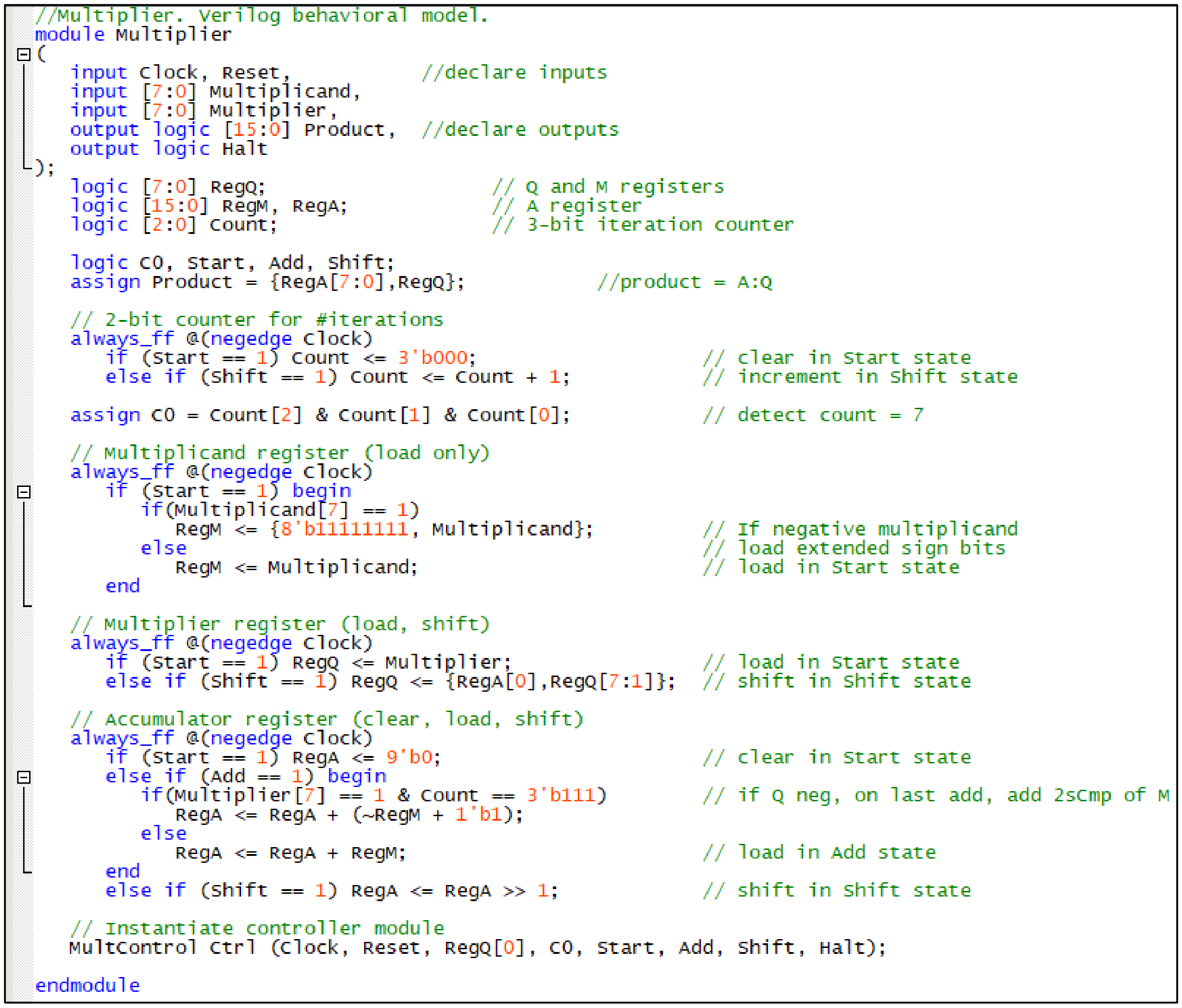
```
module Lab4_part2
(
    input CLK, CLEAR, inM, inQ, Out,
    input [7:0] X,
    output logic [15:0] Pout,
    output logic [0:13] Mout, Qout,
    output logic [0:6] HEX,
    output logic [3:0] CAT
);

    logic [7:0] M, Q;
    logic [15:0] P;
    logic Halt, clk190;

    NBitRegister Multiplicand
    (
        .D(X) ,
        .CLK(inM) ,
        .CLR(CLEAR),
        .Q(M)
    );

    binary2seven hex5
    (
        .BIN(M[7:4]),
        .MODE(1'b0),
        .SEV(Mout[0:6])
    );

    binary2seven hex4
    (
        .BIN(M[3:0]),
        .MODE(1'b0),
        .SEV(Mout[7:13])
    );


    NBitRegister Multiplier
    (
        .D(X) ,
        .CLK(inQ),
        .CLR(CLEAR),
        .Q(Q)
    );

    binary2seven hex3
    (
        .BIN(Q[7:4]),
        .MODE(1'b0),
        .SEV(Qout[0:6])
    );

    binary2seven hex2
    (
        .BIN(Q[3:0]),
        .MODE(1'b0),
        .SEV(Qout[7:13])
    );

    Multiplier Mult
    (
        .Clock(CLK),
        .Reset(Out),
        .Multiplicand(M),
        .Multiplier(Q),
        .Product(P),
        .Halt(Halt)
    );

    NBitRegister #(16) regR
    (
        .D(P),
        .CLK(~Halt),
        .CLR(CLEAR),
        .Q(Pout)
    );

    clk_ladder clock
    (
        .CLK(CLK),
        .clk190(clk190)
    );

    Controller Mux
    (
        .clk190(clk190),
        .CLEAR(CLEAR),
        .MODE(1'b1),
        .D0(Pout[3:0]),
        .D1(Pout[7:4]),
        .D2(Pout[11:8]),
        .D3(Pout[15:12]),
        .CAT(CAT),
        .HEX(HEX)
    );
endmodule
```

Register Module



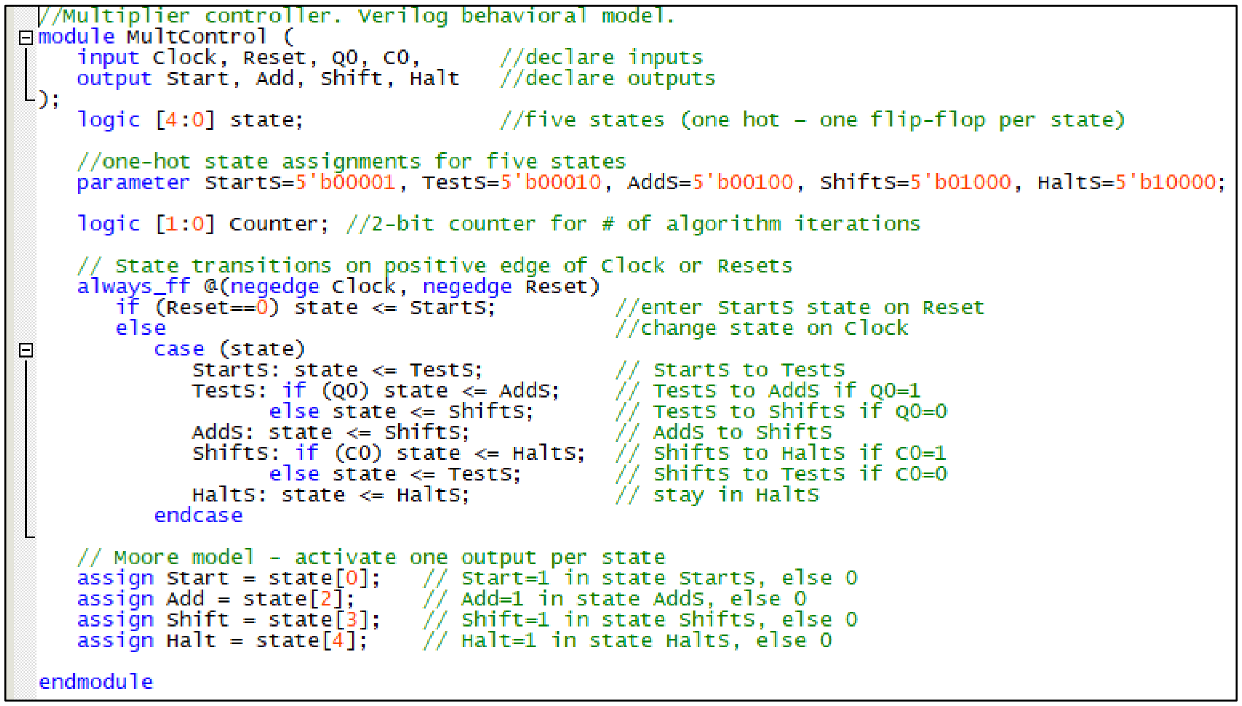
```
module NBitRegister #(parameter N = 8)
(
    input [N-1:0] D,
    input CLK, CLR,
    output logic [N-1:0] Q
);

    always @ (negedge CLK, negedge CLR) begin
        if (CLR == 1'b0)
            Q <= 0;                         //zero out register
        else if (CLK == 1'b0)
            Q <= D;                         //data input values loaded in
    end
endmodule
```

