ECE 385 Experiment 4: Lab Report

Spring 2023
Eshaan Tibrewala (eshaant2) and Shivam Patel (shivamp6)
TA: Shitao Liu

Introduction

The purpose of this experiment was to design a multiplier in System Verilog that is capable of handling two 8-bit 2's compliment numbers. We used a simple add-shift algorithm to implement the multiplication. The process was similar to pen and pencil multiplication, where we add the values as we move towards the more significant bits. However since we are dealing with 2's complement numbers, the last step in our multiplier includes subtraction, depending on the sign bit of the number. Our multiplier is also capable of performing consecutive multiplications, meaning that our control unit does not need to be reset between multiplication cycles.

Pre-Lab Questions

1. **Rework the multiplication example on page 5.2 of the lab manual, as in compute 11000101 * 00000111 in a table like the example. Note that the order of the multiplicand and multiplier are reversed from the example.**

Table 1: Example Computation of -59 x 7

FUNCTION	X	A	B	M	COMMENTS
ClrA, LdB, Reset	0	0000 0000	0000 0111	1	S added to A
ADD	1	1100 0101	0000 0111	1	1 Shift XAB
SHIFT	1	1110 0010	1 0000011	1	S added to A
ADD	1	1010 0111	1 0000011	1	2 Shift XAB
SHIFT	1	1101 0011	11 000001	1	S added to A
ADD	1	1001 1000	11 000001	1	3 Shift XAB
SHIFT	1	1100 1100	011 00000	0	4 Shift XAB
SHIFT	1	1110 0110	0011 0000	0	5 Shift XAB
SHIFT	1	1111 0011	00011 000	0	6 Shift XAB

FUNCTION	X	A	B	M	COMMENTS
SHIFT	1	1111 1001	100011 00	0	7 Shift XAB
SHIFT	1	1111 1100	1100011 0	0	8 Shift XAB
SHIFT	1	**1111 1110**	**01100011**	1	Product in AB

Multiplier Circuit

Summary of Operation

Our circuit performs multiplication using the add-shift method. It takes the 8-bit data contents in Register B, and multiplies them with with switches signal S, storing the result in Register AB. To first load Register B, the user sets the desired value using the switches and then toggles the `ClearA_LoadB` button. This signal resets the contents of Register A, and loads the values of the switches into Register B. Since Register A gets cleared, bit X is also 0 since X is the sign extended bit of Register A. M represents the least significant bit of Register B, and determines if the multiplier will ADD and then shift XAB, or skip the ADD operation and just shift XAB. If M is high, the value of the switches will be added to the value in Register A. If M is low, the multiplier will skip the ADD and just shift the registers. In the ADD state, the switches are added to Register A by sign extending the values of both the register and switches to 9 bits, and then storing the result into XA, which is a 9-bit register. In the SHIFT state, all 17-bits of XAB are shifted to the right. Since our multiplier works on 2's complement numbers, we also have to account for negative values. If A is negative, the correct partial is stored in XA and the sign will be preserved during the shift. If B is negative, then M will be 1 following the seventh shift, and a subtraction operation will be performed to calculate the correct value. We designed the adder/subtractors using full adders instead of the SystemVerilog arithmetic operations.

