ECE 385

Fall 2021

Experiment # 5

An 8-Bit Multiplier in SystemVerilog

Xu Ke / Zhu Xiaohan

LA4/Thursday & 18:00-20:50 Huang Tianhao

1. Introduction

In this lab, we complete a 2's complement 8-bit Multiplier Unit in SystemVerilog using logic operation. This Multiplier can multiply two 8-bit signed numbers. On the FPGA board, we can enter two operator and one operation will be executed when the Run press button is pressed.

2. Pre-lab 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. Note that this is different than the case when B = -59 and S = 7.

First, we transform these two numbers in 2's complement:

7 = 0000 0111

-59 = 1100 0101

Function	X	A (-59)	B (7)	M	Comments for the next step
Clear A, Load B	0	0000 0000	0000 0111	1	Since M = 1, multiplicand (available from switches S) will be added to A.
ADD	1	1100 0101	0000 0111	1	Shift XAB by one bit after ADD complete
SHIFT	1	1110 0010	1000 0011	1	Add S to A since M = 1.
ADD	1	1010 0111	1000 0011	1	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1101 0011	1100 0001	1	Add S to A since M = 1.
ADD	1	1001 1000	1100 0001	1	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1100 0011	0110 0000	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1110 0110	0011 0000	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1111 0011	0001 1000	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1111 1001	1000 1100	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1111 1110	1100 0110	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1111 1110	0110 0011	1	8 th shift done. Stop. 16-bit Product in AB.

3. Written description and diagrams of multiplier circuit

Summary of operation

a) How operands are loaded:

We will press **Reset**, Registers X, A and B will be set to 0. Then, we load the multiplier to Register B by setting the switches on board and press **ClearA_LoadB**, which will clear the Registers X and A as well as load B. After that, we set the switches to represent the multiplicand and press the **Run**, the multiplicand will be put into adder.

b) How the multiplier computes its result:

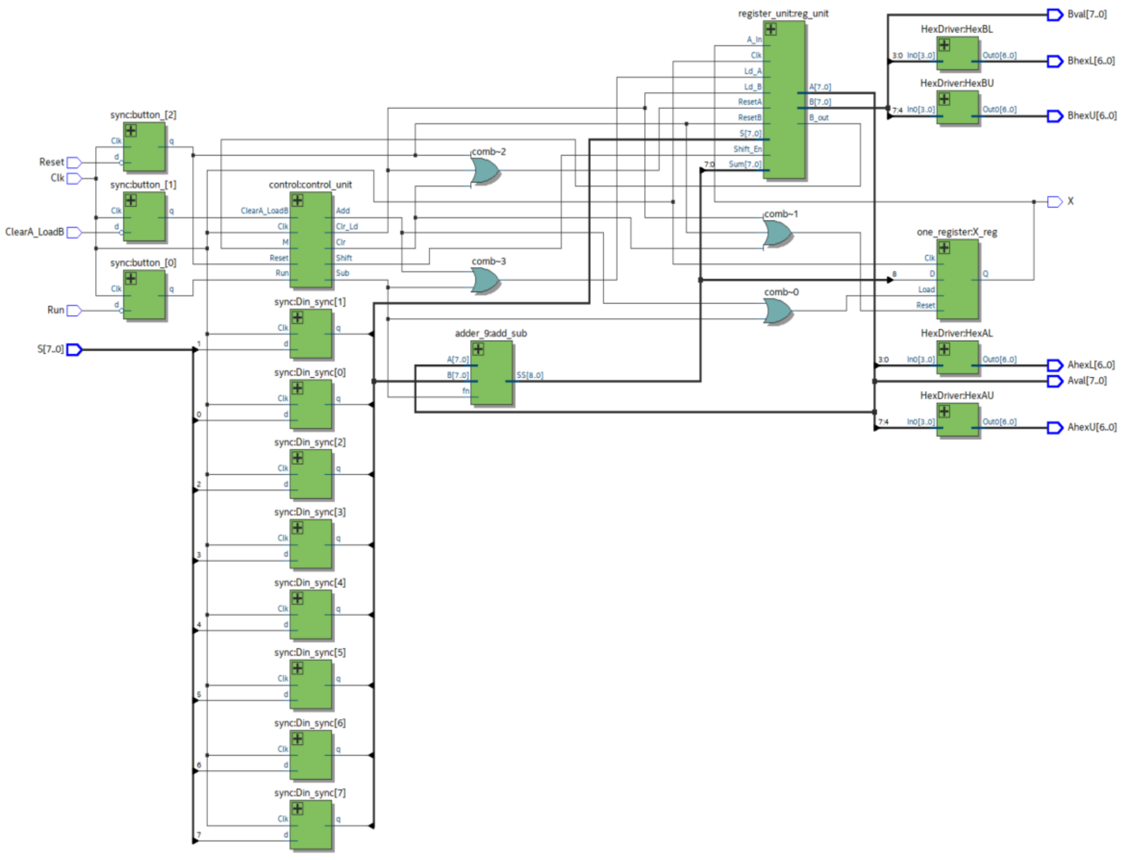
This part is mostly based on the control unit we designed. Basically, we multiply two operands using a way that is similar to the way that human do multiplication instead of adding the multiplier many times. So, for two 8-bit signed numbers, we will right shift them for eight times. Each time if the last bit of B (B[0] = M) is 1, we will add the multiplier to A then shift, if not, we just shift XAB. To complete these steps, we designed the state machine which can automatically switch between states according to the value of M. Notice that the last ADD state is actually SUB, for B[7] is a sign bit.

In our design, control unit specify the FSM model, in each state it will give the control signals to push forward the tasks of adder or registers, so that the multiplier will compute the result well. For example,