Multiplier Module



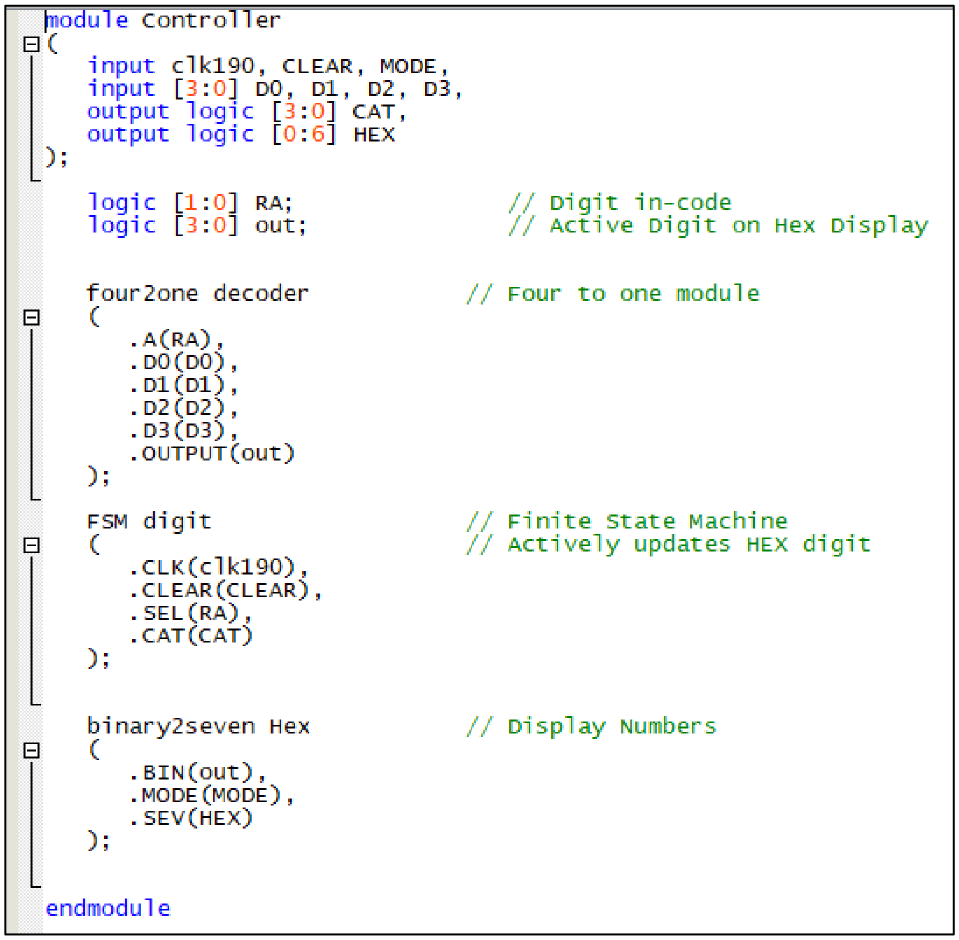
```
//Multiplier. Verilog behavioral model.
module Multiplier
(
    input Clock, Reset,             //declare inputs
    input [7:0] Multiplicand,
    input [7:0] Multiplier,
    output logic [15:0] Product,    //declare outputs
    output logic Halt
);
    logic [7:0] RegQ;                       // Q and M registers
    logic [15:0] RegM, RegA;                // A register
    logic [2:0] Count;                      // 3-bit iteration counter

    logic C0, Start, Add, Shift;
    assign Product = {RegA[7:0],RegQ};              //product = A:Q

    // 2-bit counter for #iterations
    always_ff @(negedge Clock)
       if (Start == 1) Count <= 3'b000;                 // clear in Start state
       else if (Shift == 1) Count <= Count + 1;         // increment in Shift state

    assign C0 = Count[2] & Count[1] & Count[0];         // detect count = 7

    // Multiplicand register (load only)
    always_ff @(negedge Clock)
       if (Start == 1) begin
          if(Multiplicand[7] == 1)
             RegM <= {8'b11111111, Multiplicand};       // If negative multiplicand
          else                                          // load extended sign bits
             RegM <= Multiplicand;                      // load in Start state
       end

    // Multiplier register (load, shift)
    always_ff @(negedge Clock)
       if (Start == 1) RegQ <= Multiplier;              // load in Start state
       else if (Shift == 1) RegQ <= {RegA[0],RegQ[7:1]};  // shift in Shift state

    // Accumulator register (clear, load, shift)
    always_ff @(negedge Clock)
       if (Start == 1) RegA <= 9'b0;                    // clear in Start state
       else if (Add == 1) begin
          if(Multiplier[7] == 1 & Count == 3'b111)      // if Q neg, on last add, add 2sCmp of M
             RegA <= RegA + (~RegM + 1'b1);
          else
             RegA <= RegA + RegM;                       // load in Add state
       end
       else if (Shift == 1) RegA <= RegA >> 1;          // shift in Shift state

    // Instantiate controller module
    MultControl Ctrl (Clock, Reset, RegQ[0], C0, Start, Add, Shift, Halt);

endmodule
```

Multiplier Control Module



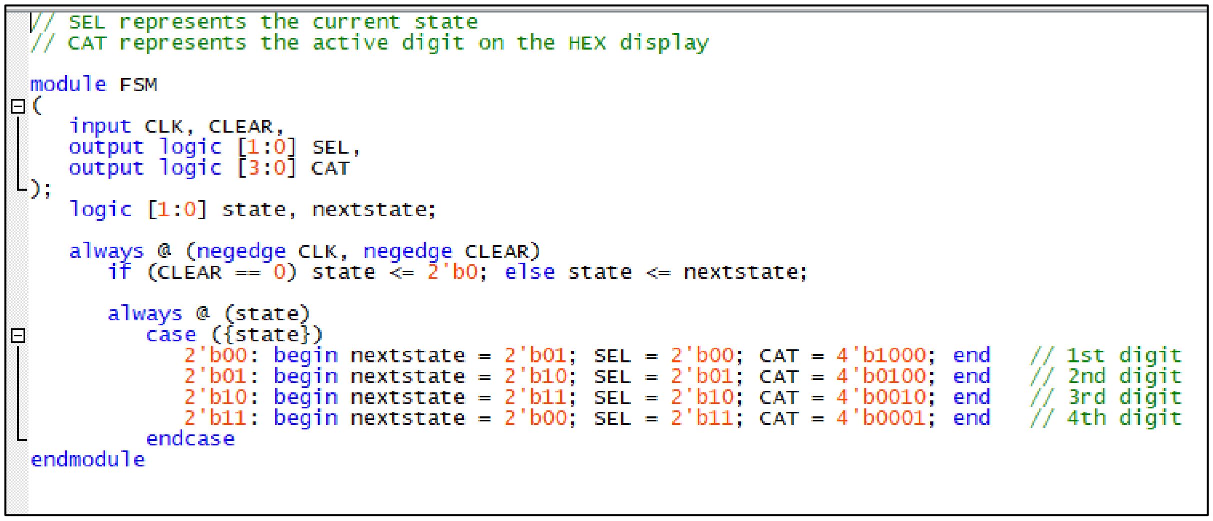
```
//Multiplier controller. Verilog behavioral model.
module MultControl (
    input Clock, Reset, Q0, C0,      //declare inputs
    output Start, Add, Shift, Halt   //declare outputs
);
    logic [4:0] state;               //five states (one hot - one flip-flop per state)

    //one-hot state assignments for five states
    parameter StartS=5'b00001, TestS=5'b00010, AddS=5'b00100, ShiftS=5'b01000, HaltS=5'b10000;

    logic [1:0] Counter; //2-bit counter for # of algorithm iterations

    // State transitions on positive edge of Clock or Resets
    always_ff @(negedge Clock, negedge Reset)
       if (Reset==0) state <= StartS;          //enter StartS state on Reset
       else                                    //change state on Clock
          case (state)
             StartS: state <= TestS;           // StartS to TestS
             TestS: if (Q0) state <= AddS;     // TestS to AddS if Q0=1
                   else state <= ShiftS;       // TestS to ShiftS if Q0=0
             AddS: state <= ShiftS;            // AddS to ShiftS
             ShiftS: if (C0) state <= HaltS;   // ShiftS to HaltS if C0=1
                   else state <= TestS;        // ShiftS to TestS if C0=0
             HaltS: state <= HaltS;            // stay in HaltS
          endcase

    // Moore model - activate one output per state
    assign Start = state[0];    // Start=1 in state StartS, else 0
    assign Add = state[2];      // Add=1 in state AddS, else 0
    assign Shift = state[3];    // Shift=1 in state ShiftS, else 0
    assign Halt = state[4];     // Halt=1 in state HaltS, else 0

endmodule
```

MUX Controller Module



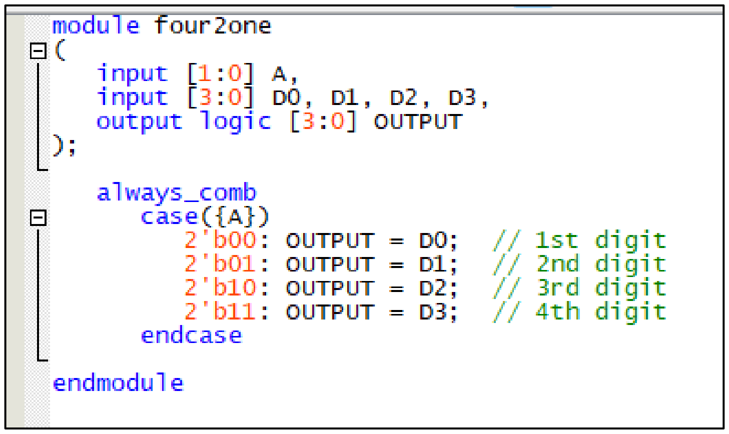
```
module Controller
(
    input clk190, CLEAR, MODE,
    input [3:0] D0, D1, D2, D3,
    output logic [3:0] CAT,
    output logic [0:6] HEX
);

    logic [1:0] RA;              // Digit in-code
    logic [3:0] out;             // Active Digit on Hex Display


    four2one decoder           // Four to one module
    (
        .A(RA),
        .D0(D0),
        .D1(D1),
        .D2(D2),
        .D3(D3),
        .OUTPUT(out)
    );

    FSM digit                  // Finite State Machine
    (                          // Actively updates HEX digit
        .CLK(clk190),
        .CLEAR(CLEAR),
        .SEL(RA),
        .CAT(CAT)
    );


    binary2seven Hex           // Display Numbers
    (
        .BIN(out),
        .MODE(MODE),
        .SEV(HEX)
    );

endmodule
```