Top Level Block Diagram

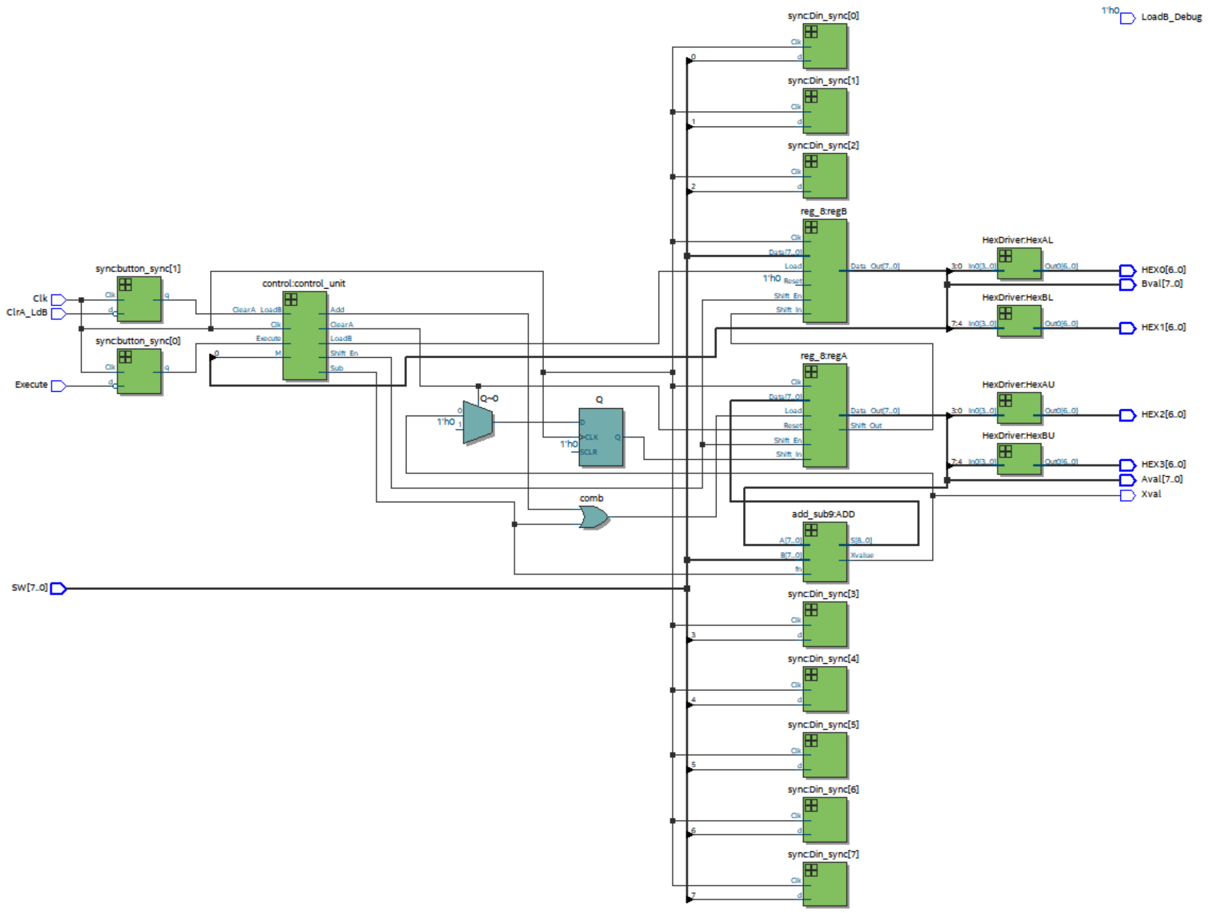

Figure 1: Multiplier Top Level RTL Diagram

Figure 1 shows the top level diagram of our circuit. The inputs from from the `Execute` and `ClrA_LdB` buttons and `SW[7:0]`. The final result is stored in registers AB, and displayed via hexadecimal values on the FPGA.

Written Description of .SV Modules

Table 2: .SV Modules for Multiplier

MODULE	UNIT/PURPOSE
Multiplier.SV	Top Level Module
Control.SV	Control Unit/FSM
Reg_8.SV	Registers
HexDriver.SV	LED output on FPGA

MODULE	UNIT/PURPOSE
Adder.SV	Adder and Subtractor
Synchronizers.SV	Asynchronous Clock Signals
Testbench.SV	Testbench

Multiplier.SV

Code Block 1: module multiplier

```
module multiplier(input logic Clk, ClrA_LdB, Execute,
            input logic [7:0] SW,
            output logic[7:0] Aval, Bval,
            output logic Xval,
            output logic [6:0] HEX0, HEX1, HEX2, HEX3);
```

`Multiplier` is the top level module of our design. The `Clk` is an internal signal, and the `ClrA_LdB` and `Execute` signal come from the buttons on the FPGA. The `SW` also come from the FPGA and are used to load in the values for register B at the beginning of our operation, and also determine S, the value of the multiplicand. The result is stored in `Aval` and `Bval`, and displayed on the `HEX` values on the FPGA. When `ClrA_LdB` is pressed, Register B is set, and once `Execute` is pressed the circuit starts the cycle to perform multiplication.

The purpose of this module is to instantiate all of our other modules so that our multiplier can function successfully. Also includes the flip flop which stores the value of X bit.

