ECE 385

Fall 2020

Experiment #5

Multiplier

Alex Wen (acwen2), Manav Agrawal (manava3)

Section ABF

Xinbo Wu

Introduction

This lab is focused on the implementation of a multiplier through our understanding of RTL Design with the utilization of SystemVerilog HDL. This multiplier will take in 2 8-bit inputs - numerical values - in 2's complement form and perform the multiplication operation by repeatedly adding and shifting through the add-shift algorithm. It can also multiply negative numbers and it has the potential to handle that. It can show 16 bit answers in XAB and it will also perform repeated multiplication.

Add-Shift Multiplication Algorithm

The multiplication operation is implemented through the add-shift bit algorithm. The idea of add-shift bit algorithm is such that the multiplier is added to each bit of the multiplicand

Consider: B = 00000111 (7 as multiplier), S = 11000101 (-59 as multiplicand). This is taken through a logical implementation of the bit-shift multiplication.

Function	X	A	B	M	Comments
Clear A, Load B, Reset	0	0000 0000	00000111	1	M = 1 so we need to add
ADD	1	1100 0101	0000 0111	1	Shift XAB by one bit after ADD complete
SHIFT	1	1110 0010	*1*000 0011	1	We want to add S again M = 1
ADD	1	1010 0111	*1*000 0011	1	Shift XAB by one bit after ADD complete
SHIFT	1	1101 0011	*11*00 0001	1	We want to add S again M = 1
ADD	1	1001 1000	*11*00 0001	1	Shift XAB by one bit after ADD complete
SHIFT	1	1100 1100	*011*0 0000	0	Since M = 0, just shift
SHIFT	1	1110 0110	*0011* 0000	0	Since M = 0, just shift
SHIFT	1	1111 0011	*0001 1*000	0	Since M = 0, just shift
SHIFT	1	1111 1001	*1000 11*00	0	Since M = 0, just shift
SHIFT	1	1111 1100	1100 0110	0	Since M = 0, just shift

SHIFT	1	1111 1110	*0110 0011*	1	This was the 8th shift so we don't need M, stop
Halt	1	1111 1110	0110 0011	1	We are at Halt, 16 bit product in XAB

Table 1 - Logical Implementation of Bit-Shift Multiplication

Multiplier Implementation

In the general scheme of the multiplier, the operands (multiplier and multiplicand) are loaded as inputs through switches on the FPGA. In this case, the multiplier is loaded into an 8-bit register B, while the 8-bit register A and sign-extension overflow X is cleared. The switches are then set to the multiplicand, and then run through the design. Once the run button is triggered, the inputs are processed through an 8-bit adder with a sign-extension bit to compute the operation using the bit-shift multiplication as defined by the established algorithm. The results are then stored cumulatively in the 2 8-bit registers (A and B).

As described, the Multiplier composed of different units such as D-flip flops for the state diagram or the control unit, a 9-bit adder/subtractor, register A and register B, flip-flop specific for X, some combinational logic etc. Below is the RTL viewer or the overall top level block diagram of the multiplier.

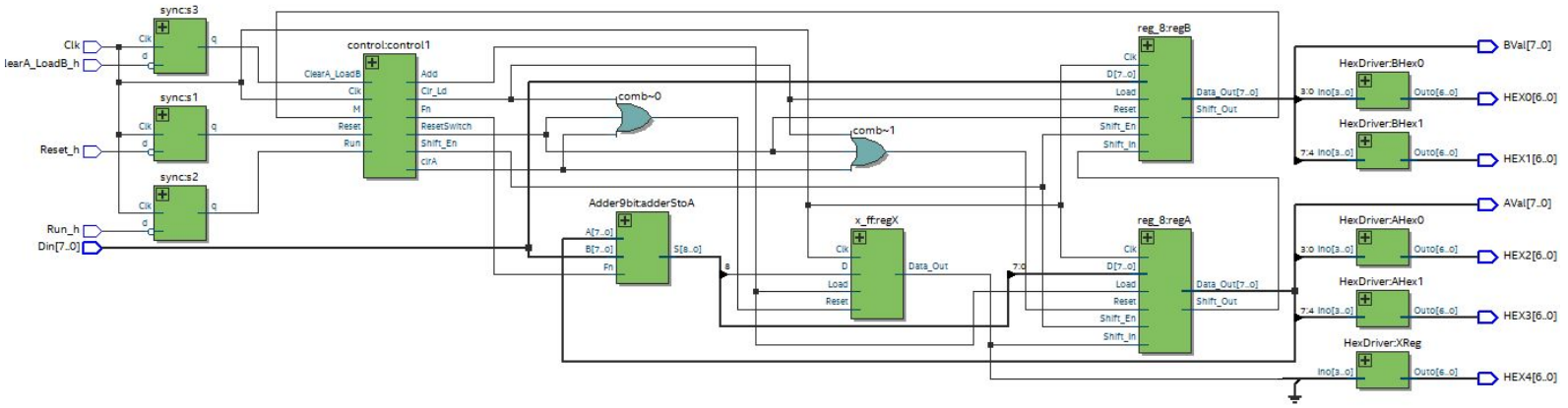


Figure 1: RTL Viewer of the Multiplier given by Intel Quartus Prime

From the block diagram, the components can be clearly seen. In addition, the synchronizers were also used to sync the inputs with the clock and make it better for the operation of the circuit.