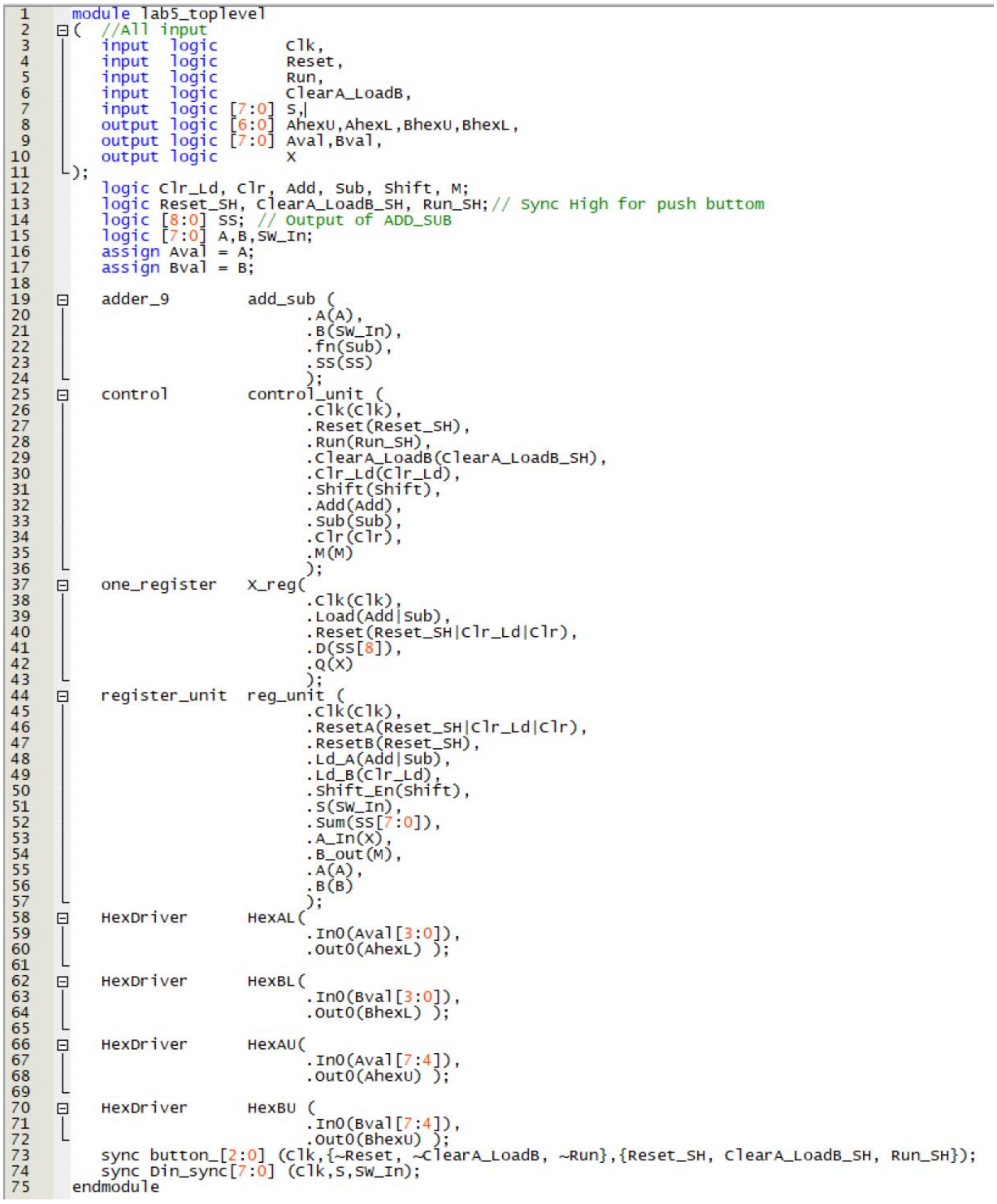
c) How the result is stored:

In our design, we have an 8-bit register to store A and B and a one bit register to store X. The final multiply result will be store in B as a multiplicand for further multiply.

Top Level Block Diagram



Description of .sv Modules



```
module lab5_toplevel
( //All input
    input  logic        Clk,
    input  logic        Reset,
    input  logic        Run,
    input  logic        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, Clr, Add, Sub, Shift, M;
    logic Reset_SH, ClearA_LoadB_SH, Run_SH;// Sync High for push buttom
    logic [8:0] SS; // Output of ADD_SUB
    logic [7:0] A,B,SW_In;
    assign Aval = A;
    assign Bval = B;

    adder_9        add_sub (
                        .A(A),
                        .B(SW_In),
                        .fn(Sub),
                        .SS(SS)
                        );
    control        control_unit (
                        .Clk(Clk),
                        .Reset(Reset_SH),
                        .Run(Run_SH),
                        .ClearA_LoadB(ClearA_LoadB_SH),
                        .Clr_Ld(Clr_Ld),
                        .Shift(Shift),
                        .Add(Add),
                        .Sub(Sub),
                        .Clr(Clr),
                        .M(M)
                        );
    one_register   X_reg(
                        .Clk(Clk),
                        .Load(Add|Sub),
                        .Reset(Reset_SH|Clr_Ld|Clr),
                        .D(SS[8]),
                        .Q(X)
                        );
    register_unit  reg_unit (
                        .Clk(Clk),
                        .ResetA(Reset_SH|Clr_Ld|Clr),
                        .ResetB(Reset_SH),
                        .Ld_A(Add|Sub),
                        .Ld_B(Clr_Ld),
                        .Shift_En(Shift),
                        .S(SW_In),
                        .Sum(SS[7:0]),
                        .A_In(X),
                        .B_out(M),
                        .A(A),
                        .B(B)
                        );
    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) );
    sync button_[2:0] (Clk,{~Reset, ~ClearA_LoadB, ~Run},{Reset_SH, ClearA_LoadB_SH, Run_SH});
    sync Din_sync[7:0] (Clk,S,SW_In);
endmodule
```

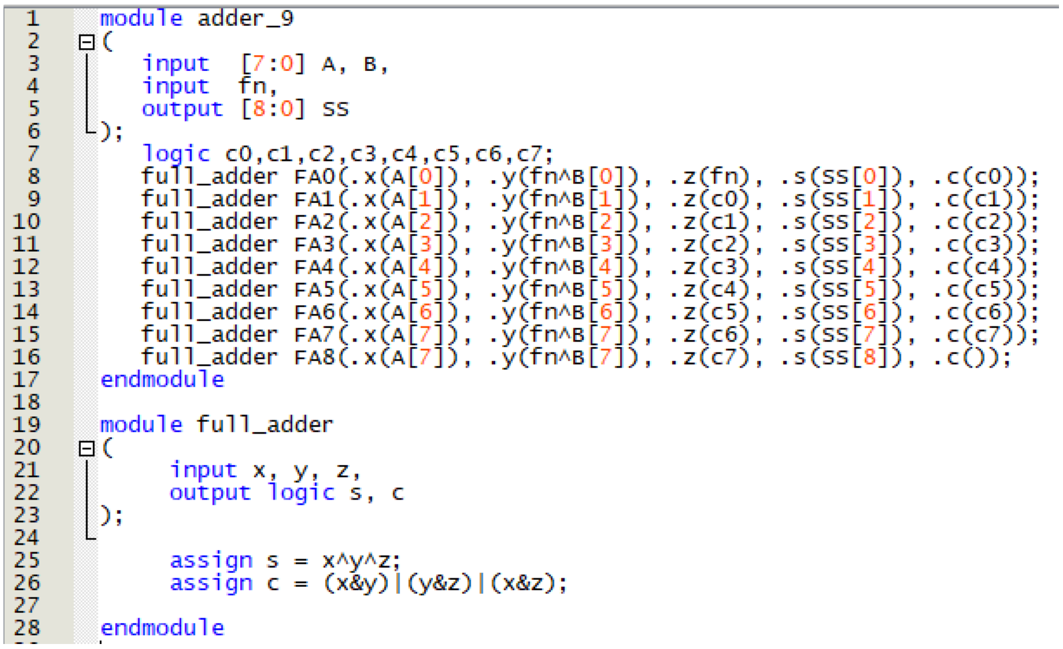
Module: lab5_toplevel.sv

Inputs: Clk, Reset, Run, ClearA_LoadB, [7:0] S

Outputs: X, [6:0] AhexL, AhexU, BhexL, BhexU

Description: This is the top level of our design, which clarify the input and output of other modules so that the circuit will work, it also reads the data input from switches and outputs it on the display. The processor can be seen as the heart of the circuitry meaning it connects everything and brings the circuit to life.