Control.SV

Code Block 2: module control

```
module control(input logic Clk, Execute, ClearA_LoadB, M,
            output logic Shift_En, Add, Sub, ClearA, LoadB);
```

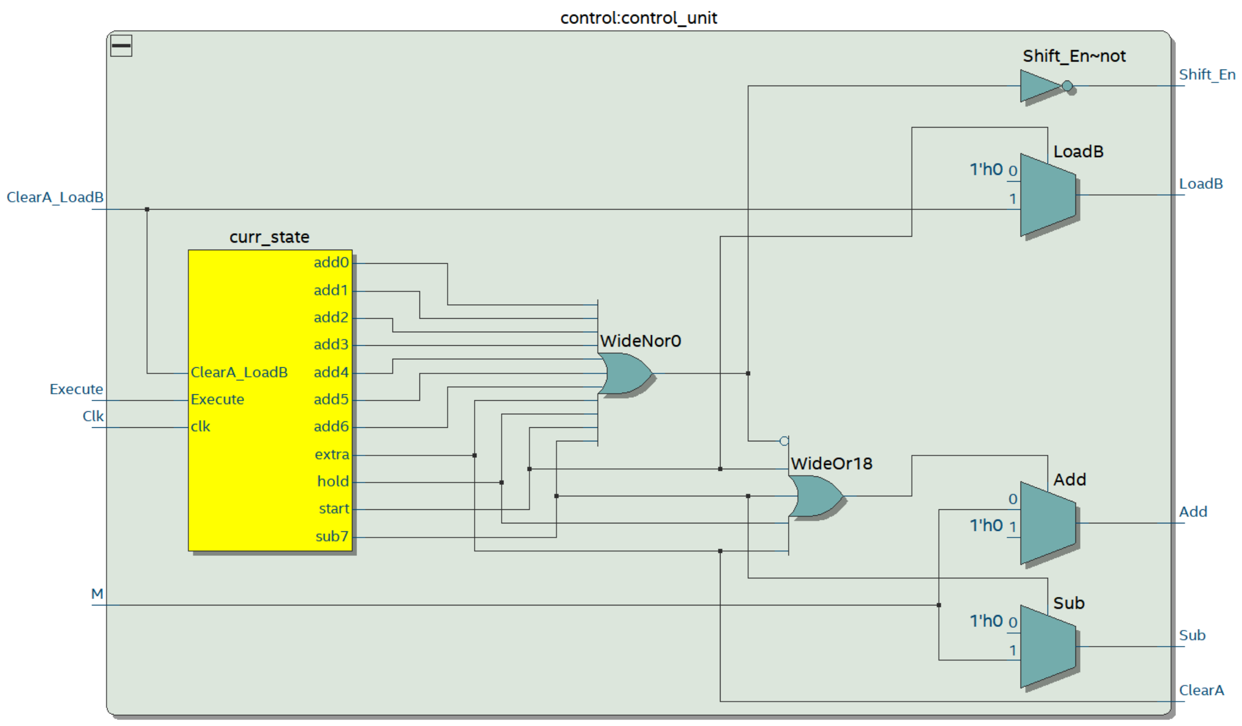

Figure 2: Control Module RTL Diagram

The `Control` module takes the button inputs `Execute` and `ClearA_LoadB` and the internal `Clk` signal along with `M`, which is the least significant bit of Register B. `M` is important for the FSM and control design because it decides whether to enter the ADD/SUB state, or immediately jump to the SHIFT state. When `Shift_En` , the registers shift right, when either `Add` or `Sub` are high, the circuit performs the corresponding operation. In our outputs, we split the `ClearA_LoadB` signals into separate signals, so that it would make debugging and designing a little easier. So when the signals are high, the circuit performs the corresponding clear/load operation.

The purpose of the module is to hold our FSM and drive the states of our multiplier based on the current state and input signals.

