ECE385

Experiment #5

An 8-Bit Multiplier in System Verilog

Chengting Yu, 3170111707
Yangkai Du, 3170112258

1. Introduction

In this experiment, we build an 2's Complement 8-bit Multiplier unit, which can support two 8-bit 2's compliment numbers multiplication.

2. Pre-lab question

Function	X	A	*B*	M	Comments for the next step
Clear A, LoadB	**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**	**Shift XAB by one bit after ADD complete**
SHIFT	**1**	**1101 0011**	**1100 0001**	**1**	**Add S to A since M = 1.**
ADD	**1**	**1001 1000**	**1100 0001**	**1**	**Shift XAB by one bit after ADD complete**
SHIFT	**1**	**1100 1100**	**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 1100**	**1100 0110**	**0**	**Do not SUB S to A since M = 0. Shift XAB.**
SHIFT	**1**	**1111 1110**	**0110 0011**	**0**	**8th shift done. Stop. 16-bit Product in AB.**

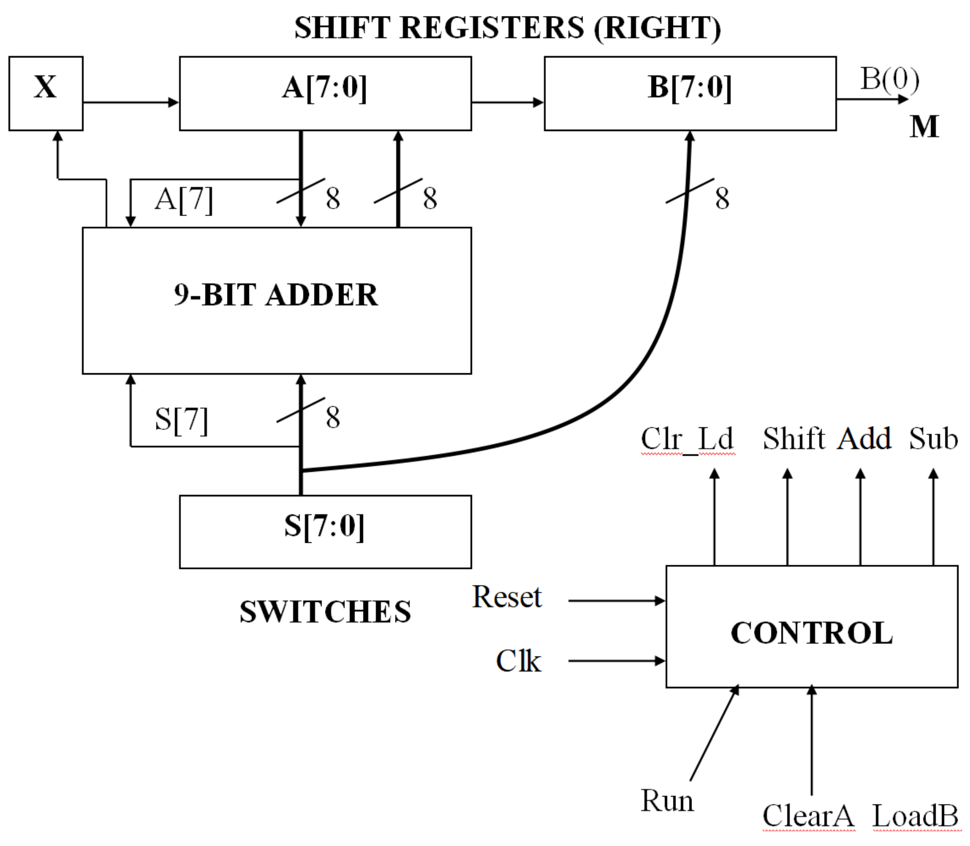
3. Written description and diagrams of multiplier circuit

a. Summary of operation

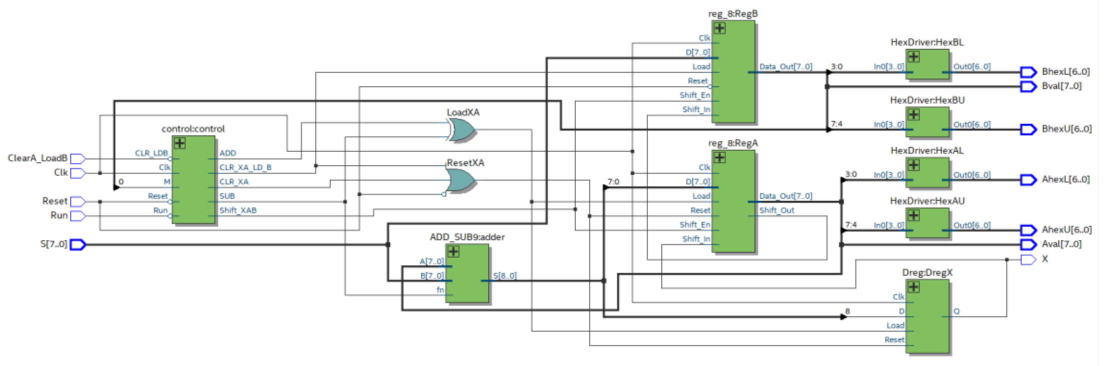
Operands Loading: First store the Multiplier in S and press ClearA_LoadB, the Multiplier in S would be loaded to Register B and Register A and DFF X would be clear to 0; then store the Multiplicand in S.

Computing: User need to press "Run" after "ClearA_LoadB" is excuted, then the "add-shift" algorithm will be executed to compute the results. There are two states, add states and shift states. In add states, if M = B[0] is 0, then we do not need to accumulate the Multiplicand, just transfer to shift states. If M = B[0] is 1, then we need to accumulate the Multiplicand, A <- S+A, the sign extention bit is stored in X, then we would transfer to shift state. In shift state, A[0]->B[7], X->A[7], every bit is shited right in Reg A and B. There would be 8 shift and 8 add/sub operation for 8 bit multiplier.

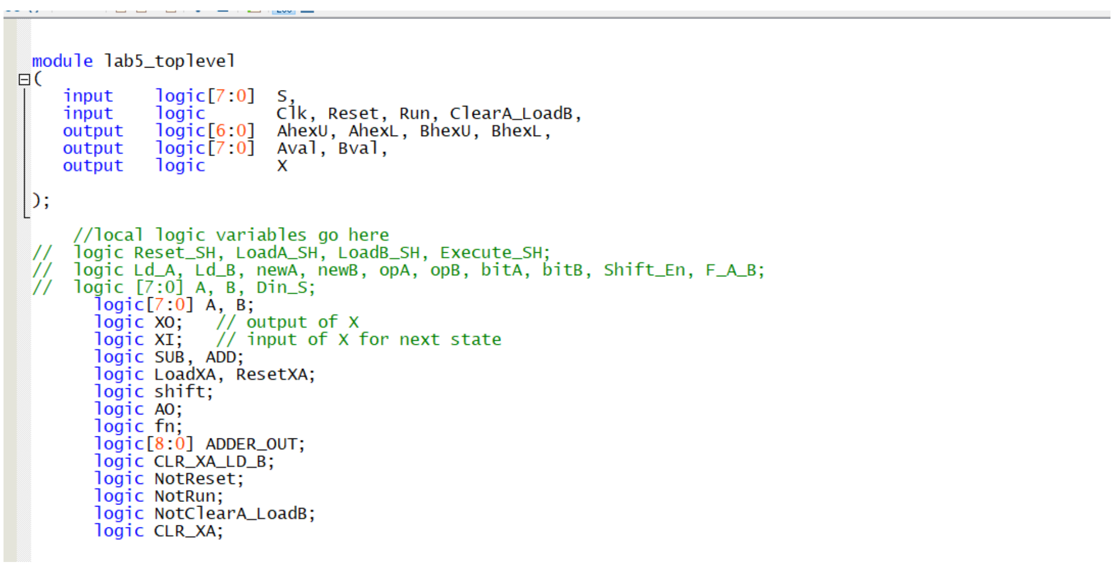
Results Store: The final results R is 16-bit, R[15:8] is stored in Reg A, and R[7:0] is stored in Reg B.