Purpose: Practically it's where all the wiring goes. And it allows us to run and test everything.



```
module adder_9
(
    input  [7:0] A, B,
    input  fn,
    output [8:0] SS
);
    logic c0,c1,c2,c3,c4,c5,c6,c7;
    full_adder FA0(.x(A[0]), .y(fn^B[0]), .z(fn), .s(SS[0]), .c(c0));
    full_adder FA1(.x(A[1]), .y(fn^B[1]), .z(c0), .s(SS[1]), .c(c1));
    full_adder FA2(.x(A[2]), .y(fn^B[2]), .z(c1), .s(SS[2]), .c(c2));
    full_adder FA3(.x(A[3]), .y(fn^B[3]), .z(c2), .s(SS[3]), .c(c3));
    full_adder FA4(.x(A[4]), .y(fn^B[4]), .z(c3), .s(SS[4]), .c(c4));
    full_adder FA5(.x(A[5]), .y(fn^B[5]), .z(c4), .s(SS[5]), .c(c5));
    full_adder FA6(.x(A[6]), .y(fn^B[6]), .z(c5), .s(SS[6]), .c(c6));
    full_adder FA7(.x(A[7]), .y(fn^B[7]), .z(c6), .s(SS[7]), .c(c7));
    full_adder FA8(.x(A[7]), .y(fn^B[7]), .z(c7), .s(SS[8]), .c());
endmodule

module full_adder
(
        input x, y, z,
        output logic s, c
);

        assign s = x^y^z;
        assign c = (x&y)|(y&z)|(x&z);

endmodule
```

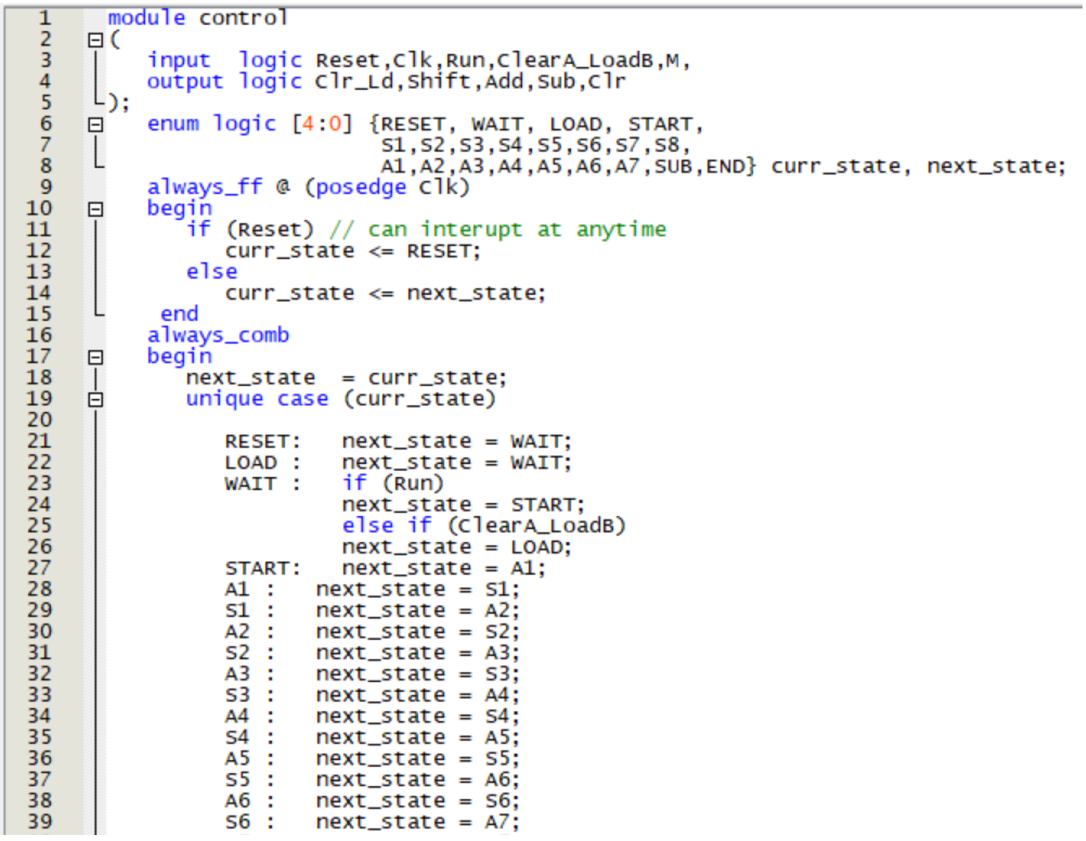
Module: adder_9.sv (Also contain a full adder module: *full_adder*)

Inputs: [7:0] A, B, fn

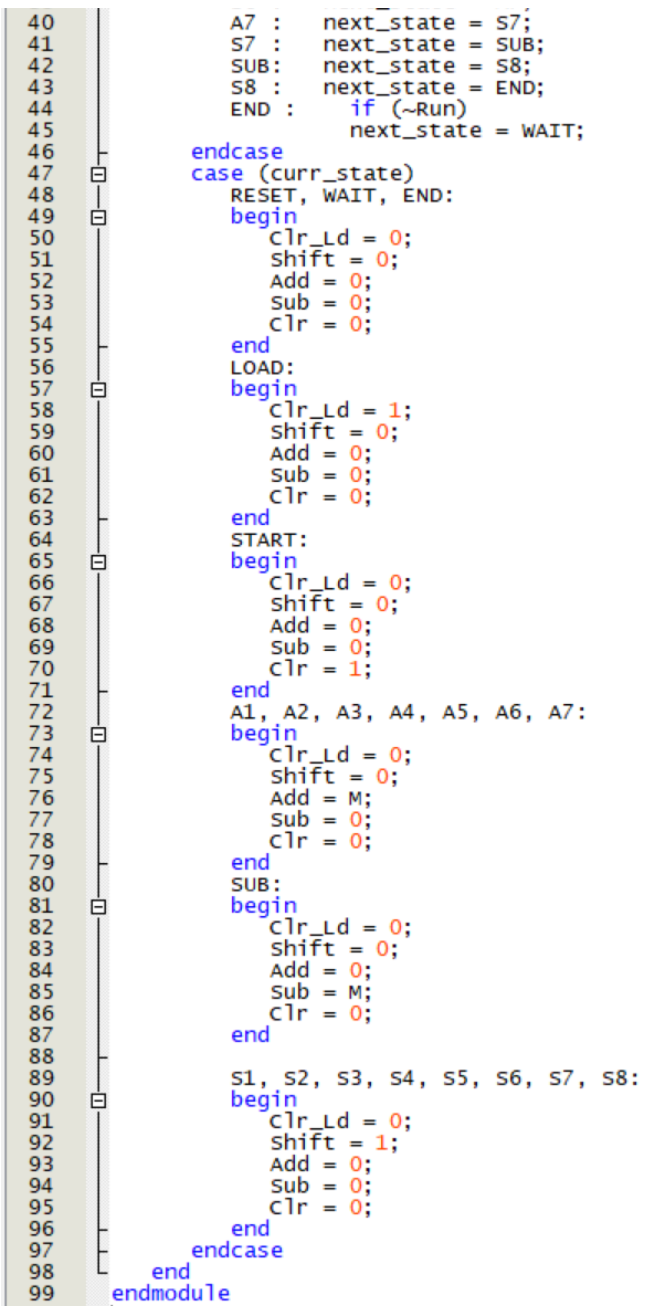
Outputs: [8:0] SS

Description: This is a nine-bit adder, which can also do subtraction. It is made of nine full adders. The XOR operation of the select bit and the bit of B will allows for subtraction to take place when needed.