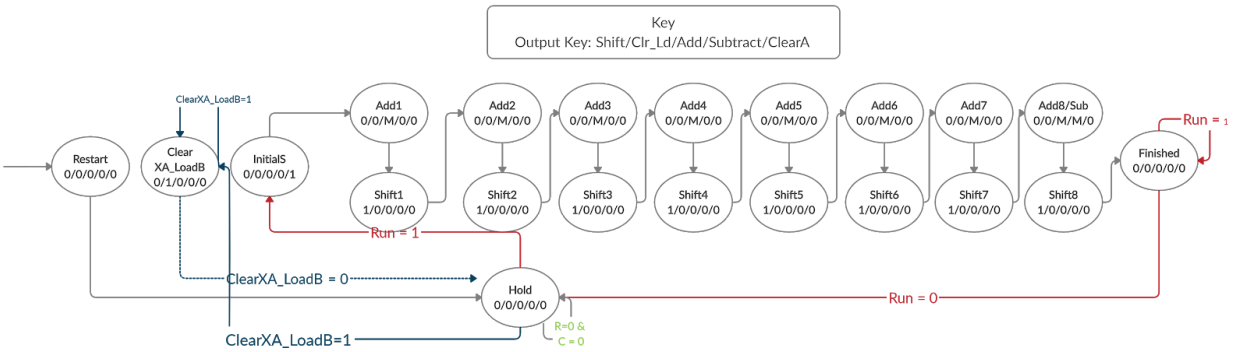


Figure 2: Implemented State Diagram of the Multiplier

This was the state diagram which was implemented. The program starts with a default reset state which then goes onto the hold state. As shown, there were 16 states where 8 were ‘add’ and 8 were ‘shift’. The other states could have been minimized but they weren’t as each state had some specific function and it gives a better demo. The reset stage was to give everything a reset but for next time we can just make it a hold state. But overall from the key, it can be seen that you clear XA and LOADB when the button is pressed from the hold state. Then it just comes back to hold. After Run is active high, the program runs and if M is 1 the add is enabled otherwise it isn’t. For the 8th add/sub state the M is also used as fn determination to add/subtract. This shows that some of the output determines from the input and seems Mealy because the M is input to this state machine. This was an easier state machine to implement on SystemVerilog and the multiplier worked during the demonstration.

When there is a transition from state hold to InitialS, it is assumed that run is 1 and the other inputs do not interfere with the state machine as instructed by the lab manual that both will never be active high. The grey transitions are the unconditional transition arcs hence no input is required.

The modules implemented for the multiplier are thoroughly described in Appendix I.

Simulation

It is very important to test and simulate SystemVerilog. Therefore a testbench was assembled and the four different operations were run and measured.