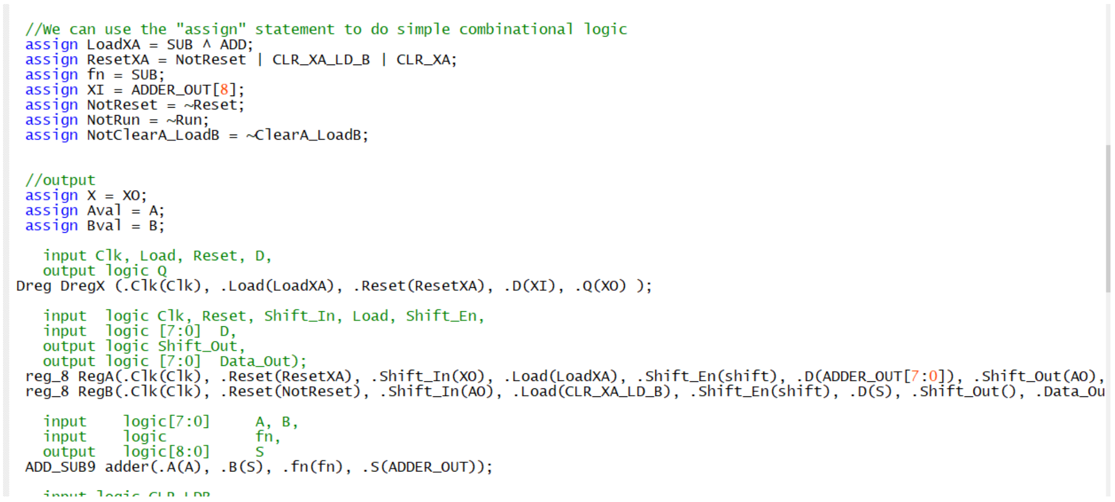
b. Top Level Block Diagram



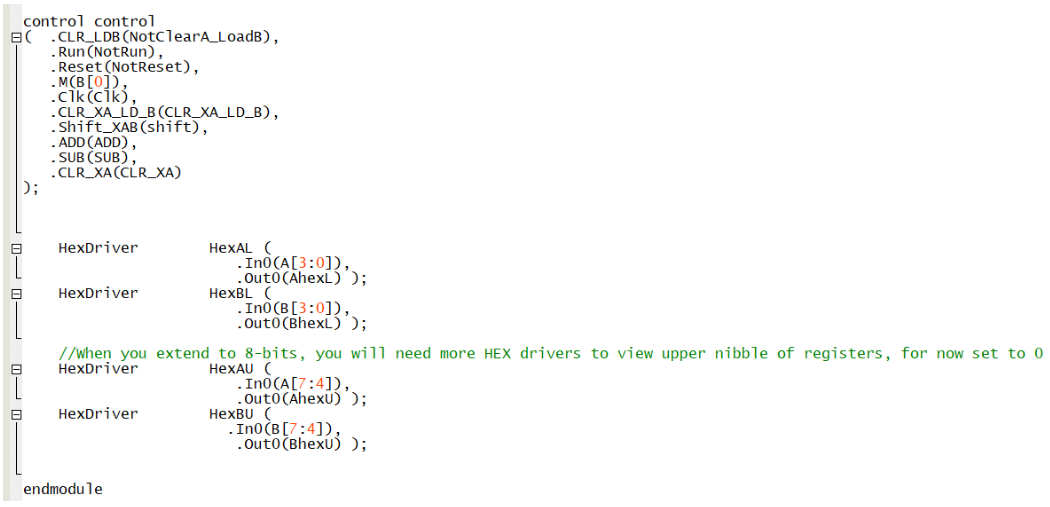
c. Written Description of .sv Modules



```
module lab5_toplevel
(
    input     logic[7:0]   S,
    input     logic        Clk, Reset, Run, ClearA_LoadB,
    output    logic[6:0]   AhexU, AhexL, BhexU, BhexL,
    output    logic[7:0]   Aval, Bval,
    output    logic        X

);

    //local logic variables go here
//  logic Reset_SH, LoadA_SH, LoadB_SH, Execute_SH;
//  logic Ld_A, Ld_B, newA, newB, opA, opB, bitA, bitB, Shift_En, F_A_B;
//  logic [7:0] A, B, Din_S;
      logic[7:0] A, B;
      logic XO;   // output of X
      logic XI;   // input of X for next state
      logic SUB, ADD;
      logic LoadXA, ResetXA;
      logic shift;
      logic AO;
      logic fn;
      logic[8:0] ADDER_OUT;
      logic CLR_XA_LD_B;
      logic NotReset;
      logic NotRun;
      logic NotClearA_LoadB;
      logic CLR_XA;
```



```
  //We can use the "assign" statement to do simple combinational logic
  assign LoadXA = SUB ^ ADD;
  assign ResetXA = NotReset | CLR_XA_LD_B | CLR_XA;
  assign fn = SUB;
  assign XI = ADDER_OUT[8];
  assign NotReset = ~Reset;
  assign NotRun = ~Run;
  assign NotClearA_LoadB = ~ClearA_LoadB;


  //output
  assign X = XO;
  assign Aval = A;
  assign Bval = B;

    input Clk, Load, Reset, D,
    output logic Q
 Dreg DregX (.Clk(Clk), .Load(LoadXA), .Reset(ResetXA), .D(XI), .Q(XO) );

    input  logic Clk, Reset, Shift_In, Load, Shift_En,
    input  logic [7:0]  D,
    output logic Shift_Out,
    output logic [7:0]  Data_Out);
  reg_8 RegA(.Clk(Clk), .Reset(ResetXA), .Shift_In(XO), .Load(LoadXA), .Shift_En(shift), .D(ADDER_OUT[7:0]), .Shift_Out(AO),
  reg_8 RegB(.Clk(Clk), .Reset(NotReset), .Shift_In(AO), .Load(CLR_XA_LD_B), .Shift_En(shift), .D(S), .Shift_Out(), .Data_Ou

    input    logic[7:0]     A, B,
    input    logic          fn,
    output   logic[8:0]     S
  ADD_SUB9 adder(.A(A), .B(S), .fn(fn), .S(ADDER_OUT));

    input logic CLR_LDB
```



```
control control
(   .CLR_LDB(NotClearA_LoadB),
    .Run(NotRun),
    .Reset(NotReset),
    .M(B[0]),
    .Clk(Clk),
    .CLR_XA_LD_B(CLR_XA_LD_B),
    .Shift_XAB(shift),
    .ADD(ADD),
    .SUB(SUB),
    .CLR_XA(CLR_XA)
);


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

    //When you extend to 8-bits, you will need more HEX drivers to view upper nibble of registers, for now set to 0
    HexDriver        HexAU (
                        .In0(A[7:4]),
                        .Out0(AhexU) );
    HexDriver        HexBU (
                       .In0(B[7:4]),
                        .Out0(BhexU) );

endmodule
```

Module: lab5_toplevel.sv

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

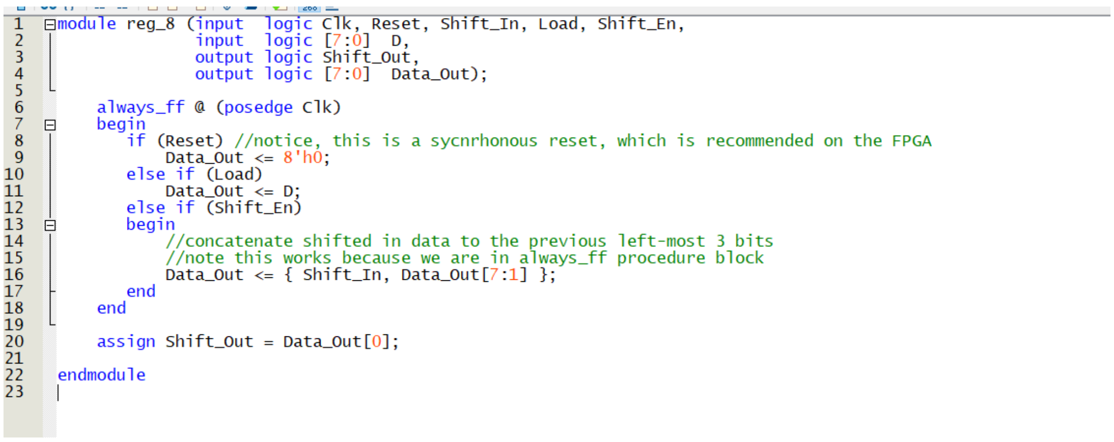
Outputs: logic[6:0]   AhexU, AhexL, BhexU, BhexL,

logic[7:0] Aval, Bval,

logic   X

Description: This module build a 8-bit Multiplier, with the modules shown below

Purpose: Top lever Module which aggreagates all the design modules.



```
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 sycnrhonous reset, which is recommended on the FPGA
           Data_Out <= 8'h0;
       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
```

