ECE385

Fall 2020

Experiment 5

An 8-Bit Multiplier in SystemVerilog

Name: Zhou Qinren
Zhang Yichi
Lab Section: LA3
TA's Name: Yu Yuqi

Oct. 27^{th} 2020

1 Introduction

In this lab, we will build an 8-bit multiplier to accomplish the function of multiplication and continuous multiplications among 8-bit numbers with a new shift algorithm using SystemVerilog and test it on the FPGA.

2 Pre-Lab Questions

Question: Rework the 8-bit multiplication example presented in the table form at the beginning of this assignment. Use Multiplier $B = 7$, and Multiplicand $S = -59$.

Function	X	A	B	M	Comments for the next step
Clear A, Load B	0	00000000	00000111	1	Since M = 1, multiplicand will be added to A.
ADD	1	11000101	00000111	1	Add S to A since M = 1.
SHIFT	1	11100010	10000011	1	Shift XAB by one bit after ADD complete
ADD	1	10100111	10000011	1	Add S to A since M = 1.
SHIFT	1	11010011	11000001	1	Shift XAB by one bit after ADD complete
ADD	1	10011000	11000001	1	Add S to A since M = 1.
SHIFT	1	11001100	01100000	0	Shift XAB by one bit after ADD complete
SHIFT	1	11100110	00110000	0	M = 0. Shift XAB.
SHIFT	1	11110011	00011000	0	M = 0. Shift XAB.
SHIFT	1	11111001	10001100	0	M = 0. Shift XAB.
SHIFT	1	11111100	11000110	0	M = 0. Shift XAB.
SHIFT	1	11111110	01100011	0	Done.

3 Written Description and Diagrams of Multiplier Circuit

3.1 Summary of Operation

First, we flip the switches to load register B. When the switches are all set up, we press key 2 (ClearA_LoadB), so that the value of the switches is loaded into register B, and A is set to 0. Then we flip the switches to represent the multiplicand S. After that, we press key 3 (Run), the multiplication begins.

The multiplication consists of three parts, ADD, SHIFT and SUB. Add S to A if the least significant bit of B is 1 and do nothing if 0. If the addition generates a carry out, store it in X. Then Shift XAB anthemically by one bit. After the first seven shifts, subtract S from A if the LSB of B is 1, and do nothing if 0. Then shift XAB by one bit, and we are done. The correct result stores in AB (16 bits).

3.2 Top Level Block Diagram

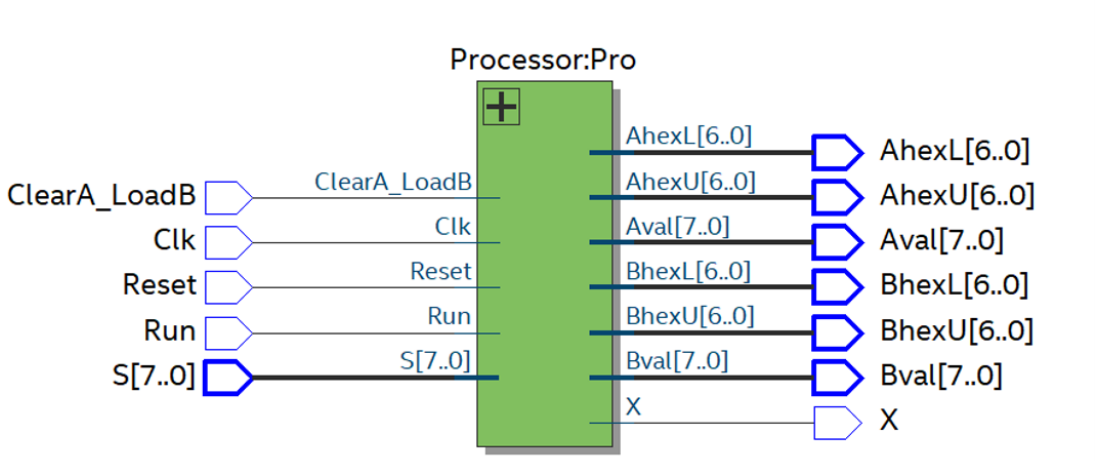

Figure 1: top level block diagram

3.3 Written Description of .sv Modules

```
module reg_8 (input  logic Clk, Reset, Shift_In, Load, Shift_En,
              input  logic [7:0]  D,
              output logic [7:0]  Data_Out);

    always_ff @ (posedge Clk)
    begin
        if (Reset)
            Data_Out <= 8'h0;
        else if (Load)
            Data_Out <= D;
        else if (Shift_En)
        begin
            Data_Out <= { Shift_In, Data_Out[7:1] };
        end
    end

endmodule
```

Figure 2: Reg_8

Module: Reg_8
Inputs: Clk, Reset, Shift_In, Load, Shift_En, [7:0] D
Outputs: [7:0] Data_Out
Description: This is a positive-edge triggered 16-bit register with synchronous reset, load and shift.
Purpose: This module is used to implement the low-level component of register unit.

```
module Register_unit (input  logic Clk, Clr_Ld, Shift_En, Shift_In, ResetA_B, Load_A, Reset_A,
                      input  logic [7:0]  D,
                      output logic [7:0]  A,
                      output logic [7:0]  B
                      );

        logic reset;
        assign reset = Clr_Ld | ResetA_B | Reset_A;

        reg_8  reg_A (.*, .Reset(reset), .Load(Load_A), .Data_Out(A));

        reg_8  reg_B (.*, .Shift_In(A[0]), .Load(Clr_Ld), .Reset(ResetA_B), .Data_Out(B));
endmodule
```