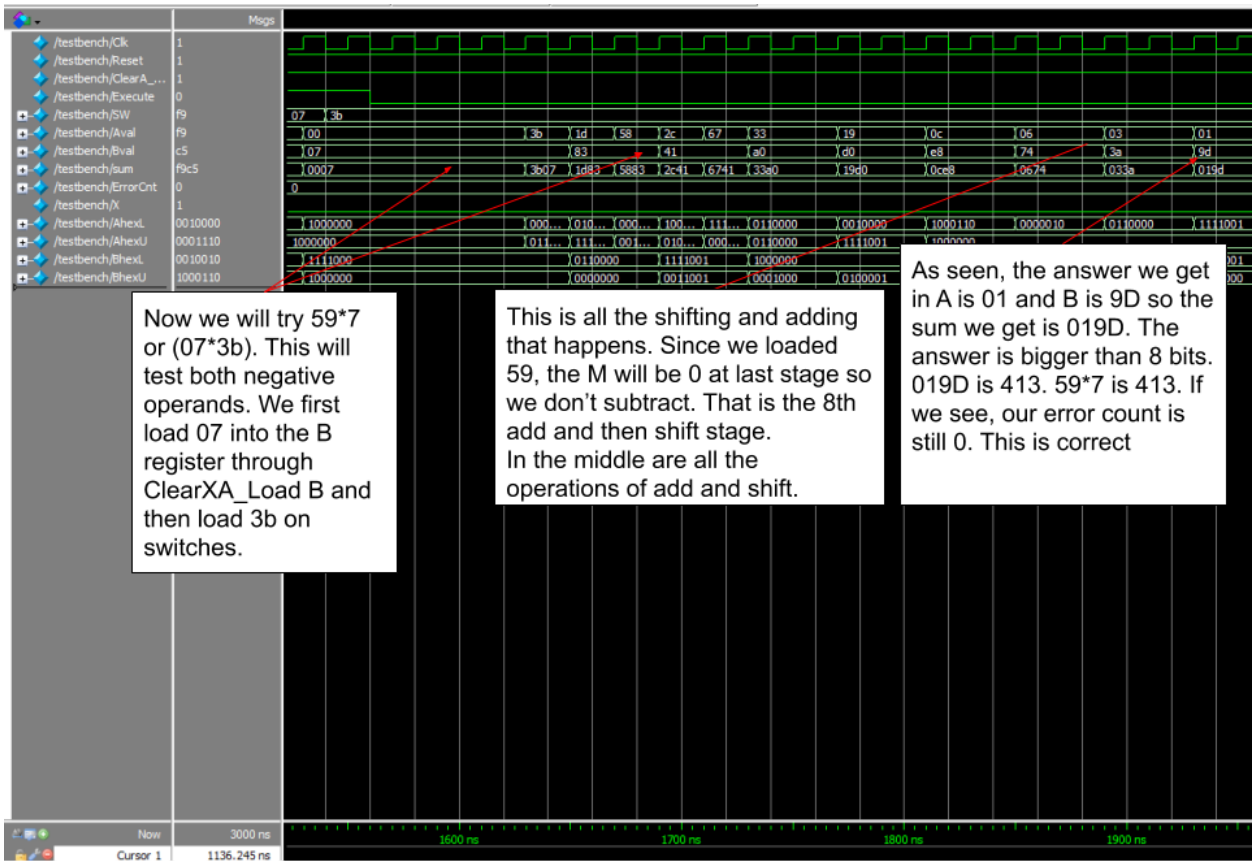


Figure 3a: Functional Simulation of +*+

From the lab, they had asked to simulate 59*7, -59*7, -7*59 and -7*-59. So firstly, from the above figure the 59*7 can be seen. This gives an answer of 413 or 019D and since it is not 8 bits, the answer can be visible in the registers A and B combined. ClearXA_Load B was used to load the value of 07 in Register B and from switches 59 was loaded and the answer seemed correct and as seen, there was no error because error count remained 0. Both were positive numbers and in the middle, you can see the 8 shifts and some additional changes due to the add operations. The next operation will be -59*7.

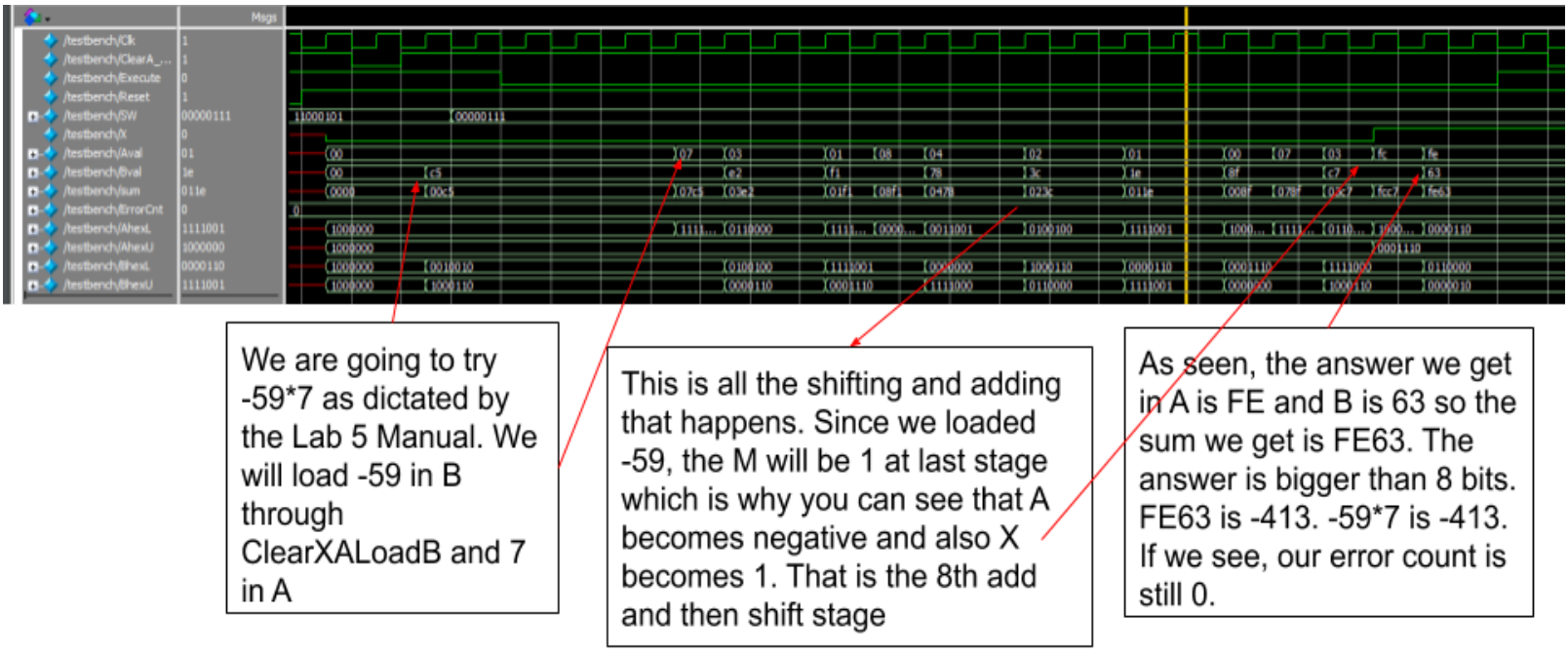


Figure 3b: Functional Simulation of +*-

This figure above shows the multiplication of -59*7. Initially, the reset was 0 which made everything to set stage but then when it became active high, the operation started from Hold Stage. From there, the operation began. As suggested by the annotations, -59 was loaded into the B register and 7 was loaded into A. This was more interesting of a simulation since the B had a negative number so M=1 during the 8th add state which caused it to subtract the number and give a negative value of -413 or FE63. Now the next figure will be of 59*-7. The operands are switched.