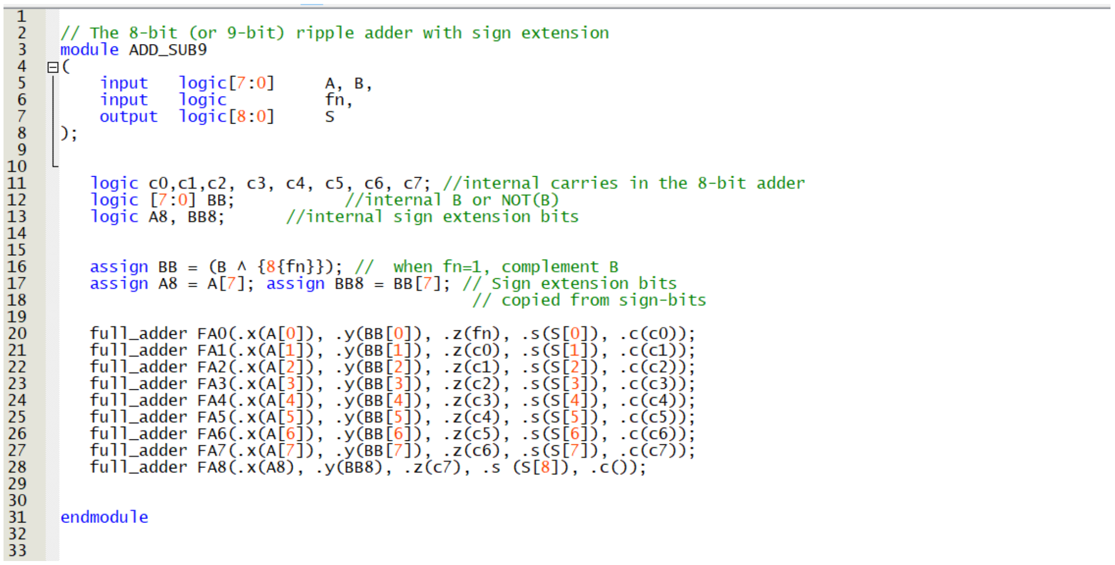
Module: reg_8

Inputs: [7:0] D, Clk, Reset, Shift_In, Load, Shift_En

Outputs: [7:0] Data_out, Shift_Out

Description: This is a positive-edge triggered 8-bit register with asynchronous reset and synchronous load. When Load is high, data is loaded from Din into the register on the positive edge of Clk.

Purpose: This module is used to create the registers that store operands A and B in the adder circuit.



```
// The 8-bit (or 9-bit) ripple adder with sign extension
module ADD_SUB9
(
    input   logic[7:0]      A, B,
    input   logic           fn,
    output  logic[8:0]      S
);


   logic c0,c1,c2, c3, c4, c5, c6, c7; //internal carries in the 8-bit adder
   logic [7:0] BB;           //internal B or NOT(B)
   logic A8, BB8;      //internal sign extension bits


   assign BB = (B ^ {8{fn}}); //  when fn=1, complement B
   assign A8 = A[7]; assign BB8 = BB[7]; // Sign extension bits
                                          // copied from sign-bits

   full_adder FA0(.x(A[0]), .y(BB[0]), .z(fn), .s(S[0]), .c(c0));
   full_adder FA1(.x(A[1]), .y(BB[1]), .z(c0), .s(S[1]), .c(c1));
   full_adder FA2(.x(A[2]), .y(BB[2]), .z(c1), .s(S[2]), .c(c2));
   full_adder FA3(.x(A[3]), .y(BB[3]), .z(c2), .s(S[3]), .c(c3));
   full_adder FA4(.x(A[4]), .y(BB[4]), .z(c3), .s(S[4]), .c(c4));
   full_adder FA5(.x(A[5]), .y(BB[5]), .z(c4), .s(S[5]), .c(c5));
   full_adder FA6(.x(A[6]), .y(BB[6]), .z(c5), .s(S[6]), .c(c6));
   full_adder FA7(.x(A[7]), .y(BB[7]), .z(c6), .s(S[7]), .c(c7));
   full_adder FA8(.x(A8), .y(BB8), .z(c7), .s (S[8]), .c());


endmodule
```

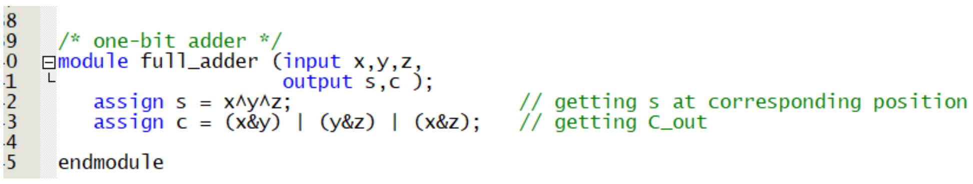
Module: ADD_SUB9.sv

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

Outputs: [8:0] S

Description: This is a 9-bit sign extention ripple Adder build by one-bit full adder. When fn is 1, the adder will perform A-B -> S, when fn is 0, the adder will perform A+B->S.

Purpose: This module is used to compute the resulsts of A+S or A-S, S[7:0] will be loaded to A and S[8] will be loaded to X.



```
/* one-bit adder */
module full_adder (input x,y,z,
                   output s,c );
   assign s = x^y^z;                  // getting s at corresponding position
   assign c = (x&y) | (y&z) | (x&z);   // getting C_out

endmodule
```

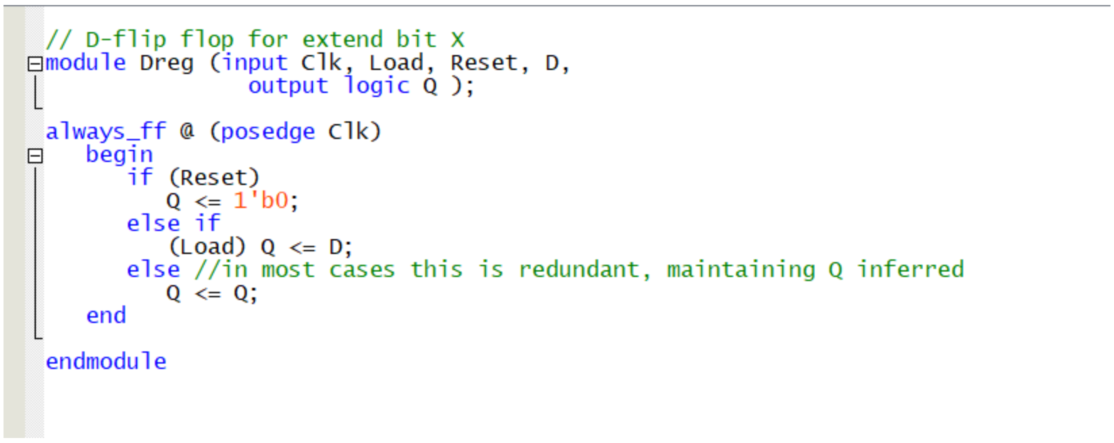
Module: full_adder

Inputs: x,y,z

Outputs: s,c

Description: one-bit full_adder, compute x+y with carry in z.