Figure 3: Register_unit

Module: Register_unit
Inputs: Clk, Clr_Ld, Shift_En, Shift_In, ResetA_B, LoadA, Reset_A, [7:0] D
Outputs: [7:0] A, [7:0] B
Description: This is a positive-edge triggered 2 16-bit-register system. The inputs oversee the behavior of the two registers A and B. All input signals are synchronized. Clr_Ld indicates clearing register A and load register B. Shift_En indicates AB shifts by one bit. ResetA_B indicates reset register A and B. LoadA indicates load register A, leaving B unchanged. Reset A indicates reset register A only, leaving B unchanged.
Purpose: This module is used to implement the registers A and B.

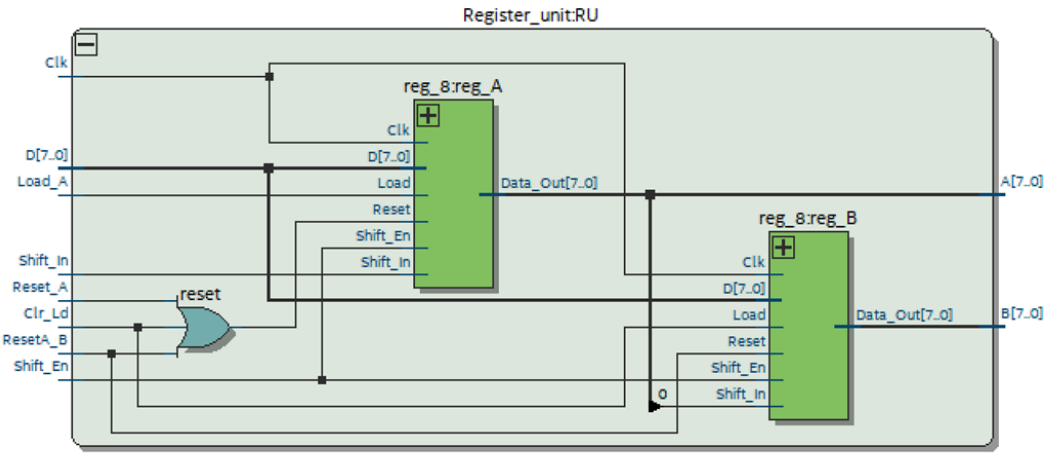

Figure 4: Register_unit RTL

```
module full_adder (input logic A, B, C_In,
                   output logic S, C_Out);

        assign S = A ^ B ^ C_In;
        assign C_Out = (A & B) | (B & C_In) | (A & C_In);
endmodule
```

Figure 5: full_adder

Module: full_adder
Inputs: A, B, C_In
Outputs: S, C_Out
Description: A simple full adder for 1-bit inputs.
Purpose: This module is used to implement the low-level component of a 9-bit adder.

```
module adder9 (input logic [8:0] A, B,
               input logic C_In,
               output logic [8:0] S,
               output logic C_Out);

        logic [7:0] c;

        full_adder FA0 (.A(A[0]), .B(B[0]), .C_In(C_In), .S(S[0]), .C_Out(c[0]));
        full_adder FA1 (.A(A[1]), .B(B[1]), .C_In(c[0]), .S(S[1]), .C_Out(c[1]));
        full_adder FA2 (.A(A[2]), .B(B[2]), .C_In(c[1]), .S(S[2]), .C_Out(c[2]));
        full_adder FA3 (.A(A[3]), .B(B[3]), .C_In(c[2]), .S(S[3]), .C_Out(c[3]));

        full_adder FA4 (.A(A[4]), .B(B[4]), .C_In(c[3]), .S(S[4]), .C_Out(c[4]));
        full_adder FA5 (.A(A[5]), .B(B[5]), .C_In(c[4]), .S(S[5]), .C_Out(c[5]));
        full_adder FA6 (.A(A[6]), .B(B[6]), .C_In(c[5]), .S(S[6]), .C_Out(c[6]));
        full_adder FA7 (.A(A[7]), .B(B[7]), .C_In(c[6]), .S(S[7]), .C_Out(c[7]));

        full_adder FA8 (.A(A[8]), .B(B[8]), .C_In(c[7]), .S(S[8]), .C_Out(C_Out));
endmodule
```

Figure 6: adder9

Module: adder9
Inputs: [8:0] A, [8:0] B, C_In
Outputs: [8:0] S, C_Out
Description: A 9-bit adder made of 9 full adders.
Purpose: This module is used to add the contents in registers S and A.

