EEE 248 | Logic Design

Final Lab Report

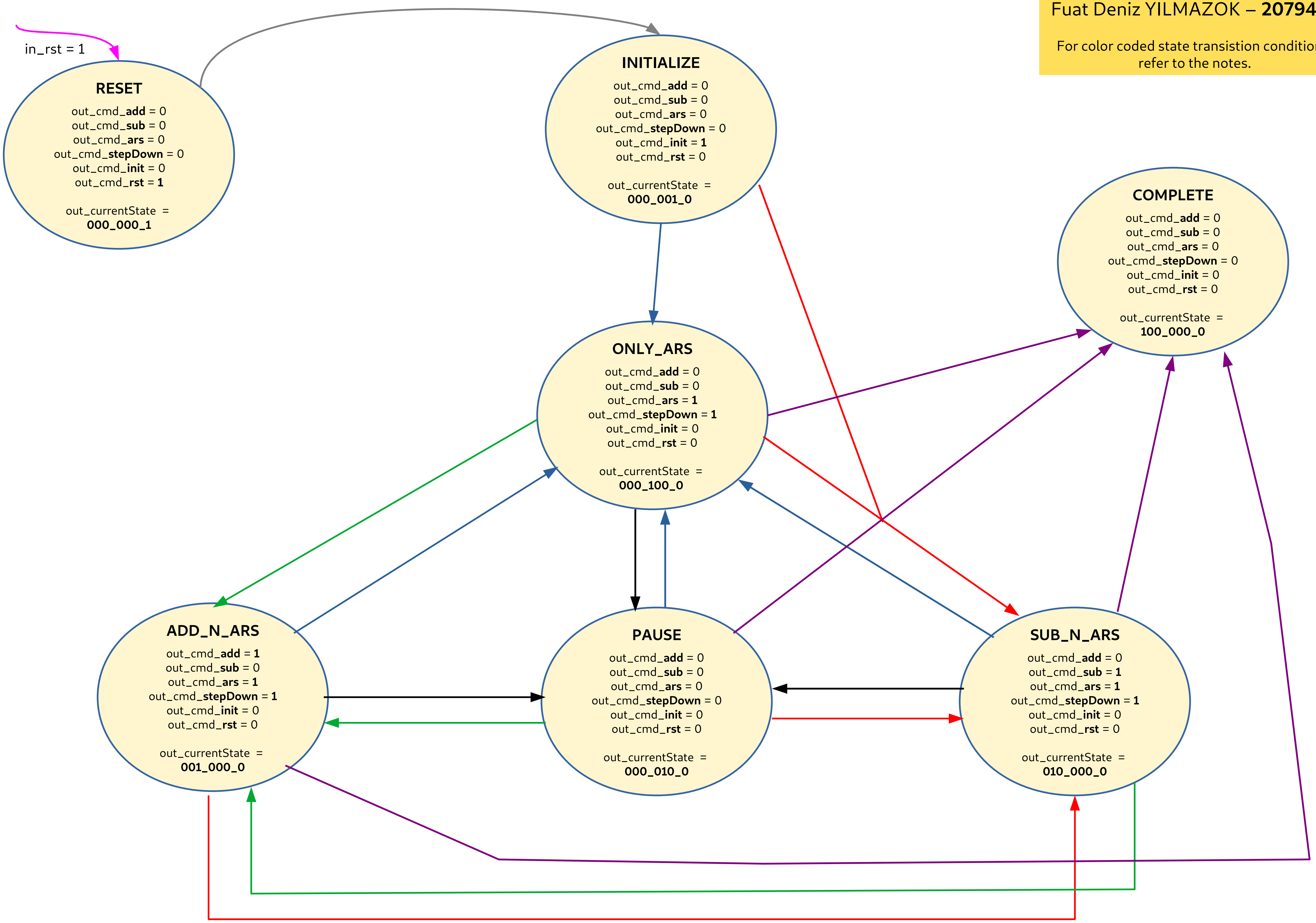
Fuat Deniz YILMAZOK - **2079465**

Part 1 | Control Unit

Documents in this part, already included in the part 1 report.
They're added here to refer from **part 2**.

Fuat Deniz YILMAZOK – **2079465**

For color coded state transistion conditions, refer to the notes.



Notes for State Diagram

Inputs to the circuit:

```
in_start
in_Q_0
in_Q_M1
in_currentStep // Output of drop-down counter.
in_rst
```

Color Coded Conditionals:

Pink conditional showed on the diagram to save space.

condition_gray = in_rst == 0

condition_black = in_start == 0 && in_currentStep != 0

condition_green = in_start == 1 && in_currentStep != 0 && Q_0Q_{-1} == 01

condition_blue = in_start == 1 && in_currentStep != 0 && (Q_0Q_{-1} == 00 || Q_0Q_{-1} == 11)

condition_red = in_start == 1 && in_currentStep != 0 && Q_0Q_{-1} == 10

condition_purple = in_currentStep == 0

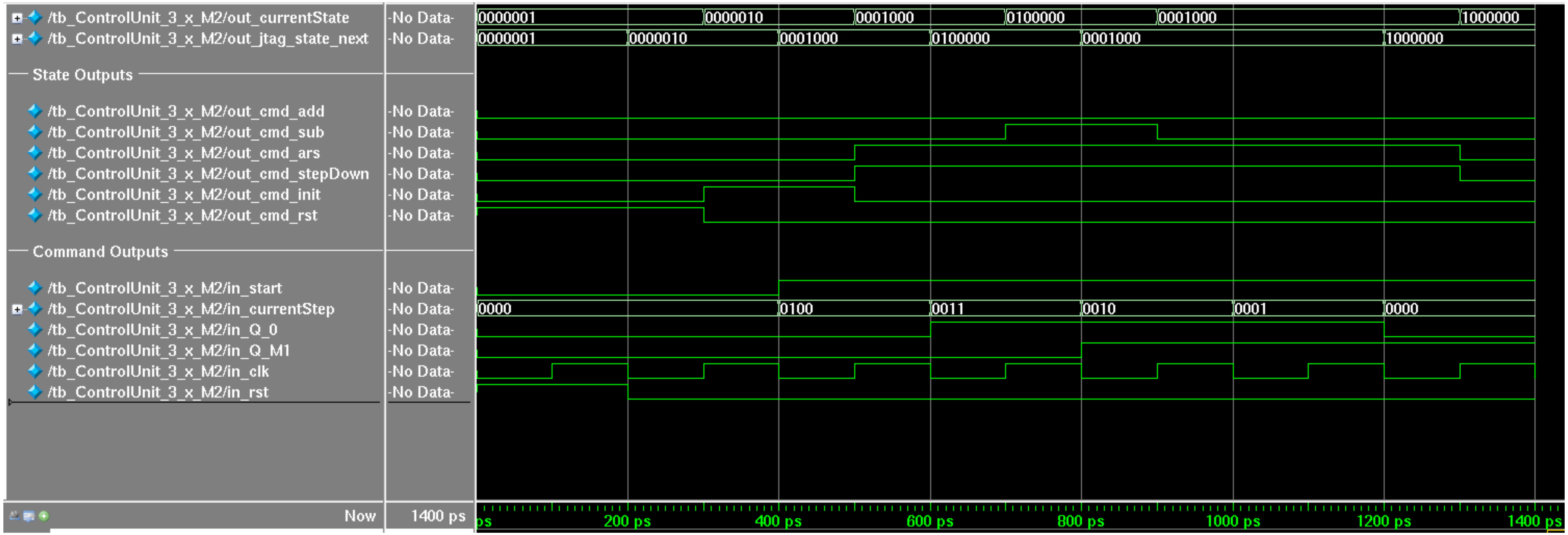
```
// Verilog code for moore fsm.
// For color coded information, refer to state diagram.
// Fuat Deniz YILMAZOK - 2079465
module ControlUnit(
                   out_cmd_add,
                   out_cmd_sub,
                   out_cmd_ars,
                   out_cmd_stepDown,
                   out_cmd_init,
                   out_cmd_rst,

                   out_currentState,

                   out_jtag_state_next,

                   in_start,
                   in_Q_0,
                   in_Q_M1,
                   in_currentStep,
                   in_clk,
                   in_rst
                  );

output reg out_cmd_add,
           out_cmd_sub,
           out_cmd_ars,
           out_cmd_stepDown,
           out_cmd_init,
           out_cmd_rst;

// Does not use any parameter,
//    because there is no need expose number of state constants.
output [6:0] out_currentState, out_jtag_state_next;

input in_start,
      in_Q_0,
      in_Q_M1,
      in_clk,
      in_rst;

parameter CONST_DATA_SIZE = 4; // Exposed parameter to the outside.

input[CONST_DATA_SIZE - 1 : 0] in_currentStep;


// Local constants.
// I did not declare them as parameter, because they're not needed in the outside world
//    and they make symbol for block diagram unnecessarily complicated.
localparam CONST_STATE_COUNT = 7,

           CONST_STATE_RESET = 7'b000_000_1,
           CONST_STATE_INITIALIZE = 7'b000_001_0,
           CONST_STATE_PAUSE = 7'b000_010_0,
           CONST_STATE_ONLY_ARS = 7'b000_100_0,
           CONST_STATE_ADD_N_ARS = 7'b001_000_0,
           CONST_STATE_SUB_N_ARS = 7'b010_000_0,
           CONST_STATE_COMPLETE = 7'b100_000_0;

reg [CONST_STATE_COUNT - 1 : 0] state_current, state_next;

// Dataflow outputs.
assign out_currentState = state_current;
assign out_jtag_state_next = state_next;