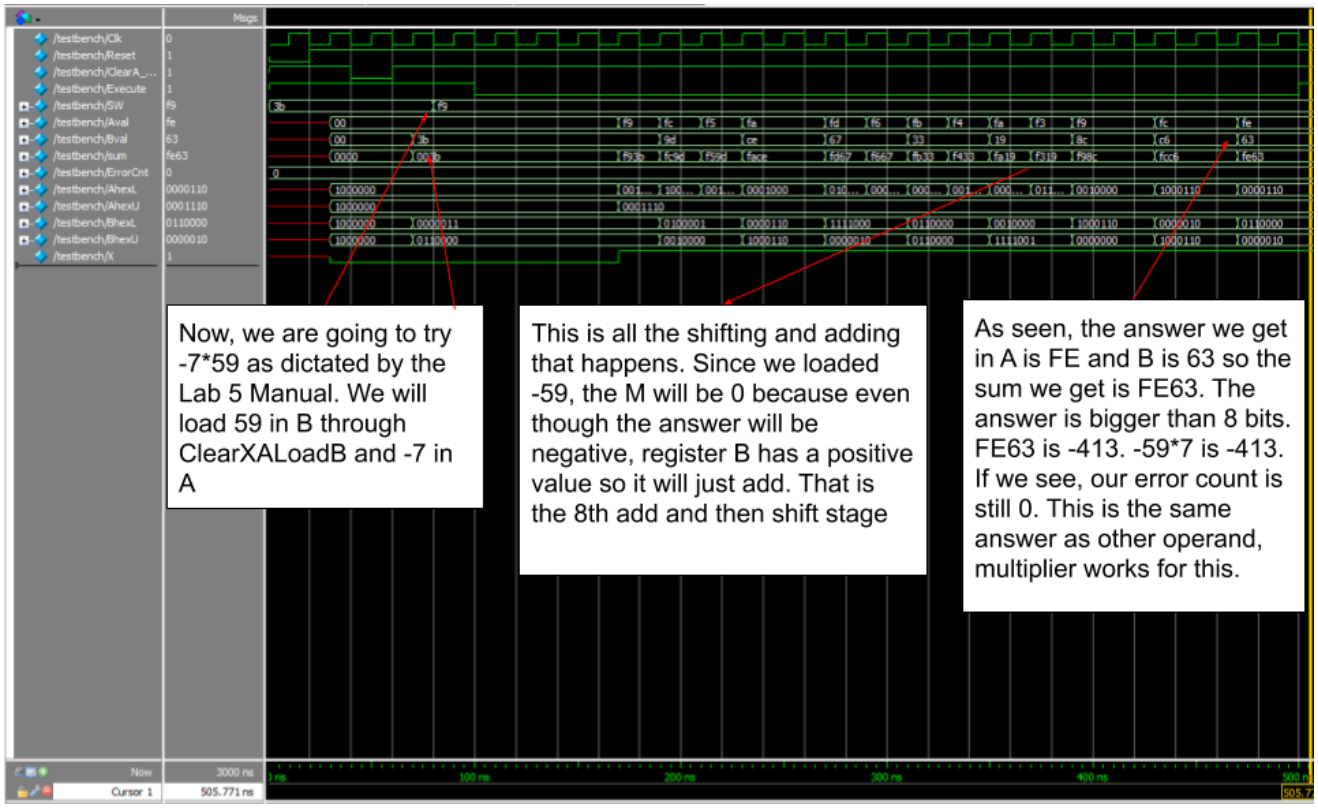


Figure 3c: Functional Simulation of -*+

The reset is 0, then it becomes active high from where it goes to hold state. This figure shows that now we load the positive 59 into register B and -7 through switches and multiply. We get the same answer as expected which is -413 or FE63. The error count is 0 making sure that there is correct and consistent multiplication. It also seems that the register values are stored correctly.