Reg_8.SV

Code Block 3: module reg_8

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

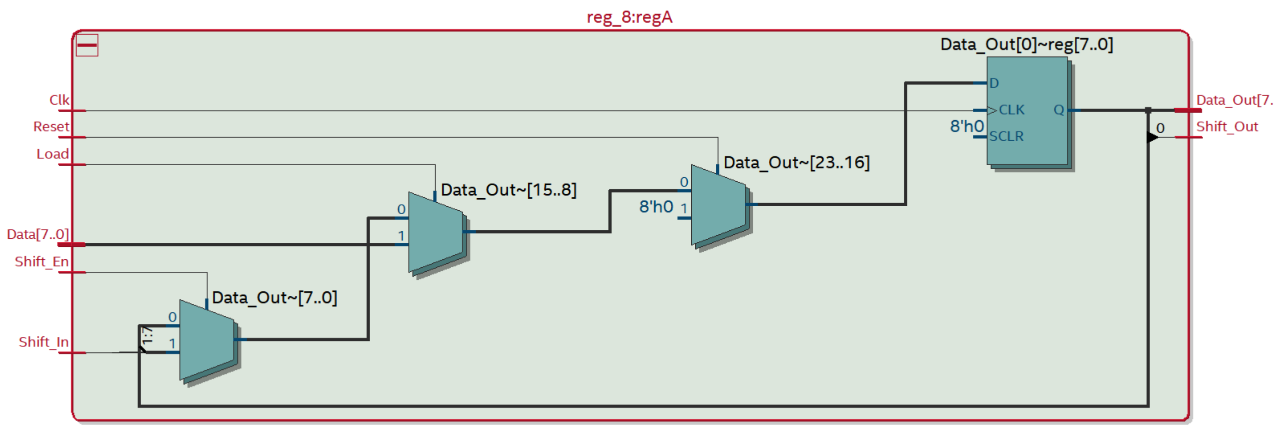

Figure 3: Reg_8 Module RTL Diagram

Figure 3 shows the RTL diagram for the Register A instantiation of the `reg_8` module. Register B also as the same diagram so it is not displayed here, but can be seen in the top level diagram in Figure 1. When `Load` is high, the 8-bit `Data` is loaded into the register. When `Shift_En` is high, the register performs a right shift and the value of `Shift_In` is shifted into the register unit as the most significant bit. When `Reset` is high the contents of the register are cleared and set to zero.

The purpose of this module is to instantiate the two 8-bit registers A and B which will hold our final result of the multiplication operation performed by our circuit.

HexDriver.SV

Code Block 4: module HexDriver

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

Assigns 4-bit inputs to corresponding hex values. The purpose of the `HexDriver` is to output the sum to the FPGA.

Adder.SV

Code Block 5: module adder, add_sub9

```
module adder (input A, B, cin, output S, cout);

module add_sub9(input logic [7:0] A, B,
                                    input logic fn,
                                    output logic [8:0] S, output logic
Xvalue);
```