```

```
// State register.
always@(posedge in_clk, posedge in_rst) begin
   if(in_rst)
      state_current <= CONST_STATE_RESET;
   else
      state_current <= state_next;
end


wire [1:0] qValues= {in_Q_0, in_Q_M1};

// Implicit assign statements.
wire condition_gray = in_rst == 0,
     condition_black = in_start == 0 && in_currentStep != 0,
     condition_green = in_start == 1 && in_currentStep != 0 && qValues == 2'b01,
     condition_blue = in_start == 1 && in_currentStep != 0 && (qValues == 2'b00 || qValues == 2'b11),
     condition_red = in_start == 1 && in_currentStep != 0 && qValues == 2'b10,
     condition_purple = in_currentStep == 0;


// Next state logic.
always@(condition_gray,
        condition_black,
        condition_green,
        condition_blue,
        condition_red,
        condition_purple,

        state_current
       ) begin

// Retain current state.
state_next <= state_current;

   // Change state if needed.
   case(state_current)
      CONST_STATE_RESET: if(condition_gray) state_next <= CONST_STATE_INITIALIZE;

      CONST_STATE_INITIALIZE: begin
         if(condition_blue) state_next <= CONST_STATE_ONLY_ARS;
         else if(condition_red) state_next <= CONST_STATE_SUB_N_ARS;
      end

      CONST_STATE_PAUSE: begin
         if(condition_purple) state_next <= CONST_STATE_COMPLETE; // Prioritzed for ending calculations if, no more steps left.
         else if(condition_blue) state_next <= CONST_STATE_ONLY_ARS;
         else if(condition_red) state_next <= CONST_STATE_SUB_N_ARS;
         else if(condition_green) state_next <= CONST_STATE_ADD_N_ARS;
      end

      CONST_STATE_ONLY_ARS: begin
         if(condition_purple) state_next <= CONST_STATE_COMPLETE; // Prioritzed for ending calculations if, no more steps left.
         else if(condition_black) state_next <= CONST_STATE_PAUSE;
         else if(condition_green) state_next <= CONST_STATE_ADD_N_ARS;
         else if(condition_red) state_next <= CONST_STATE_SUB_N_ARS;
      end

      CONST_STATE_ADD_N_ARS: begin
         if(condition_purple) state_next <= CONST_STATE_COMPLETE; // Prioritzed for ending calculations if, no more steps left.
         else if(condition_black) state_next <= CONST_STATE_PAUSE;
         else if(condition_blue) state_next <= CONST_STATE_ONLY_ARS;
         else if(condition_red) state_next <= CONST_STATE_SUB_N_ARS;
      end

      CONST_STATE_SUB_N_ARS: begin
         if(condition_purple) state_next <= CONST_STATE_COMPLETE; // Prioritzed for ending calculations if, no more steps left.
         else if(condition_black) state_next <= CONST_STATE_PAUSE;
         else if(condition_blue) state_next <= CONST_STATE_ONLY_ARS;
         else if(condition_green) state_next <= CONST_STATE_ADD_N_ARS;
      end
   endcase
end


// Output logic.
always@(state_current) begin

   out_cmd_add <= 0;
   out_cmd_sub <= 0;
   out_cmd_ars <= 0;
   out_cmd_stepDown <= 0;
   out_cmd_init <= 0;
   out_cmd_rst <= 0;

   // CONST_STATE_PAUSE and CONST_STATE_COMPLETE
   //    are handled by assignment statements above.
   case(state_current)
      CONST_STATE_RESET: out_cmd_rst <= 1;

      CONST_STATE_INITIALIZE: out_cmd_init <= 1;

      CONST_STATE_ONLY_ARS: begin
         out_cmd_ars <= 1;
         out_cmd_stepDown <= 1;
      end

      CONST_STATE_ADD_N_ARS: begin
         out_cmd_add <= 1;
         out_cmd_ars <= 1;
         out_cmd_stepDown <= 1;
      end

      CONST_STATE_SUB_N_ARS: begin
         out_cmd_sub <= 1;
         out_cmd_ars <= 1;
         out_cmd_stepDown <= 1;
      end
   endcase
end

endmodule
```

```
// Testbech for FSM.
//    Tests the case of 3 x (-2)
//    Test is performed independent from datapath. That is, inputs from datapath are imitated.
// Fuat Deniz YILMAZOK - 2079465
module tb_ControlUnit_3_x_M2();

localparam CONST_STATE_COUNT = 7,
           CONST_DATA_SIZE = 4;

wire  out_cmd_add,
      out_cmd_sub,
      out_cmd_ars,
      out_cmd_stepDown,
      out_cmd_init,
      out_cmd_rst;


wire [CONST_STATE_COUNT - 1 : 0] out_currentState,
                                 out_jtag_state_next;

reg   in_start,
      in_Q_0,
      in_Q_M1,
      in_clk,
      in_rst;

reg [CONST_DATA_SIZE - 1 : 0] in_currentStep;

ControlUnit DUT(out_cmd_add,
                out_cmd_sub,
                out_cmd_ars,
                out_cmd_stepDown,
                out_cmd_init,
                out_cmd_rst,

                out_currentState,
                out_jtag_state_next,

                in_start,
                in_Q_0,
                in_Q_M1,
                in_currentStep,
                in_clk,
                in_rst
               );
```

```

localparam DEFAULT_DELAY = 100,
           ONE_CLOCK_CYCLE = 200;

always begin
   in_clk = 0; #DEFAULT_DELAY;
   in_clk = ~in_clk; #DEFAULT_DELAY;
end

initial begin

   // NOTE: Run this simulation for 7 clock cycles.
   in_rst = 1; in_start = 0; in_Q_0 = 0; in_Q_M1 = 0; in_currentStep = 4'd0; #ONE_CLOCK_CYCLE; // Start in reset state.

   in_rst = 0; #ONE_CLOCK_CYCLE; // Go to initialize state.

   // After multiplicand and multiplier is loaded into datapath, start calculation.
   in_start = 1; in_currentStep = 4'd4; in_Q_0 = 0; in_Q_M1 = 0; #ONE_CLOCK_CYCLE; // Ars only.

   in_start = 1; in_currentStep = 4'd3; in_Q_0 = 1; in_Q_M1 = 0; #ONE_CLOCK_CYCLE; // Sub & shift.

   in_start = 1; in_currentStep = 4'd2; in_Q_0 = 1; in_Q_M1 = 1; #ONE_CLOCK_CYCLE; // Ars only.

   in_start = 1; in_currentStep = 4'd1; in_Q_0 = 1; in_Q_M1 = 1; #ONE_CLOCK_CYCLE; // Ars only.

   in_start = 1; in_currentStep = 4'd0; in_Q_0 = 0; in_Q_M1 = 1; #ONE_CLOCK_CYCLE; // Test whether out_currentState ends up on 1_000_000.
end
endmodule
```

Simulation Results for Control Unit


/tb_ControlUnit_3_x_M2/out_currentState
/tb_ControlUnit_3_x_M2/out_jtag_state_next
State Outputs
/tb_ControlUnit_3_x_M2/out_cmd_add
/tb_ControlUnit_3_x_M2/out_cmd_sub
/tb_ControlUnit_3_x_M2/out_cmd_ars
/tb_ControlUnit_3_x_M2/out_cmd_stepDown
/tb_ControlUnit_3_x_M2/out_cmd_init
/tb_ControlUnit_3_x_M2/out_cmd_rst
Command Outputs
/tb_ControlUnit_3_x_M2/in_start
/tb_ControlUnit_3_x_M2/in_currentStep
/tb_ControlUnit_3_x_M2/in_Q_0
/tb_ControlUnit_3_x_M2/in_Q_M1
/tb_ControlUnit_3_x_M2/in_clk
/tb_ControlUnit_3_x_M2/in_rst
-No Data-
0000001
0000010
0001000
0100000
0001000
1000000
0000001
0000010
0001000
0100000
0001000
1000000
0000
0100
0011
0010
0001
0000
Now
1400 ps
ps
200 ps
400 ps
600 ps
800 ps
1000 ps
1200 ps
1400 ps

0-200ps: Controller is in reset state because, in_rst is asserted.

200-400ps: Here, it can be seen that, controller changes it's state from Reset to Initialize. This happens because, in this interval Rst is disabled.

400-600ps: Here, we see another state transition which is from Initialize state to Arithmetic Right Shift state. This is also can be seen from the output transitions in this interval. Note that, now output from counter is 0100 (4 in decimal), this because, changes to the datapath from previous steps is now reflected to the datapath.

600-800ps: In this interval Controller is performing subtract ant right shift operation. This is pretty normal, because, when you perform 3 x -2 by hand, you'll see that 3rd step is exactly the same. This can also be understood from the counter value.

Simulation Results for Control Unit

800-1000ps: Last two steps of the calculation should be two consecutive right shifts. And this, can be proven by looking at the state outputs. Current state and next state corresponds to the plain right shift operation. And, we got the first one in this interval.

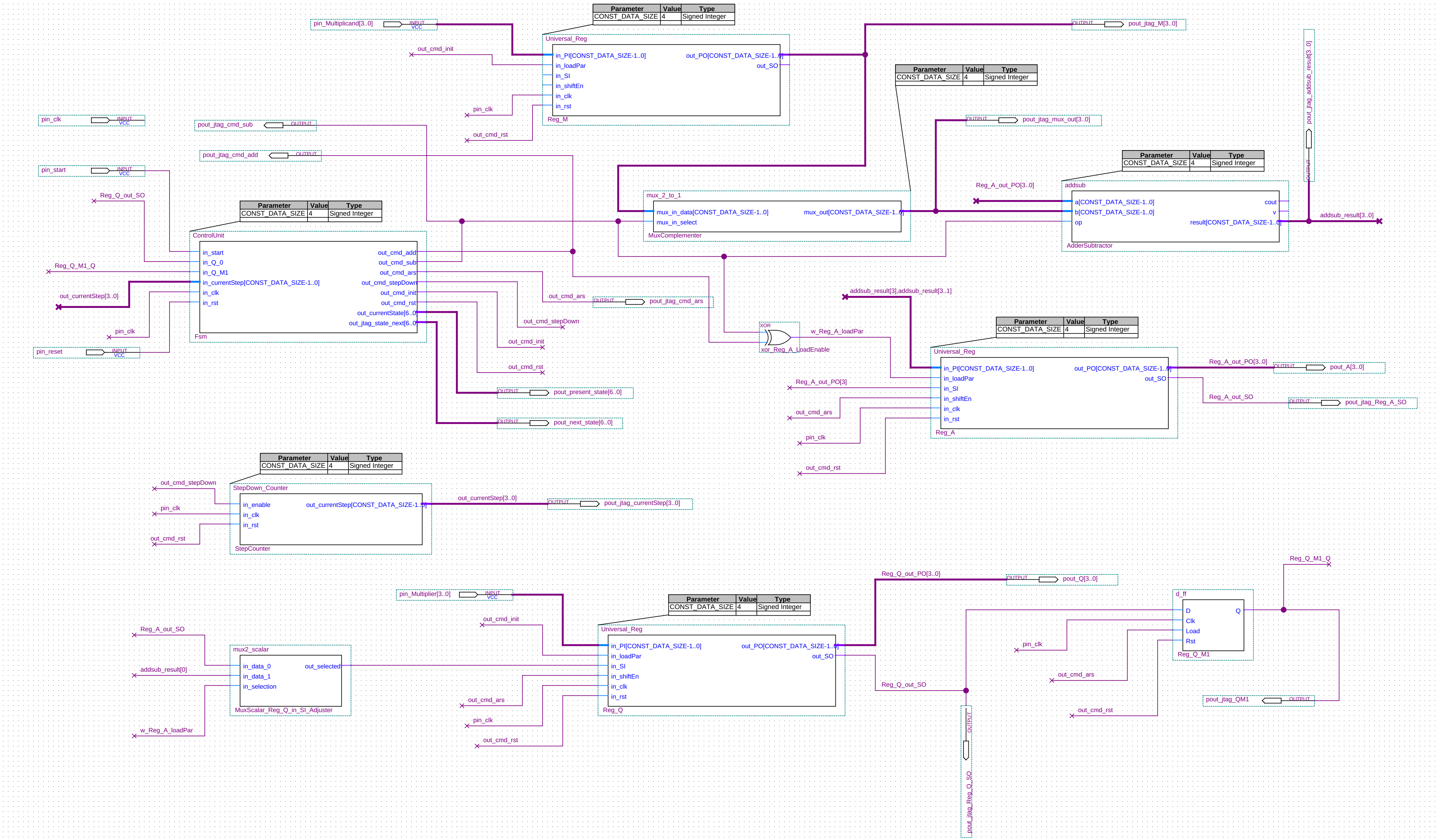
1000-1200ps: This interval follows from the previous one. So there is no change in the state outputs. But, we performed the last shift operation which is also final operation for calculation.

1200-1400ps: Now, controller is in complete state, this is because, in previous interval counter is decremented one last time. But, this new counter value read in this interval. Which in turn causes special condition to be triggered to make next state equal to the complete state. After that, because there is no transition in complete state controller will stay there until reset input is activated. This statement can be proved by looking at the state outputs which both have the same value.

Part 2 | Control Unit combined with Datapath

Booth Multiplier Schematic

Fuat Deniz YILMAZOK - 2079465



Notes for **Booth Multiplier Schematic**

Additional output & input pins in diagram:

Necessary output & input pins for booth's multiplier are:
pin_clk, **pin_start**, **pin_reset**, **pin_Multiplicand**, **pin_Multiplier**, **pout_A**, **pout_Q**, **pout_present_state** and **pout_next_state**.

Other pins are not relevant to the outside world, but they're are needed internally to operate the system or they're here to debug the circuit.

Which components are used from the previous experiments ?

From lab 2 & 3, **mux_2_to_1** and **addsub** are used. And, **d_ff** is used from lab 4 & 5.

How parallel load and arithmetic right shift (ars) achieved on Reg_A ?

Whenever, **only** ars is needed, right shift started by **in_shiftEn** of **Reg_A**. At the same time **Reg_A_out_PO[3]** is fed-back into the register via **in_SI**, which converts right shift to arithmetic right shift.

Parallel load is achieved via **w_Reg_A_loadPar**, it's equal to 1 when there is a subtraction or addition takes place. But, parallel load and arithmetic right shift needs to be done at the same clock cycle. To achieve this in one clock cycle, **addsub_result** is connected to **in_PI** by repeating the most significant bit. That is, shifted value is loaded into the **Reg_A** by using parallel load capability in one clock cycle. Although **Reg_A** is in correct state, **Reg_Q** is not. To fix that, primitive **MuxScalar_Reg_Q_in_SI_Adjuster** added into to schematic. Whenever, **w_Reg_A_loadPar** is active, it dictates mux to load least significant bit of **addsub_result** into **Reg_Q** from **in_SI** of **Reg_Q**. That's how parallel load and shift is achieved in same clock cycle.