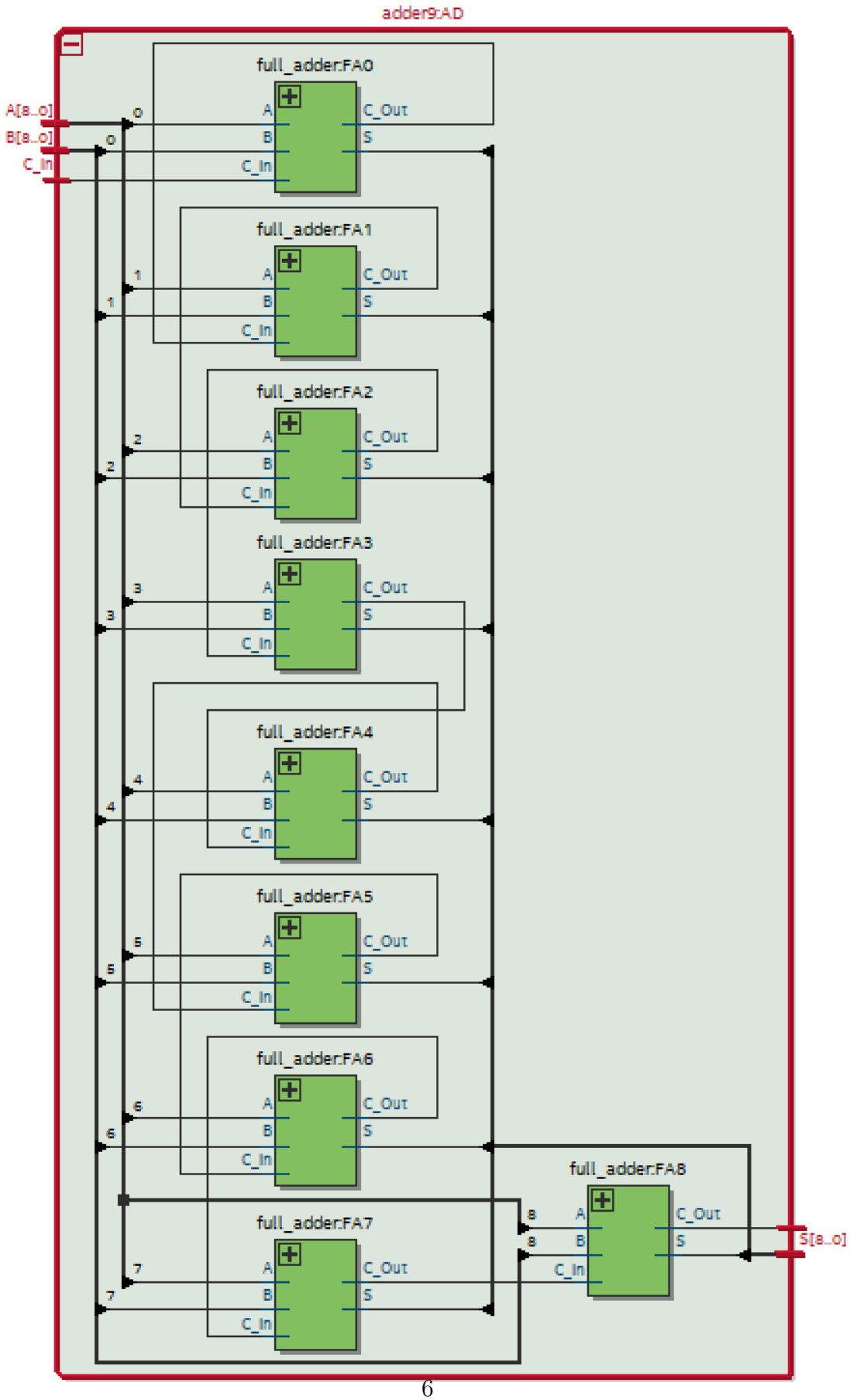

Figure 7: adder9 RTL

```
module multiplier (input logic Clk, Shift, Add, Sub, ResetA_B, Clr_Ld, Reset_A,
                   input logic [7:0] D,
                   output logic [7:0] A,
                   output logic [7:0] B,
                   output logic X
                   );

        logic X0, temp_X, Load_A, C_In;
        logic [7:0] addition_result, Data;
        logic [8:0] S_addend;

        Register_unit RU (.*, .Shift_En(Shift), .Shift_In(X), .Load_A(Load_A), .D(Data));
        adder9 AD (.A({A[7], A}), .B(S_addend), .C_In(C_In), .S({temp_X, addition_result}), .C_out());

        assign X = X0;


        always_ff @ (posedge Clk)
           begin
              if (Shift)
                 X0 <= X0;
              if ((Add) || (Sub))
                 X0 <= temp_X;
           end


        always_comb
           begin
              if (!((Add) || (Sub))) begin
                 Data = D;
                 Load_A = 1'b0;
                 S_addend = 9'h000;
                 C_In = 1'b0;
              end
              else begin
                 if (B[0] == 0) begin
                    S_addend = 9'h000;
                    C_In = 1'b0;
                 end
                 else begin
                    if (Add) begin
                       S_addend = {D[7], D};
                       C_In = 1'b0;
                    end
                    else begin
                       if (Add) begin
                          S_addend = {D[7], D};
                          C_In = 1'b0;
                       end
                       else begin
                          S_addend = ~{D[7], D};
                          C_In = 1'b1;
                       end
                    end
                    Load_A = 1'b1;-
                    Data = addition_result;
                 end
              end

endmodule
```

Figure 8: multiplier

Module: multiplier
Inputs: Clk, Shift, Add, Sub, ResetA_B, Clr_Ld, Reset_A, [7:0] D
Outputs: [7:0] A, [7:0] B, X
Description: This module consists of a register unit, a 9-bit adder, an extra bit-storer X and some combinational logic. The system shifts XAB by one bit if Shift signal is high. It adds S and A if Add signal is high. It subtracts S from A if Sub signal is high. Note when performing Add and Sub, S and A are firstly sign-extended to 9 bits. ResetA_B indicates reset A and B, Reset_A indicates reset A only, and Clr_Ld indicates clear A and Load B.
Purpose: This module implements the multiplication function, and several behaviors about the registers such as load and reset.