Finite State Machine Module



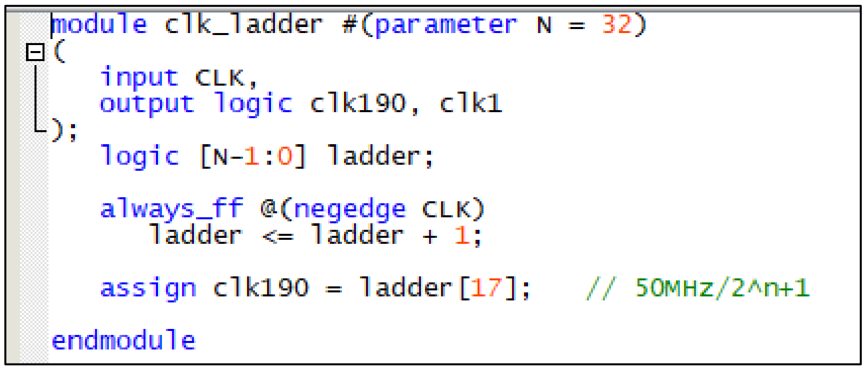
```
// SEL represents the current state
// CAT represents the active digit on the HEX display

module FSM
(
   input CLK, CLEAR,
   output logic [1:0] SEL,
   output logic [3:0] CAT
);
   logic [1:0] state, nextstate;

   always @ (negedge CLK, negedge CLEAR)
      if (CLEAR == 0) state <= 2'b0; else state <= nextstate;

      always @ (state)
         case ({state})
            2'b00: begin nextstate = 2'b01; SEL = 2'b00; CAT = 4'b1000; end   // 1st digit
            2'b01: begin nextstate = 2'b10; SEL = 2'b01; CAT = 4'b0100; end   // 2nd digit
            2'b10: begin nextstate = 2'b11; SEL = 2'b10; CAT = 4'b0010; end   // 3rd digit
            2'b11: begin nextstate = 2'b00; SEL = 2'b11; CAT = 4'b0001; end   // 4th digit
         endcase
endmodule
```

MUX/ Four to One Decoder Module



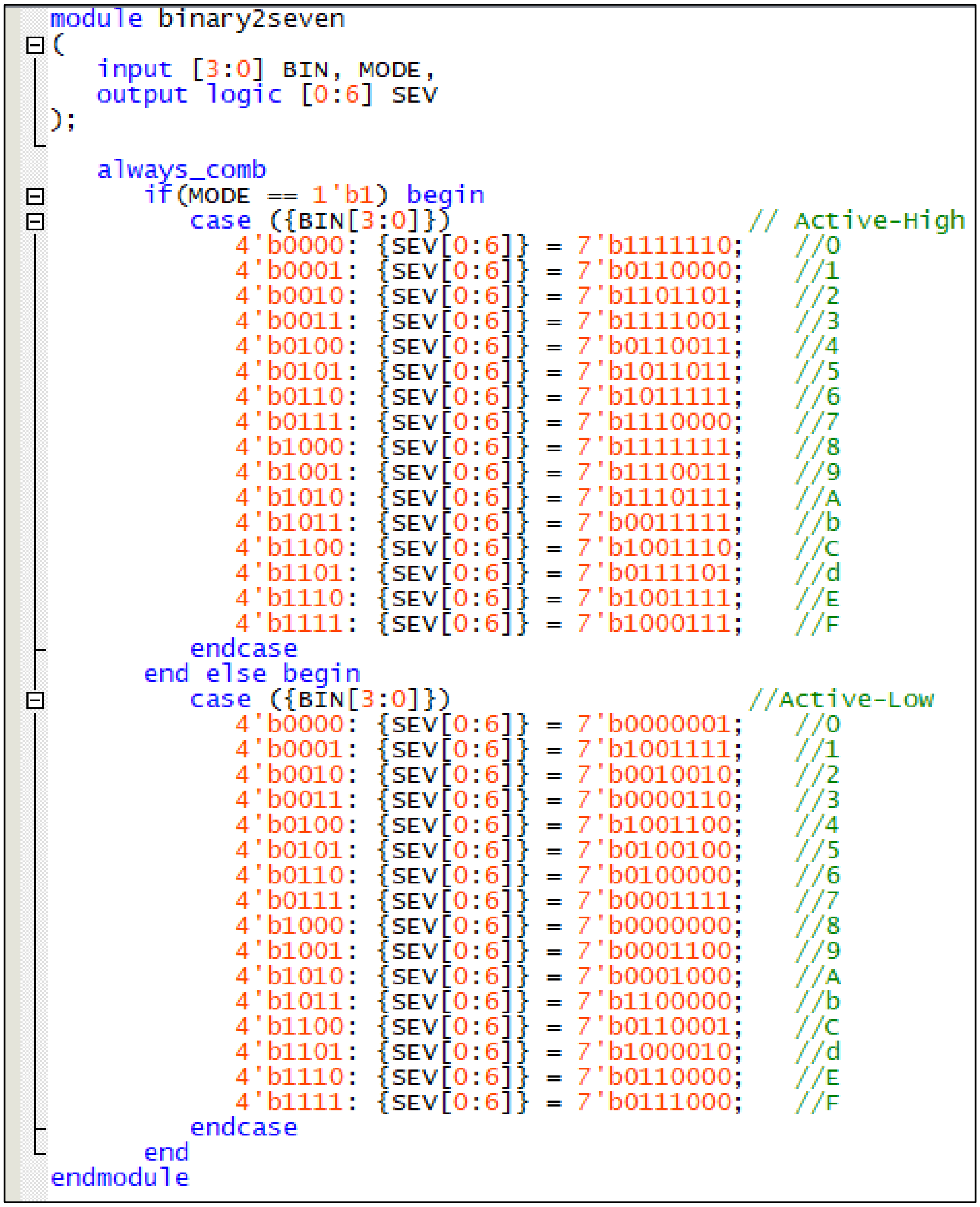
```
module four2one
(
   input [1:0] A,
   input [3:0] D0, D1, D2, D3,
   output logic [3:0] OUTPUT
);

   always_comb
      case({A})
         2'b00: OUTPUT = D0;   // 1st digit
         2'b01: OUTPUT = D1;   // 2nd digit
         2'b10: OUTPUT = D2;   // 3rd digit
         2'b11: OUTPUT = D3;   // 4th digit
      endcase

endmodule
```

Clock Ladder Module



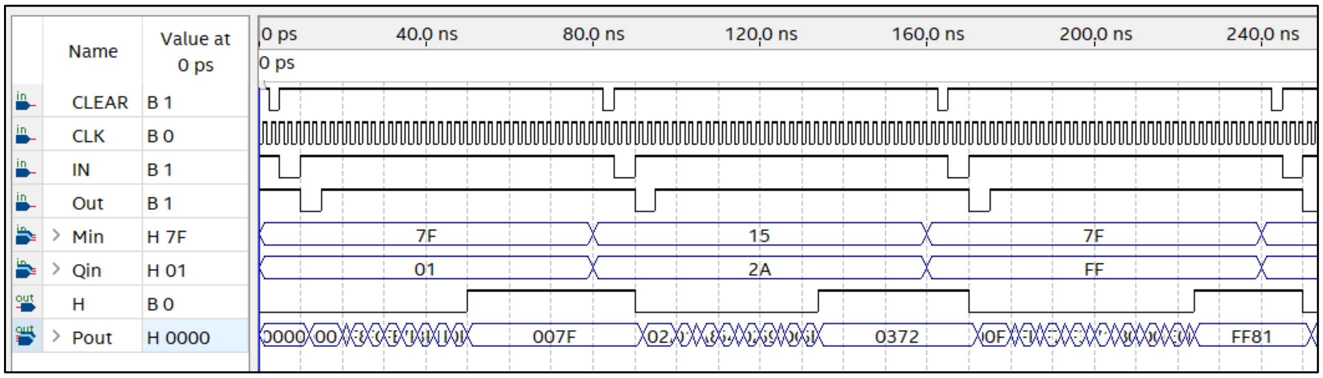
```
module clk_ladder #(parameter N = 32)
(
   input CLK,
   output logic clk190, clk1
);
   logic [N-1:0] ladder;

   always_ff @(negedge CLK)
      ladder <= ladder + 1;

   assign clk190 = ladder[17];    // 50MHz/2^n+1

endmodule
```