Purpose: Basic build element for ADD_SUB9



```
// D-flip flop for extend bit X
module Dreg (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
```

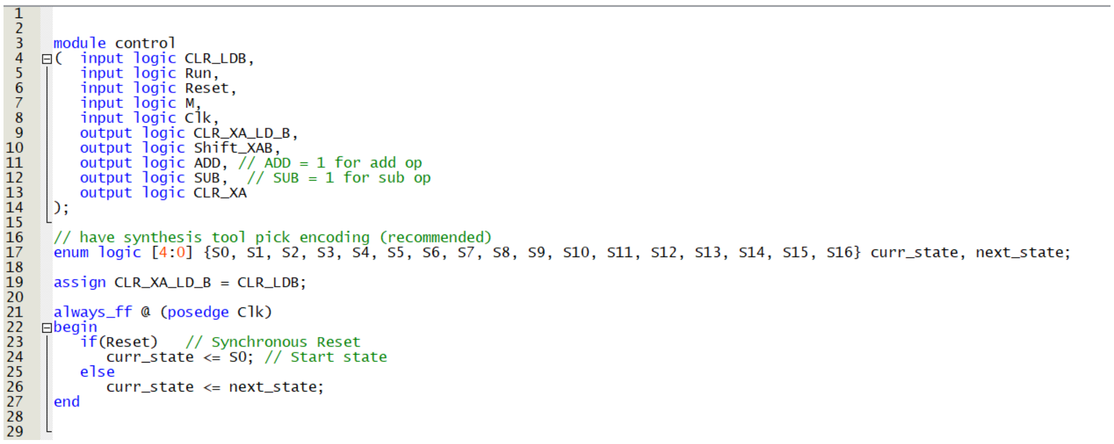
Module: Dreg

Inputs: Clk, Load, Reset, D

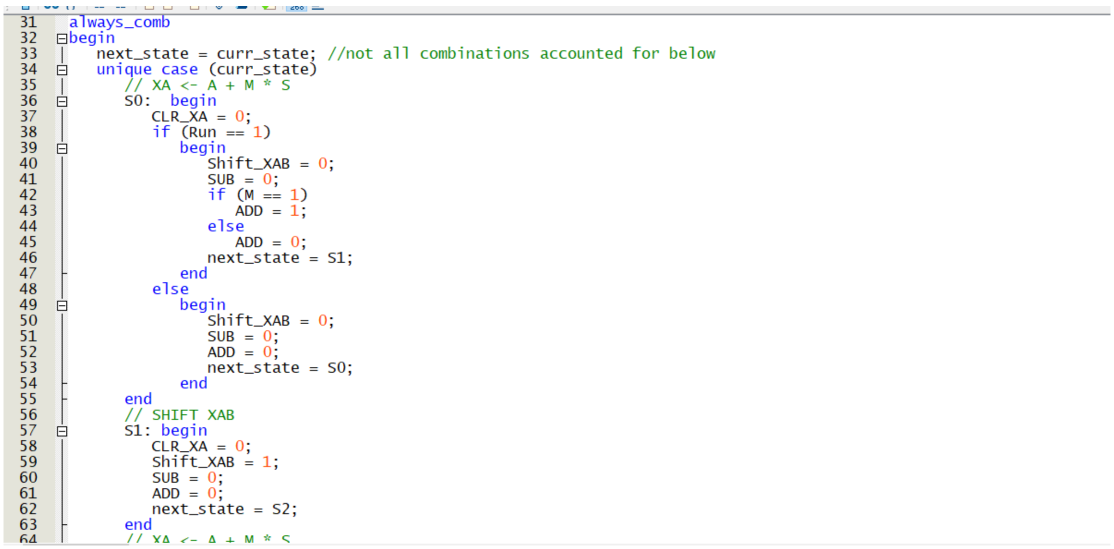
Outputs: Q

Description: This is a positive-edge triggered 1-bit D-Flip-Flog with asynchronous reset and synchronous load. When Load is high, data is loaded from D into the register on the positive edge of Clk.

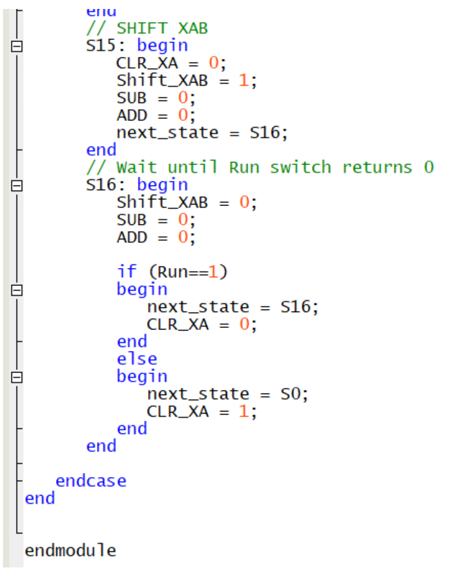
Purpose: Store the sign extension bit X.



```


module control
(  input logic CLR_LDB,
   input logic Run,
   input logic Reset,
   input logic M,
   input logic Clk,
   output logic CLR_XA_LD_B,
   output logic Shift_XAB,
   output logic ADD, // ADD = 1 for add op
   output logic SUB,  // SUB = 1 for sub op
   output logic CLR_XA
);

// have synthesis tool pick encoding (recommended)
enum logic [4:0] {S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16} curr_state, next_state;

assign CLR_XA_LD_B = CLR_LDB;

always_ff @ (posedge Clk)
begin
   if(Reset)   // Synchronous Reset
      curr_state <= S0; // Start state
   else
      curr_state <= next_state;
end

```



```
always_comb
begin
    next_state = curr_state; //not all combinations accounted for below
    unique case (curr_state)
        // XA <- A + M * S
        S0:  begin
            CLR_XA = 0;
            if (Run == 1)
                begin
                    Shift_XAB = 0;
                    SUB = 0;
                    if (M == 1)
                        ADD = 1;
                    else
                        ADD = 0;
                    next_state = S1;
                end
            else
                begin
                    Shift_XAB = 0;
                    SUB = 0;
                    ADD = 0;
                    next_state = S0;
                end
        end
        // SHIFT XAB
        S1: begin
            CLR_XA = 0;
            Shift_XAB = 1;
            SUB = 0;
            ADD = 0;
            next_state = S2;
        end
        // XA <- A + M * S
```



```
        end
        // SHIFT XAB
        S15: begin
            CLR_XA = 0;
            Shift_XAB = 1;
            SUB = 0;
            ADD = 0;
            next_state = S16;
        end
        // Wait until Run switch returns 0
        S16: begin
            Shift_XAB = 0;
            SUB = 0;
            ADD = 0;

            if (Run==1)
            begin
                next_state = S16;
                CLR_XA = 0;
            end
            else
            begin
                next_state = S0;
                CLR_XA = 1;
            end
        end

    endcase
end

endmodule
```

(Here I only show part of the states to reduce reduntancy, there are 17 states in total, you can check the entire code in project file)