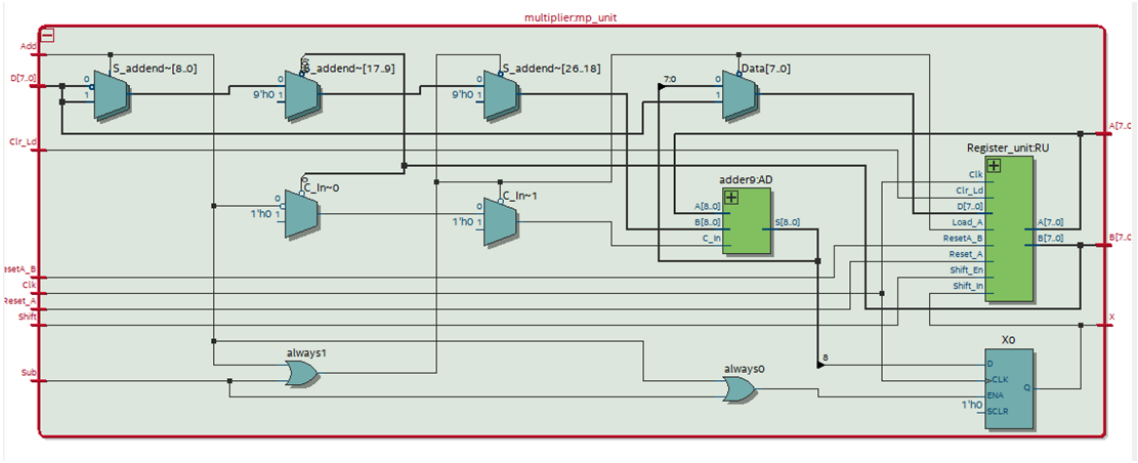

Figure 9: multiplier RTL

```
module Control (input  logic Clk, Reset, Run, ClearA_LoadB,
                output logic Clr_Ld, Shift, Add, Sub, ResetA_B, Reset_A);

    enum logic [3:0] {A, B, C, D, E, F}   curr_state, next_state;

    logic [3:0] counter, next_counter, temp;

    adder3 counter_adder (.A(counter), .B(4'b01), .S(temp));


    // below is the state representation
    always_ff @ (posedge Clk)
    begin
         if (Reset)
         begin
            curr_state <= A;
            counter <= 4'h0;
         end else
         begin
            curr_state <= next_state;
            counter <= next_counter;
         end
    end


    // below is the next state logic
   always_comb
    begin
        case (curr_state)
            // A is the initial state
            A :
            begin
               if (Run)
                  next_state = F;
               else
                  next_state = curr_state;
            end

            F:
            begin
               next_state = B;
            end

            // B is the Add state
            B :    next_state = C;
```

Figure 10: control1

```
    // C is the Shift state
    C :
    begin
            unique case (counter)
              4'b1111: next_state = E;
              4'b1101: next_state = D;
              default: next_state = B;
            endcase
    end

    // D is the Sub state
    D :    next_state = C;

    // E is the end state
    E :
    begin
            if (~Run)
                next_state = A;
            else
                next_state = E;
    end
endcase


// Assign outputs based on state and inputs
unique case (curr_state)
    A:
    begin
       Shift = 1'b0;
       Add = 1'b0;
       Sub = 1'b0;
       next_counter = 4'h0;
       Reset_A = 1'b0;
    end

    F:
    begin
       Shift = 1'b0;
       Add = 1'b0;
       Sub = 1'b0;
       next_counter = 4'h0;
       Reset_A = 1'b1;
    end
```

Figure 11: control2

```
            B:
            begin
                Shift = 1'b0;
                Sub = 1'b0;
                next_counter = temp;
                Add = 1'b1;
                Reset_A = 1'b0;
            end

            C:
            begin
                Shift = 1'b1;
                Add = 1'b0;
                Sub = 1'b0;
                next_counter = temp;
                Reset_A = 1'b0;
            end

            D:
            begin
                Shift = 1'b0;
                Add = 1'b0;
                next_counter = temp;
                Sub = 1'b1;
                Reset_A = 1'b0;
            end

            E:
            begin
                Shift = 1'b0;
                Add = 1'b0;
                Sub = 1'b0;
                next_counter = counter;
                Reset_A = 1'b0;
            end
        endcase
    end

    assign Clr_Ld = ClearA_LoadB;
    assign ResetA_B = Reset;

endmodule
```

Figure 12: control3

Module: Control
Inputs: Clk, Reset, Run, ClearA_LoadB
Outputs: Clr_Ld, Shift, Add, Sub, ResetA_B, Reset_A
Description: This module is essentially a finite state machine. We use enum to create 6 states, representing initial, reset_A, Add, Shift, Sub, and halt. The state is reset to initial once reset is pressed. Once Run is pressed, the initial state transits to reset_A state to clear the content in A. Then it goes to Add, Shift or Sub state depending on the least significant bit of register B. We made use of a counter to count the times of shifts. After the seventh shift, the machine decides whether to go to Sub state. After the eighth shift, the machine stays in the End state if Run is still pressed, and it goes to initial state if Run is released.
Purpose: This module is the control unit of the project. It deals with the input information and decides the transition and generates the corresponding outputs.