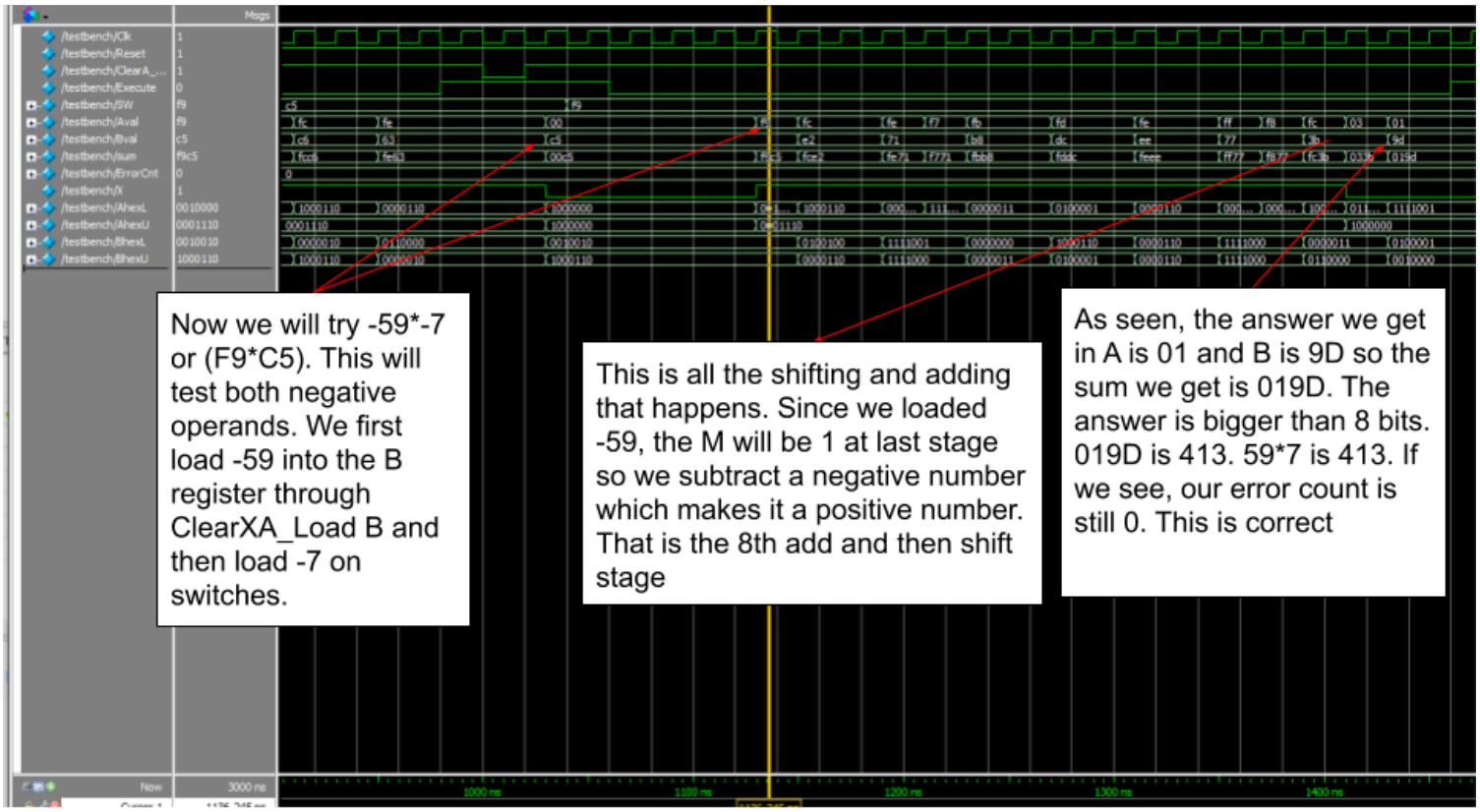


Figure 3d: Functional Simulation of -*-

This was one of the last operand that as simulated that shows -59*-7. It should yield a positive answer. The -59 was loaded into B and -7 through switches and then it was run. In addition to that, we observe that this time, the 8th state of ADD will have to subtract since M will be 1. So when they will subtract a negative number, it will become positive which is what we expect to get which is 019D or +413.

Post-Lab Inquiries

Like the last lab, it is important to analyze the statistics and the power/memory usage of the circuit and think how can this be improved and make sense of the actual results. Hence the next discussion will be regarding the Intel Quartus Prime's analysis of these measures.

	Multiplier
LUT	81
DSP	No DSP Blocks were used
Memory (BRam)	0
Flip-Flop	41
Frequency	82.37 MHz
Static Power	89.97 mW
Dynamic Power	1.56 mW
Total Power	104.17 mW

Table 2: Statistical Analysis of the Multiplier

As you can see, these were the statistics recorded through Quartus Prime about the Multiplier function. There are a lot of improvements that we could think of as partners which can improve the overall Mealy state diagram and improve the performance by removing the number of Flip-Flop potentially. Below is the improved FSM diagram which shows that we do not need to go to some of the ADD states and overall less states.

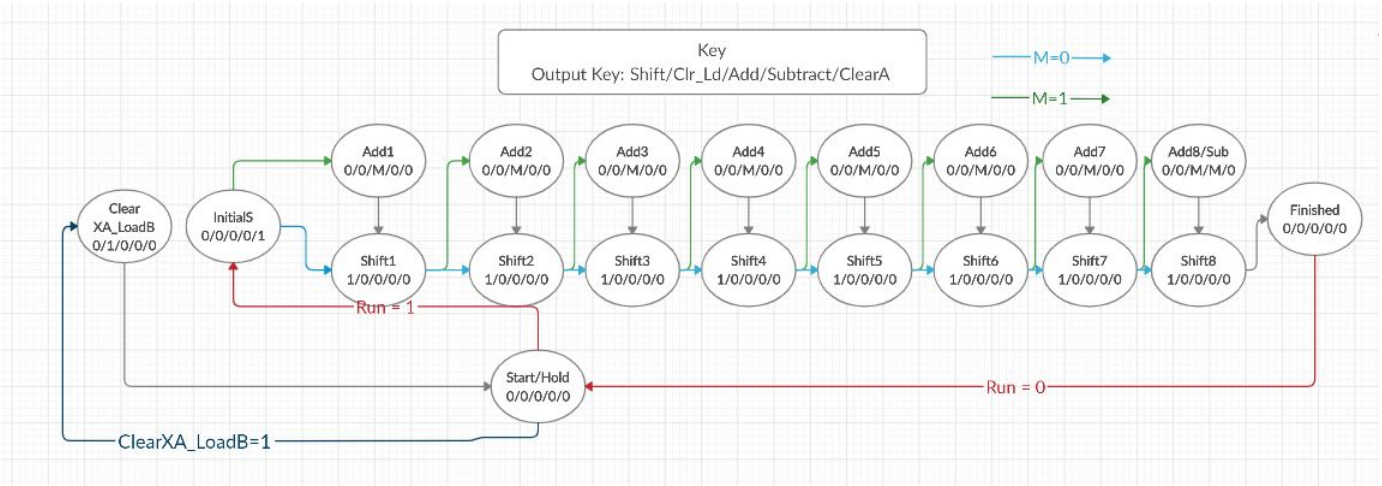


Figure 4: Potential Optimized State Diagram

This is one of the diagrams with lower states, better transitions and we do not have to go through the 16 states. This was the plan but there was some problem initially and for safeguard, more states were added but this can act as a more efficient design.