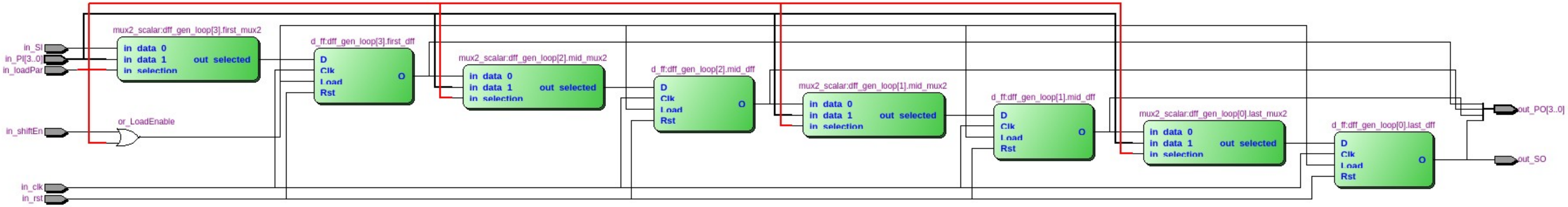
Primitive elements:

To achieve requested functionality of datapath, some custom primitive components are defined. These are: **add1bit**, **d_ff** and **mux2_scalar**.

Components inside the schematic may directly or indirectly contain those primitives. Also, **MuxScalar_Reg_Q_in_SI_Adjuster** is one of those primitives, which is directly instantiated in the schematic.

Synthesized schematic of **Universal_Reg**

Below you see the schematic of **Universal_Reg** which is generated from verilog code. This is included to simplify the understanding of code.


in_SI
in_PI[3..0]
in_loadPar
in_shiftEn
in_clk
in_rst
or_LoadEnable
mux2_scalar:dff_gen_loop[3].first_mux2
in data 0
in data 1
in selection
out selected
d_ff:dff_gen_loop[3].first_dff
D
Clk
Load
Rst
O
mux2_scalar:dff_gen_loop[2].mid_mux2
d_ff:dff_gen_loop[2].mid_dff
mux2_scalar:dff_gen_loop[1].mid_mux2
d_ff:dff_gen_loop[1].mid_dff
mux2_scalar:dff_gen_loop[0].last_mux2
d_ff:dff_gen_loop[0].last_dff
out_PO[3..0]
out_SO

```verilog
// Universal register used in the datapath.
//    Includes primitives d_ff and mux2_scalar.
// NOTE: This register has ordering of [3:0] (4 bit assumed).

// Fuat Deniz YILMAZOK - 2079465

module Universal_Reg(out_PO,
                     out_SO,

                     in_PI,
                     in_loadPar,
                     in_SI,
                     in_shiftEn,

                     in_clk,
                     in_rst
                    );

parameter CONST_DATA_SIZE = 4;

output [CONST_DATA_SIZE - 1 : 0] out_PO;
output out_SO;

input[CONST_DATA_SIZE - 1 : 0] in_PI;
input in_SI, in_shiftEn, in_loadPar, in_clk, in_rst;

wire [CONST_DATA_SIZE - 2 : 0] w_mid_SOSI; // Connection between d_ffs for shift register.
wire [CONST_DATA_SIZE - 1 : 0] w_mux_out; // Input to every d_ff.
wire w_load_enable; // Enable input to the register.

or or_LoadEnable(w_load_enable, in_shiftEn, in_loadPar);

// In loop below, indexing of registers start from 3. Whereas, indexing of w_mid_SOSI starts from 2.
//    That is, first_dff (index 3) gives output to w_mid_SOSI[2].
//
// Every d_ff receives it's input from a mux. Whenever, in_shiftEn or in_loadPar is activated,
//    corresponding mux gives necessary bit to the matching d_ff.
// in_loadPar has higher priority than in_shiftEn. That is, whenever both inputs are activated,
//    value of in_loadPar is selected by the mux.
```

```
genvar k;
generate for(k = CONST_DATA_SIZE - 1; k >= 0; k = k - 1) begin : dff_gen_loop

   // NOTE: d_ff pin order is d_ff(Q, D, in_clk, in_Load, in_rst);

   // If we're creating the first dff.
   if(k == CONST_DATA_SIZE - 1) begin
      mux2_scalar first_mux2(w_mux_out[k], in_SI, in_PI[k], in_loadPar);

      d_ff first_dff(w_mid_SOSI[k - 1], w_mux_out[k], in_clk, w_load_enable, in_rst);

      assign out_PO[k] = w_mid_SOSI[k - 1];
   end

   // If we're creating the last dff.
   else if(k == 0) begin
      mux2_scalar last_mux2(w_mux_out[k], w_mid_SOSI[k], in_PI[k], in_loadPar);

      d_ff last_dff(out_SO, w_mux_out[k], in_clk, w_load_enable, in_rst);

      assign out_PO[k] = out_SO;
   end

   // If we're creating the mid dffs.
   else begin
      mux2_scalar mid_mux2(w_mux_out[k], w_mid_SOSI[k], in_PI[k], in_loadPar);

      d_ff mid_dff(w_mid_SOSI[k - 1], w_mux_out[k], in_clk, w_load_enable, in_rst);

      assign out_PO[k] = w_mid_SOSI[k - 1];
   end

end // End of the loop.
endgenerate

endmodule
```

```verilog
// Wrap around down counter used in the datapath.
// Whenever, this clock is reset,
//       it's initialized to binary equivalent of decimal CONST_DATA_SIZE.

// Fuat Deniz YILMAZOK - 2079465

module StepDown_Counter(out_currentStep,

                        in_enable,
                        in_clk,
                        in_rst
                       );

parameter CONST_DATA_SIZE = 4;

output reg [CONST_DATA_SIZE - 1 : 0] out_currentStep;

input in_enable, in_clk, in_rst;

always@(posedge in_clk, posedge in_rst) begin

   if(in_rst)
      out_currentStep <= CONST_DATA_SIZE; // Initialize counter with decimal number.

   else if(in_enable)
      out_currentStep <= out_currentStep - 1;
end

endmodule
```

```
// This is taken from experiment 2 & 3.
//    NOTE: Parameter name changed to CONST_DATA_SIZE.

// Fuat Deniz YILMAZOK - 2079465

module mux_2_to_1(mux_out,

                  mux_in_data,
                  mux_in_select
                 );

parameter CONST_DATA_SIZE = 4;

output reg [CONST_DATA_SIZE - 1 : 0] mux_out;

input [CONST_DATA_SIZE - 1 : 0] mux_in_data;
input mux_in_select;

always@(mux_in_data, mux_in_select)
begin
   if(mux_in_select == 0)
      mux_out = mux_in_data;
   else
      mux_out = ~mux_in_data; // 1's complement is calculated with bitwise complement operator.
end

endmodule
```

```verilog
// This is taken from lab 2 & 3.
// Includes primitive add1bit.
//    NOTE: Parameter name changed to CONST_DATA_SIZE.

// Fuat Deniz YILMAZOK - 2079465

module addsub(cout, v,
              result,

              a, b,
              op
             );

parameter CONST_DATA_SIZE = 4;

output cout, v;
output [CONST_DATA_SIZE - 1 : 0] result;

input [CONST_DATA_SIZE - 1 : 0] a, b;
input op;

wire [CONST_DATA_SIZE - 1 : 0] c;

genvar i;
generate for(i = 0; i < CONST_DATA_SIZE; i = i+1) begin: addsub_loop

// NOTE: port configuration of add1bt is add1bit(sum, cout, a, b, cin);

   if(i == 0) // If we are creating the first adder.
      add1bit fa(result[i], c[i], a[i], b[i], op);

   else // If we are creating the other adders including the last one.
      add1bit fa(result[i], c[i], a[i], b[i], c[i-1]);

end endgenerate

// When there is no overflow,
//    this xor gate is used to ignore carry or borrow out of the sign bit position.
// But, when there is an overflow,
//     cout represents the actual sign bit of the 5-bit number.
xor (cout, c[CONST_DATA_SIZE - 1], op);

// Represnts overflow situation,
//     overflow can only occur when both numbers are have the same sign.
// It detects overflow by checking carry into the sign bit and carry out of the sign bit.
//    When they're different, that means there is an overflow situation.
xor (v, c[CONST_DATA_SIZE - 1], c[CONST_DATA_SIZE - 2]);

endmodule
```

```verilog
// 1 bit full adder, which is primitive for other designs.

// Fuat Deniz YILMAZOK - 207945

module add1bit(output sum,
               output cout,

               input a,
               input b,
               input cin
              );

assign {cout, sum} = a + b + cin;

endmodule
```

```verilog
// Taken from experiment 4 & 5.
//    NOTE: Reset condition is changed to the active high.

// Fuat Deniz YILMAZOK - 2079465

module d_ff(output reg Q,

            input D,
            input Clk,
            input Load,
            input Rst
           );

always@(posedge Clk, posedge Rst) begin
   if(Rst) // If async reset input is asserted.
      Q <= 1'b0;

   else if(Load) // If Load was active before the rising edge of the Clk.
      Q <= D;
end

endmodule
```

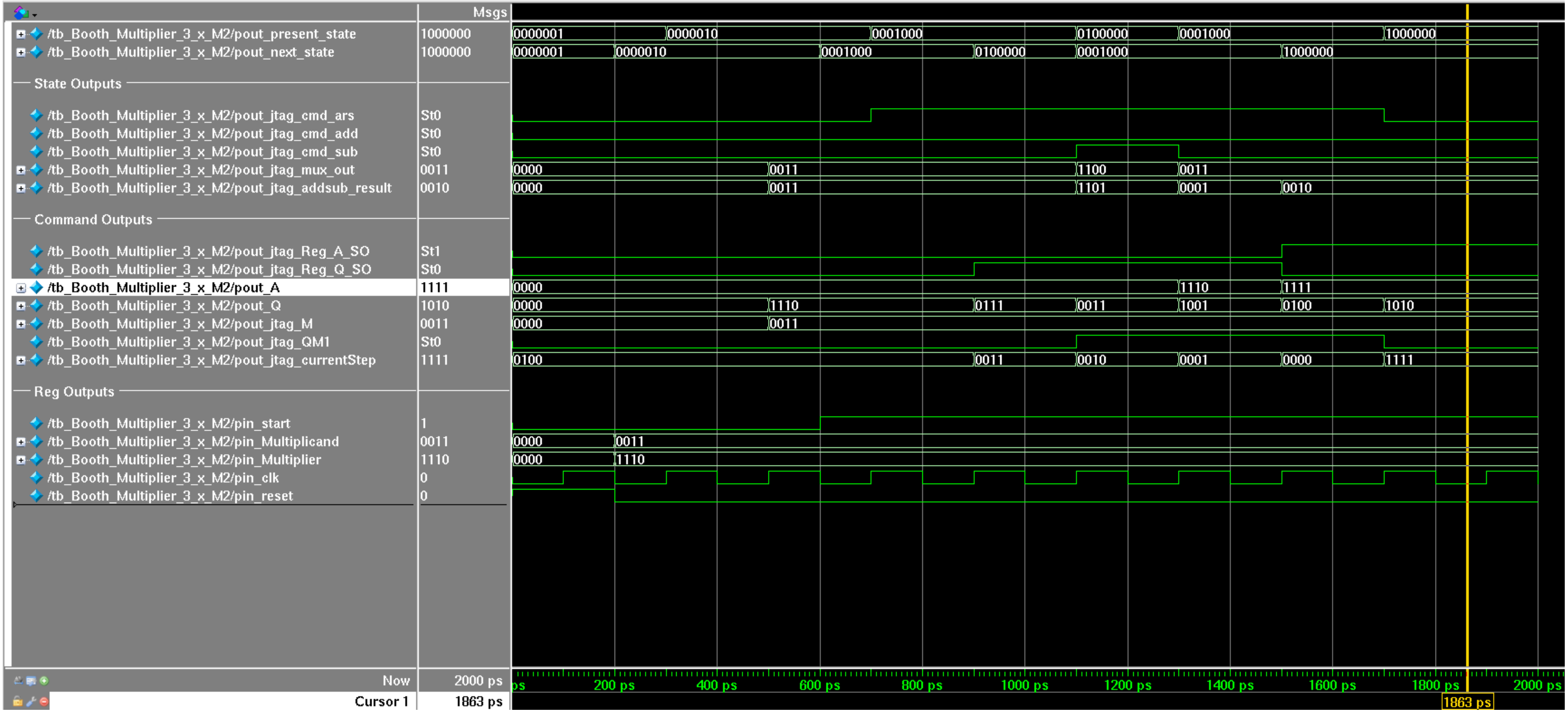
```verilog
// Primitive scalar 2 to 1 mux.

// Fuat Deniz YILMAZOK - 2079465.

module mux2_scalar(out_selected,

                   in_data_0,
                   in_data_1,
                   in_selection
                  );

output reg out_selected;

input in_data_0, in_data_1, in_selection;

always@(in_data_0, in_data_1, in_selection) begin

   if(in_selection == 1) out_selected <= in_data_1;

   else out_selected <= in_data_0;

end

endmodule
```

```
// Test bench for module Booth_Multiplier.
//    This testbench tests control unit combined with the datapath.

// Fuat Deniz YILMAZOK - 2079465

module tb_Booth_Multiplier_3_x_M2();

localparam CONST_STATE_COUNT = 7,
           CONST_DATA_SIZE = 4,
           CONST_DEFAULT_DELAY = 100,
           CONST_CLOCK_CYCLE = 2 * CONST_DEFAULT_DELAY;

reg pin_start;
reg pin_reset;
reg pin_clk;
reg [3:0] pin_Multiplicand;
reg [3:0] pin_Multiplier;
wire pout_jtag_QM1;
wire pout_jtag_cmd_ars;
wire pout_jtag_cmd_add;
wire pout_jtag_cmd_sub;
wire pout_jtag_Reg_A_SO;
wire pout_jtag_Reg_Q_SO;
wire [3:0] pout_A;
wire [3:0] pout_jtag_addsub_result;
wire [3:0] pout_jtag_currentStep;
wire [3:0] pout_jtag_M;
wire [3:0] pout_jtag_mux_out;
wire [6:0] pout_next_state;
wire [6:0] pout_present_state;
wire [3:0] pout_Q;

Booth_Multiplier DUT(
   pin_start,
   pin_reset,
   pin_clk,
   pin_Multiplicand,
   pin_Multiplier,

   pout_jtag_QM1,
   pout_jtag_cmd_ars,
   pout_jtag_cmd_add,
   pout_jtag_cmd_sub,
   pout_jtag_Reg_A_SO,
   pout_jtag_Reg_Q_SO,
   pout_A,
   pout_jtag_addsub_result,
   pout_jtag_currentStep,
   pout_jtag_M,
   pout_jtag_mux_out,
   pout_next_state,
   pout_present_state,
   pout_Q
);