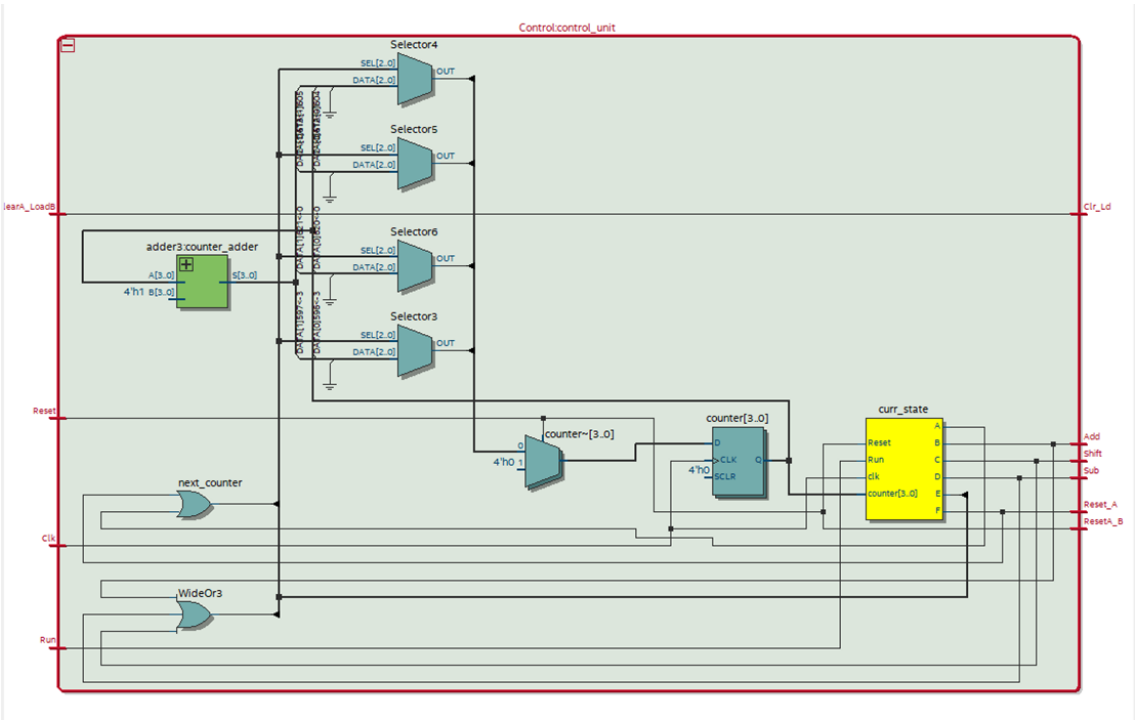

Figure 13: control_RTL

```
module Processor (input logic   Clk,
                                Reset,
                                Run,
                                ClearA_LoadB,
                  input  logic [7:0]  S,
                  output logic [6:0] AhexU, AhexL, BhexU, BhexL,
                  output logic [7:0] Aval, Bval,
                  output logic X);

        logic Clr_Ld, Shift, Add, Sub, ResetA_B, Reset_A;
        logic Reset_SH, Run_SH, ClearA_LoadB_SH;
        logic [7:0] D_S;

        multiplier mp_unit (.*, .D(S), .A(Aval), .B(Bval));

        Control control_unit (.*, .Reset(Reset_SH), .Run(Run_SH), .ClearA_LoadB(ClearA_LoadB_SH));

        sync button_sync[2:0] (Clk, {~Reset, ~Run, ~ClearA_LoadB}, {Reset_SH, Run_SH, ClearA_LoadB_SH});
        sync Din_sync[7:0] (Clk, S, D_S);

        HexDriver       HexAL (
                               .In0(Aval[3:0]),
                               .Out0(AhexL) );
        HexDriver       HexBL (
                               .In0(Bval[3:0]),
                               .Out0(BhexL) );

        HexDriver       HexAU (
                               .In0(Aval[7:4]),
                               .Out0(AhexU) );
        HexDriver       HexBU (
                               .In0(Bval[7:4]),
                               .Out0(BhexU) );

endmodule
```

Figure 14: Processor

Module: Processor
Inputs: Clk, Reset, Run, ClearA_LoadB, [7:0] S
Outputs: [6:0] Ahexu, [6:0] AhexL, [6:0] BhexU, [6:0] BhexL, [7:0] Aval, [7:0] Bval, X

Description: This module consists of control unit, multiplier, hexdriver and synchronizer. Control unit and multiplier processes the inputs and generated outputs. Hexdriver displays the output information. Synchronizers are used to synchronize the user input.
Purpose: Process the input information and generates the desired outputs.