Another improvement is that in experiment 3 we had a counter to keep track of 4 cycles for an operation to happen. This can be the same for this lab also as if we have a counter that counts up to 16 to 2 counters that count up to 8 then we can combine the add and shift states to be one entire large state. The disadvantage of this is the FSM is simple to implement but the counter would have added extra work and it needed to be checked whether synchronized, working properly and FSM would have had more inputs and combinational logic.

In terms of design, the purpose of the X register is to store the sign-extended bit. It essentially functions as a storage for the sign-extension of A, XA. It gets updated subsequent to every add-operation; in other words, as A shifts right, the sign-extension gets preserved.

The limitation of continuous multiplication is such that there is only a definite number of bits that each register can hold. The registers can hold up to 8-bits each, along with the sign-extension X (total of 17 bits); if continuous multiplication is extended such that it overflowed into XA, the contents may be zeroed out and the resulting multiplication would therefore be incorrect. Therefore it can only handle 8 bits of continuous/repeated multiplications.

The advantage of implementing the multiplication algorithm over pencil-and-paper is such that it easily covers the fundamental principle and process of yielding the accurate result. On the other hand, it is infeasible to implement this in terms of digital design, especially with large binary numbers as it is very complex - it is more practical to implement decimal multiplication if using the pencil-paper method, which does not compute easily in digital design. The pencil and paper

method, you can add Xs at the back and keep on adding with one large sum like in the introduction. But in digital or logic design, there cannot be just an X added randomly towards the end hence we need to create some defined space at registers A and B. There also needs to be state diagram checking the last bit and making sure we shifted the 8 hence we implemented this algorithm. The paper-pencil works the best for the decimal number system however we need to think of digital logic design or binary.

Conclusion

The multiplier worked well in the demonstration but the testbench did not work well during the demonstration because the inputs of reset and ClearXA_LoadB were messed up. The other parts of the design did work. Some of the errors which were observed were the state diagram because if we did not have hold or finished, there was a problem with the reset stage and transitioning to the initial start state. We were required to have repeated multiplication which is why an Initial Start state and hold state was devised. In hindsight, the reset and hold could be combined and there could be better transitions to improve efficiency.

In terms of the logistics of this lab, it is mostly as straightforward as it can get; however, this lab is built on the premise of a full and complete understanding of the bit-shift multiplier. It would perhaps be better to have a general overview of the *logical* implementation of the bit-shift multiplication in its entirety before having to understand the step-by-step process with subsequent explanation.

APPENDIX

1. Description of SV Modules

```
module multiplier(input logic Clk, ClearA_LoadB_h, Run_h, Reset_h,
                  input logic [7:0] Din,
                  //output logic M, D, //This will go to LEDs for testing the debugging
                  output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4,
                  output logic [7:0] AVal, BVal
                  );

      logic addornot, fn, Shift_en, Clear_load, Mthbit, XtoA, clrA, ResetSwitch;
      logic Reset, Run, ClearA_LoadB;
      logic bitB;
      logic [8:0] sumA;
      logic [7:0] A, B;

      sync s1(.Clk(Clk), .d(~Reset_h), .q(Reset));
      sync s2(.Clk(Clk), .d(~Run_h), .q(Run));
      sync s3(.Clk(Clk), .d(~ClearA_LoadB_h), .q(ClearA_LoadB));


      control control1(
         .Clk(Clk),
         .ClearA_LoadB(ClearA_LoadB),
         .ResetSwitch(ResetSwitch),
         .Reset(Reset),
         .clrA(clrA),
         .Run(Run), .M(Mthbit),
         .Add(addornot),
         .Fn(fn),
         .Shift_En(Shift_en),
         .Clr_Ld(Clear_load)
      );

      Adder9bit adderStoA(
         .A(A),
         .B(Din[7:0]),
         .Fn(fn),
         //.CO(),
         .S(sumA[8:0])
      );

      x_ff regX(
      .Clk(Clk),
      .Reset(Clear_load | ResetSwitch|clrA),
      .Load(addornot),
      .D(sumA[8]),
      .Data_Out(XtoA)
      );

      reg_8 regA(
         .Clk(Clk), .Reset(ResetSwitch|Clear_load|clrA), .Load(addornot), .Shift_In(XtoA), .Shift_En(Shift_en),
         .D(Din[7:0]),
         .Shift_Out(Mthbit),
         .Data_Out(B[7:0])
      );

      always_comb
         begin
            AVal = A;
            BVal = B;
         end

      HexDriver      AHex0 (
                        .In0(A[3:0]),
                        .Out0(HEX2) );

      HexDriver      AHex1 (
                        .In0(A[7:4]),
                        .Out0(HEX3) );

      HexDriver      BHex0 (
                        .In0(B[3:0]),
                        .Out0(HEX0) );

      HexDriver      BHex1 (
                        .In0(B[7:4]),
                        .Out0(HEX1) );
      HexDriver      XReg (
                        .In0({3'b0, XtoA}),
                        .Out0(HEX4) );

endmodule
```