```

```

always begin
   pin_clk = 0; #CONST_DEFAULT_DELAY;
   pin_clk = ~pin_clk; #CONST_DEFAULT_DELAY;
end

localparam CONST_MULTIPLICAND = 4'b0011,
           CONST_MULTIPLIER = 4'b1110;

initial begin
   // NOTE: Run this test for 10 clock cycles.

   // Start booth multiplier in reset state.
   pin_reset = 1; pin_start = 0; pin_Multiplicand = 0; pin_Multiplier = 0; #
CONST_CLOCK_CYCLE;

   // Go to initialize state.
   pin_reset = 0; pin_start = 0; pin_Multiplicand = CONST_MULTIPLICAND;
pin_Multiplier = CONST_MULTIPLIER; #CONST_CLOCK_CYCLE;

   // Load Multiplicand +3 = 0011 and Multiplier -2 = 1110
   pin_reset = 0; pin_start = 0; pin_Multiplicand = CONST_MULTIPLICAND;
pin_Multiplier = CONST_MULTIPLIER; #CONST_CLOCK_CYCLE;

   // Start calculation with ars.
   pin_reset = 0; pin_start = 1; pin_Multiplicand = CONST_MULTIPLICAND;
pin_Multiplier = CONST_MULTIPLIER; #CONST_CLOCK_CYCLE;

   // Subtract and shift.
   pin_reset = 0; pin_start = 1; pin_Multiplicand = CONST_MULTIPLICAND;
pin_Multiplier = CONST_MULTIPLIER; #CONST_CLOCK_CYCLE;

   // Ars only.
   pin_reset = 0; pin_start = 1; pin_Multiplicand = CONST_MULTIPLICAND;
pin_Multiplier = CONST_MULTIPLIER; #CONST_CLOCK_CYCLE;

   // Ars only.
   pin_reset = 0; pin_start = 1; pin_Multiplicand = CONST_MULTIPLICAND;
pin_Multiplier = CONST_MULTIPLIER; #CONST_CLOCK_CYCLE;
end

endmodule
```

Simulation Results for **module tb_Booth_Multiplier_3_x_M2**

Verilog code for **DUT** is generated from schematic, which can be found in the **appendix A**.



```
                                                   Msgs
/tb_Booth_Multiplier_3_x_M2/pout_present_state     1000000
/tb_Booth_Multiplier_3_x_M2/pout_next_state        1000000
State Outputs
/tb_Booth_Multiplier_3_x_M2/pout_jtag_cmd_ars      St0
/tb_Booth_Multiplier_3_x_M2/pout_jtag_cmd_add      St0
/tb_Booth_Multiplier_3_x_M2/pout_jtag_cmd_sub      St0
/tb_Booth_Multiplier_3_x_M2/pout_jtag_mux_out      0011
/tb_Booth_Multiplier_3_x_M2/pout_jtag_addsub_result 0010
Command Outputs
/tb_Booth_Multiplier_3_x_M2/pout_jtag_Reg_A_SO     St1
/tb_Booth_Multiplier_3_x_M2/pout_jtag_Reg_Q_SO     St0
/tb_Booth_Multiplier_3_x_M2/pout_A                 1111
/tb_Booth_Multiplier_3_x_M2/pout_Q                 1010
/tb_Booth_Multiplier_3_x_M2/pout_jtag_M            0011
/tb_Booth_Multiplier_3_x_M2/pout_jtag_QM1          St0
/tb_Booth_Multiplier_3_x_M2/pout_jtag_currentStep  1111
Reg Outputs
/tb_Booth_Multiplier_3_x_M2/pin_start              1
/tb_Booth_Multiplier_3_x_M2/pin_Multiplicand       0011
/tb_Booth_Multiplier_3_x_M2/pin_Multiplier         1110
/tb_Booth_Multiplier_3_x_M2/pin_clk                0
/tb_Booth_Multiplier_3_x_M2/pin_reset              0
Now       2000 ps
Cursor 1  1863 ps
```

0-200ps: Module started in a reset state. This can be proven by looking to the all of the registers. In addition, Counter is set to decimal 4 (0100 in binary) and all of the state outputs gives reset state.

200-400ps: Now, reset is disabled. This means condition gray should take place. That is, control unit should switch to the initialize state. And, as you can this transition happens in this interval.

400-600ps: In this interval, **pin_Multiplicand** and **pin_Multiplier** are loaded into corresponding registers. This loading happened in this interval because, although during initialize state **out_cmd_init = 1** (**not shown in the graph**) it needs an setup time to update registers. That's why now registers are initialized.

Simulation Results for **module tb_Booth_Multiplier_3_x_M2**

600-800ps: In this interval, **pin_start = 1**. Which means, operation should start. And this can be proven by looking at the state outputs. Before the clock edge, next state is indicated as state ars_only. And after clock hits, present state updated accordingly. As a result, **pout_jtag_cmd_ars = 1.** But, ars will happen in the next clock edge and counter will be decremented at the next clock edge. Because, there is a setup time for registers.

800-1000ps: This the interval where previously described operations will occur. Namely, shift occurred and counter is decremented. Note that, **Reg_A** is also shifted in this interval but, because it was initialized to zero, changes will be indistinguishable. Moreover, next state indicates that, next operation is subtract and ars. This is true, because, **pout_Q[0] = 1** and **pout_jtag_QM1 = 0.**

1000-1200ps: Here present state becomes sub_n_ars state, because of this, **pout_jtag_cmd_sub = 1** and **pout_jtag_cmd_ars = 1**. After clock edge encountered, Q stores newly shifted value, which is **0011 (+3)**. When you look at the **pout_jtag_addsub_result**, you'll see that, result is **1101 (-3)**. That means, -Q calculated successfully. Now, it's time to load new value into **Reg_A** and shift it. But, this will happen on the next interval, because of the setup time.

1200-1400ps: When ars applied to the 1101 result is **1110**. If you look at **pout_A**, you'll verify that, state of **Reg_A** is correct. After 1101 shifted, rightmost 1 needs put into leftmost bit of **Reg_Q**. In previous interval **Reg_Q** was 0011, when 1 is fed into **Reg_Q** from serial input result is **1001**. Which can be clearly can be seen from the graph. In addition, this interval leads to next state ars_only.

1400-1600ps: Here **pout_jtag_currentStep = 0** that means, it's time to end calculation. But, after sub_n_ars happened we need to perform 2 consecutive ars (can be seen from manual calculation). To understand this situation, look at the 1300ps. At 1300ps sub_n_shift ends and ars_only starts and counter has value of 1. So, whenever state is ars_only and counter != 0, we need to perform ars and decrement counter. This is what happened between **1300ps and 1500ps**. So, in current interval (**1400-1600ps)** we need to the perform 2nd ars. This can be seen by looking at the **pout_present_state** which is still at state ars_only. So, next clock edge will give us the final shift.

But, although counter = 0, why ars still continues in current interval ? This is related to how state machine works. When counter = 1, state machine already decided to perform an ars, so, it performs an ars then decides to stay at the same state, then decrements the counter. And, when counter = 0, performs another ars, then decides to go to next state which is the complete state.

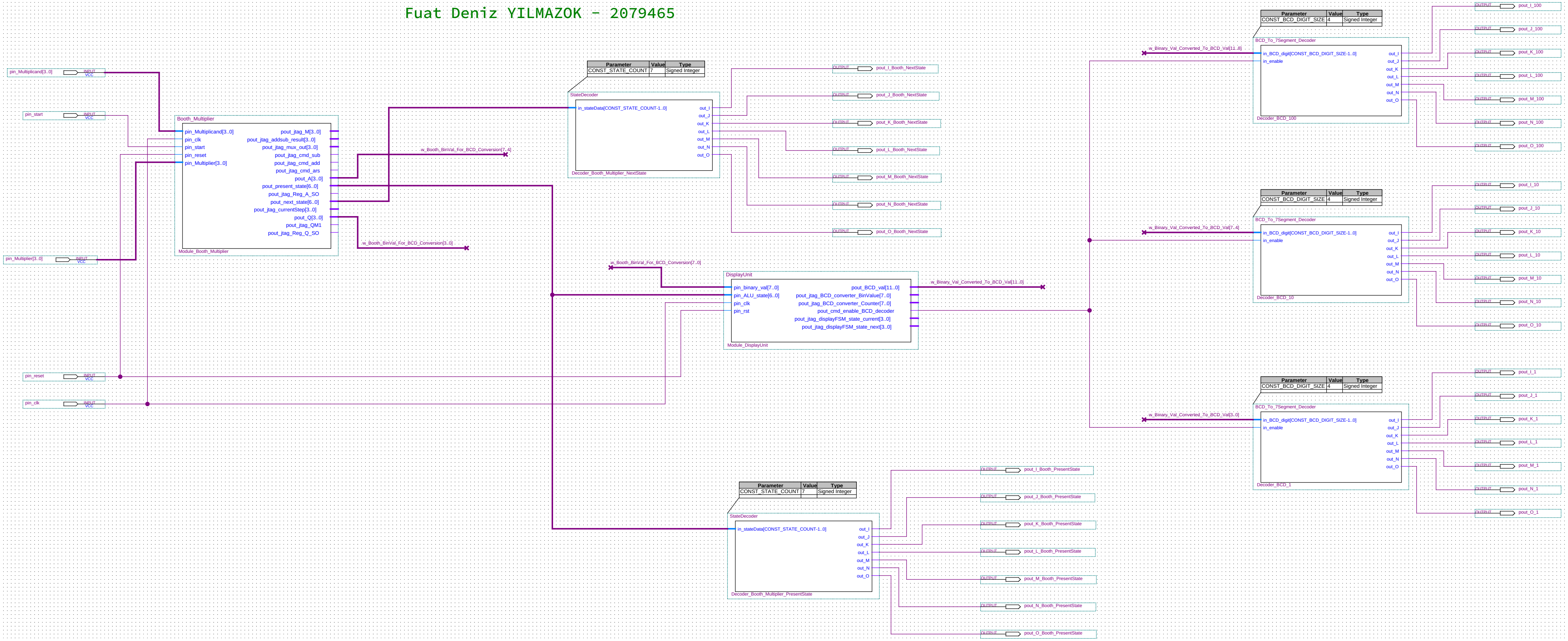
1600-1800ps: Effects of the second shift which is performed in the previous interval is apparent here. That is, after clock edge, register values give the result. When clock transition happened in current interval, state machine goes to complete state. And because, clock edge occurred, counter is decremented one more time. But, this won't be problem because operation is completed.