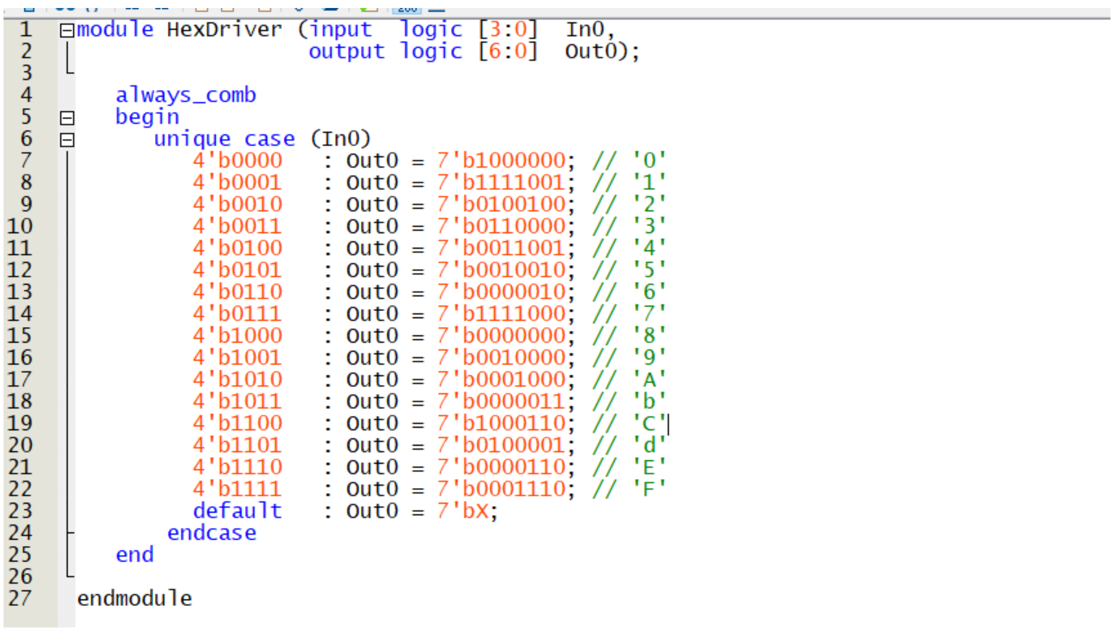
Module: control.sv

Inputs: CLR_LDB, Run, Reset, M, Clk

Outputs: CLR_XA_LD_B, Shift_XAB, ADD, SUB, CLR_XA

Description: The control module implement a state machine for the circuit control, there are 17 states in total, the state machine implement the "add-shift" algorithm and ensure Multiplication is only executed once.