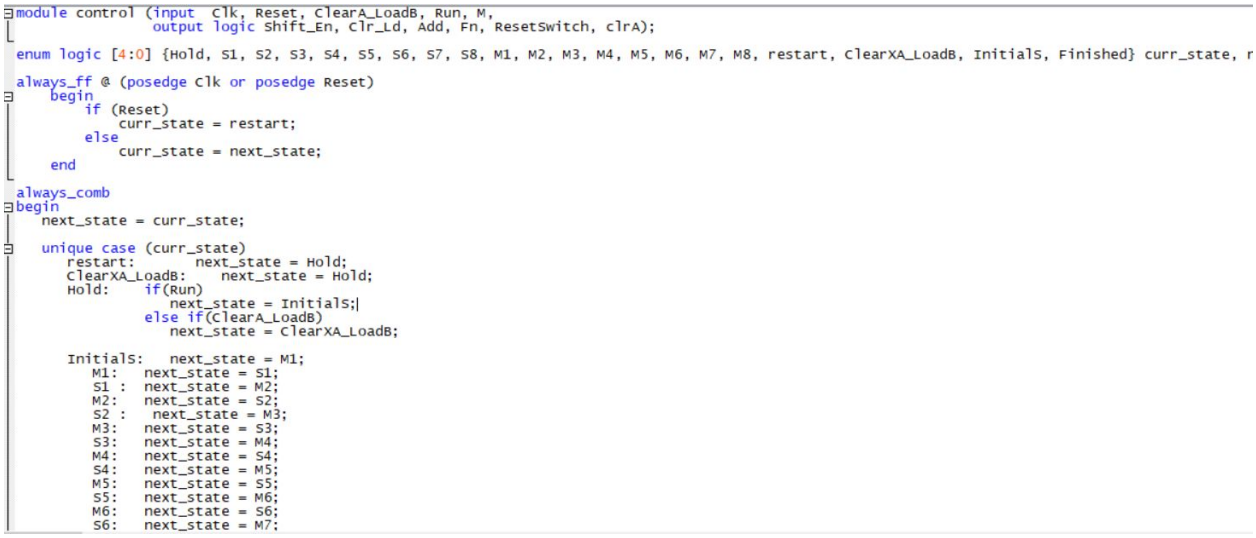
Module: multiplier.sv

Input: Clk, ClearA_LoadB_h, Run_h, Reset_h, [7:0] Din

Output: [6:0] Hex0 Hex1 Hex2 Hex3 Hex4, [7:0] Aval Bval

Description: Top-level entity. This is the overall multiplier, where it takes in the inputs from switches and buttons, and yields the output on the Hex Boards and LEDs. It also serves to function as the designator in terms of which unit is running, among the storage, control, and operation units.

Purpose: This module is used to oversee the entirety of the multiplier. It takes the inputs from switches and buttons, along with a clock cycle, and runs it through the control, operation, and storage units to yield outputs.



```
module control (input  Clk, Reset, ClearA_LoadB, Run, M,
                output logic Shift_En, Clr_Ld, Add, Fn, ResetSwitch, clrA);

enum logic [4:0] {Hold, S1, S2, S3, S4, S5, S6, S7, S8, M1, M2, M3, M4, M5, M6, M7, M8, restart, ClearXA_LoadB, Initials, Finished} curr_state, r

always_ff @ (posedge Clk or posedge Reset)
    begin
        if (Reset)
            curr_state = restart;
        else
            curr_state = next_state;
    end

always_comb
begin
    next_state = curr_state;

    unique case (curr_state)
       restart:       next_state = Hold;
       ClearXA_LoadB:    next_state = Hold;
       Hold:     if(Run)
                    next_state = Initials;
                 else if(ClearA_LoadB)
                    next_state = ClearXA_LoadB;

       Initials:   next_state = M1;
          M1:   next_state = S1;
          S1 :  next_state = M2;
          M2:   next_state = S2;
          S2 :   next_state = M3;
          M3:   next_state = S3;
          S3:   next_state = M4;
          M4:   next_state = S4;
          S4:   next_state = M5;
          M5:   next_state = S5;
          S5:   next_state = M6;
          M6:   next_state = S6;
          S6:   next_state = M7;
```

Module: Control.sv (This is a control sv file for our state diagram)

Input: Clock, Reset, ClearALoadB, Run, M(If the Multiplicand is negative or positive)

Output: Shift_En (This output tells whether to shift or not), Clr_Ld(This tells whether to clear or load), Add (This will tell whether to add or not), Fn (This will tell whether subtract or not), ClrA (Sometimes you have to clear A), ResetSwitch(This resets the entire circuit)

Purpose: The purpose of this Control.Sv is to store the entire diagram which will dictate what operation will happen when and what registers will store what. This has 16 (8 ADD and Shift States) plus 4 to five additional states for reset, start, clearXA_LoadB.

Description: The state diagram is implemented through 2-always. The first is a flip-flop which stores the current state. This is a sequential logic. Then the combinational logic is implemented which dictates the outputs and the next-state transitions which is done through always_comb blocks.

```
module sync (
    input  logic clk, d,
    output logic q
);

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

endmodule
```

Module: sync.sv

Input: Clk, dataInput

Output: q

Purpose: The purpose of sync.sv is to synchronize the input switches to the clock so we do not have meta-stability or asynchronous inputs.

Description: A D flip-flop was used to synchronize the input with the clock.

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

              always_ff @ (posedge Clk or posedge Reset)
              begin

                 if(Reset)
                    Data_Out <= 8'b00000000;

                 else if(Load)
                    Data_Out <= D;

                 else if(Shift_En)
                    Data_Out <= {Shift_In, Data_Out[7:1]};


              end

              assign Shift_Out = Data_Out[0];

endmodule

module x_ff(input Clk, Reset, Load,
         input D,
         output logic Data_Out);
         logic din;

         always_ff @ (posedge Clk)
         begin
            Data_Out <= din;
         end

         always_comb begin
            if(Reset)
               din = 1'b0;
            else if(Load)
               din = D;
            else
               din = Data_Out;
         end
endmodule
```

Module: reg_8.sv

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

Output: Shift_Out, [7:0] Data_Out

Description: This is the standard positive-edge triggered 8-bit register storage unit that takes in and maintains values from switches and adders. As the operation moves along, the registers will either parallel-load values determined by the operation block or shift accordingly.

Purpose: This module is used to construct 8-bit registers in order to maintain A and B values for multiplier and multiplicand.

Module: reg_8.sv → x_ff

Input: Clk, Reset, Load, D

Output: Data_Out

Description: This is the basic 1-bit register storage that takes in a single bit from the adder in cohesion with register A. The operation based in the adder will yield 9-bits for XA, A for register value and X for sign-extension

Purpose: This module is used to form a 1-bit register storage unit to store the sign-extension from register A.