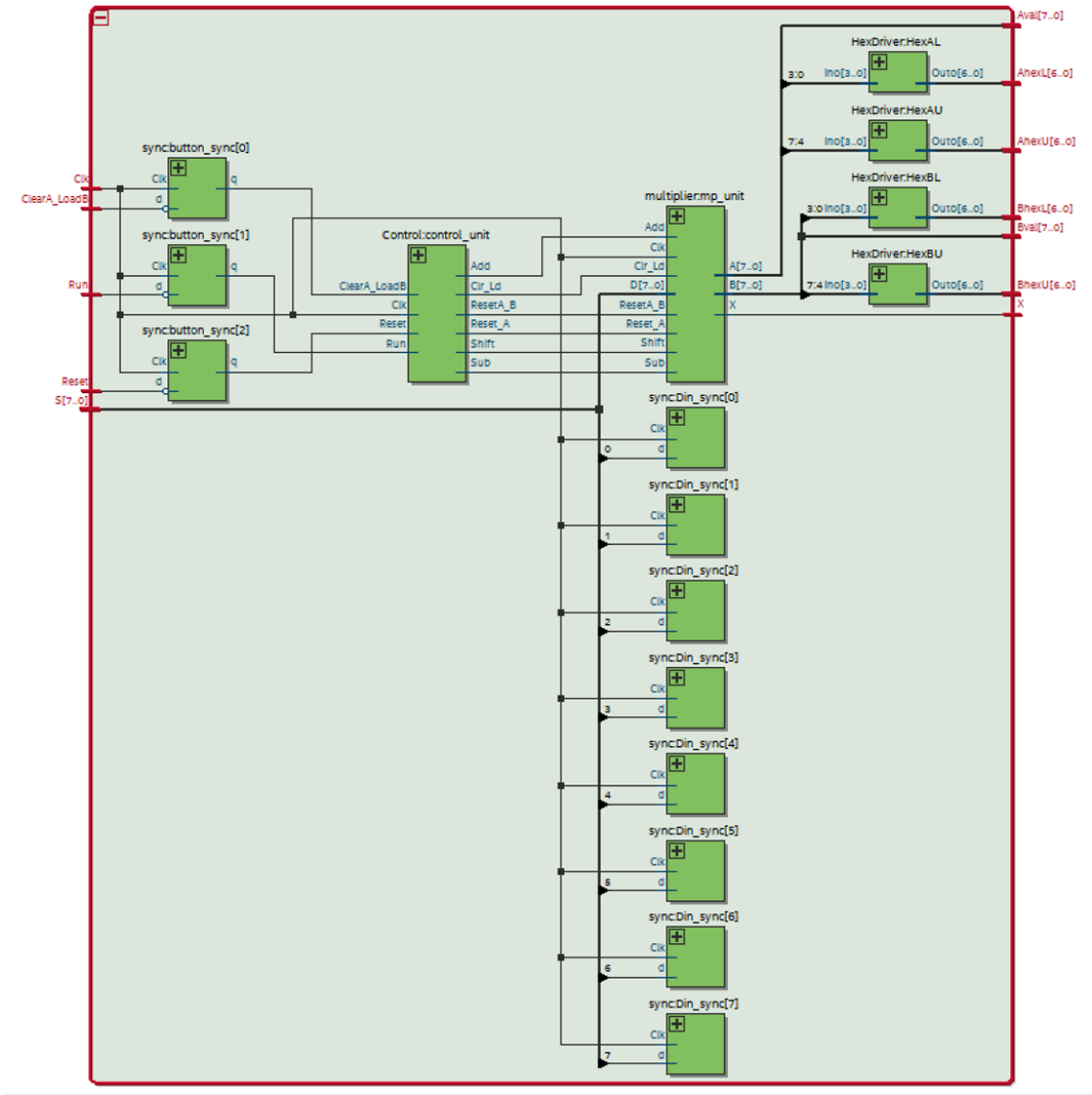

Figure 15: Processor RTL

3.4 State Diagram for Control Unit

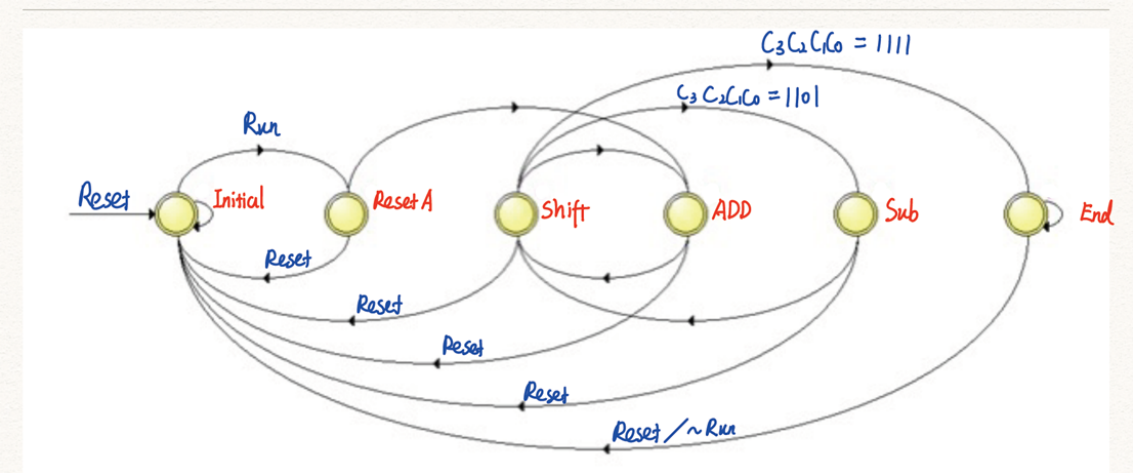

Figure 16: State Diagram for Control Unit. A counter of 4 bits counts the times of shift operations that have been done.