Purpose: Provide control for the entire circuit.



```
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

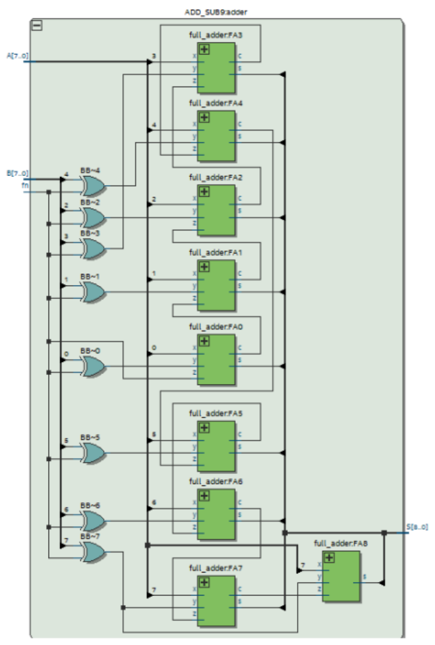
Inputs: [3:0] In0

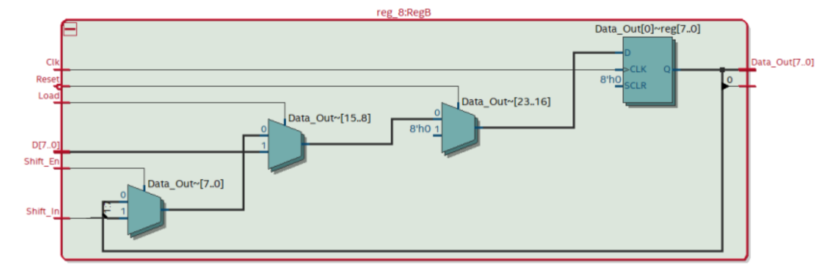
Outputs: [6:0] Out0

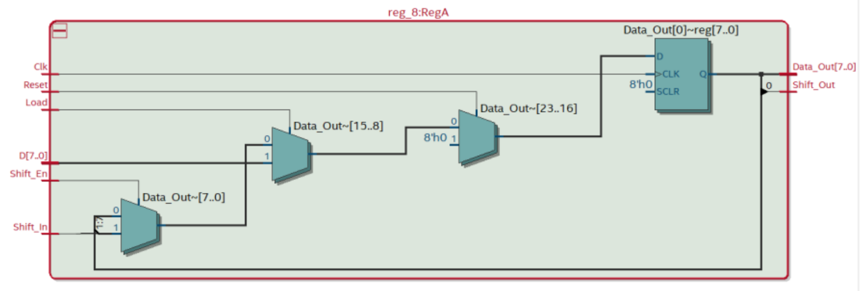
Description: Input a value and Output a code for displaying corresponding Hex representation.

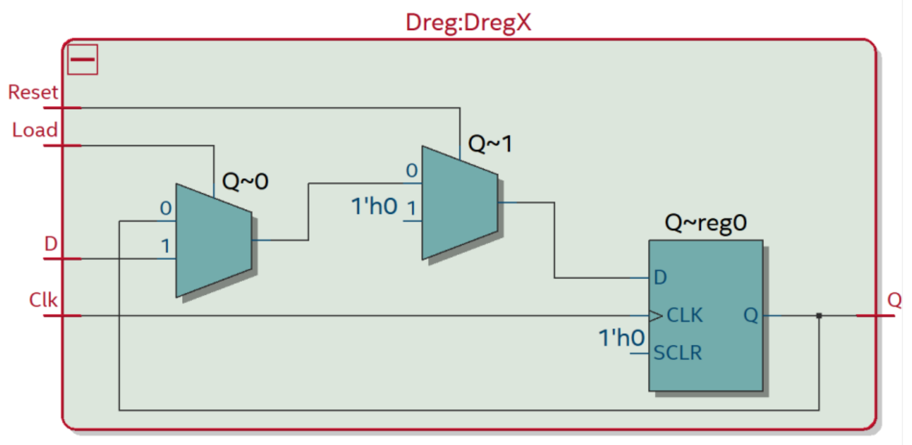
Purpose: To make the screen show the result of Multiplication in Hex Format.

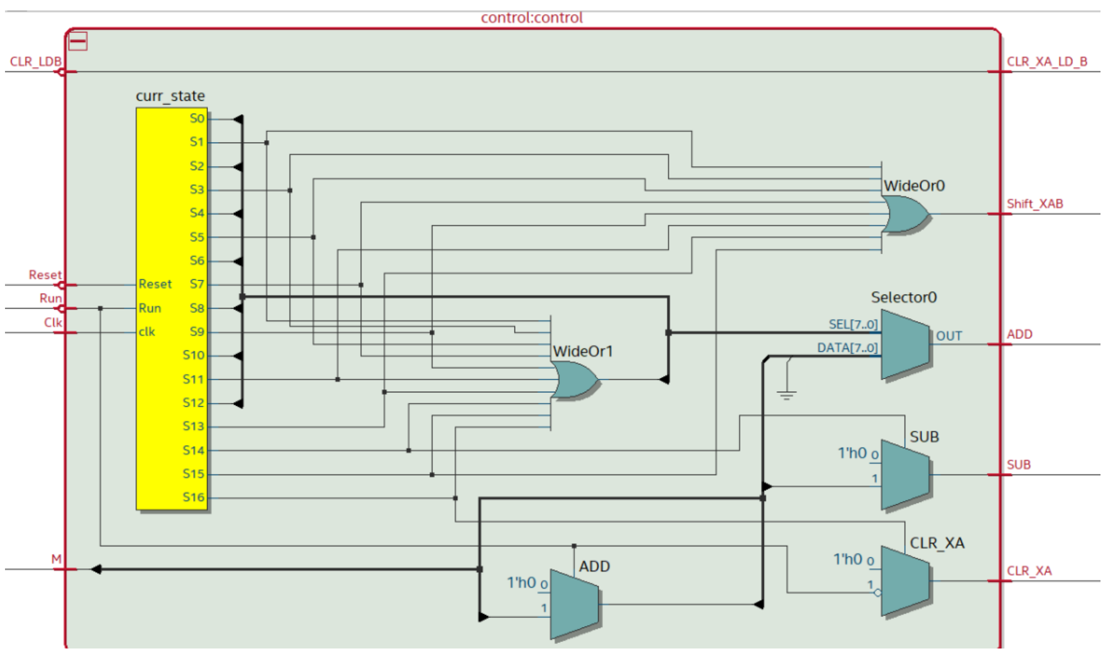
Block Diagram for each module:

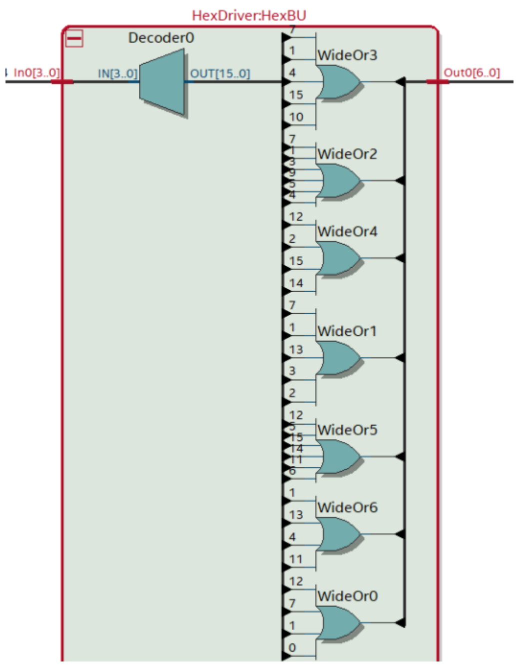


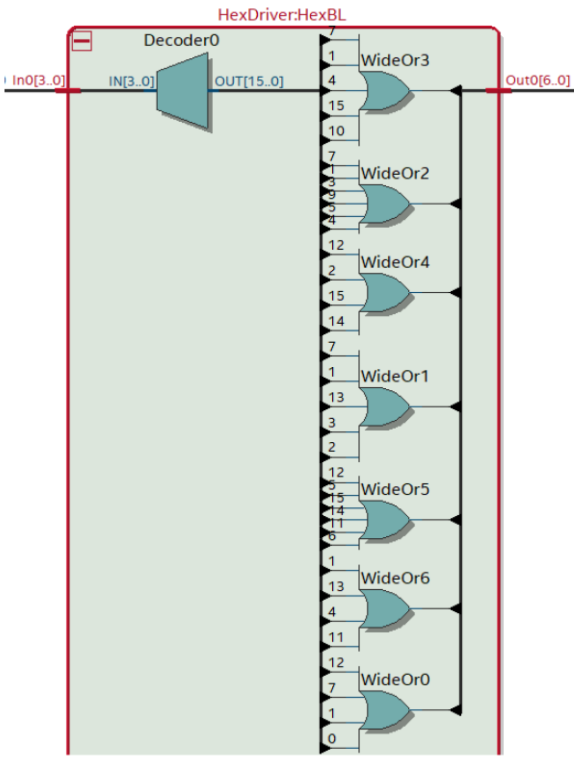

reg_8:RegB
Data_Out[0]~reg[7..0]
Clk
Reset
Load
D[7..0]
Shift_En
Shift_In
Data_Out~[15..8]
Data_Out~[23..16]
Data_Out~[7..0]
8'h0
D
CLK
SCLR
Q
Data_Out[7..0]
0
1

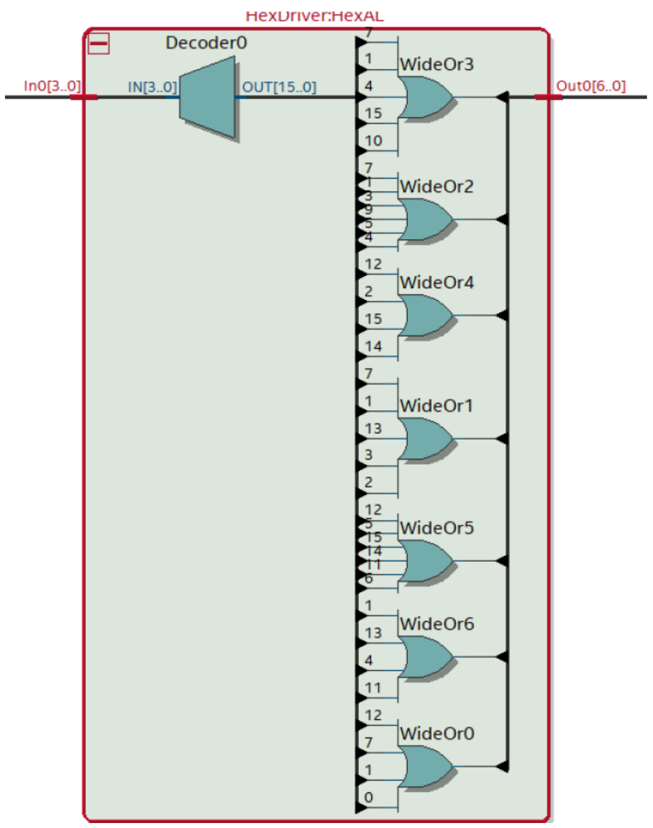
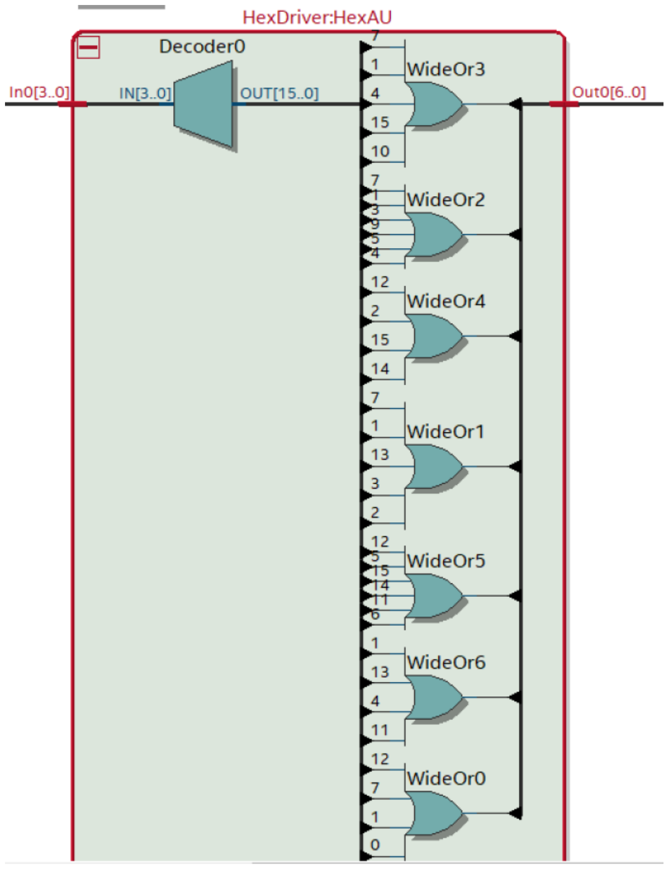

reg_8:RegA
Data_Out[0]~reg[7..0]
Clk
Reset
Load
D[7..0]
Shift_En
Shift_In
Data_Out~[15..8]
Data_Out~[23..16]
Data_Out~[7..0]
8'h0
D
CLK
SCLR
Q
Data_Out[7..0]
Shift_Out
0
1


Dreg:DregX
Reset
Load
D
Clk
Q~0
Q~1
Q~reg0
0
1
1'h0
D
CLK
SCLR
Q

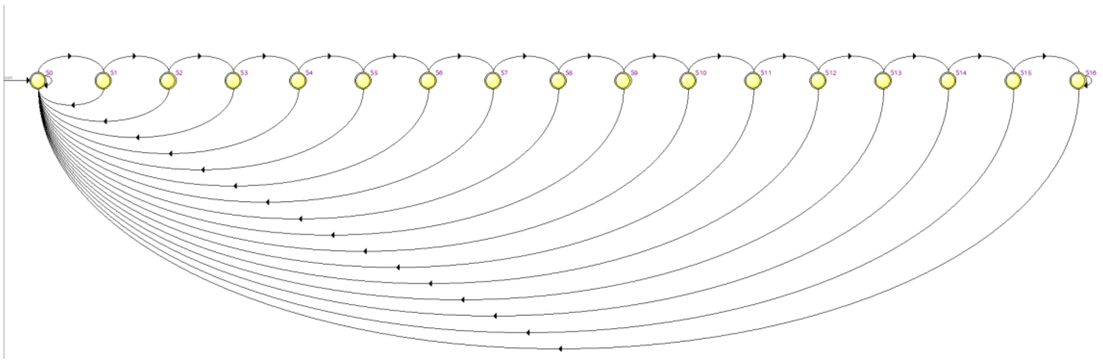

control:control
CLR_LDB
CLR_XA_LD_B
curr_state
S0
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
S11
S12
S13
S14
S15
S16
Reset
Run
clk
Reset
Run
Clk
M
WideOr0
Shift_XAB
Selector0
SEL[7..0]
DATA[7..0]
OUT
ADD
WideOr1
SUB
1'h0
SUB
CLR_XA
CLR_XA
ADD
0
1


HexDriver:HexBU
Decoder0
In0[3..0]
IN[3..0]
OUT[15..0]
Out0[6..0]
WideOr3
WideOr2
WideOr4
WideOr1
WideOr5
WideOr6
WideOr0

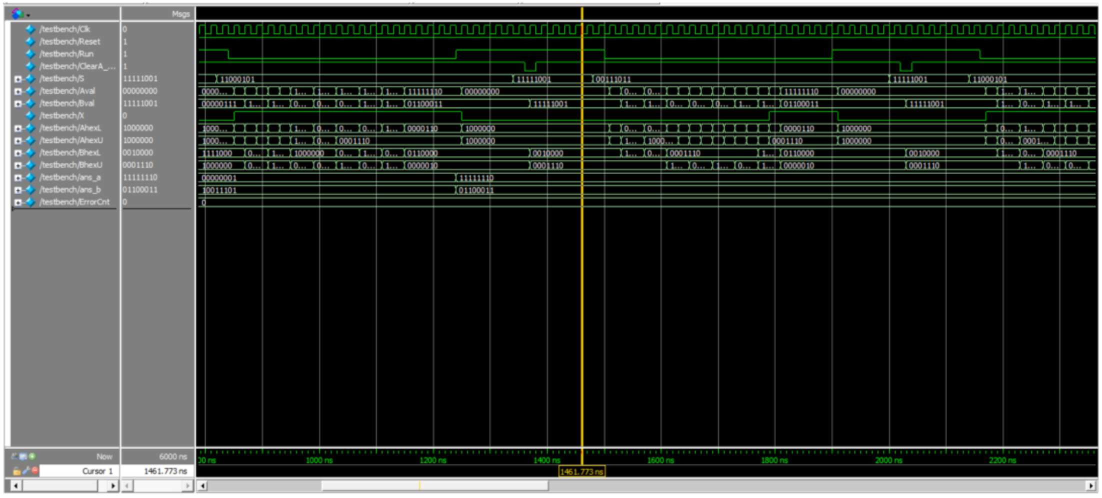

HexDriver:HexBL
Decoder0
In0[3..0]
IN[3..0]
OUT[15..0]
Out0[6..0]
WideOr3
WideOr2
WideOr4
WideOr1
WideOr5
WideOr6
WideOr0


HexDriver:HexAL
Decoder0
In0[3..0]
IN[3..0]
OUT[15..0]
WideOr3
WideOr2
WideOr4
WideOr1
WideOr5
WideOr6
WideOr0
Out0[6..0]

HexDriver:HexAU
Decoder0
In0[3..0]
IN[3..0]
OUT[15..0]
WideOr3
WideOr2
WideOr4
WideOr1
WideOr5
WideOr6
WideOr0
Out0[6..0]

d. State Diagram for Control Unit



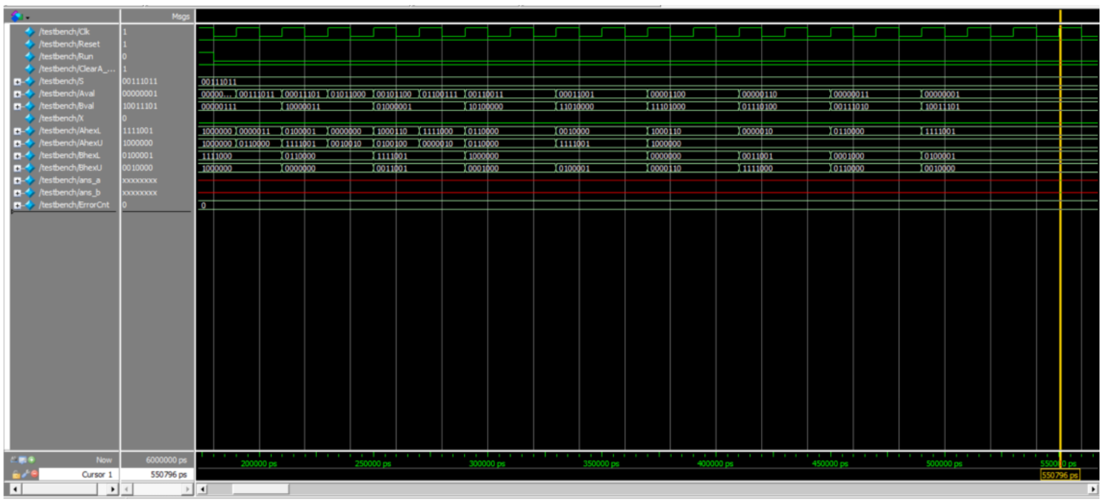
4. Annotated simulation waveforms using standard testbench



	Msgs
/testbench/Clk	0
/testbench/Reset	1
/testbench/Run	1
/testbench/ClearA_...	1
/testbench/S	11111001
/testbench/Aval	00000000
/testbench/Bval	11111001
/testbench/X	0
/testbench/AhexL	1000000
/testbench/AhexU	1000000
/testbench/BhexL	0010000
/testbench/BhexU	0001110
/testbench/ans_a	11111110
/testbench/ans_b	01100011
/testbench/ErrorCnt	0