```
module Adder9bit ( input [7:0] A, B,
                   input Fn,
                   output [8:0] S);

logic c0,c1;
logic [7:0] BTemp;
logic A9, BTemp9;

assign BTemp = (B ^ {8{Fn}});
assign A9 = A[7];
assign BTemp9 = BTemp[7];

   adder4 AD0(.A(A[3:0]), .B(BTemp[3:0]), .cin(Fn), .S(S[3:0]), .cout(c0));
   adder4 AD1(.A(A[7:4]), .B(BTemp[7:4]), .cin(c0), .S(S[7:4]), .cout(c1));
   full_adder AD2(.x(A9), .y(BTemp9), .z(c1), .s(S[8]), .c());

endmodule

module adder4 (input [3:0] A, B,
               input cin,
               output cout,
               output [3:0] S);
   logic c0, c1, c2;

   full_adder FA0(.x(A[0]), .y(B[0]), .z(cin), .s(S[0]), .c(c0));
   full_adder FA1(.x(A[1]), .y(B[1]), .z(c0), .s(S[1]), .c(c1));
   full_adder FA2(.x(A[2]), .y(B[2]), .z(c1), .s(S[2]), .c(c2));
   full_adder FA3(.x(A[3]), .y(B[3]), .z(c2), .s(S[3]), .c(cout));
endmodule


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

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

endmodule
```

Module: Adder9bit.sv

Input: [7:0] A B, Fn

Output: [8:0] S

Description: This is the design formulation of the 9-bit adder used for the operation unit, where it takes in two 8-bit inputs and provides 9-bit output.

Purpose: This module provides the adder based on adder4 and full_adder to provide the unit where the bit-shift multiplication takes place, leading to an 8-bit result going to A and 1 sign-extended bit going to register X.

Module: Adder9bit.sv → adder4

Inputs: [3:0] A, [3:0] B, c_in

Outputs: c_out, [3:0] S

Description: This is the design block for one 4-bit hierarchical adder that takes in 2-inputs A and B, and a carry-in value to yield a sum. The output is the result S of said sum along with a carry-out overflow.

Purpose: This module sequentially takes 4 individual binary adder units from full_adder to form the 4-bit hierarchical units.

Module: Adder9bit.sv → full_adder

Inputs: x, y, z

Outputs: s, c

Description: This is the fundamental single-bit adder. It takes in 1-bit values x and y, and a carry-in z, and produces outputs consisting of the sum s and the carry-out bit c.

Purpose: This module is used to create logic for a single-bit adder for the 9-bit adder in which the bit-shift algorithm takes place.

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

      // This module has hardcoded values that convert 4bit binary numbers from registers into hexadecimal
      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: This is the Hex Board on the FPGA that indicates the input switches and outputs, which are shown on the Hex Board on the FPGA.

Purpose: This module shows switches and registers on the Hex Board.

```
module testbench();

timeunit 10ns;

timeprecision 1ns;
logic Clk = 0;
logic Reset, ClearA_LoadB, Execute;
logic X;
logic [7:0] SW;
logic [7:0] Aval,Bval;
logic [6:0] Ahex1,Ahex0,Bhex1,Bhex0;
logic [15:0] sum;

integer ErrorCnt = 0;

multiplier multiplier0(.*);


always begin : CLOCK_GENERATION
#1 Clk = ~Clk;
end

initial begin: CLOCK_INITIALIZATION
    Clk = 0;
end
```

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

#2 SW = 8'hf9;

#2 Execute = 0;    // -59*7

#40 Execute = 1;
if(Aval != 8'b11111110)
begin
ErrorCnt++;
end
if(Bval != 8'b01100011)
begin
ErrorCnt++;
end
```

Module: Testbench.sv

Input: Clk, Reset, ClearA_LoadB, Execute which will go into the multiplier

Output: AValues, BValues (These are from register), the 4 Hex Values of AHex1, AHex0, BHex1, BHex0, the sum which is just A+B combined and the X register/one bit flip-flop

Purpose: The purpose of testbench.sv is just to simulate and see if it works for different operands or not. We tried -59*7, -7*59, 59*7, -7*-59. A block was shown on the side that was that tested. We also used if and else to detect error as it is better for us to show whether an error is caused or not.

Description: All the variables were defined at the start. 4 blocks like the right picture was used to test each operation. If was used to count the error.

Last file: Multiplier.sdc

```
create_clock -name {clk} -period 20ns -waveform {0.000 5.000} [get_ports {clk}]
# Constrain the input I/O path
set_input_delay -clock {clk} -max 3 [all_inputs]
set_input_delay -clock {clk} -min 2 [all_inputs]
# Constrain the output I/O path
set_output_delay -clock {clk} 2 [all_outputs]
```

This is a constraint file that creates a 50 mHz clock and is used for testing and analysis purposes.