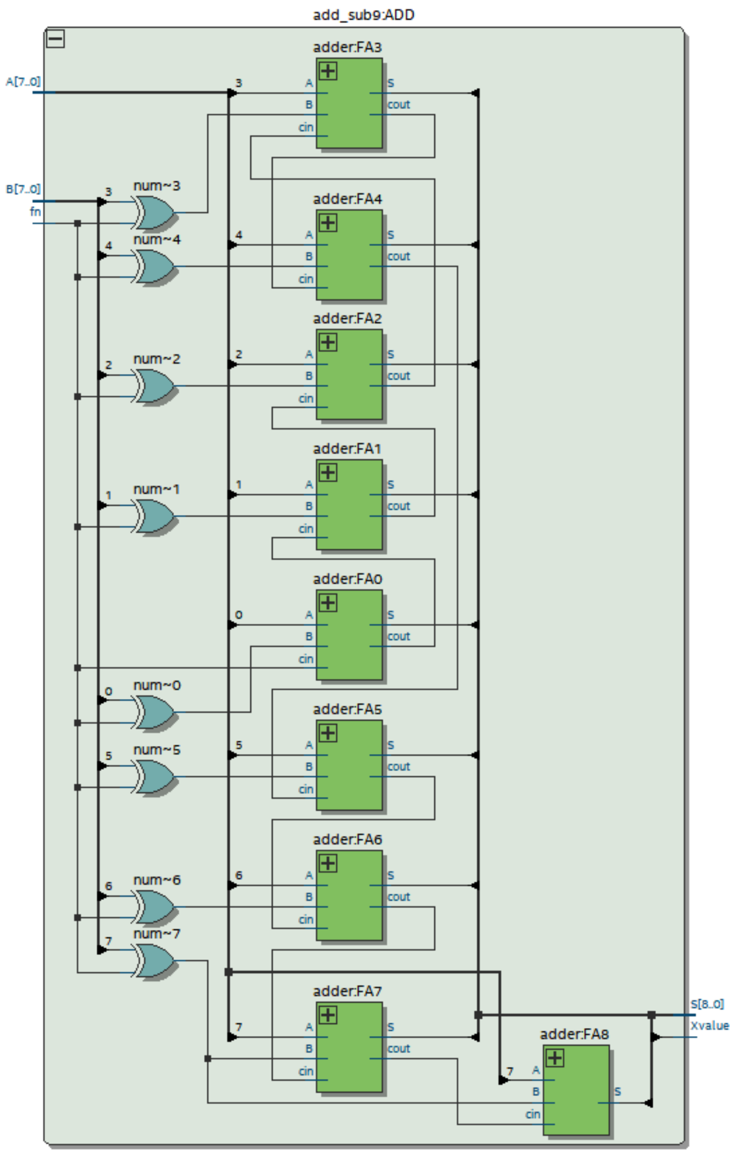

Figure 4: Adder Module RTL Diagram

`Adder.SV` contains the full adder and the performs the 9-bit addition/subtraction between `S` and `A`. The `adder` module is the full adder, and the `add_sub9` does the sign extension and the either adds or subtracts the input values. `fn` determines the operation, so when high, `add_sub9` performs subtraction by sign extending the value to 9 bits, and then instantiates 9 full adders where the value of register A is added to the value of `fn`. The `fn` bit is also the `cin` of the first `adder` instantiation, and performs subtraction since adders can subtract when the `cin` is high, as it is in this scenario.

The purpose is to perform add or subtract when the FSM is in the ADD or SUB state. The module also assigns the most significant bit of the result as the X value (`Xvalue`) so that it can be shifted later.