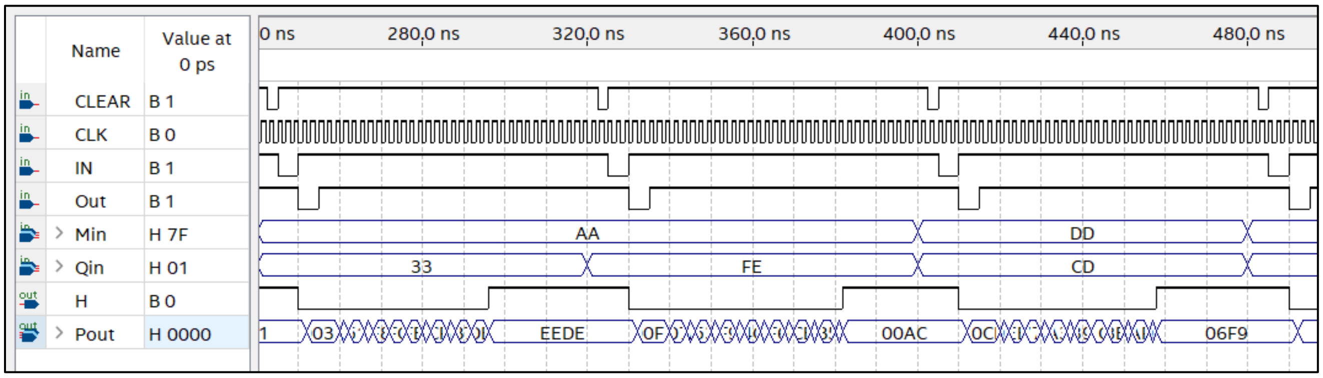
Binary to Seven-Seg Display Decoder Module



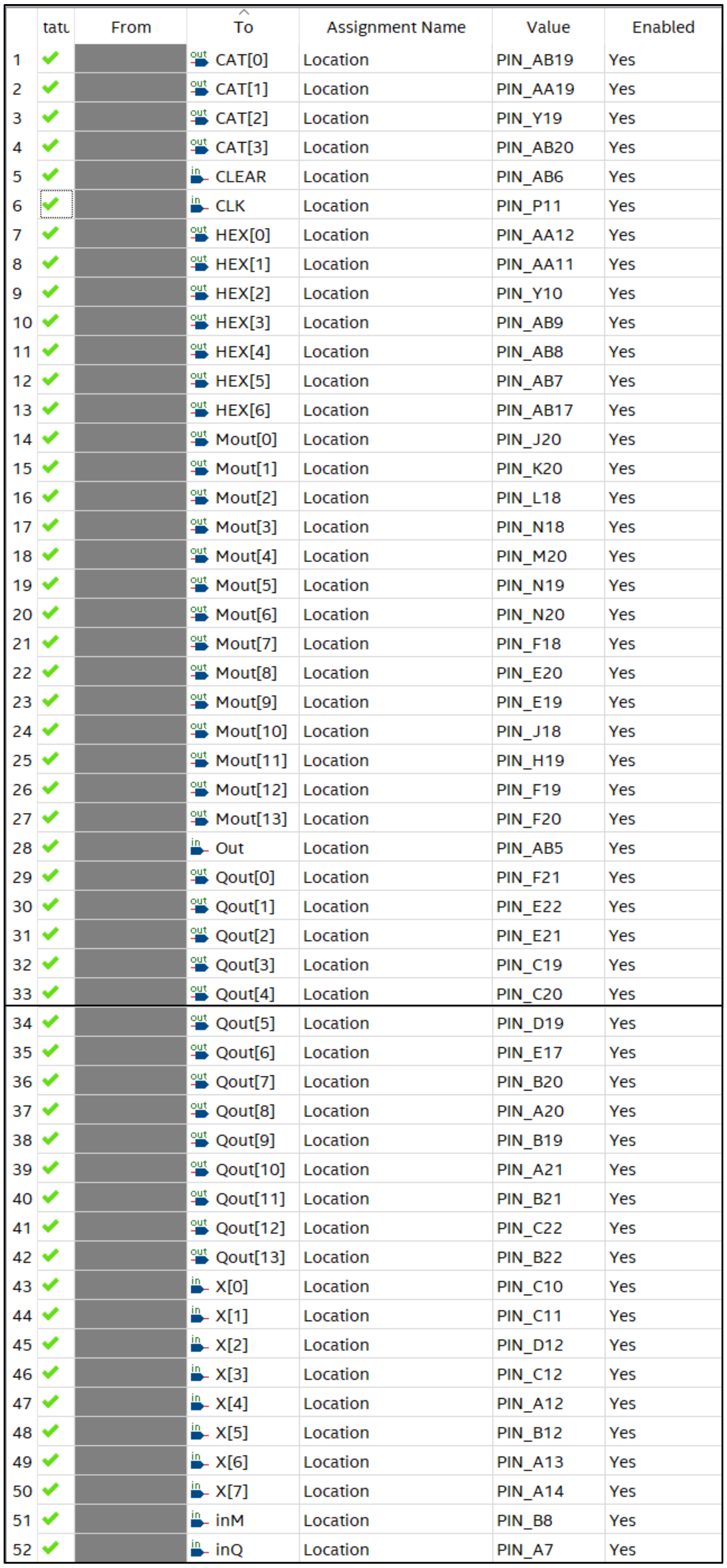
```
module binary2seven
(
    input [3:0] BIN, MODE,
    output logic [0:6] SEV
);

    always_comb
        if(MODE == 1'b1) begin
            case ({BIN[3:0]})                              // Active-High
                4'b0000: {SEV[0:6]} = 7'b1111110;    //0
                4'b0001: {SEV[0:6]} = 7'b0110000;    //1
                4'b0010: {SEV[0:6]} = 7'b1101101;    //2
                4'b0011: {SEV[0:6]} = 7'b1111001;    //3
                4'b0100: {SEV[0:6]} = 7'b0110011;    //4
                4'b0101: {SEV[0:6]} = 7'b1011011;    //5
                4'b0110: {SEV[0:6]} = 7'b1011111;    //6
                4'b0111: {SEV[0:6]} = 7'b1110000;    //7
                4'b1000: {SEV[0:6]} = 7'b1111111;    //8
                4'b1001: {SEV[0:6]} = 7'b1110011;    //9
                4'b1010: {SEV[0:6]} = 7'b1110111;    //A
                4'b1011: {SEV[0:6]} = 7'b0011111;    //b
                4'b1100: {SEV[0:6]} = 7'b1001110;    //C
                4'b1101: {SEV[0:6]} = 7'b0111101;    //d
                4'b1110: {SEV[0:6]} = 7'b1001111;    //E
                4'b1111: {SEV[0:6]} = 7'b1000111;    //F
            endcase
        end else begin
            case ({BIN[3:0]})                              //Active-Low
                4'b0000: {SEV[0:6]} = 7'b0000001;    //0
                4'b0001: {SEV[0:6]} = 7'b1001111;    //1
                4'b0010: {SEV[0:6]} = 7'b0010010;    //2
                4'b0011: {SEV[0:6]} = 7'b0000110;    //3
                4'b0100: {SEV[0:6]} = 7'b1001100;    //4
                4'b0101: {SEV[0:6]} = 7'b0100100;    //5
                4'b0110: {SEV[0:6]} = 7'b0100000;    //6
                4'b0111: {SEV[0:6]} = 7'b0001111;    //7
                4'b1000: {SEV[0:6]} = 7'b0000000;    //8
                4'b1001: {SEV[0:6]} = 7'b0001100;    //9
                4'b1010: {SEV[0:6]} = 7'b0001000;    //A
                4'b1011: {SEV[0:6]} = 7'b1100000;    //b
                4'b1100: {SEV[0:6]} = 7'b0110001;    //C
                4'b1101: {SEV[0:6]} = 7'b1000010;    //d
                4'b1110: {SEV[0:6]} = 7'b0110000;    //E
                4'b1111: {SEV[0:6]} = 7'b0111000;    //F
            endcase
        end
endmodule
```