Purpose: This module is used to do the adding and subtracting for the shift-add algorithm.



```
module control
(
    input  logic Reset,Clk,Run,ClearA_LoadB,M,
    output logic Clr_Ld,Shift,Add,Sub,Clr
);
    enum logic [4:0] {RESET, WAIT, LOAD, START,
                      S1,S2,S3,S4,S5,S6,S7,S8,
                      A1,A2,A3,A4,A5,A6,A7,SUB,END} curr_state, next_state;
    always_ff @ (posedge Clk)
    begin
        if (Reset) // can interupt at anytime
            curr_state <= RESET;
        else
            curr_state <= next_state;
     end
    always_comb
    begin
        next_state  = curr_state;
        unique case (curr_state)

            RESET:   next_state = WAIT;
            LOAD :   next_state = WAIT;
            WAIT :   if (Run)
                     next_state = START;
                     else if (ClearA_LoadB)
                     next_state = LOAD;
            START:   next_state = A1;
            A1 :   next_state = S1;
            S1 :   next_state = A2;
            A2 :   next_state = S2;
            S2 :   next_state = A3;
            A3 :   next_state = S3;
            S3 :   next_state = A4;
            A4 :   next_state = S4;
            S4 :   next_state = A5;
            A5 :   next_state = S5;
            S5 :   next_state = A6;
            A6 :   next_state = S6;
            S6 :   next_state = A7;
```



```
            A7 :   next_state = S7;
            S7 :   next_state = SUB;
            SUB:   next_state = S8;
            S8 :   next_state = END;
            END :    if (~Run)
                     next_state = WAIT;
        endcase
        case (curr_state)
            RESET, WAIT, END:
            begin
                Clr_Ld = 0;
                Shift = 0;
                Add = 0;
                Sub = 0;
                Clr = 0;
            end
            LOAD:
            begin
                Clr_Ld = 1;
                Shift = 0;
                Add = 0;
                Sub = 0;
                Clr = 0;
            end
            START:
            begin
                Clr_Ld = 0;
                Shift = 0;
                Add = 0;
                Sub = 0;
                Clr = 1;
            end
            A1, A2, A3, A4, A5, A6, A7:
            begin
                Clr_Ld = 0;
                Shift = 0;
                Add = M;
                Sub = 0;
                Clr = 0;
            end
            SUB:
            begin
                Clr_Ld = 0;
                Shift = 0;
                Add = 0;
                Sub = M;
                Clr = 0;
            end

            S1, S2, S3, S4, S5, S6, S7, S8:
            begin
                Clr_Ld = 0;
                Shift = 1;
                Add = 0;
                Sub = 0;
                Clr = 0;
            end
        endcase
    end
endmodule
```

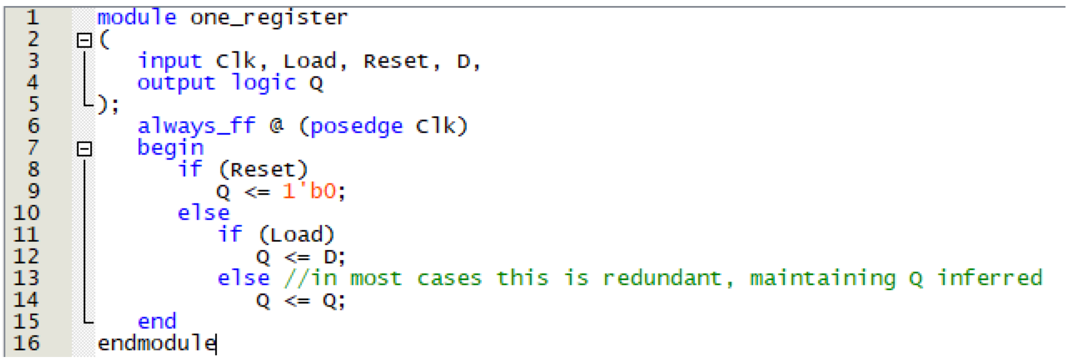
Module: control.sv

Inputs: Reset, Clk, Run, ClearA_LoadB, M,

Outputs: Clr_Ld, Shift, Add, Sub, Clr

Description: This is module is a finite state machine implementation. According to the input, it will go different states, and for each state it will produce its own output and next level logic. Here, it goes to RESET state anytime it receives the Reset signal, and then it will go to WAIT state to wait for Run or ClearA_LoadB signal. After that, it will either start the process or load the value. When the shift-add process begin, the states will switch between A_n state and S_n state, until reach the END state. The state diagram of the control unit will shown later.

Purpose: This module is used to give the control signal when we need to add, shift, subtract, or other state.



```
module one_register
(
    input Clk, Load, Reset, D,
    output logic Q
);
    always_ff @ (posedge Clk)
    begin
        if (Reset)
            Q <= 1'b0;
        else
            if (Load)
                Q <= D;
            else //in most cases this is redundant, maintaining Q inferred
                Q <= Q;
    end
endmodule
```

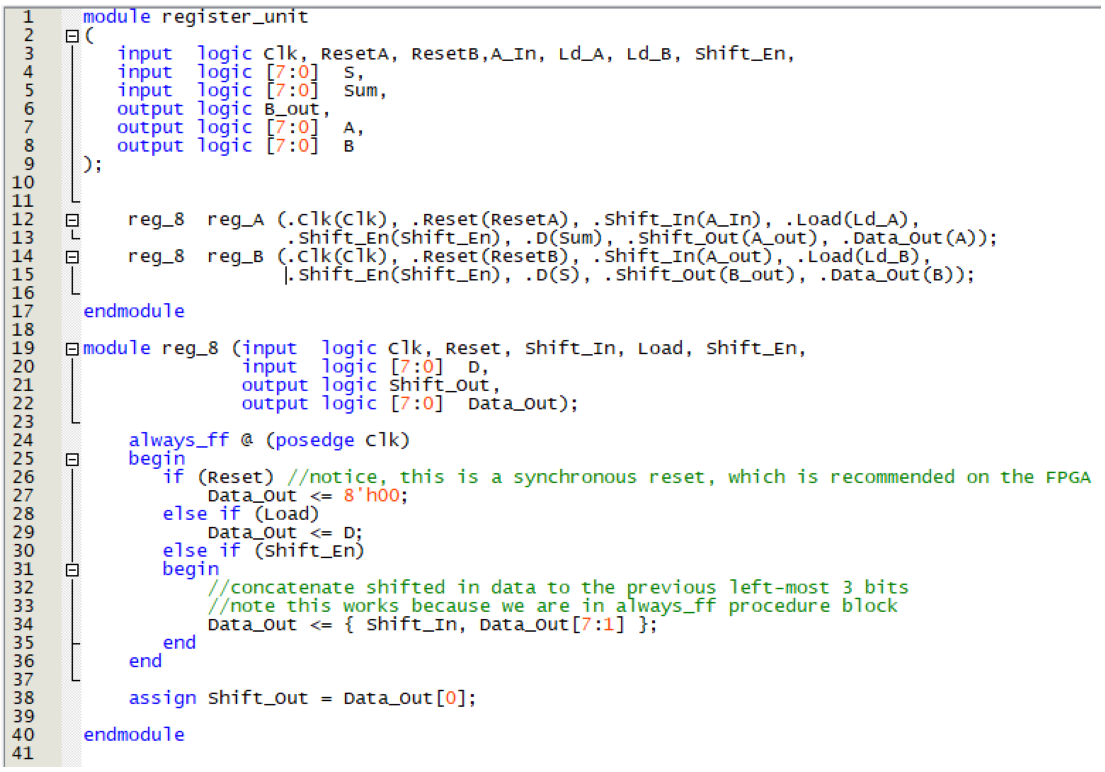
Module: one_register.sv

Inputs: Clk, Load, Reset, D

Outputs: Q

Description: It is a 1-bit register, which can store 1 bit of information. It will get updated on positive edge of clock based on reset or load select bit.