Synchronizers.SV

Code Block 6: module sync, sync_r0, sync_r1

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

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

//synchronizer with reset to 1 (d_ff)
module sync_r1 (
  input  logic Clk, Reset, d,
  output logic q
);
```

The purpose of the `Synchronizers` module is to bring the asynchronous signals into the FPGA. For switches and buttons it does not require a reset, but has a reset signal to 0 and 1 for d flip flops.

Testbench.SV

This is the testbench module. It creates all the local logic variables for the top level `multiplier` module and then instantiates it. The purpose of the testbench is to debug and test our circuit to ensure that it is performing the multiplication operation.

State Diagram of Control Unit

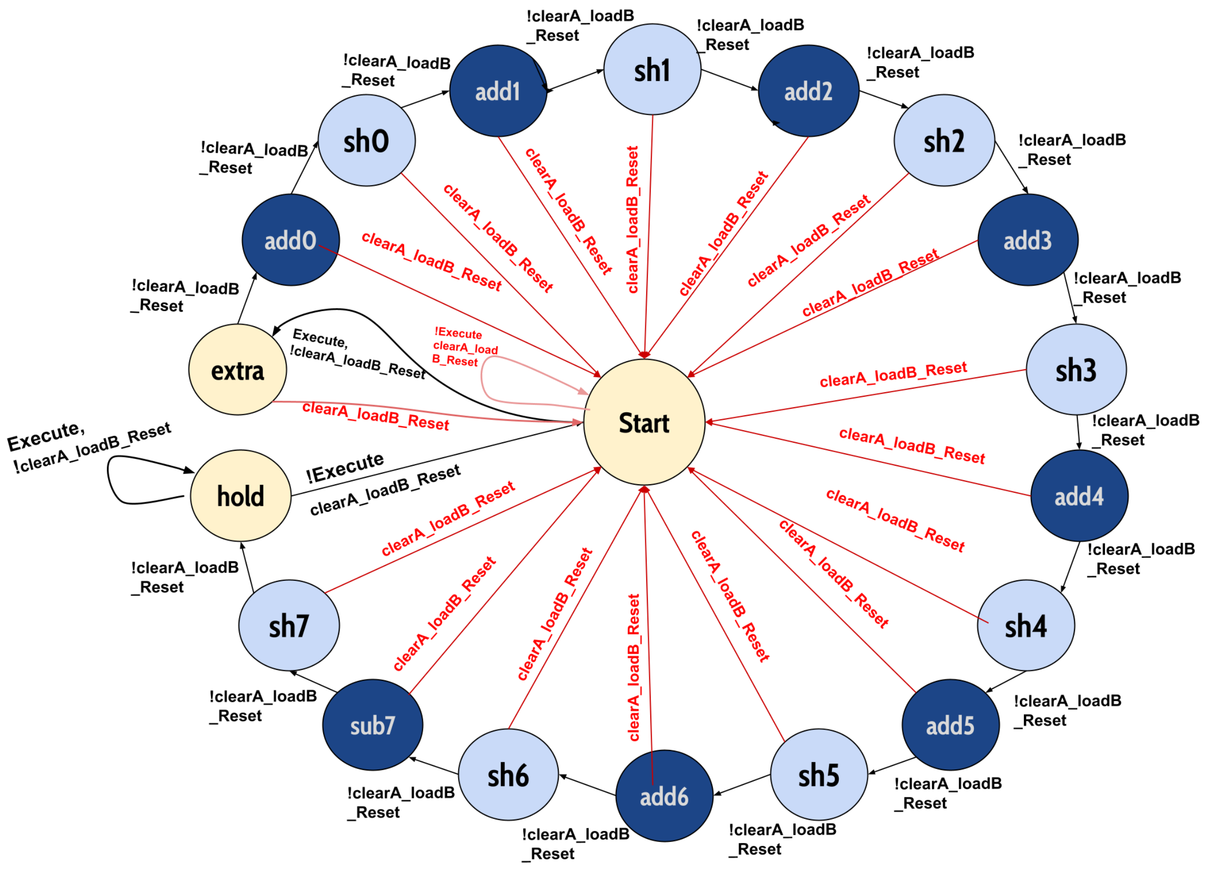

Figure 5: FSM Diagram

Figure 5 shows the FSM diagram for our control unit. It is made up of 19 states, with 7 ADD states, 1 SUBTRACT state, 8 SHIFT states, the START, EXTRA, and HOLD state. Once in the EXTRA state, the state machine iterates through the ADD/SUBTRACT/SHIFT states until it arrives at the HOLD state where the result is stored in Registers AB. The FSM then starts again when `Execute` is toggled or the registers are cleared and reset.

Pre-Lab Simulation Waveforms

Table 3: Waveform Operations

FIGURE	OPERATION	OPERATION(HEX)	EXPECTED RESULT (HEX)
6	7 ∗ 59	07 ∗ 3B	x019D
7	7 ∗ -59	07 * C5	xFE63
8	-7 ∗ 59	F9 ∗ 3B	xFE63
9	-7 ∗ -59	F9 ∗ C5	x019D

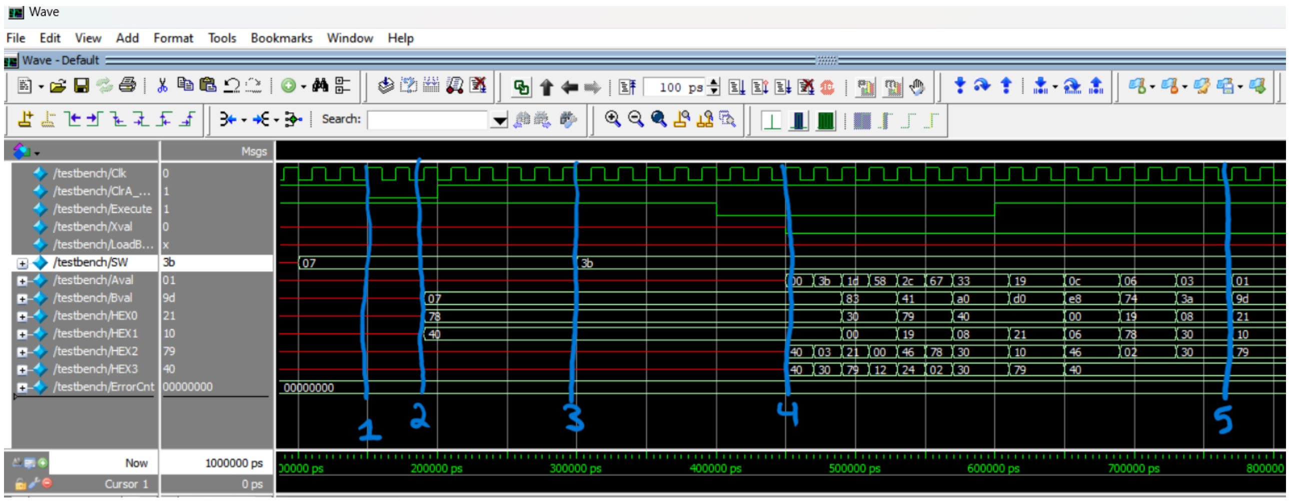

Figure 6: (+7) * (+59) Waveform

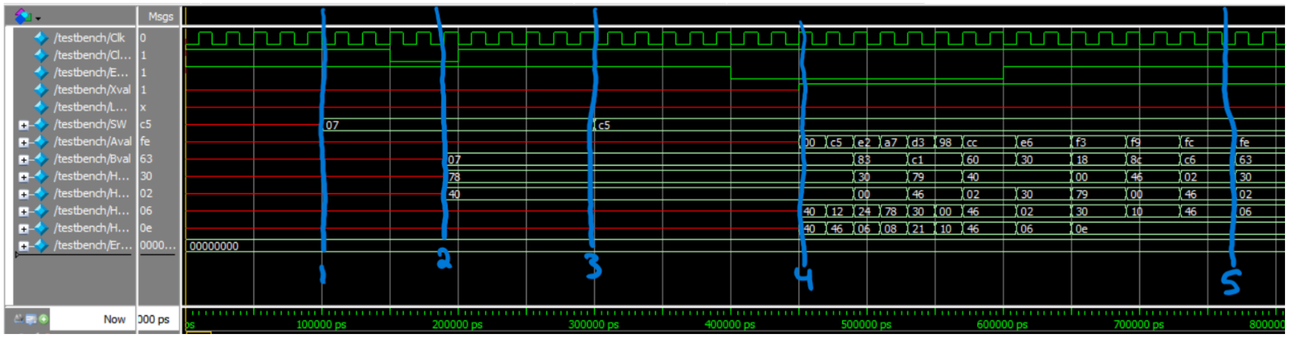

Figure 7: (+7) * (-59) Waveform

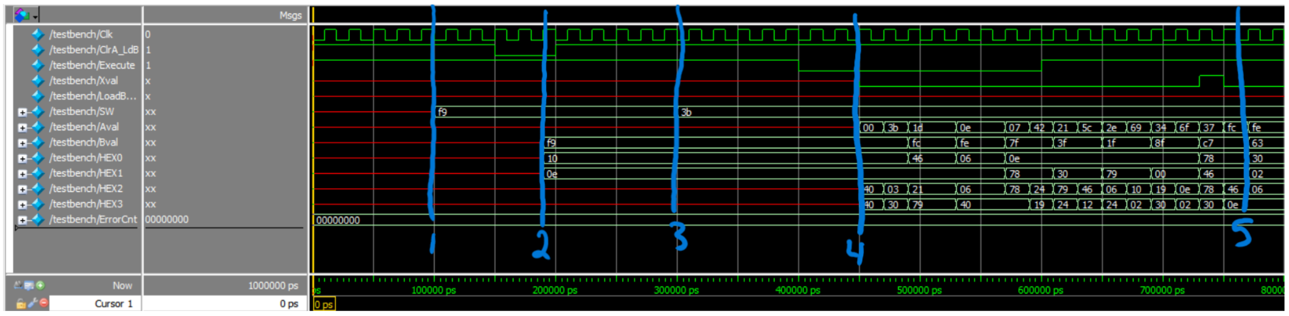

Figure 8: (-7) * (+59) Waveform

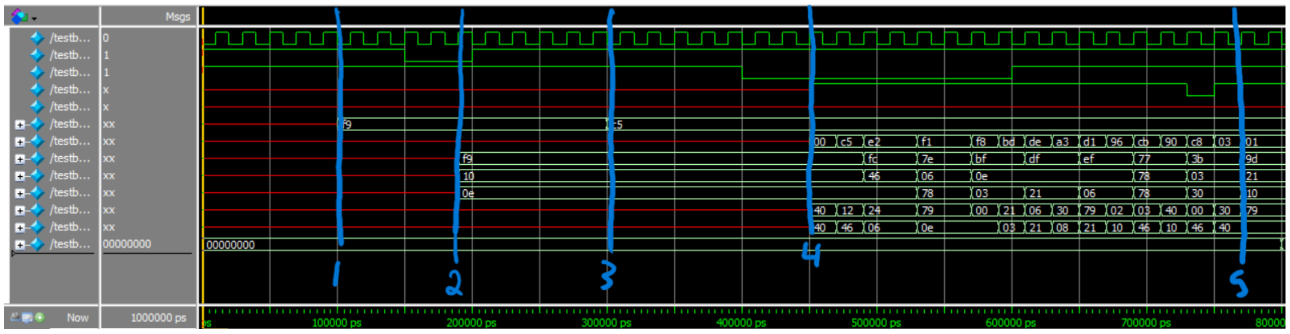

Figure 9: (-7) * (-59) Waveform

Figures 6-9 show four different operations all using different signs. Table 3 shows that values used in each waveform, and the expected result in hexadecimal. All four waveforms have the same time delay between signals. Line 1 shows the initial setting of the `SW` signal, which is the switch input. At Line 2, `ClrA_LdB` is being toggled, and we see the value of the `SW` signal load into Register B. At Line 3, the `SW` signal is set to the new value, or the multiplicand. At Line 4, the `Execute` signal has been toggled, and see see the start of the shifting and adding that performs our actual multiplication. At Line 5, we see the product stored in Registers A and B. Ultimately, AB is displayed on the FPGA and represents our final answer.