SIMULATION RESULTS WAVEFORM





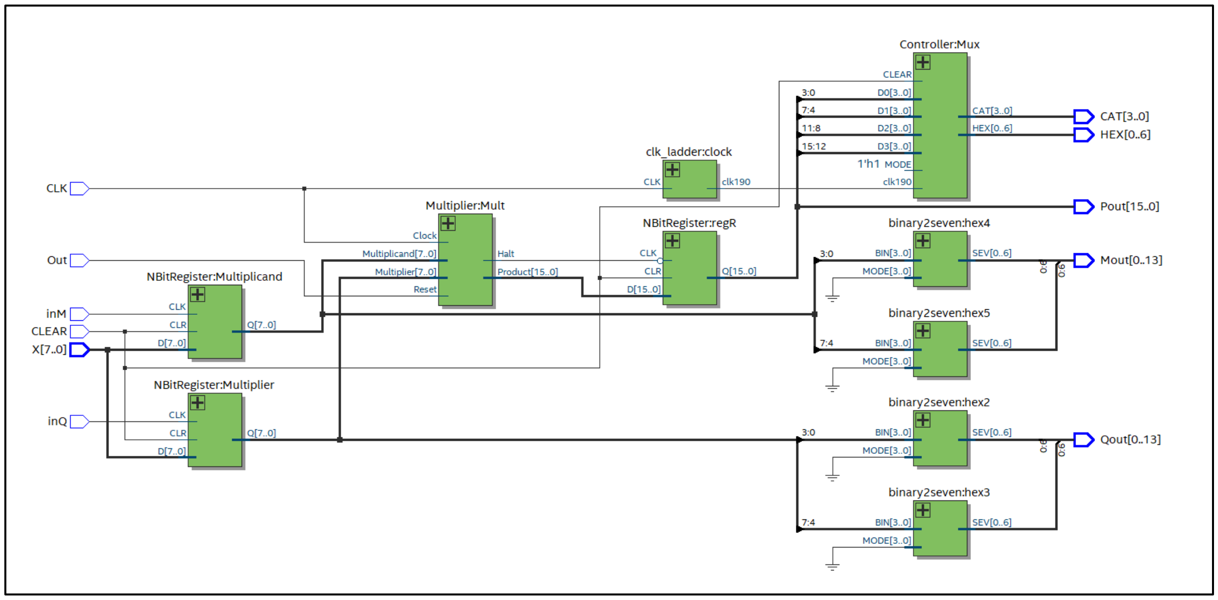
PIN ASSIGMENTS



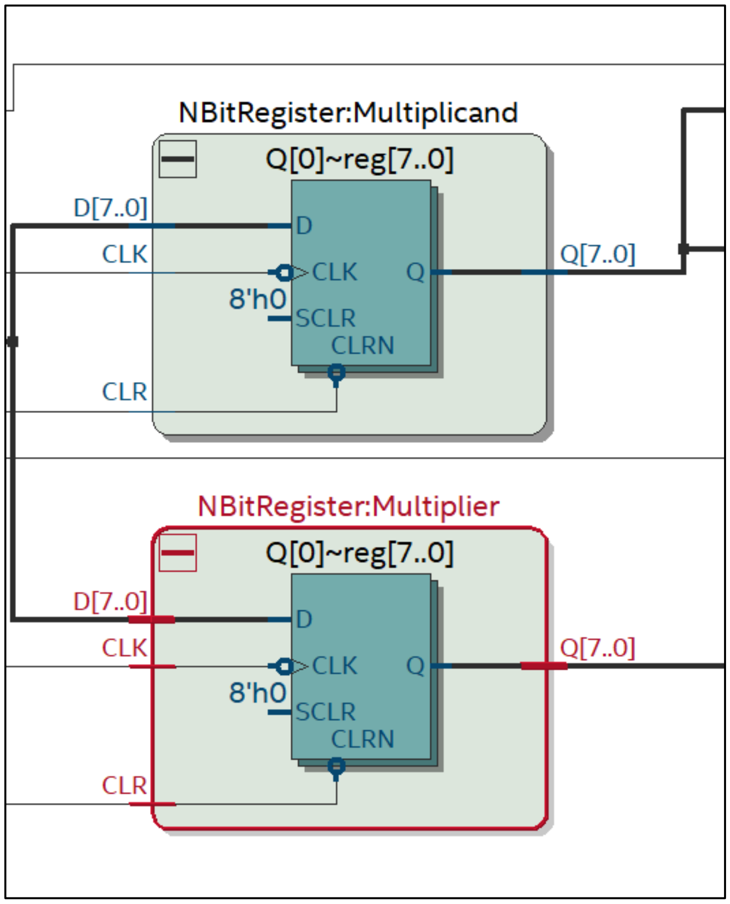
	tatu	From	To	Assignment Name	Value	Enabled
1	✓		CAT[0]	Location	PIN_AB19	Yes
2	✓		CAT[1]	Location	PIN_AA19	Yes
3	✓		CAT[2]	Location	PIN_Y19	Yes
4	✓		CAT[3]	Location	PIN_AB20	Yes
5	✓		CLEAR	Location	PIN_AB6	Yes
6	✓		CLK	Location	PIN_P11	Yes
7	✓		HEX[0]	Location	PIN_AA12	Yes
8	✓		HEX[1]	Location	PIN_AA11	Yes
9	✓		HEX[2]	Location	PIN_Y10	Yes
10	✓		HEX[3]	Location	PIN_AB9	Yes
11	✓		HEX[4]	Location	PIN_AB8	Yes
12	✓		HEX[5]	Location	PIN_AB7	Yes
13	✓		HEX[6]	Location	PIN_AB17	Yes
14	✓		Mout[0]	Location	PIN_J20	Yes
15	✓		Mout[1]	Location	PIN_K20	Yes
16	✓		Mout[2]	Location	PIN_L18	Yes
17	✓		Mout[3]	Location	PIN_N18	Yes
18	✓		Mout[4]	Location	PIN_M20	Yes
19	✓		Mout[5]	Location	PIN_N19	Yes
20	✓		Mout[6]	Location	PIN_N20	Yes
21	✓		Mout[7]	Location	PIN_F18	Yes
22	✓		Mout[8]	Location	PIN_E20	Yes
23	✓		Mout[9]	Location	PIN_E19	Yes
24	✓		Mout[10]	Location	PIN_J18	Yes
25	✓		Mout[11]	Location	PIN_H19	Yes
26	✓		Mout[12]	Location	PIN_F19	Yes
27	✓		Mout[13]	Location	PIN_F20	Yes
28	✓		Out	Location	PIN_AB5	Yes
29	✓		Qout[0]	Location	PIN_F21	Yes
30	✓		Qout[1]	Location	PIN_E22	Yes
31	✓		Qout[2]	Location	PIN_E21	Yes
32	✓		Qout[3]	Location	PIN_C19	Yes
33	✓		Qout[4]	Location	PIN_C20	Yes
34	✓		Qout[5]	Location	PIN_D19	Yes
35	✓		Qout[6]	Location	PIN_E17	Yes
36	✓		Qout[7]	Location	PIN_B20	Yes
37	✓		Qout[8]	Location	PIN_A20	Yes
38	✓		Qout[9]	Location	PIN_B19	Yes
39	✓		Qout[10]	Location	PIN_A21	Yes
40	✓		Qout[11]	Location	PIN_B21	Yes
41	✓		Qout[12]	Location	PIN_C22	Yes
42	✓		Qout[13]	Location	PIN_B22	Yes
43	✓		X[0]	Location	PIN_C10	Yes
44	✓		X[1]	Location	PIN_C11	Yes
45	✓		X[2]	Location	PIN_D12	Yes
46	✓		X[3]	Location	PIN_C12	Yes
47	✓		X[4]	Location	PIN_A12	Yes
48	✓		X[5]	Location	PIN_B12	Yes
49	✓		X[6]	Location	PIN_A13	Yes
50	✓		X[7]	Location	PIN_A14	Yes
51	✓		inM	Location	PIN_B8	Yes
52	✓		inQ	Location	PIN_A7	Yes