Purpose: This module is used to store the ‘X’ value. It can maintain its value for sign extend when we are not do adding and shifting.



```
module register_unit
(
    input  logic Clk, ResetA, ResetB,A_In, Ld_A, Ld_B, Shift_En,
    input  logic [7:0]  S,
    input  logic [7:0]  Sum,
    output logic B_out,
    output logic [7:0]  A,
    output logic [7:0]  B
);


     reg_8  reg_A (.Clk(Clk), .Reset(ResetA), .Shift_In(A_In), .Load(Ld_A),
                   .Shift_En(Shift_En), .D(Sum), .Shift_Out(A_out), .Data_Out(A));
     reg_8  reg_B (.Clk(Clk), .Reset(ResetB), .Shift_In(A_out), .Load(Ld_B),
                   .Shift_En(Shift_En), .D(S), .Shift_Out(B_out), .Data_Out(B));

endmodule

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

    always_ff @ (posedge Clk)
    begin
        if (Reset) //notice, this is a synchronous reset, which is recommended on the FPGA
            Data_Out <= 8'h00;
        else if (Load)
            Data_Out <= D;
        else if (Shift_En)
        begin
            //concatenate shifted in data to the previous left-most 3 bits
            //note this works because we are in always_ff procedure block
            Data_Out <= { Shift_In, Data_Out[7:1] };
        end
    end

    assign Shift_Out = Data_Out[0];

endmodule
```

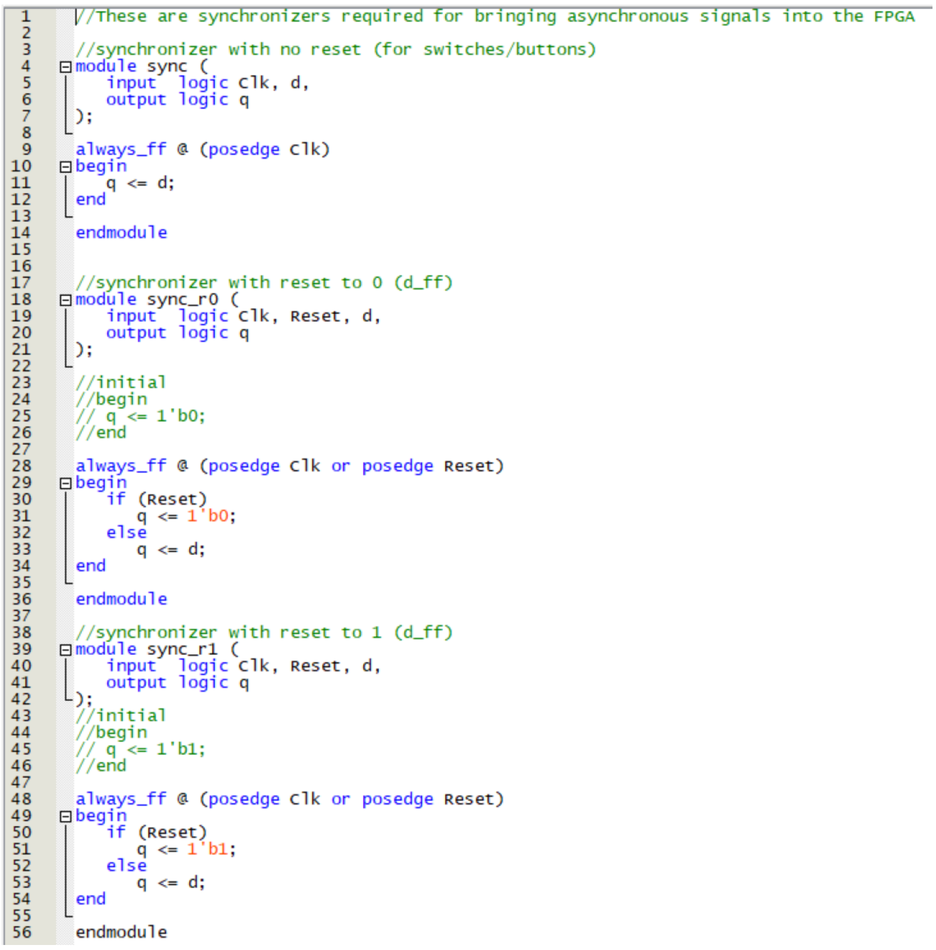
Module: register_unit.sv (Also contain an 8-bit register module: *reg_8*)

Inputs: Clk, ResetA, ResetB, A_In, Ld_A, Ld_B, Shift_En, [7:0] S, [7:0] Sum,

Outputs: B_out, [7:0] A, [7:0] B

Description: This is an 8-bit shift register that is positive edge triggered. It can clear the register and a load bit by a reset bit to load the data to the register.

Purpose: This module is used to store A and B and shift data.



```
//These are synchronizers required for bringing asynchronous signals into the FPGA

//synchronizer with no reset (for switches/buttons)
module sync (
    input  logic Clk, d,
    output logic q
);

always_ff @ (posedge Clk)
begin
    q <= d;
end

endmodule


//synchronizer with reset to 0 (d_ff)
module sync_r0 (
    input  logic Clk, Reset, d,
    output logic q
);

//initial
//begin
// q <= 1'b0;
//end

always_ff @ (posedge Clk or posedge Reset)
begin
    if (Reset)
        q <= 1'b0;
    else
        q <= d;
end

endmodule

//synchronizer with reset to 1 (d_ff)
module sync_r1 (
    input  logic Clk, Reset, d,
    output logic q
);
//initial
//begin
// q <= 1'b1;
//end

always_ff @ (posedge Clk or posedge Reset)
begin
    if (Reset)
        q <= 1'b1;
    else
        q <= d;
end

endmodule
```

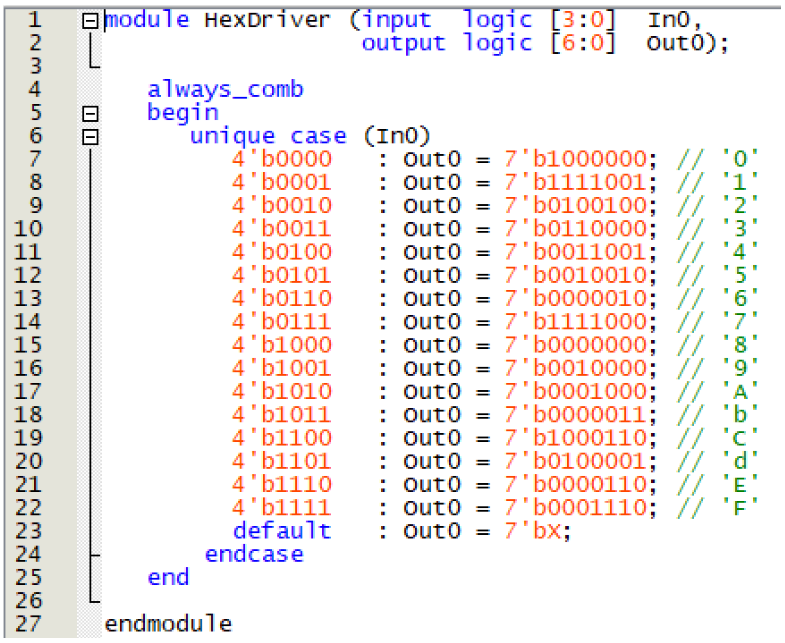
Module: sync.sv

Inputs: Clk, d, Reset

Outputs: q

Description: It will make asynchronous inputs synchronize at rising edge of the clock or rising edge of the Reset signal.