Post-Lab Questions

Table 4: Design Resources and Statistics

LUT	**93**
DSP	0
Flip-Flops	38
Memory (BRAM)	0
Frequency	58.7 MHz
Dynamic Power	1.43 mW
Static Power	89.94 mW
Total Power	103.65 mW

To optimize our design and decrease the total gate count we can possibly combine add and shift states or we can combine our two registers for A and X into one bigger register. In doing so, we can significantly decrease the number of gates implemented in the design, optimizing it. Furthermore, we can change the type of adder we utilized to increase the maximum frequency. As we noticed in Experiment 3, the ripple adder is the simplest with the least amount of gates but it is also has the slowest frequency. In this lab we chained together nine full adders to create a 9-bit ripple carry adder and to increase the frequency we can change it to either a carry lookahead adder or a carry select adder. We have the logic for implementing the three adders in SystemVerilog from experiment three and can easily implement them for our design so we may increase the frequency, further optimizing the design.

Question 1

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

When we shift in our multiplier, we want the most significant bit of A to be the same before and after we shift so the sign of A is maintained and we can continue our computation. We can do this by sign-extending A before we shift and then shifting all of our bits. In our lab, the X register has the purpose of storing this sign extension of A so that we may shift the value of X into A and keep the sign of A the same during a SHIFT state. We set the value of the X register during any of our ADD or SUB states since when we compute an addition or subtraction the most significant bit of A could change and we want it to update every time we finish a computation between S and A. We can also set the value of X when we initially start our computation and the bits of A are all 0. X is cleared when we press the `ClearA_LoadB_Reset` button since we want to clear X at the same time that we clear A.

Question 2

What would happen if you used the carry-out of an 8-bit adder instead of the output of a 9-bit adder for X?