1800-2000ps: This interval proves, until reset is activated, machine will stay in complete state and continue to give result.

Part 3 | Final Design

Final Design Schematic

Fuat Deniz YILMAZOK - 2079465



Notes for **Final Design Schematic**

Explanation of modules:

Booth_Multiplier: This module contains control unit for booth’s multiplier algorithm and corresponding datapath. Internal details can be found on previous part.

StateDecoder: This is a combinational circuit which decodes state data received from **Booth_Multiplier** to segments of a 7-segment display.

BCD_To_7Segment_Decoder: Another combinational circuit, which takes a bcd digit as an input and decodes it to 7-segment display.

DisplayUnit: This a sequential circuit which contains FSM. It converts binary result of **Booth_Multiplier** to array of BCD digits. Then, distributes those individual BCD digits to corresponding **BCD_To_7Segment_Decoder**. And, controls those decoders operation. That is, this module determines when decoders should display the BCD digits. Implementation details for this module explained on next part.

How BCD digits are displayed on 7-segment displays ?

First of all, **Booth_Multiplier** performs signed multiplcation. Therefore, 8-bit result received from it will be signed. For example, let's take **1111 1010** (-6 in signed 2's complement system or 250 in unsigned binary) as an example. This value then, given to **DisplayUnit.** Here, 1111 1010 is converted to BCD array with the help of **Double Dabble** algorithm, which is **0010 0101 0000** (each BCD digit corresponds to digits of 250). Then, **DisplayUnit** distributes those BCD digits according to their place value (Hundreads, tens and ones). Finally, when binary to bcd conversion is completed, **DisplayUnit** activates decoders to display final result.

Why result is displayed as BCD ?

This because, decoding a number in signed 2's complement form will cost a lot. Such that, decoding numbers in this form will require a lookup table which corresponds to a case statement in verilog. So, this case statement will be to long to handle. But, by using **double dabble** algorithm for converting binary to BCD, we streamline conversion procedure with well-defined steps instead of costly lookup table.

How humans should translate the result ?

First, convert displayed decimal number into **un-signed binary** by dividing 2. Then, treat resulting number as 8-bit **signed** number in **2's complement** form.

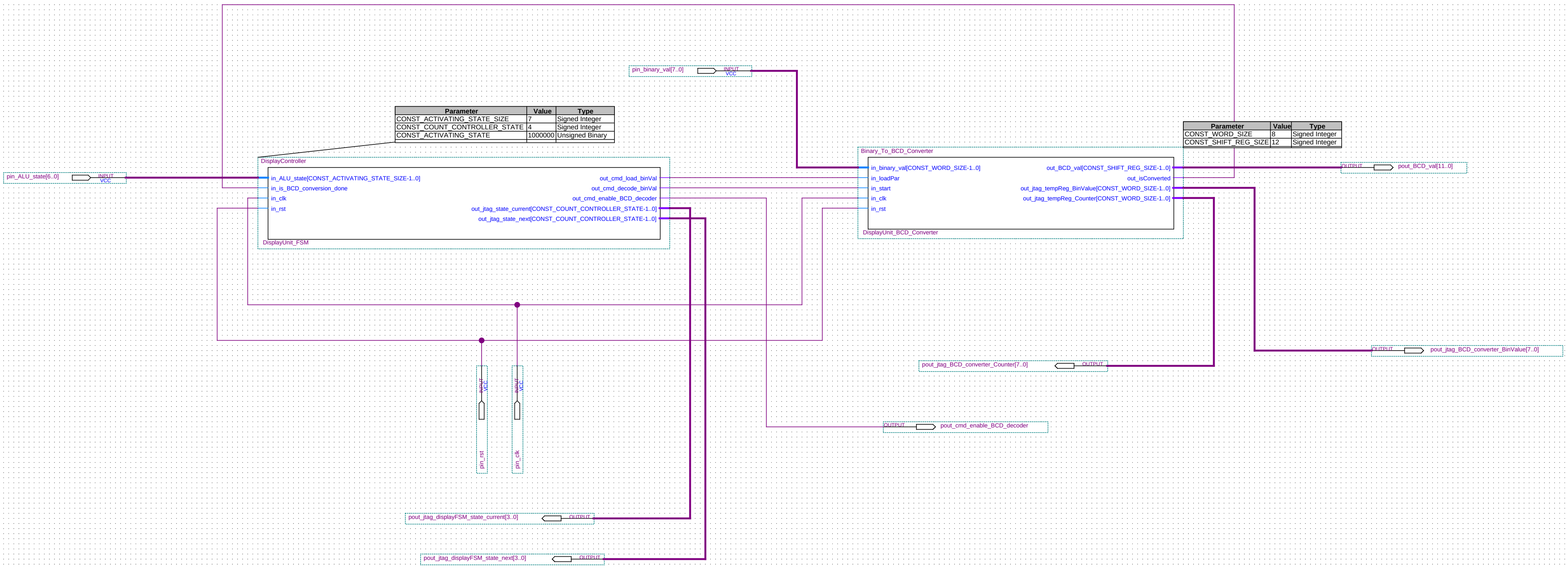
```
// Fuat Deniz YILMAZOK - 2079465

module BCD_To_7Segment_Decoder(out_I,
                               out_J,
                               out_K,
                               out_L,
                               out_M,
                               out_N,
                               out_O,

                               in_BCD_digit,
                               in_enable
                              );

parameter CONST_BCD_DIGIT_SIZE = 4;

output reg out_I,
           out_J,
           out_K,
           out_L,
           out_M,
           out_N,
           out_O;

input [CONST_BCD_DIGIT_SIZE - 1 : 0] in_BCD_digit;
input in_enable;

always@(in_BCD_digit, in_enable) begin
   if(!in_enable) begin
      // Display "-" (dash)
      out_I = 1; out_J = 1; out_K = 1; out_L = 1; out_M = 1; out_N = 1; out_O = 0;
   end

   else begin // If decoder enabled.
      case(in_BCD_digit)
         4'b0001: begin out_I = 1; out_J = 0; out_K = 0; out_L = 1; out_M = 1; out_N = 1; out_O = 1; end // 1
         4'b0010: begin out_I = 0; out_J = 0; out_K = 1; out_L = 0; out_M = 0; out_N = 1; out_O = 0; end // 2
         4'b0011: begin out_I = 0; out_J = 0; out_K = 0; out_L = 0; out_M = 1; out_N = 1; out_O = 0; end // 3
         4'b0100: begin out_I = 1; out_J = 0; out_K = 0; out_L = 1; out_M = 1; out_N = 0; out_O = 0; end // 4
         4'b0101: begin out_I = 0; out_J = 1; out_K = 0; out_L = 0; out_M = 1; out_N = 0; out_O = 0; end // 5
         4'b0110: begin out_I = 0; out_J = 1; out_K = 0; out_L = 0; out_M = 0; out_N = 0; out_O = 0; end // 6
         4'b0111: begin out_I = 0; out_J = 0; out_K = 0; out_L = 1; out_M = 1; out_N = 1; out_O = 1; end // 7
         4'b1000: begin out_I = 0; out_J = 0; out_K = 0; out_L = 0; out_M = 0; out_N = 0; out_O = 0; end // 8
         4'b1001: begin out_I = 0; out_J = 0; out_K = 0; out_L = 0; out_M = 1; out_N = 0; out_O = 0; end // 9
         // 0000
         default: begin out_I = 0; out_J = 0; out_K = 0; out_L = 0; out_M = 0; out_N = 0; out_O = 1; end
      endcase
   end
end

endmodule
```

```
// State decoder, which will receive the input from state outputs of booth multiplier.

// Fuat Deniz YILMAZOK - 2079465

module StateDecoder(out_I, out_J, out_K, out_L, out_M, out_N, out_O,

                    in_currentState
                   );

parameter CONST_STATE_COUNT = 7;

localparam CONST_STATE_RESET = 7'b000_000_1,
           CONST_STATE_INITIALIZE = 7'b000_001_0,
           CONST_STATE_PAUSE = 7'b000_010_0,
           CONST_STATE_ONLY_ARS = 7'b000_100_0,
           CONST_STATE_ADD_N_ARS = 7'b001_000_0,
           CONST_STATE_SUB_N_ARS = 7'b010_000_0,
           CONST_STATE_COMPLETE = 7'b100_000_0;

output reg out_I, out_J, out_K, out_L, out_M, out_N, out_O;

input [CONST_STATE_COUNT - 1 : 0] in_currentState;

always@(in_currentState) begin

   case(in_currentState)
      // Display 2
      CONST_STATE_INITIALIZE: begin out_I = 0; out_J = 0; out_K = 1; out_L = 0; out_M = 0; out_N = 1; out_O = 0; end

      // Display 3
      CONST_STATE_PAUSE: begin out_I = 0; out_J = 0; out_K = 0; out_L = 0; out_M = 1; out_N = 1; out_O = 0; end

      // Display 4
      CONST_STATE_ONLY_ARS: begin out_I = 1; out_J = 0; out_K = 0; out_L = 1; out_M = 1; out_N = 0; out_O = 0; end

      // Display 5
      CONST_STATE_ADD_N_ARS: begin out_I = 0; out_J = 1; out_K = 0; out_L = 0; out_M = 1; out_N = 0; out_O = 0; end

      // Display 6
      CONST_STATE_SUB_N_ARS: begin out_I = 0; out_J = 1; out_K = 0; out_L = 0; out_M = 0; out_N = 0; out_O = 0; end

      // Display 7
      CONST_STATE_COMPLETE: begin out_I = 0; out_J = 0; out_K = 0; out_L = 1; out_M = 1; out_N = 1; out_O = 1; end

      // CONST_STATE_RESET - Display 1
      default: begin out_I = 1; out_J = 0; out_K = 0; out_L = 1; out_M = 1; out_N = 1; out_O = 1; end
   endcase

end

endmodule
```

Part 4 | Display Unit

Display Unit Schematic

Fuat Deniz YILMAZOK - 2079465



Parameter	Value	Type
CONST_ACTIVATING_STATE_SIZE	7	Signed Integer
CONST_COUNT_CONTROLLER_STATE	4	Signed Integer
CONST_ACTIVATING_STATE	1000000	Unsigned Binary

Parameter	Value	Type
CONST_WORD_SIZE	8	Signed Integer
CONST_SHIFT_REG_SIZE	12	Signed Integer