Purpose: This module is used to make everything to be synchronous.



```
module HexDriver (input  logic [3:0]  In0,
                  output logic [6:0]  Out0);

    always_comb
    begin
        unique case (In0)
            4'b0000   : Out0 = 7'b1000000; // '0'
            4'b0001   : Out0 = 7'b1111001; // '1'
            4'b0010   : Out0 = 7'b0100100; // '2'
            4'b0011   : Out0 = 7'b0110000; // '3'
            4'b0100   : Out0 = 7'b0011001; // '4'
            4'b0101   : Out0 = 7'b0010010; // '5'
            4'b0110   : Out0 = 7'b0000010; // '6'
            4'b0111   : Out0 = 7'b1111000; // '7'
            4'b1000   : Out0 = 7'b0000000; // '8'
            4'b1001   : Out0 = 7'b0010000; // '9'
            4'b1010   : Out0 = 7'b0001000; // 'A'
            4'b1011   : Out0 = 7'b0000011; // 'b'
            4'b1100   : Out0 = 7'b1000110; // 'C'
            4'b1101   : Out0 = 7'b0100001; // 'd'
            4'b1110   : Out0 = 7'b0000110; // 'E'
            4'b1111   : Out0 = 7'b0001110; // 'F'
            default   : Out0 = 7'bX;
        endcase
    end

endmodule
```

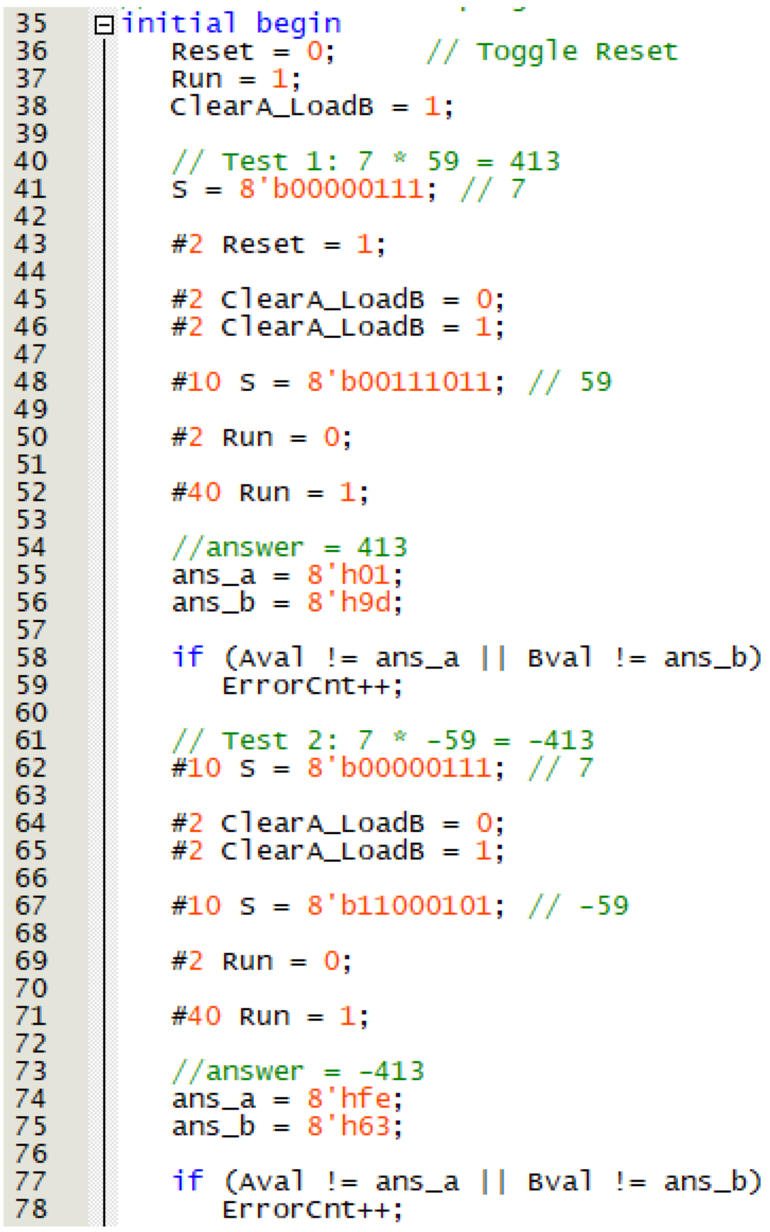
Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: The HexDriver was provided in Experiment 4, which can translate a number from binary to the display so that we can see the number directly in Hex.

Purpose: This module is used to display the final answer on the FPGA board.



```
initial begin
    Reset = 0;      // Toggle Reset
    Run = 1;
    ClearA_LoadB = 1;

    // Test 1: 7 * 59 = 413
    S = 8'b00000111; // 7

    #2 Reset = 1;

    #2 ClearA_LoadB = 0;
    #2 ClearA_LoadB = 1;

    #10 S = 8'b00111011; // 59

    #2 Run = 0;

    #40 Run = 1;

    //answer = 413
    ans_a = 8'h01;
    ans_b = 8'h9d;

    if (Aval != ans_a || Bval != ans_b)
       ErrorCnt++;

    // Test 2: 7 * -59 = -413
    #10 S = 8'b00000111; // 7

    #2 ClearA_LoadB = 0;
    #2 ClearA_LoadB = 1;

    #10 S = 8'b11000101; // -59

    #2 Run = 0;

    #40 Run = 1;

    //answer = -413
    ans_a = 8'hfe;
    ans_b = 8'h63;

    if (Aval != ans_a || Bval != ans_b)
       ErrorCnt++;
```

Module: testbench.sv (part of)