4 Annotated Pre-lab Simulation Waveforms

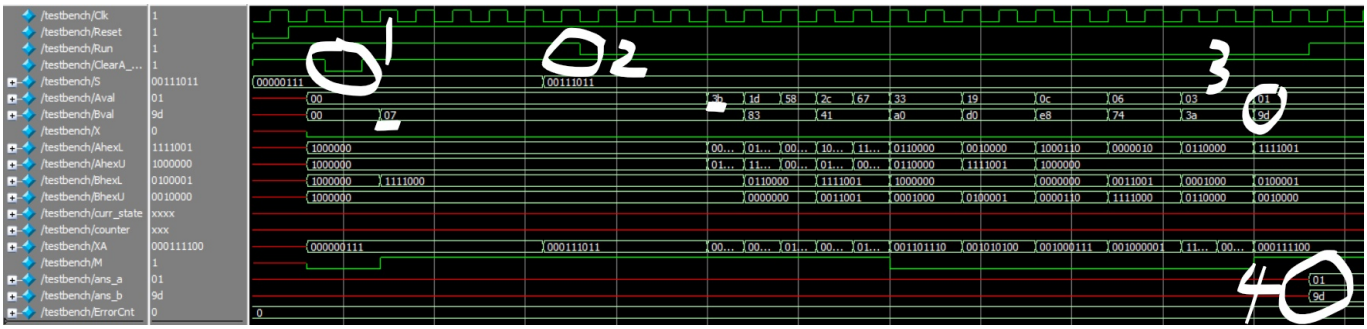

Figure 17: test case1

This is test case 1, where we test two positive numbers' multiplication, $7 * 59 = 413$. The multiplier loads B from S at 1 as the ClearA_LoadB is pressed. Then it loads A from S at 2 as the Run is pressed. After the calculation, the result we get at 3 is the same as the result we know at 4. Therefore, it passed the test case 1.

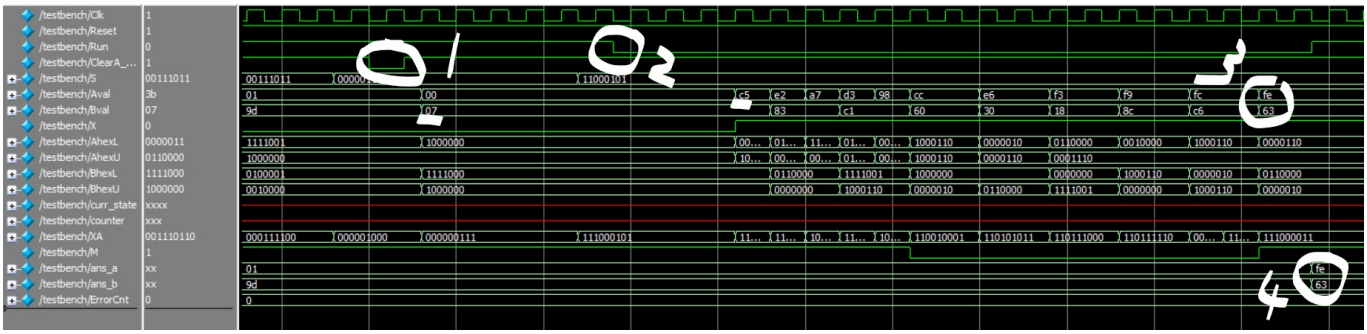

Figure 18: test case2

This is test case 2, where we test a positive number times a negative number, $7 * (-59) = -413$. The multiplier loads B from S at 1 as the ClearA_LoadB is pressed. Then it loads A from S at 2 as the Run is pressed. After the calculation, the result we get at 3 is the same as the result we know at 4. Therefore, it passed the test case 2.

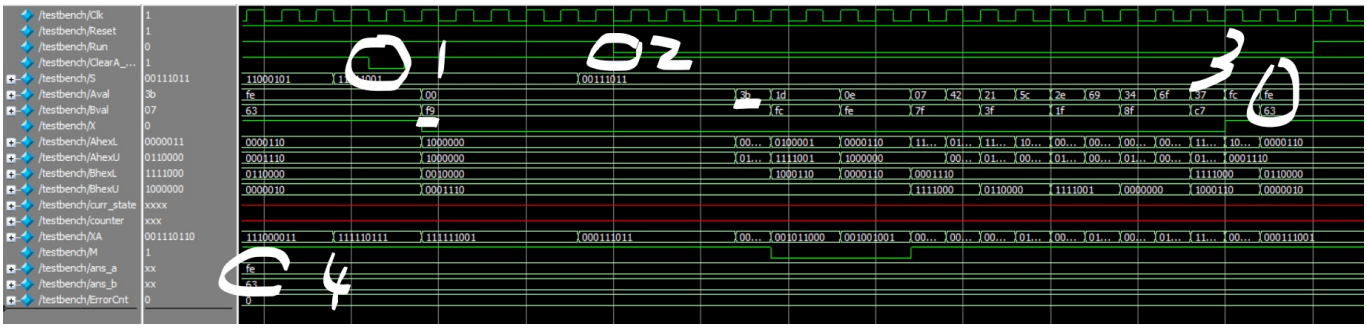

Figure 19: test case3

This is test case 3, where we test a negative number times a positive number, $-7 * 59 = -413$. The multiplier loads B from S at 1 as the ClearA_LoadB is pressed. Then it loads A from S at 2 as the Run is pressed. After the calculation, the result we get at 3 is the same as the result we know at 4. Therefore, it passed the test case 3.