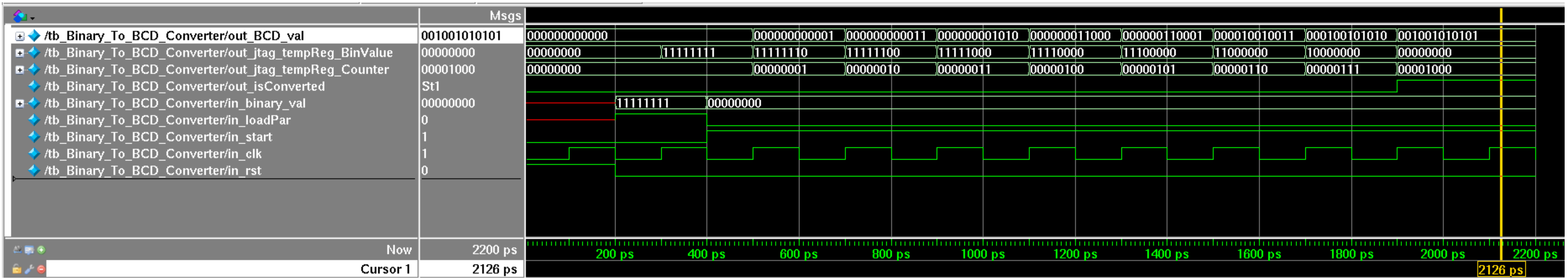
pin_binary_val[7..0]
INPUT
VCC
pin_ALU_state[6..0]
INPUT
VCC
DisplayController
in_ALU_state[CONST_ACTIVATING_STATE_SIZE-1..0]
in_is_BCD_conversion_done
in_clk
in_rst
out_cmd_load_binVal
out_cmd_decode_binVal
out_cmd_enable_BCD_decoder
out_jtag_state_current[CONST_COUNT_CONTROLLER_STATE-1..0]
out_jtag_state_next[CONST_COUNT_CONTROLLER_STATE-1..0]
DisplayUnit_FSM
Binary_To_BCD_Converter
in_binary_val[CONST_WORD_SIZE-1..0]
in_loadPar
in_start
in_clk
in_rst
out_BCD_val[CONST_SHIFT_REG_SIZE-1..0]
out_isConverted
out_jtag_tempReg_BinValue[CONST_WORD_SIZE-1..0]
out_jtag_tempReg_Counter[CONST_WORD_SIZE-1..0]
DisplayUnit_BCD_Converter
OUTPUT
pout_BCD_val[11..0]
OUTPUT
pout_jtag_BCD_converter_BinValue[7..0]
pout_jtag_BCD_converter_Counter[7..0]
OUTPUT
INPUT
VCC
pin_rst
INPUT
VCC
pin_clk
OUTPUT
pout_cmd_enable_BCD_decoder
pout_jtag_displayFSM_state_current[3..0]
OUTPUT
pout_jtag_displayFSM_state_next[3..0]
OUTPUT

```verilog
// Uses double dabble algorithm for conversion.
// Provides conversion logic for datapath of DisplayController.

// Fuat Deniz YILMAZOK - 2079465

module Binary_To_BCD_Converter(out_BCD_val,
                               out_isConverted,

                               out_jtag_tempReg_BinValue,
                               out_jtag_tempReg_Counter,

                               in_binary_val,
                               in_loadPar,
                               in_start,
                               in_clk,
                               in_rst
                              );

parameter CONST_WORD_SIZE = 8, // Input binary data size.
          CONST_SHIFT_REG_SIZE = 12; // Holds the partial and resulting BCD array.

localparam CONST_MAX_ALLOWED_DIGIT = 4;

output [CONST_SHIFT_REG_SIZE - 1 : 0] out_BCD_val;
output reg out_isConverted; // Output flag to indicate conversion is done.

output [CONST_WORD_SIZE - 1 : 0] out_jtag_tempReg_BinValue,
                                 out_jtag_tempReg_Counter;

input [CONST_WORD_SIZE - 1 : 0] in_binary_val;
input in_loadPar, in_start, in_clk, in_rst;

reg [CONST_WORD_SIZE - 1 : 0] tempReg_BinValue,
                              tempReg_Counter;

reg [CONST_SHIFT_REG_SIZE - 1 : 0] tempReg_Digits;

assign out_BCD_val = tempReg_Digits;
assign out_jtag_tempReg_BinValue = tempReg_BinValue;
assign out_jtag_tempReg_Counter = tempReg_Counter;

always@(posedge in_clk, posedge in_rst) begin

   if(in_rst) begin
```

```
      tempReg_BinValue = 0;
      tempReg_Digits = 0;
      tempReg_Counter = 0;
      out_isConverted = 0;
   end

   else if(in_loadPar) begin
      tempReg_BinValue = in_binary_val;
      tempReg_Digits = 0;
      tempReg_Counter = 0;
      out_isConverted = 0;
   end

   else if(in_start == 1 && out_isConverted != 1) begin
      tempReg_Counter = tempReg_Counter + 1; // Synchronize algorithm step with word size index.

      // Shift left by 1 bit regardless of everything.
      tempReg_Digits = tempReg_Digits << 1;
      tempReg_Digits[0] = tempReg_BinValue[CONST_WORD_SIZE - 1];
      tempReg_BinValue = tempReg_BinValue << 1;

      // Prevent adding 3 in the last step.
      if(tempReg_Counter == CONST_WORD_SIZE) out_isConverted = 1;

      // Add 3 to exceeding digits, except for the last step.
      if(!out_isConverted) begin
         // Check for Ones
         if (tempReg_Digits[3:0] >= CONST_MAX_ALLOWED_DIGIT)
            tempReg_Digits[3:0] = tempReg_Digits[3:0] + 3;

         // Check for Tens
         if(tempReg_Digits[7:4] >= CONST_MAX_ALLOWED_DIGIT)
            tempReg_Digits[7:4] = tempReg_Digits[7:4] + 3;

         // No need to check for Hundreads, because maximum of hundreads can get is 0011.
         // Actually, CONST_SHIFT_REG_SIZE = 10 is enough for algorithm but,
        //      12 is set, because, decoders expect 3 BCD digits.
        //  And, number 10 comes from the addittion,
        //      that is, because we may add 3 to two 4-bit numbers in algoritm, we need places for carries.
      end
   end
end

endmodule
```

```verilog
// Testbench for Binary_To_BCD_Converter

// Fuat Deniz YILMAZOK - 2079465

module tb_Binary_To_BCD_Converter();

parameter CONST_WORD_SIZE = 8,
          CONST_SHIFT_REG_SIZE = 12;

localparam CONST_MAX_ALLOWED_DIGIT = 4;

wire [CONST_SHIFT_REG_SIZE - 1 : 0] out_BCD_val;

wire [CONST_WORD_SIZE - 1 : 0] out_jtag_tempReg_BinValue,
                               out_jtag_tempReg_Counter;
wire out_isConverted;

reg [CONST_WORD_SIZE - 1 : 0] in_binary_val;
reg in_loadPar, in_start, in_clk, in_rst;

Binary_To_BCD_Converter DUT(out_BCD_val,
                            out_isConverted,

                            out_jtag_tempReg_BinValue,
                            out_jtag_tempReg_Counter,

                            in_binary_val,
                            in_loadPar,
                            in_start,
                            in_clk,
                            in_rst
                           );


localparam CONST_DEFAULT_DELAY = 100,
           CONST_CLOCK_PERIOD = 2 * CONST_DEFAULT_DELAY,
           CONST_BINARY_TEST_VALUE = 8'b1111_1111;

always begin
   in_clk = 0; #CONST_DEFAULT_DELAY;
   in_clk = ~in_clk; #CONST_DEFAULT_DELAY;
end

initial begin
   // NOTE: Run this simulation for 11 clock cycles.

   in_rst = 1; in_start = 0; #CONST_CLOCK_PERIOD;

   // Load value 1111_1111 from parallel input.
   in_rst = 0; in_start = 0; in_loadPar = 1; in_binary_val = CONST_BINARY_TEST_VALUE; #CONST_CLOCK_PERIOD;

   // Double dabble until counter value becomes 8.
   in_rst = 0; in_start = 1; in_loadPar = 0; in_binary_val = 0; #CONST_CLOCK_PERIOD;
end

endmodule
```

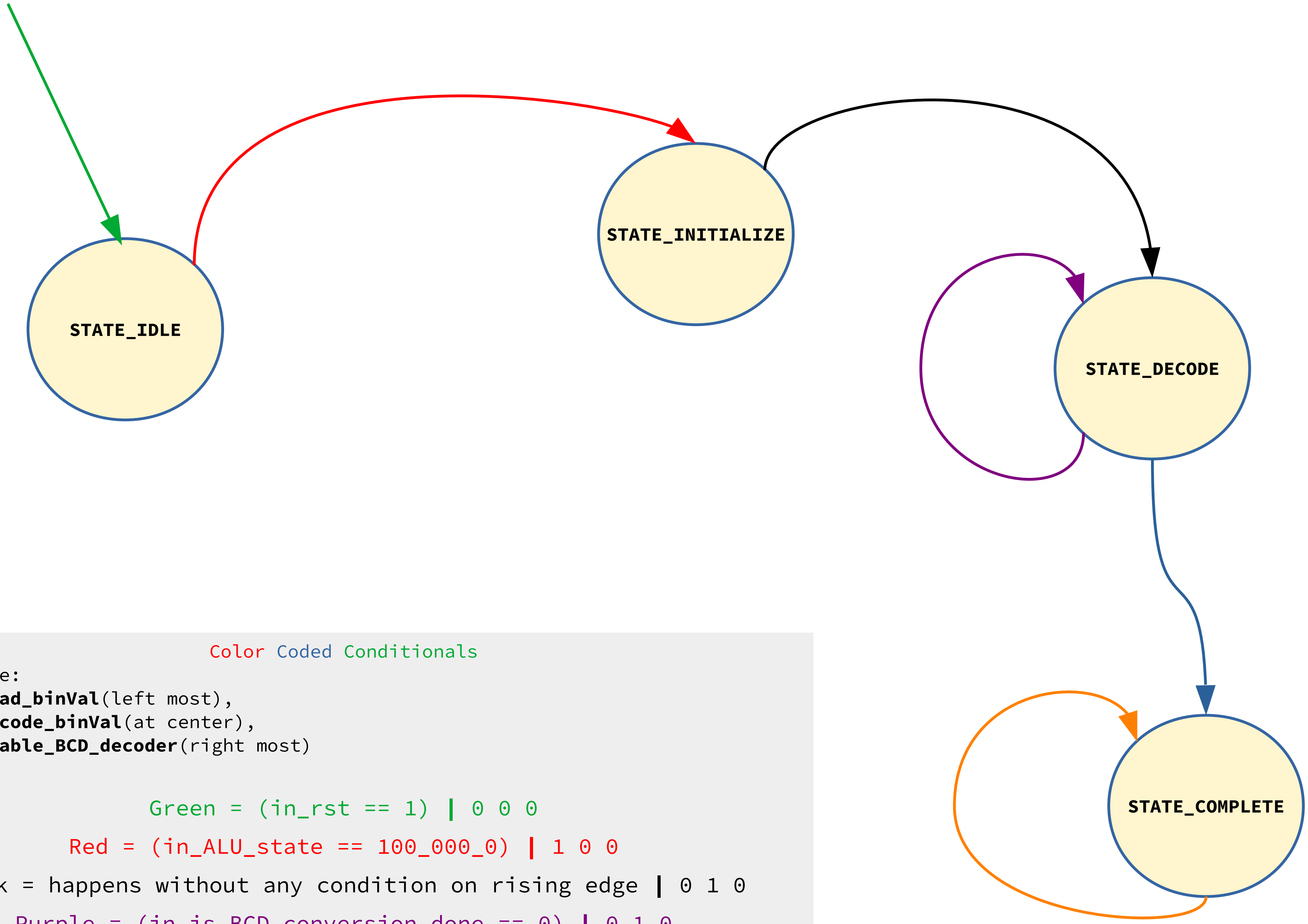
Simulation Results for **Binary_To_BCD_Converter**



	Msgs
/tb_Binary_To_BCD_Converter/out_BCD_val	001001010101
/tb_Binary_To_BCD_Converter/out_jtag_tempReg_BinValue	00000000
/tb_Binary_To_BCD_Converter/out_jtag_tempReg_Counter	00001000
/tb_Binary_To_BCD_Converter/out_isConverted	St1
/tb_Binary_To_BCD_Converter/in_binary_val	00000000
/tb_Binary_To_BCD_Converter/in_loadPar	0
/tb_Binary_To_BCD_Converter/in_start	1
/tb_Binary_To_BCD_Converter/in_clk	1
/tb_Binary_To_BCD_Converter/in_rst	0
Now	2200 ps
Cursor 1	2126 ps

Simulation results are self-explanatory. With careful inspection correctness of the DUT can be verified. Output value 255 (**0010 0101 0101**) is given at the last step as expected.