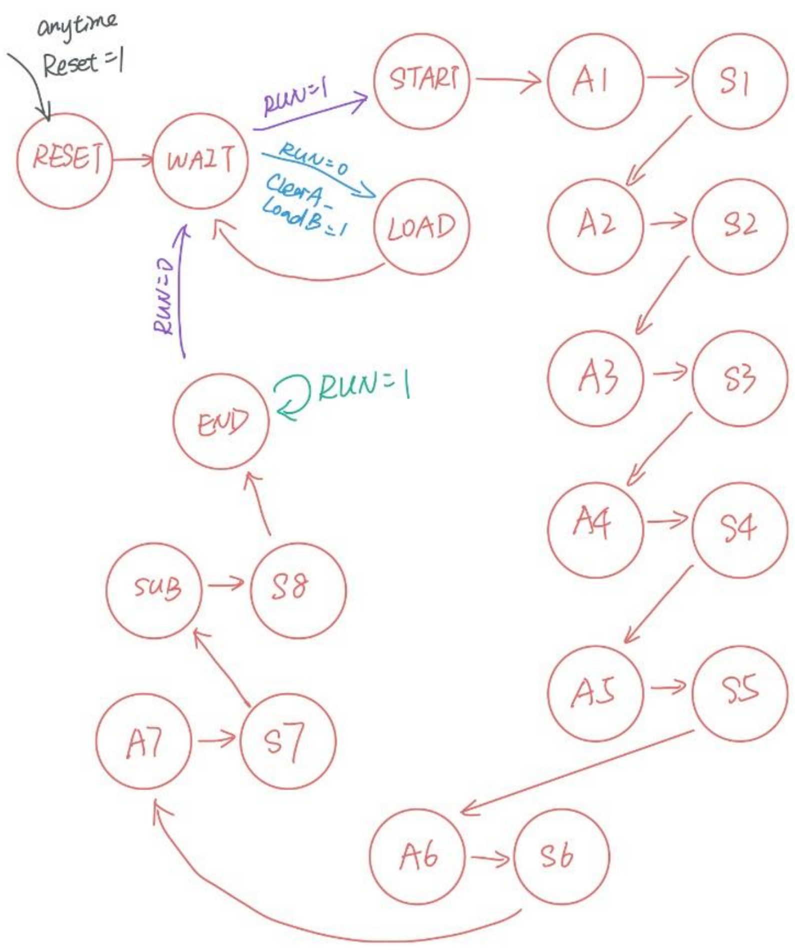
Inputs: NONE

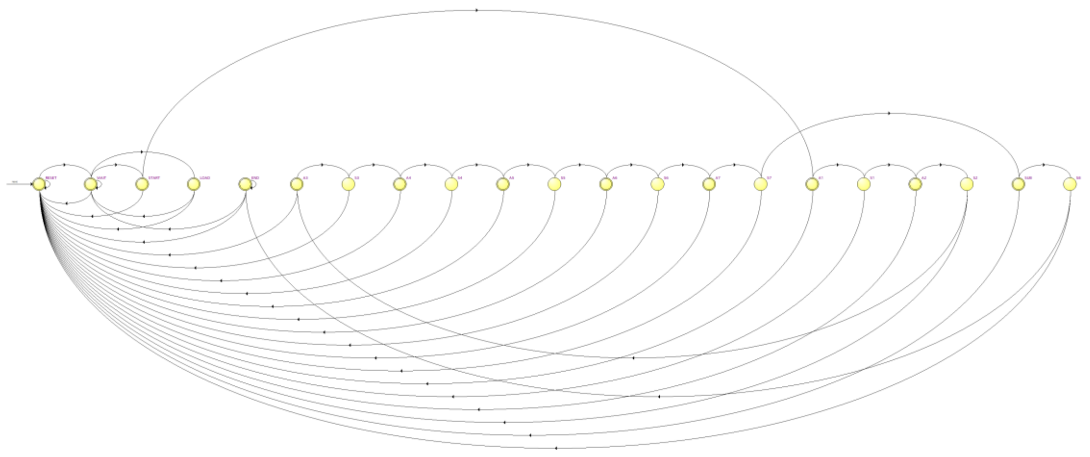
Outputs: NONE

Description: First we load A and B and start the process, then we compare our output to correct output, if not correct, we count it in ErrorCnt, which will tell us if there’s any error at the end.

Purpose: This module is used to test the different cases of ++/+-/-+/-- to figure whether our modules can work.

State Diagram for Control Unit

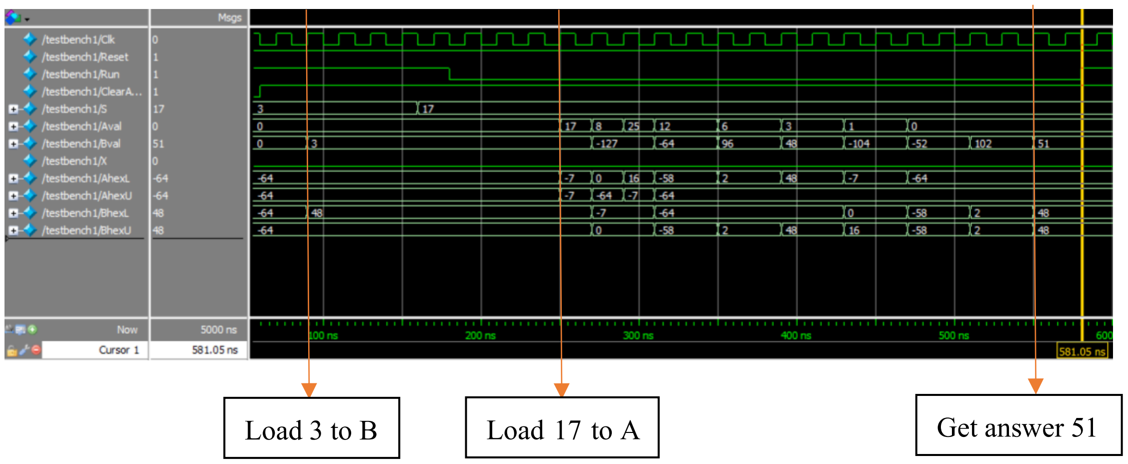




4. Annotated pre-lab simulation waveforms

Case 1: 3 * 17 = 51

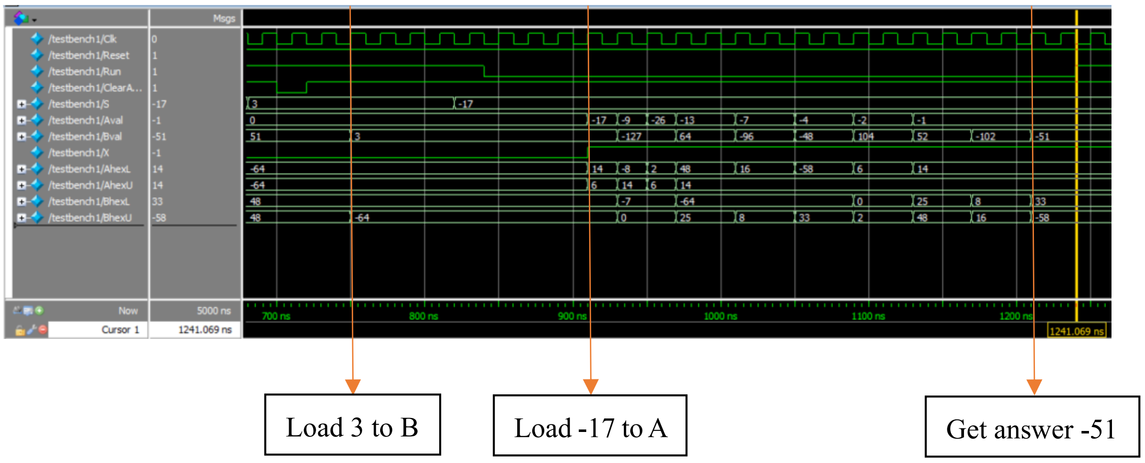
```
// Test 1: 3 * 17 = 51
S = 8'b00000011; // 3

#2 Reset = 1;

#2 ClearA_LoadB = 0;
#2 ClearA_LoadB = 1;

#10 S = 8'b00010001; // 17

#2 Run = 0;

#40 Run = 1;
```



First, RESET is pressed and at the next cycle, X, A and B will be cleared. Then, when ClearA_LoadB is pressed, it will clear X and A as well as load 3 to B in next cycle. After that, we reset the switches to 17 and press RUN, it will clear X and A. FSM then automatically transitions between the states of adding and shifting. At the end, we compute the correct answer of 51.