```
Now       6000 ns
Cursor 1  1461.773 ns
```

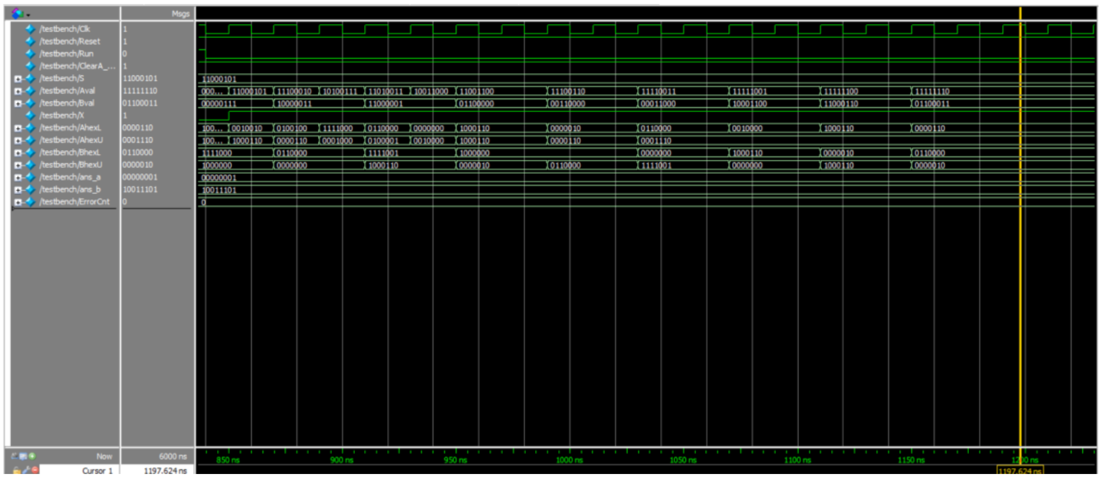
ClearA_LoadB



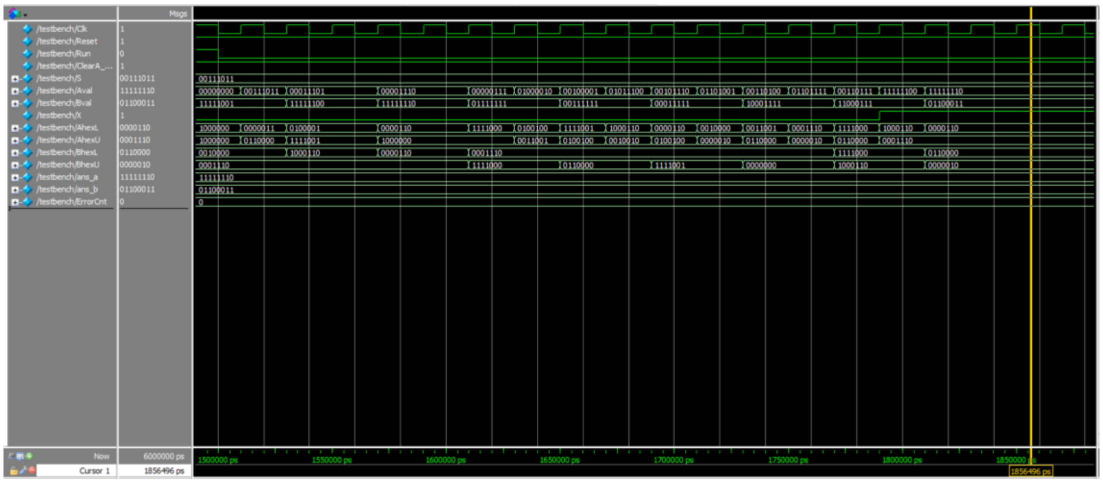
	Msgs
/testbench/Clk	1
/testbench/Reset	1
/testbench/Run	0
/testbench/ClearA_...	1
/testbench/S	00111011
/testbench/Aval	00000001
/testbench/Bval	10011101
/testbench/X	0
/testbench/AhexL	1111001
/testbench/AhexU	1000000
/testbench/BhexL	0100001
/testbench/BhexU	0010000
/testbench/ans_a	xxxxxxxxx
/testbench/ans_b	xxxxxxxxx
/testbench/ErrorCnt	0