RTL DIAGRAMS

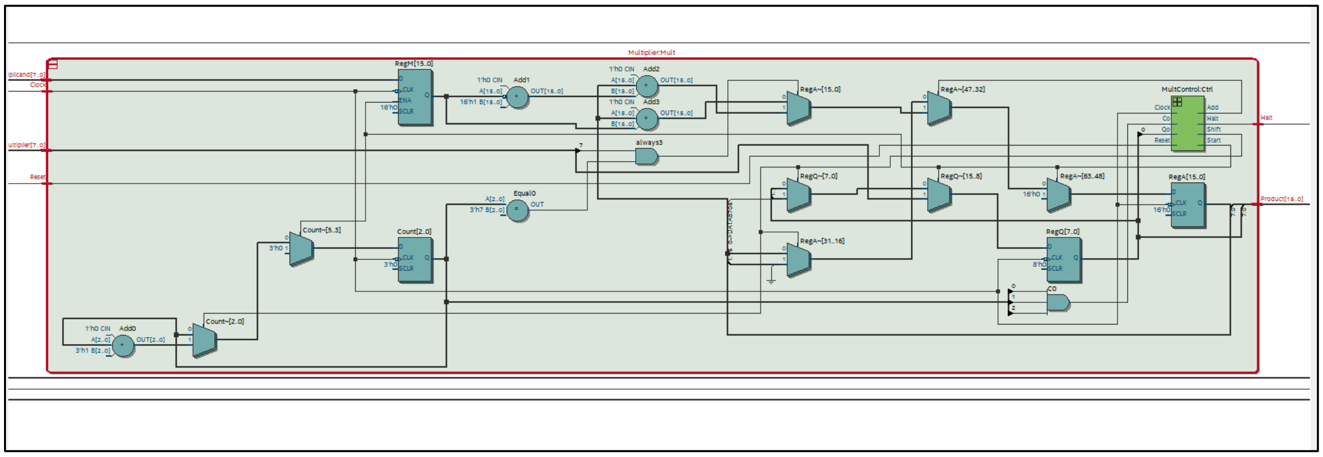
Top Module



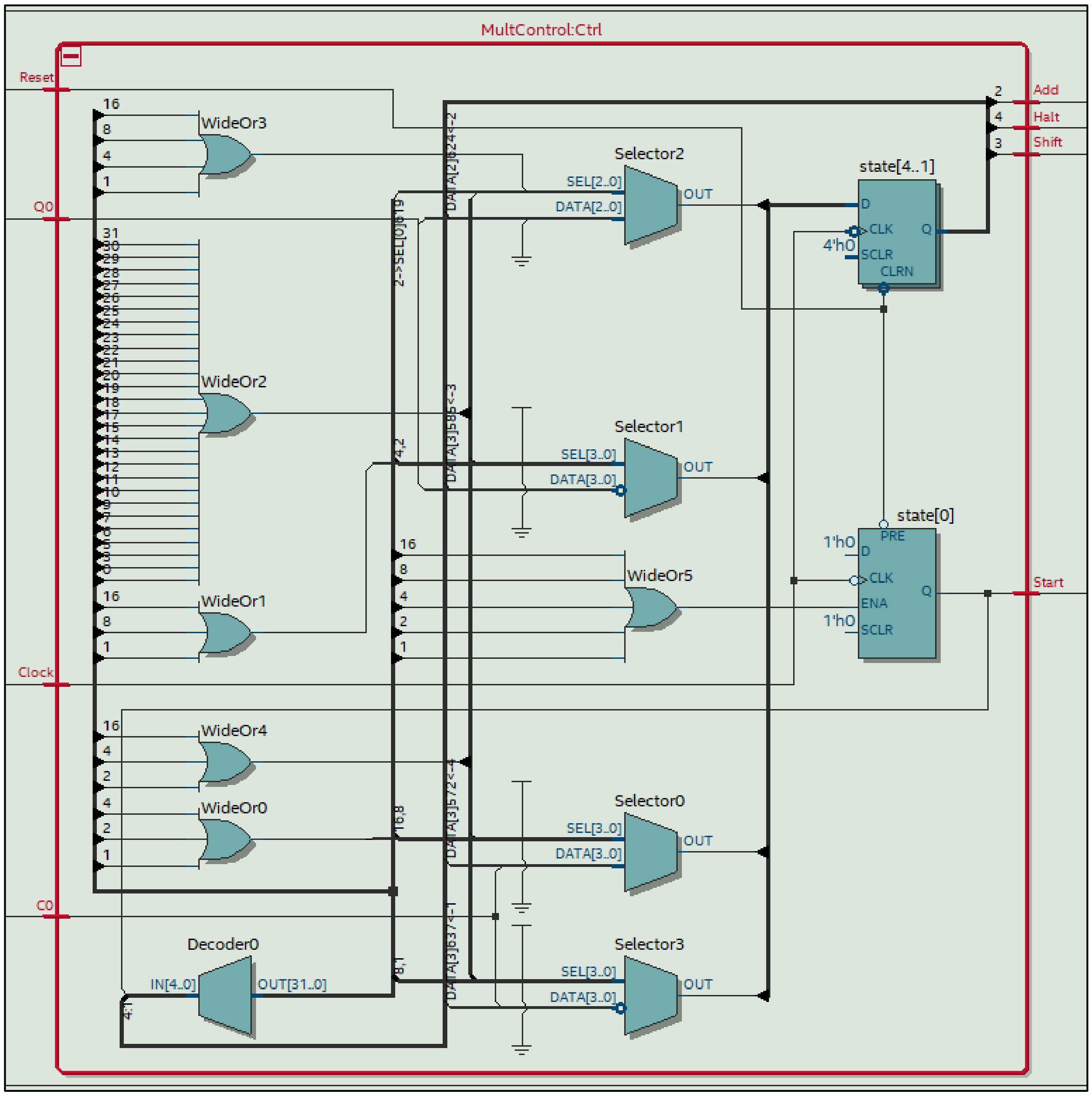
Registers A & B



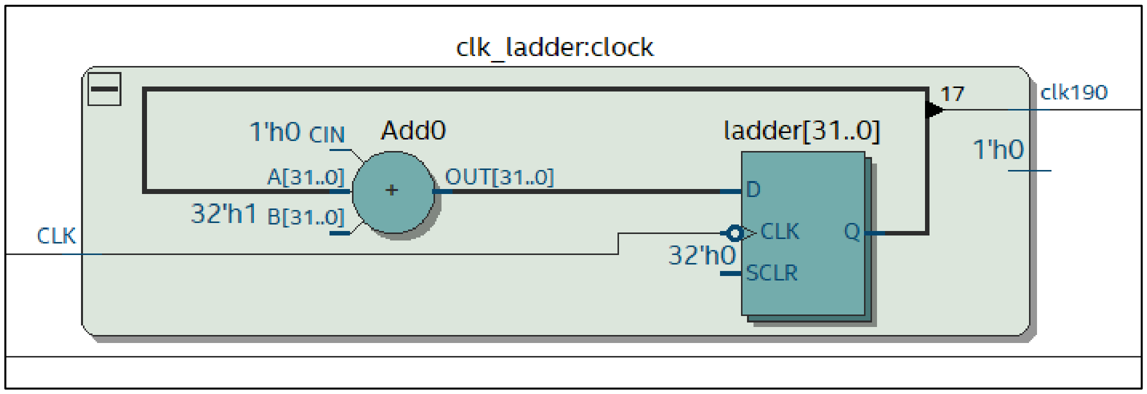
Multiplier



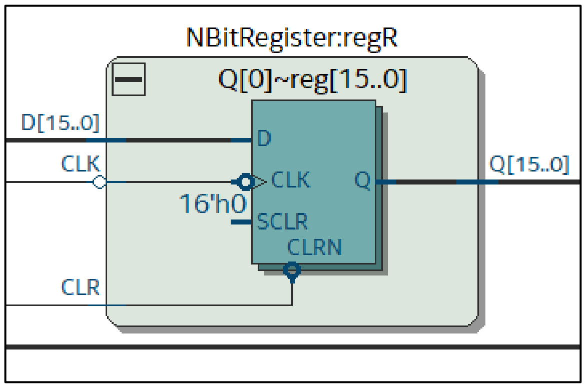
Multiplier Control



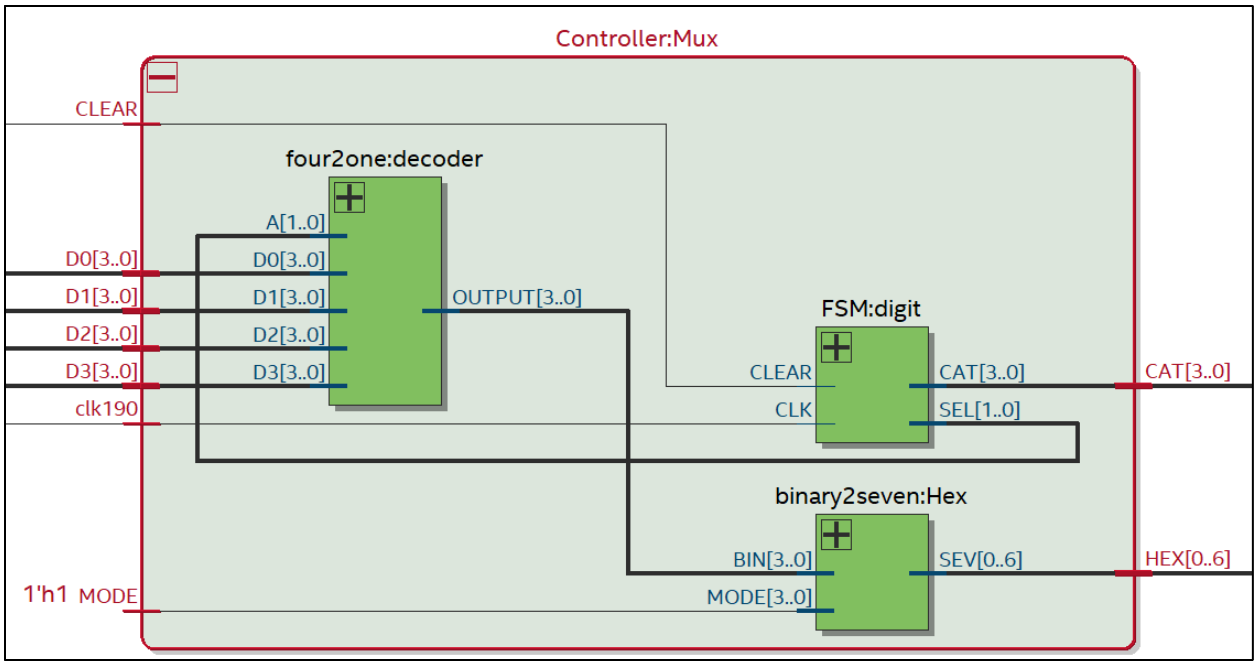
Clock Ladder



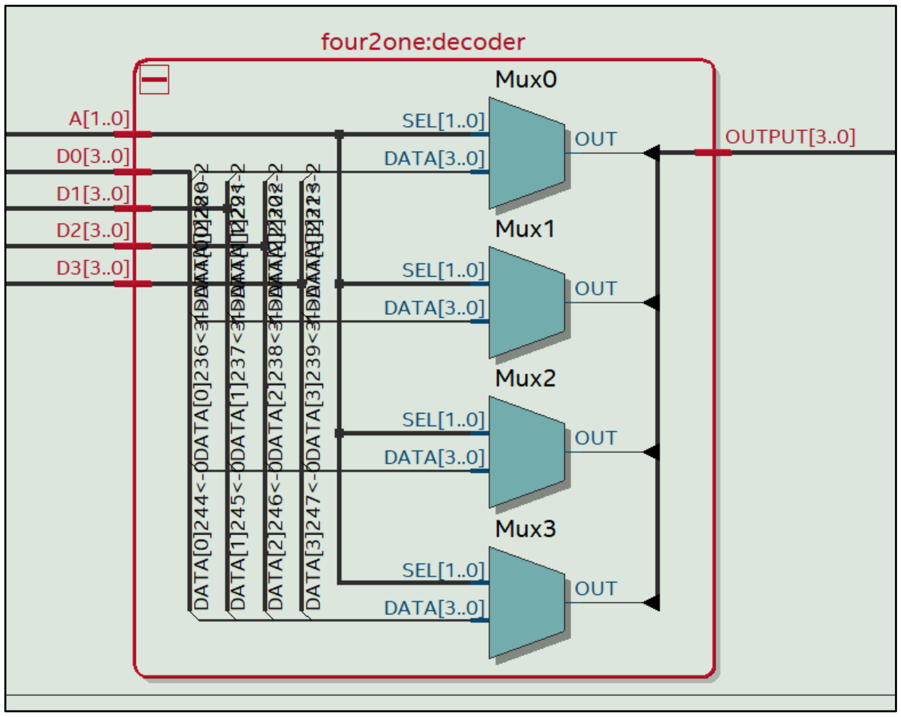
Product Register