State Diagram for **DisplayController**



Color Coded Conditionals

Outputs are:
out_cmd_load_binVal(left most),
out_cmd_decode_binVal(at center),
out_cmd_enable_BCD_decoder(right most)

Green = (in_rst == 1) | 0 0 0

Red = (in_ALU_state == 100_000_0) | 1 0 0

Black = happens without any condition on rising edge | 0 1 0

Purple = (in_is_BCD_conversion_done == 0) | 0 1 0

Blue = (in_is_BCD_conversion_done == 1) | 0 0 1

Orange = happens without any condition on rising edge | 0 0 1

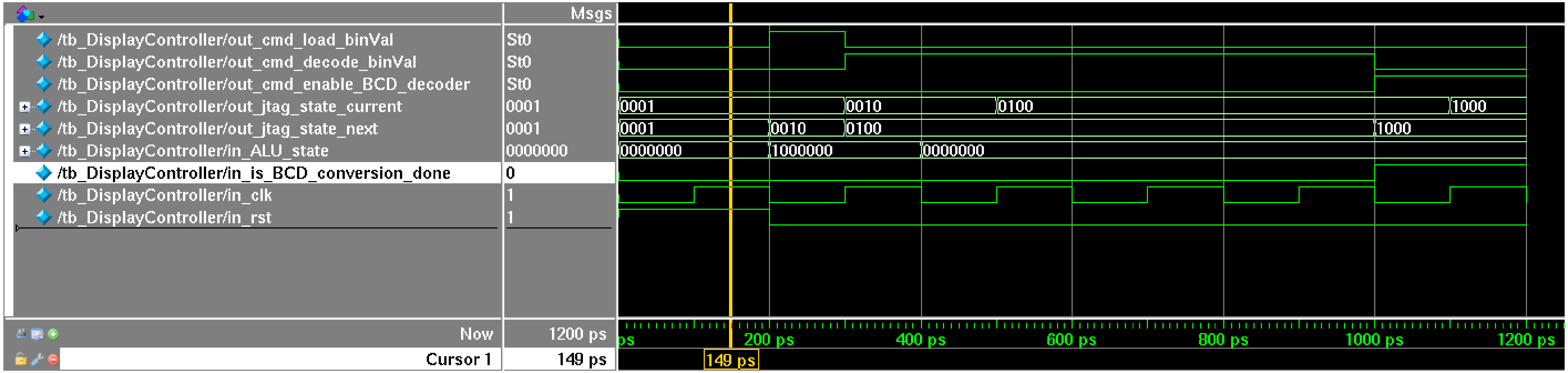
Fuat Deniz YILMAZOK - **2079465**

```
module DisplayController(out_cmd_load_binVal,
                         out_cmd_decode_binVal, // BCD converter control outputs.

                         out_cmd_enable_BCD_decoder, // Output for activating BCD to seven segment decoders.

                         out_jtag_state_current,
                         out_jtag_state_next,

                         in_ALU_state, // Input from present state output of Booth_Multiplier.
                         in_is_BCD_conversion_done, // Input from conversion status of BCD converter.
                         in_clk,
                         in_rst
                        );

parameter CONST_ACTIVATING_STATE_SIZE = 7,
          CONST_COUNT_CONTROLLER_STATE = 4,
          CONST_ACTIVATING_STATE = 7'b100_000_0;

output reg out_cmd_load_binVal,
           out_cmd_decode_binVal,
           out_cmd_enable_BCD_decoder;

input [CONST_ACTIVATING_STATE_SIZE - 1 : 0] in_ALU_state;

input in_is_BCD_conversion_done,
      in_clk,
      in_rst;

localparam CONST_STATE_IDLE = 4'b0001,
           CONST_STATE_INITIALIZE = 4'b0010,
           CONST_STATE_DECODE = 4'b0100,
           CONST_STATE_COMPLETE = 4'b1000;

reg [CONST_COUNT_CONTROLLER_STATE - 1 : 0] state_current, state_next;

output [CONST_COUNT_CONTROLLER_STATE - 1 : 0] out_jtag_state_current,
                                              out_jtag_state_next;

assign out_jtag_state_current = state_current;
assign out_jtag_state_next = state_next;


// State Register.
always@(posedge in_clk, posedge in_rst) begin
```

```
   if(in_rst)
      state_current <= CONST_STATE_IDLE;
   else
      state_current <= state_next;
end


// Next state and output logic.
always@(state_current,
        in_ALU_state,
        in_is_BCD_conversion_done
       ) begin

   // Default operations for don't care conditions of mealy machine.
   state_next <= state_current;
   out_cmd_load_binVal <= 0;
   out_cmd_decode_binVal <= 0;
   out_cmd_enable_BCD_decoder <= 0;

   case(state_current)
      CONST_STATE_IDLE: begin
         if(in_ALU_state == CONST_ACTIVATING_STATE) begin
            out_cmd_load_binVal <= 1;
            state_next <= CONST_STATE_INITIALIZE;
         end
      end

      CONST_STATE_INITIALIZE: begin
         out_cmd_decode_binVal <= 1;
         state_next <= CONST_STATE_DECODE;
      end

      CONST_STATE_DECODE: begin
         if(in_is_BCD_conversion_done) begin
            state_next <= CONST_STATE_COMPLETE;
            out_cmd_enable_BCD_decoder <= 1;
         end
         else out_cmd_decode_binVal <= 1;
      end

      CONST_STATE_COMPLETE: out_cmd_enable_BCD_decoder <= 1; // Preserve mealy output.
   endcase
end
endmodule
```

```verilog
// Testbech for DisplayController.

// Fuat Deniz YILMAZOK - 2079465

module tb_DisplayController();

parameter CONST_ACTIVATING_STATE_SIZE = 7,
          CONST_ACTIVATING_STATE = 7'b100_000_0;

localparam CONST_COUNT_CONTROLLER_STATE = 4;

wire out_cmd_load_binVal,
     out_cmd_decode_binVal,
     out_cmd_enable_BCD_decoder;

wire [CONST_COUNT_CONTROLLER_STATE - 1 : 0] out_jtag_state_current,
                                            out_jtag_state_next;

reg [CONST_ACTIVATING_STATE_SIZE - 1 : 0] in_ALU_state;

reg in_is_BCD_conversion_done,
      in_clk,
      in_rst;

DisplayController DUT(out_cmd_load_binVal,
                      out_cmd_decode_binVal,
                      out_cmd_enable_BCD_decoder,

                      out_jtag_state_current,
                      out_jtag_state_next,

                      in_ALU_state,
                      in_is_BCD_conversion_done,
                      in_clk,
                      in_rst
                     );


localparam CONST_DEFAULT_DELAY = 100,
           CONST_CLOCK_PERIOD = 2 * CONST_DEFAULT_DELAY;

always begin
   in_clk = 0; #CONST_DEFAULT_DELAY;
   in_clk = ~in_clk; #CONST_DEFAULT_DELAY;
end

initial begin
// NOTE: Run this simulation for 6 clock cycles.

   // Start DUT in reset state.
   in_rst = 1; in_ALU_state = 0; in_is_BCD_conversion_done = 0; #CONST_CLOCK_PERIOD;

   // Go to initialize state.
   in_rst = 0; in_ALU_state = CONST_ACTIVATING_STATE; in_is_BCD_conversion_done = 0; #CONST_CLOCK_PERIOD;

   // Go to decode state start decoding for 3 clock cycles.
   in_rst = 0; in_ALU_state = 0; in_is_BCD_conversion_done = 0; #(3*CONST_CLOCK_PERIOD);

   // Go to state complete.
   in_rst = 0; in_ALU_state = 0; in_is_BCD_conversion_done = 1; #CONST_CLOCK_PERIOD;
end

endmodule
```

Simulation Results for **DisplayController**



	Msgs
/tb_DisplayController/out_cmd_load_binVal	St0
/tb_DisplayController/out_cmd_decode_binVal	St0
/tb_DisplayController/out_cmd_enable_BCD_decoder	St0
/tb_DisplayController/out_jtag_state_current	0001
/tb_DisplayController/out_jtag_state_next	0001
/tb_DisplayController/in_ALU_state	0000000
/tb_DisplayController/in_is_BCD_conversion_done	0
/tb_DisplayController/in_clk	1
/tb_DisplayController/in_rst	1
Now	1200 ps
Cursor 1	149 ps

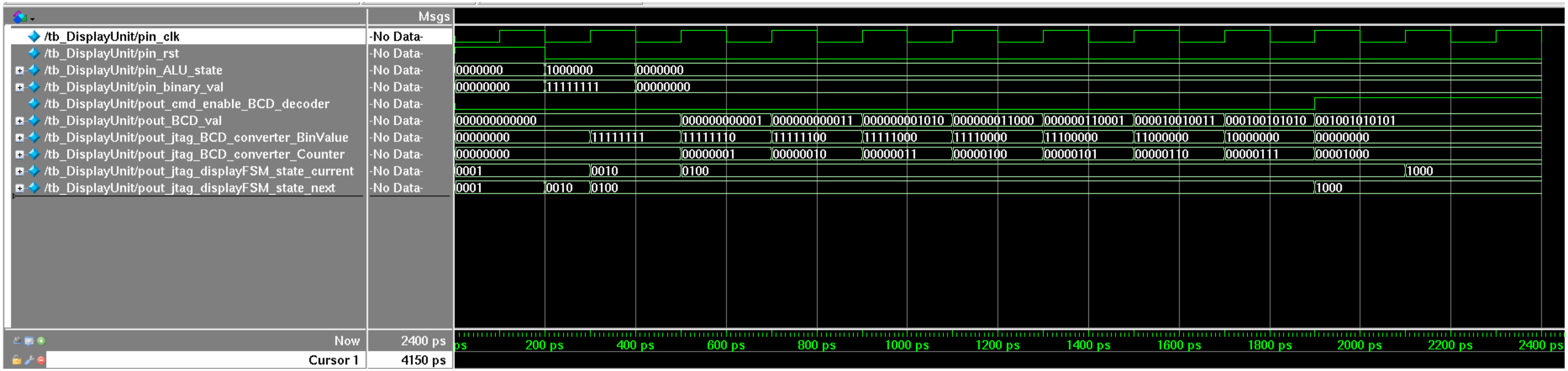
When referred to the state diagram simulation results will make itself clear to reader.
Notice that, between **200ps-300ps**, there is a momentarily change in **out_cmd_load_binVal**, which is independent from clock. This pretty normal because, this is an **mealy machine**. Although this seem like an undesired behavior, this abrupt changes won't be a problem in **DisplayUnit**, because, all inputs to the **DisplayController** in **DisplayUnit** will be synchronized. So, that will eleminate our instability problem.

Note: In this simulation input values are imitated by the testbench, they're not from the actual datapath.

```verilog
// Testbench for DisplayUnit

module tb_DisplayUnit();

reg pin_clk;
reg pin_rst;
reg [6:0] pin_ALU_state;
reg [7:0] pin_binary_val;

wire pout_cmd_enable_BCD_decoder;
wire [11:0] pout_BCD_val;
wire [7:0] pout_jtag_BCD_converter_BinValue;
wire [7:0] pout_jtag_BCD_converter_Counter;
wire [3:0] pout_jtag_displayFSM_state_current;
wire [3:0] pout_jtag_displayFSM_state_next;


DisplayUnit DUT(
   pin_clk,
   pin_rst,
   pin_ALU_state,
   pin_binary_val,
   pout_cmd_enable_BCD_decoder,
   pout_BCD_val,
   pout_jtag_BCD_converter_BinValue,
   pout_jtag_BCD_converter_Counter,
   pout_jtag_displayFSM_state_current,
   pout_jtag_displayFSM_state_next
);


localparam CONST_DEFAULT_DELAY = 100,
           CONST_CLOCK_PERIOD = 2 * CONST_DEFAULT_DELAY,
           CONST_ACTIVATING_STATE = 7'b100_000_0,
           CONST_BINARY_TEST_VALUE = 8'b1111_1010; // -6 or 250 in unsigned binary

always begin
   pin_clk = 0; #CONST_DEFAULT_DELAY;
   pin_clk = ~pin_clk; #CONST_DEFAULT_DELAY;
end

initial begin
   // Start DUT in reset state.
   pin_rst = 1; pin_ALU_state = 0; pin_binary_val = 0; #CONST_CLOCK_PERIOD;

   // Go to initialize state and load data.
   pin_rst = 0; pin_ALU_state = CONST_ACTIVATING_STATE; pin_binary_val = CONST_BINARY_TEST_VALUE; #CONST_CLOCK_PERIOD;

   // Go to decode
   pin_rst = 0; pin_ALU_state = 0; pin_binary_val = 0; #CONST_CLOCK_PERIOD;

end

endmodule
```

Simulation Results for **DisplayUnit**

Verilog code for DUT is generated from schematic, which can be found in the **appendix B**.


Msgs
/tb_DisplayUnit/pin_clk -No Data-
/tb_DisplayUnit/pin_rst -No Data-
/tb_DisplayUnit/pin_ALU_state -No Data- 0000000 1000000 0000000
/tb_DisplayUnit/pin_binary_val -No Data- 00000000 11111111 00000000
/tb_DisplayUnit/pout_cmd_enable_BCD_decoder -No Data-
/tb_DisplayUnit/pout_BCD_val -No Data- 000000000000 000000000001 000000000011 000000001010 000000011000 000000110001 000010010011 000100101010 001001010101
/tb_DisplayUnit/pout_jtag_BCD_converter_BinValue -No Data- 00000000 11111111 11111110 11111100 11111000 11110000 11100000 11000000 10000000 00000000
/tb_DisplayUnit/pout_jtag_BCD_converter_Counter -No Data- 00000000 00000001 00000010 00000011 00000100 00000101 00000110 00000111 00001000
/tb_DisplayUnit/pout_jtag_displayFSM_state_current -No Data- 0001 0010 0100 1000
/tb_DisplayUnit/pout_jtag_displayFSM_state_next -No Data- 0001 0010 0100 1000
Now 2400 ps
Cursor 1 4150 ps
ps
200 ps
400 ps
600 ps
800 ps
1000 ps
1200 ps
1400 ps
1600 ps
1800 ps
2000 ps
2200 ps
2400 ps

Results match with the simulation of **DisplayController** and **Binary_To_BCD_Converter**. When **pout_jtag_BCD_converter_BinValue** followed simulation steps can be followed. At **1900ps**, **pout_cmd_enable_BCD_decoder** is enabled and **pout_BCD_val** is (255 = **0100 0101 0101**). This results verifies the functionality of **DisplayUnit**.