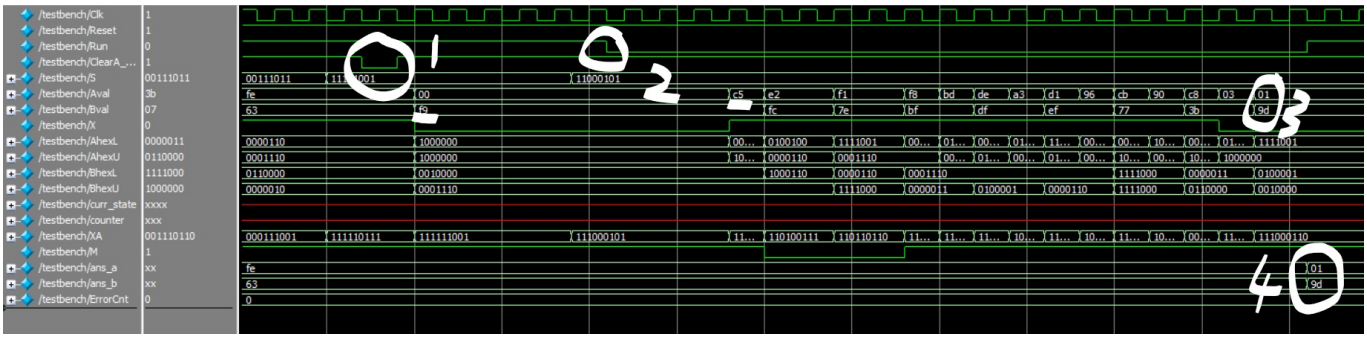

Figure 20: test case4

This is test case 4, where we test two negative numbers' multiplication, $-7 * (-59) = 413$. The multiplier loads B from S at 1 as the ClearA_LoadB is pressed. Then it loads A from S at 2 as

the Run is pressed. After the calculation, the result we get at 3 is the same as the result we know at 4. Therefore, it passed the test case 4.

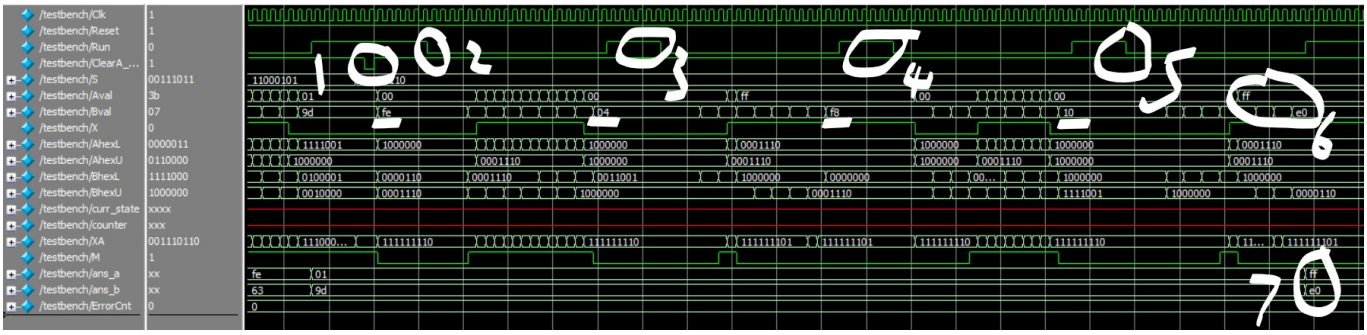

Figure 21: test case5

This is test case 5, where we test continuous multiplication, $(-2)*(-2)*(-2)*(-2)*(-2) = -32$. The multiplier loads B from S at 1 as the ClearA_LoadB is pressed. Then it loads A from S at 2, 3, 4, 5 as the Run is pressed. We can see the result in AB changes from fe(-2) to 0004(4) to fff8(-8) to 0010(16) to ffe0(-32) at 6. The result we get at 6 is the same as the result we know at 7. Therefore, it passed the test case 5.

5 Post-Lab Questions

Question: What is the purpose of the X register. When does the X register get set/cleared?

Answer: X register stores the first bit of A. Therefore, it is used in shift process. When shifting, A shift in X as the first bit. X is set when the add process occurs $XA = A + S$ and cleared when the reset button is pressed.

Question: What are the limitations of continuous multiplications? Under what circumstances will the implemented algorithm fail?
Answer: Continuous multiplications might result in overflow since the registers can only store 8-bit multiplicand and 8-bit multiplier. However, continuous multiplications would take the result of the last multiplication as a multiplier which might be over 8 bits. Thus, when the result of the multiplication has already reached over 8 bits, the following continuous multiplication would fail.
Question: What are the advantages (and disadvantages?) of the implemented multiplication algorithm over the pencil-and-paper method discussed in the introduction?

Answer: We can use computers to do the repetitive tasks for us to make our lives easier.

LUT	82
DSP	0
Memory	0
Flip-Flop	39
Frequency	233.64 MHz
Static Power	98.55mW
Dynamic Power	2.77mW
Total Power	150.64mW

Table 1: Design statistics table for the multiplier.

Question: Write down several ideas on how the maximum frequency of your design could be increased or the gate count could be decreased.

Answer: We could use CLA or CSA to replace the 9-bit CRA we use in the design calculating $XA = A + S$ to increase the maximum frequency since CLA and CSA could do the calculation parallelly. Also, if we could find a more efficient algorithm to do the multiplications, it will enhance the maximum frequency and probably the gate numbers. In addition, if we use Mealy machine instead of Moore machine, the gate count will be decreased.

6 Conclusion

6.1 Functionality

In this lab, we develop our own state machine to build a 8-bit multiplier to do the multiplication and continuous multiplications among 8-bit numbers. We use add-shift method to implement the multiplications.

6.2 About Lab Manual

We hope the lab manual can introduce more about the SystemVerilog.