```
Now       6000000 ps
Cursor 1  550796 ps
```

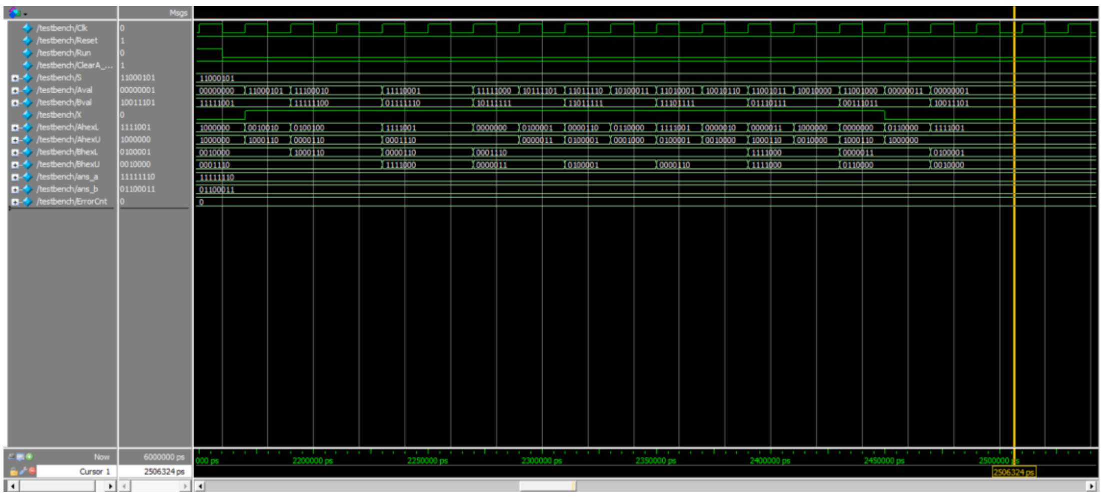
Single_Run Execution(Test1: 59*7 = 413)



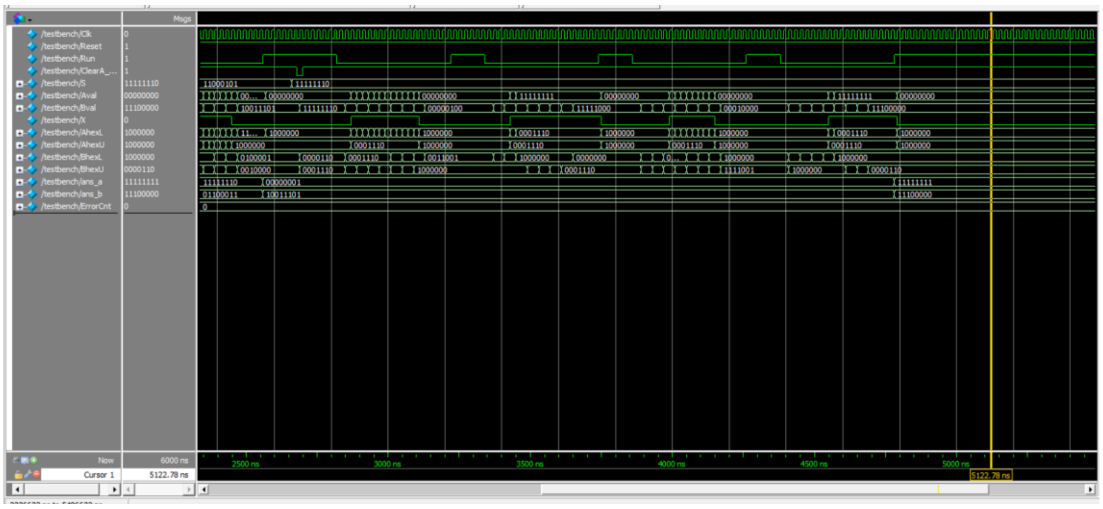
Single_Run Execution(Test2: -59*7 = 413)



Single_Run Execution(Test3: 59*-7 = -413)



Single_Run Execution(Test4: -59*-7 = 413)



Consecutive Multiplication(Test5: -2*-2*-2*-2*-2 = -32)

5. Answers to post-lab questions

1.) Refer to the Design Resources and Statistics in IQT.30-32 and complete the following design statistics table.

LUT	89
DSP	0
Memory (BRAM)	0
Flip-Flop	51
Frequency	67.98Mhz
Static Power	98.51mW
Dynamic Power	3.97mW
Total Power	146.50mW

2）

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

The X register is used to store the sign bit of value in the register A, which will produce the extended bit to register A during the shift operations. X will get cleared when Reset button or ClearA_LoadB botton is pressed, or when FSM translate from terminated state (S16) to start state (S0) once Run=0 (button Run released) detected in terminated state (Note: the setting allows to perform Consecutive Multiplication as requested).

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

The valid range of multiplications is limited, since the register X and A will be cleared after each multiplication and the result should be contained into the register B. The continuous multiplications will cause overflow when the expected result (during the multiplications) runs out of range -- (-128, 127) for 8-bit 2's complement, that is, at the beginning of next multiplication, the bits over [15:8] will be lost in this case.

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

Compared with pencil-and-paper method, the implemented multiplication algorithm helps save the memory space, that is, using only three 8-bit registers and one DFF in the implementation instead of memorizing all values of each step. However, the valid range of consecutive multiplication of this algorithm is quite limited, since the output of each multiplication (except the final result) is limited in 8-bit 2's Complement within (-128,127), which could simply cause overflow by the multiplications of two 8-bit inputs.

6. Conclusion

Discuss functionality of your design. If parts of your design didn't work, discuss what could be done to fix it

We passed all the five given tests, I think our circuits can perform all the basic 8-bit multiplication functionality as required. However, our design can be futher improved to decrease the latency, that is we can skip the add state if M=0.

Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right so it doesn't get changed.

Run, Reset and ClearA_LoadB are inverted on the given testbench, however we didn't find any declaration on the lab manual, which is quite confusing and takes us some time to figure it out.