Appendix A | Verilog code generated from Booth_Multiplier.bdf

```
// Copyright (C) 1991-2013 Altera Corporation
// Your use of Altera Corporation's design tools, logic functions 
// and other software and tools, and its AMPP partner logic 
// functions, and any output files from any of the foregoing 
// (including device programming or simulation files), and any 
// associated documentation or information are expressly subject 
// to the terms and conditions of the Altera Program License 
// Subscription Agreement, Altera MegaCore Function License 
// Agreement, or other applicable license agreement, including, 
// without limitation, that your use is for the sole purpose of 
// programming logic devices manufactured by Altera and sold by 
// Altera or its authorized distributors.  Please refer to the 
// applicable agreement for further details.

// PROGRAM     "Quartus II 64-Bit"
// VERSION     "Version 13.0.1 Build 232 06/12/2013 Service Pack 1 SJ Web Edition"
// CREATED     "Fri Sep 25 05:21:43 2020"

module Booth_Multiplier(
   pin_start,
   pin_reset,
   pin_clk,
   pin_Multiplicand,
   pin_Multiplier,
   pout_jtag_QM1,
   pout_jtag_cmd_ars,
   pout_jtag_cmd_add,
   pout_jtag_cmd_sub,
   pout_jtag_Reg_A_SO,
   pout_jtag_Reg_Q_SO,
   pout_A,
   pout_jtag_addsub_result,
   pout_jtag_currentStep,
   pout_jtag_M,
   pout_jtag_mux_out,
   pout_next_state,
   pout_present_state,
   pout_Q
);


input wire  pin_start;
input wire  pin_reset;
input wire  pin_clk;
input wire  [3:0] pin_Multiplicand;
input wire  [3:0] pin_Multiplier;
output wire pout_jtag_QM1;
output wire pout_jtag_cmd_ars;
output wire pout_jtag_cmd_add;
output wire pout_jtag_cmd_sub;
output wire pout_jtag_Reg_A_SO;
output wire pout_jtag_Reg_Q_SO;
output wire [3:0] pout_A;
output wire [3:0] pout_jtag_addsub_result;
output wire [3:0] pout_jtag_currentStep;
output wire [3:0] pout_jtag_M;
output wire [3:0] pout_jtag_mux_out;
output wire [6:0] pout_next_state;
output wire [6:0] pout_present_state;
output wire [3:0] pout_Q;

wire  [3:0] addsub_result;
wire  out_cmd_ars;
wire  out_cmd_init;
wire  out_cmd_rst;
wire  out_cmd_stepDown;
wire  [3:0] out_currentStep;
wire  [3:0] Reg_A_out_PO;
wire  Reg_A_out_SO;
wire  Reg_Q_M1_Q;
wire  [3:0] Reg_Q_out_PO;
wire  Reg_Q_out_SO;
wire  w_Reg_A_loadPar;
wire  SYNTHESIZED_WIRE_7;
wire  [3:0] SYNTHESIZED_WIRE_1;
wire  [3:0] SYNTHESIZED_WIRE_3;
wire  SYNTHESIZED_WIRE_4;
wire  SYNTHESIZED_WIRE_6;

assign   pout_jtag_cmd_add = SYNTHESIZED_WIRE_6;
assign   pout_jtag_cmd_sub = SYNTHESIZED_WIRE_7;
assign   pout_jtag_M = SYNTHESIZED_WIRE_3;
assign   pout_jtag_mux_out = SYNTHESIZED_WIRE_1;
wire  [3:0] GDFX_TEMP_SIGNAL_0;


assign   GDFX_TEMP_SIGNAL_0 = {addsub_result[3],addsub_result[3:1]};


addsub   b2v_AdderSubtractor(
   .op(SYNTHESIZED_WIRE_7),
   .a(Reg_A_out_PO),
   .b(SYNTHESIZED_WIRE_1),


   .result(addsub_result));
   defparam b2v_AdderSubtractor.CONST_DATA_SIZE = 4;


ControlUnit b2v_Fsm(
   .in_start(pin_start),
   .in_Q_0(Reg_Q_out_SO),
   .in_Q_M1(Reg_Q_M1_Q),
   .in_clk(pin_clk),
   .in_rst(pin_reset),
   .in_currentStep(out_currentStep),
   .out_cmd_add(SYNTHESIZED_WIRE_6),
   .out_cmd_sub(SYNTHESIZED_WIRE_7),
   .out_cmd_ars(out_cmd_ars),
   .out_cmd_stepDown(out_cmd_stepDown),
   .out_cmd_init(out_cmd_init),
   .out_cmd_rst(out_cmd_rst),
   .out_currentState(pout_present_state),
   .out_jtag_state_next(pout_next_state));
   defparam b2v_Fsm.CONST_DATA_SIZE = 4;


mux_2_to_1  b2v_MuxComplementer(
   .mux_in_select(SYNTHESIZED_WIRE_7),
   .mux_in_data(SYNTHESIZED_WIRE_3),
   .mux_out(SYNTHESIZED_WIRE_1));
   defparam b2v_MuxComplementer.CONST_DATA_SIZE = 4;


mux2_scalar b2v_MuxScalar_Reg_Q_in_SI_Adjuster(
   .in_data_0(Reg_A_out_SO),
   .in_data_1(addsub_result[0]),
   .in_selection(w_Reg_A_loadPar),
   .out_selected(SYNTHESIZED_WIRE_4));


Universal_Reg  b2v_Reg_A(
   .in_loadPar(w_Reg_A_loadPar),
   .in_SI(Reg_A_out_PO[3]),
   .in_shiftEn(out_cmd_ars),
   .in_clk(pin_clk),
   .in_rst(out_cmd_rst),
   .in_PI(GDFX_TEMP_SIGNAL_0),
   .out_SO(Reg_A_out_SO),
   .out_PO(Reg_A_out_PO));
   defparam b2v_Reg_A.CONST_DATA_SIZE = 4;


Universal_Reg  b2v_Reg_M(
   .in_loadPar(out_cmd_init),


   .in_clk(pin_clk),
   .in_rst(out_cmd_rst),
   .in_PI(pin_Multiplicand),

   .out_PO(SYNTHESIZED_WIRE_3));
   defparam b2v_Reg_M.CONST_DATA_SIZE = 4;


Universal_Reg  b2v_Reg_Q(
   .in_loadPar(out_cmd_init),
   .in_SI(SYNTHESIZED_WIRE_4),
   .in_shiftEn(out_cmd_ars),
   .in_clk(pin_clk),
   .in_rst(out_cmd_rst),
   .in_PI(pin_Multiplier),
   .out_SO(Reg_Q_out_SO),
   .out_PO(Reg_Q_out_PO));
   defparam b2v_Reg_Q.CONST_DATA_SIZE = 4;
```

```


d_ff  b2v_Reg_Q_M1(
   .D(Reg_Q_out_SO),
   .Clk(pin_clk),
   .Load(out_cmd_ars),
   .Rst(out_cmd_rst),
   .Q(Reg_Q_M1_Q));


StepDown_Counter  b2v_StepCounter(
   .in_enable(out_cmd_stepDown),
   .in_clk(pin_clk),
   .in_rst(out_cmd_rst),
   .out_currentStep(out_currentStep));
   defparam b2v_StepCounter.CONST_DATA_SIZE = 4;

assign   w_Reg_A_loadPar = SYNTHESIZED_WIRE_7 ^ SYNTHESIZED_WIRE_6;

assign   pout_jtag_QM1 = Reg_Q_M1_Q;
assign   pout_jtag_cmd_ars = out_cmd_ars;
assign   pout_jtag_Reg_A_SO = Reg_A_out_SO;
assign   pout_jtag_Reg_Q_SO = Reg_Q_out_SO;
assign   pout_A = Reg_A_out_PO;
assign   pout_jtag_addsub_result = addsub_result;
assign   pout_jtag_currentStep = out_currentStep;
assign   pout_Q = Reg_Q_out_PO;

endmodule

```

Appendix B | Verilog code generated from DisplayUnit.bdf

```
// Copyright (C) 1991-2013 Altera Corporation
// Your use of Altera Corporation's design tools, logic functions 
// and other software and tools, and its AMPP partner logic 
// functions, and any output files from any of the foregoing 
// (including device programming or simulation files), and any 
// associated documentation or information are expressly subject 
// to the terms and conditions of the Altera Program License 
// Subscription Agreement, Altera MegaCore Function License 
// Agreement, or other applicable license agreement, including, 
// without limitation, that your use is for the sole purpose of 
// programming logic devices manufactured by Altera and sold by 
// Altera or its authorized distributors.  Please refer to the 
// applicable agreement for further details.

// PROGRAM     "Quartus II 64-Bit"
// VERSION     "Version 13.0.1 Build 232 06/12/2013 Service Pack 1 SJ Web Edition"
// CREATED     "Sun Sep 27 21:29:49 2020"

module DisplayUnit(
   pin_clk,
   pin_rst,
   pin_ALU_state,
   pin_binary_val,
   pout_cmd_enable_BCD_decoder,
   pout_BCD_val,
   pout_jtag_BCD_converter_BinValue,
   pout_jtag_BCD_converter_Counter,
   pout_jtag_displayFSM_state_current,
   pout_jtag_displayFSM_state_next
);


input wire  pin_clk;
input wire  pin_rst;
input wire  [6:0] pin_ALU_state;
input wire  [7:0] pin_binary_val;
output wire pout_cmd_enable_BCD_decoder;
output wire [11:0] pout_BCD_val;
output wire [7:0] pout_jtag_BCD_converter_BinValue;
output wire [7:0] pout_jtag_BCD_converter_Counter;
output wire [3:0] pout_jtag_displayFSM_state_current;
output wire [3:0] pout_jtag_displayFSM_state_next;

wire  SYNTHESIZED_WIRE_0;
wire  SYNTHESIZED_WIRE_1;
wire  SYNTHESIZED_WIRE_2;





Binary_To_BCD_Converter b2v_DisplayUnit_BCD_Converter(
   .in_loadPar(SYNTHESIZED_WIRE_0),
   .in_start(SYNTHESIZED_WIRE_1),
   .in_clk(pin_clk),
   .in_rst(pin_rst),
   .in_binary_val(pin_binary_val),
   .out_isConverted(SYNTHESIZED_WIRE_2),
   .out_BCD_val(pout_BCD_val),
   .out_jtag_tempReg_BinValue(pout_jtag_BCD_converter_BinValue),
   .out_jtag_tempReg_Counter(pout_jtag_BCD_converter_Counter));
   defparam b2v_DisplayUnit_BCD_Converter.CONST_SHIFT_REG_SIZE = 12;
   defparam b2v_DisplayUnit_BCD_Converter.CONST_WORD_SIZE = 8;


DisplayController b2v_DisplayUnit_FSM(
   .in_is_BCD_conversion_done(SYNTHESIZED_WIRE_2),
   .in_clk(pin_clk),
   .in_rst(pin_rst),
   .in_ALU_state(pin_ALU_state),
   .out_cmd_load_binVal(SYNTHESIZED_WIRE_0),
   .out_cmd_decode_binVal(SYNTHESIZED_WIRE_1),
   .out_cmd_enable_BCD_decoder(pout_cmd_enable_BCD_decoder),
   .out_jtag_state_current(pout_jtag_displayFSM_state_current),
   .out_jtag_state_next(pout_jtag_displayFSM_state_next));
   defparam b2v_DisplayUnit_FSM.CONST_ACTIVATING_STATE = 7'b1000000;
   defparam b2v_DisplayUnit_FSM.CONST_ACTIVATING_STATE_SIZE = 7;
   defparam b2v_DisplayUnit_FSM.CONST_COUNT_CONTROLLER_STATE = 4;


endmodule

```