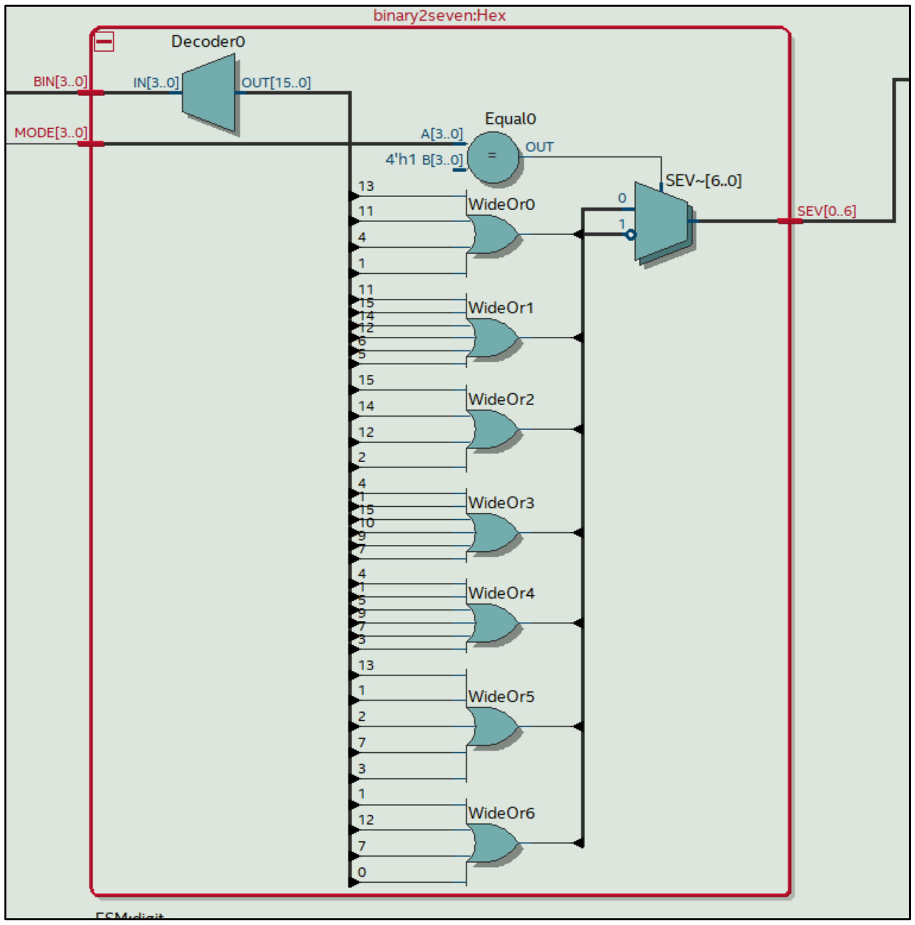
MUX/Controller



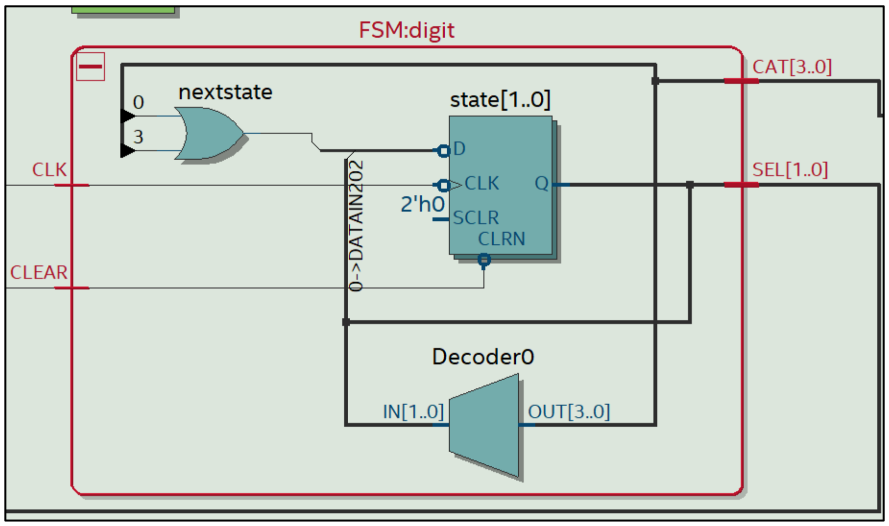
Four to One Decoder



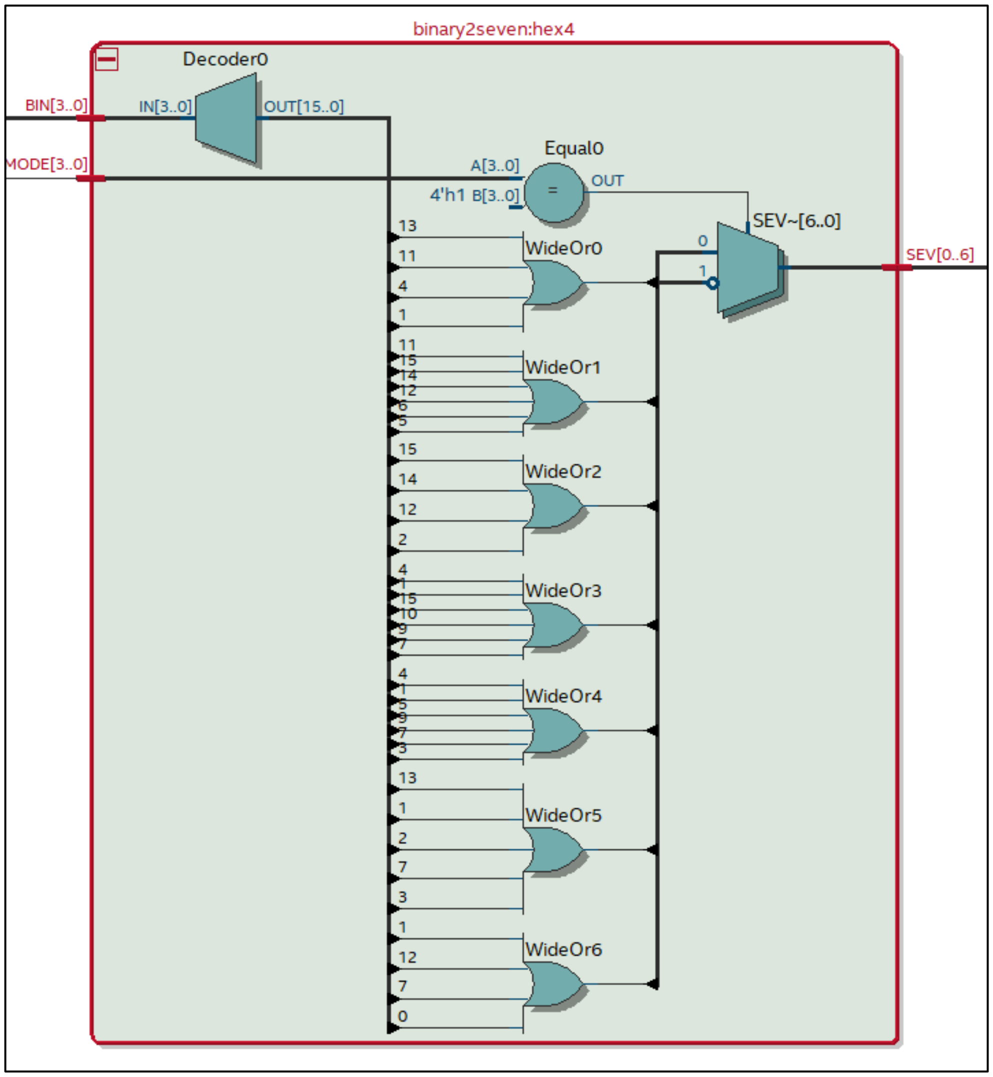
Binary To Seven Segment Display Decoder (Product)



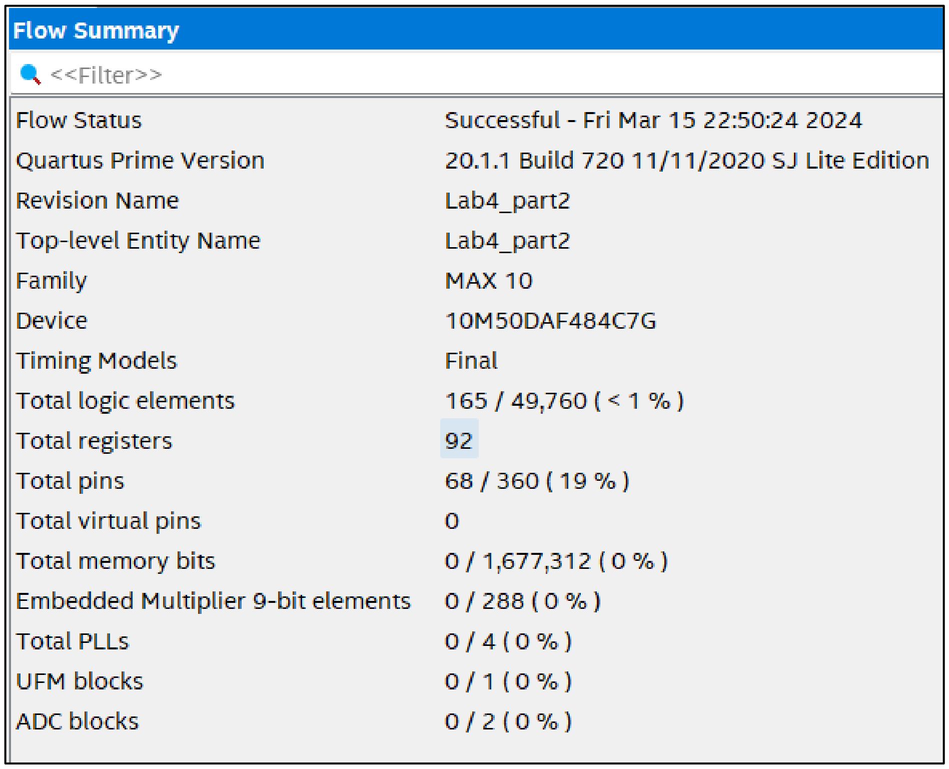
Finite State Machine



Binary To Seven Segment Display Decoder (Used 4 times to properly display M_{out} & Q_{out})