Case 2: 3 * -17 = -51

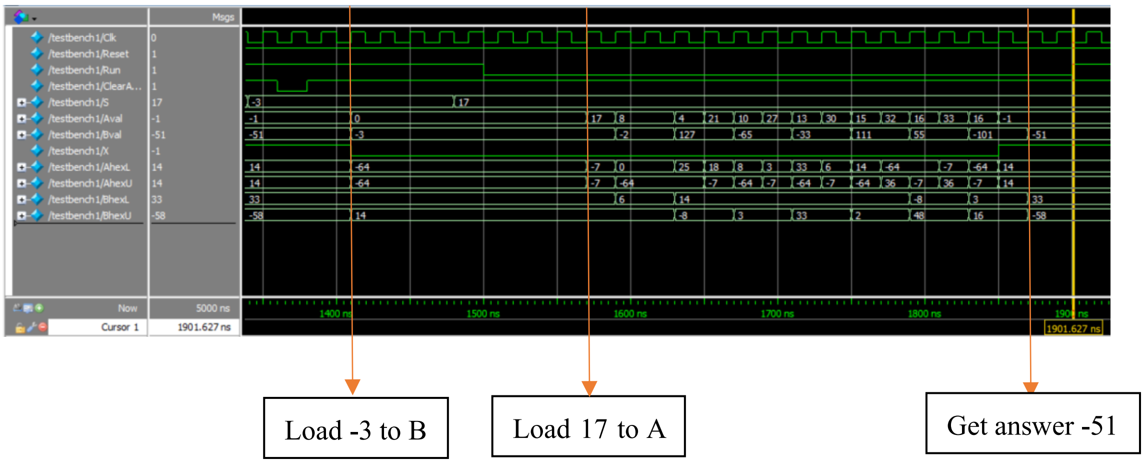
```
// Test 2: 3 * -17 = -51
#10 S = 8'b00000011; // 3

#2 ClearA_LoadB = 0;
#2 ClearA_LoadB = 1;

#10 S = 8'b11101111; // -17

#2 Run = 0;

#40 Run = 1;
```



First, RESET is pressed and at the next cycle, X, A and B will be cleared. Then, when ClearA_LoadB is pressed, it will clear X and A as well as load 3 to B in next cycle. After that, we reset the switches to -17 and press RUN, it will clear X and A. FSM then automatically transitions between the states of adding and shifting. At the end, we compute the correct answer of -51.

Case 3: -3 * 17 = -51

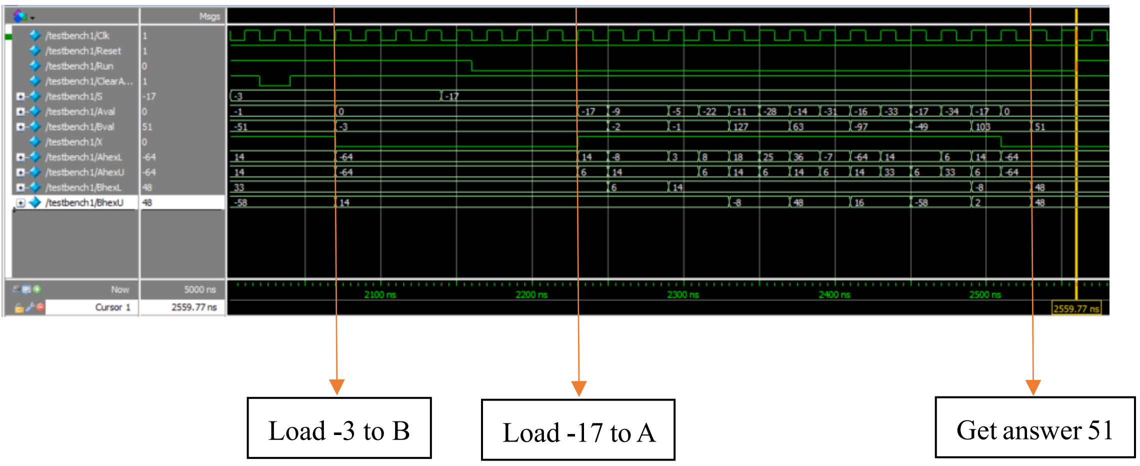
```
// Test 3: -3 * 17 = -51
#10 S = 8'b11111101; // -3

#2 ClearA_LoadB = 0;
#2 ClearA_LoadB = 1;

#10 S = 8'b00010001; // 17

#2 Run = 0;

#40 Run = 1;
```



First, RESET is pressed and at the next cycle, X, A and B will be cleared. Then, when ClearA_LoadB is pressed, it will clear X and A as well as load -3 to B in next cycle. After that, we reset the switches to 17 and press RUN, it will clear X and A. FSM then automatically transitions between the states of adding and shifting. At the end, we compute the correct answer of -51.

Case 4: -3 * -17 = 51

```
// Test 4: -3 * -17 = 51
#10 S = 8'b11111101; // -3

#2 ClearA_LoadB = 0;
#2 ClearA_LoadB = 1;

#10 S = 8'b11101111; // -17

#2 Run = 0;

#40 Run = 1;
```



First, RESET is pressed and at the next cycle, X, A and B will be cleared. Then, when ClearA_LoadB is pressed, it will clear X and A as well as load -3 to B in next cycle. After that, we reset the switches to -17 and press RUN, it will clear X and A. FSM then automatically transitions between the states of adding and shifting. At the end, we compute the correct answer of 51.

5. Answers to two post-lab questions

1)

LUT	96
DSP	N/A
Memory (BRAM)	0
Flip-Flop	47
Frequency	74.13 MHz
Static Power	99.53 mW
Dynamic Power	4.01 mW
Total Power	142.47 mW

Here are some ideas about how to optimize our design to decrease the total gate count and/or to increase maximum frequency by changing your code for the design shown as following:

Above all, we cannot allow resets in the middle of calculations to decrease unnecessary transitions and thus simplify our design. Furthermore, we notice that we will have the exact same output control signals in each same-type state (i.e.: every SHIFT state had SHIFT_EN =1, etc.) , thus we can optimize our design by simply using less states. However, this may make it difficult to synchronous and may introduce delay into design, thus there should be tradeoff.

2）**a) What is the purpose of the X register. When does the X register get set/cleared?**

In our design, we use X-register to preserve the sign bit while shifting and its value will be updated in every addition, i.e. ClearA_LoadB button is pressed (that is ClearA_LoadB = 1), X will be reset to 0; Afterwards at each step, X is the extend of A.

b) What are the limitations of continuous multiplications? Under what circumstances will the implemented algorithm fail?

According to our understanding, the limitations of continuous multiplications is that we need to clear XA at the start of the multiplication cycle since may easily overflow. In our design, XAB can store a 17-bit answer as a result of multiplication, when we consecutively multiply a number “overflowed” into XA, resetting the bits in XA to 0 and using the wrong multiplier, and thus coming up with wrong answer. In short, our algorithm will fail if the product that we will use in consecutive multiplication cannot be completely contained in B and cause overflow.

c) What are the advantages (and disadvantages?) of the implemented multiplication algorithm over the pencil-and-paper method discussed in the introduction?

The advantage of the paper-pencil method is that it is very intuitive and thus easier to understand conceptually. Also, it is relatively fast since there is no need to process a lot of multiplications with very large numbers and save memory space. However, it also has some disadvantage i.e. it is very hard to practically implement the paper-pencil method (in decimal) in a digital system (in binary).

6. Conclusion

In conclusion, in this lab, we complete a 2’s complement 8-bit Multiplier Unit in SystemVerilog using logic operation. This Multiplier can multiply two 8-bit signed numbers. On the FPGA board, we can enter two operator and one operation will be executed when the Run press button is pressed. Our design can work on all cases of (+*+), (-*+), (+*-), (-*-) if no overflow happens.

During the implementation process, we did encounter with some mistakes, i.e. forget about initiate state, use the wrong pins, forget to reset X and A to 0 at the beginning, but finally we find them out and fix them. The most difficult part is to debug the issue which makes our design not synchronous.

With many problems faced with, we realize that the tutorial is over-simplified and hope to get more detailed and intuitive information. Furthermore, we want to know more about why we need to do this rather than do this from the lecture, which we think will be more meaningful.