Four to One Decoder



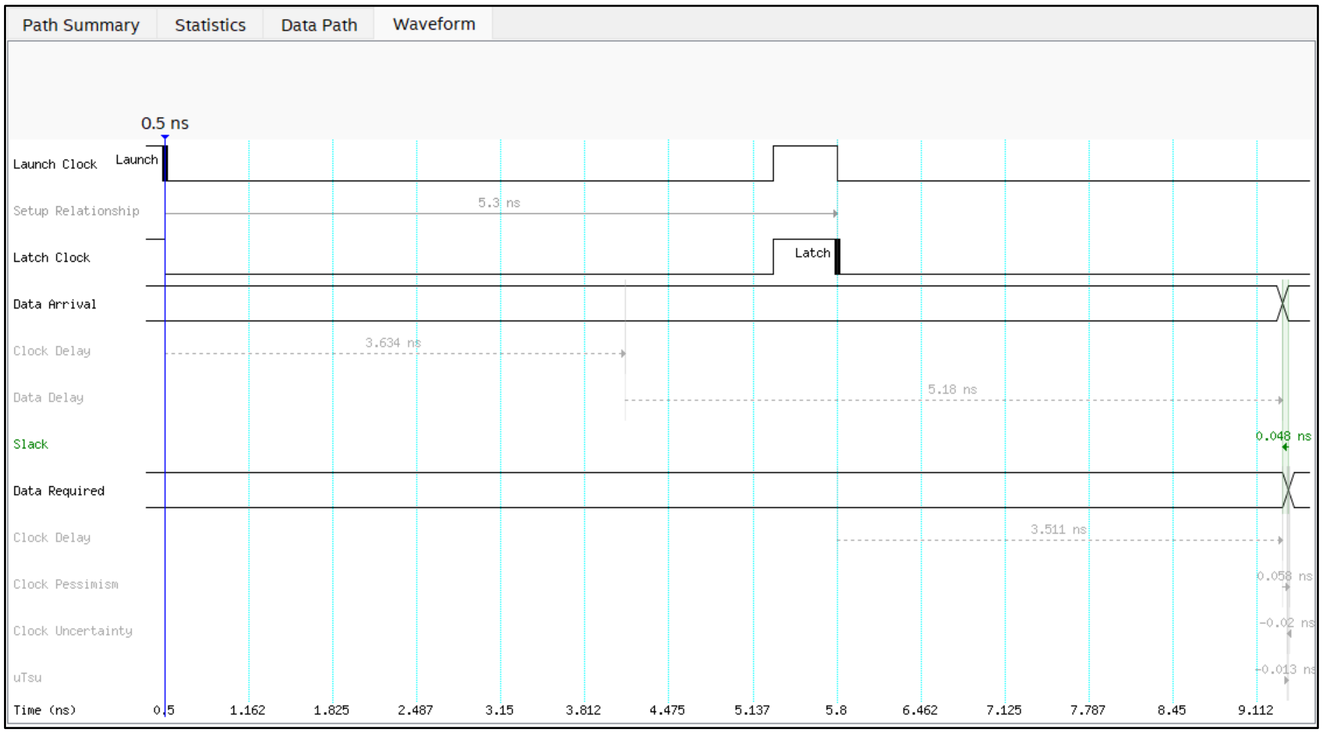
Flow Summary

<<Filter>>

Flow Status	Successful - Fri Mar 15 22:50:24 2024
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	Lab4_part2
Top-level Entity Name	Lab4_part2
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	165 / 49,760 (< 1 %)
Total registers	92
Total pins	68 / 360 (19 %)
Total virtual pins	0
Total memory bits	0 / 1,677,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

# of ALMs = 165
# of REGISTERS = 92
# of PINS = 68

TIMING ANALYSIS DIAGRAMS



Max Operating Speed: ~ 0.19 GHz

Fastest Clock Speed: 5.3 ns

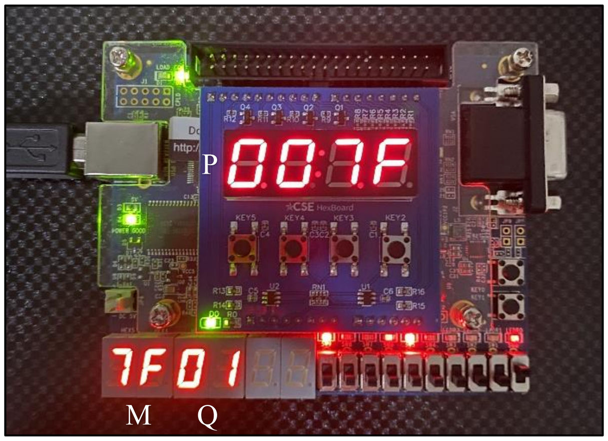
TEST RESULTS

Signed Tests

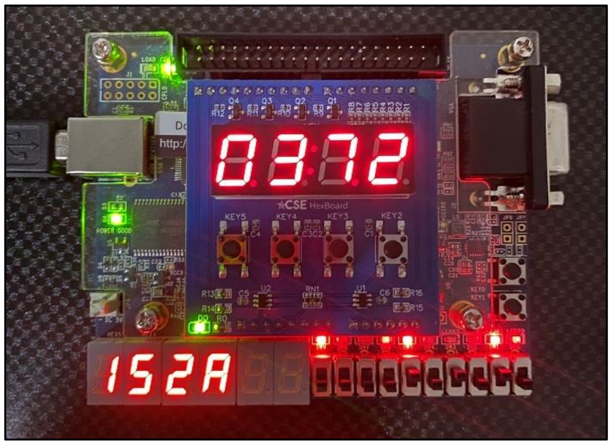
Test	M x Q (Binary)	M x Q (Hex)	Product (Hex)	Clock Cycles
a)	0111 1111 x 0000 0001	7F x 01	007F	20
b)	0001 0101 x 0010 1010	15 x 2A	0372	22
c)	0111 1111 x 1111 1111	7F x FF	FF81	27
d)	1010 1010 x 0011 0011	AA x 33	EEDE	23
e)	1010 1010 x 1111 1110	AA x FE	00AC	26
f)	1101 1101 x 1100 1101	DD x CD	06F9	24

PHOTOS OF TEST RESULTS

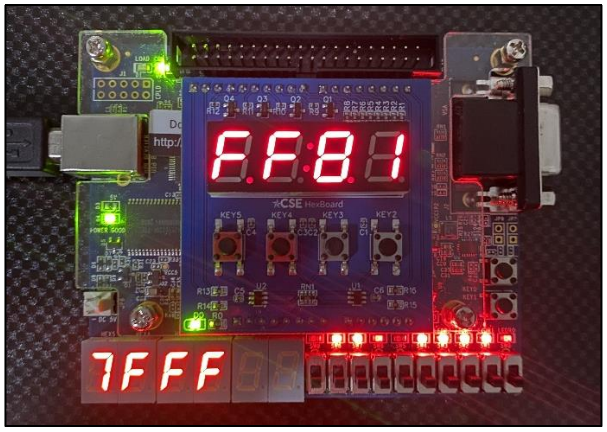
a)



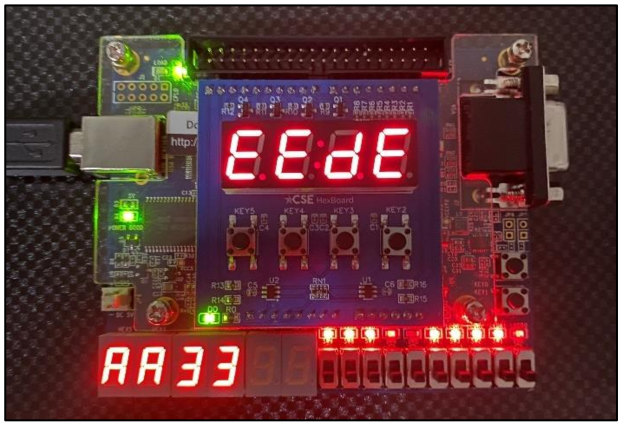
b)



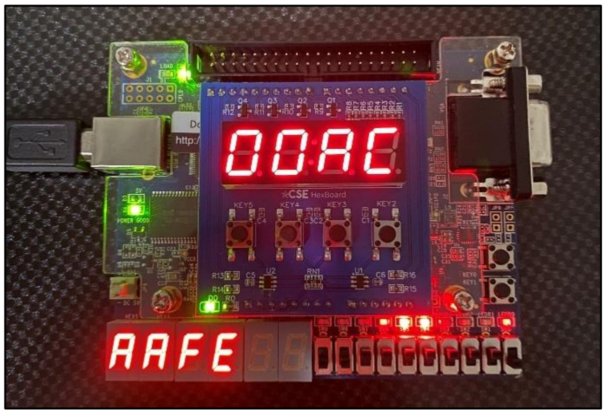
c)



d)



e)



f)