If we were to use the carry-out bit of an 8-bit adder rather than the output of a 9-bit adder for X then we would get the wrong bit for X and shift in the wrong value, failing to preserve the sign of A throughout our computations. For example, if we were adding S = 00000001 and A = 11111110 then we would get a result of 11111111 with a carry-out value of 0. Once we set the carry-out value to X and shift then that will give us A = 011111111 which is incorrect as now is positive rather than negative and having a wrong value of X to shift in fails to maintain the sign of A. For this reason, we sign extended A and S in our lab so we may set the value of X to the ninth bit and shift it in, successfully preserving the sign of X.

Question 3

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

One of the limitations of continuous multiplication is overflow. As we multiply time and time again, we eventually run into the problem that our value cannot be represented in 16 bits and that we would need more. In our case, however, we are unable to preserve the numbers being multiplied as we need more than 16 bits to store our previous product and run into an overflow problem when we attempt to. Once overflow occurs our X and A registers get cleared to 0 displaying the limit of our multiplication and when our algorithm finally fails.

Question 4

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

A disadvantage of the our multiplication algorithm over the pencil-and-paper method would be an overflow in bits. When we have an overflow in bits in our algorithm the X and A registers get cleared to 0 and we cannot compute the multiplication; however, on the pencil-and-paper method we are able to compute as many bits as we would like even with an overflow. An advantage of the multiplication algorithm is that it recognizes if we should subtract at the final part of the computation and also that it computes the product much faster than if we were to compute by pencil-and-paper.

Conclusions

Lab Manual Reflection

Any doubts we may have had after reading over the manual were resolved by attending lectures and watching past Q/A video streams as we were able to grasp a better visual understanding of the lab and how we can start implementing it. The manual provided us with knowledge of what was required of the design in the lab and what it must accomplish whereas the lectures and Q&A videos provided us with an understanding of how we may get started and with hints on designing our program. We had to take some time to understand how we can use the X and M bits to design our state machine. We believe that the lab manual was quite direct and does not have to be changed for future semesters.

Summary

In this lab we were tasked with designing a multiplier that can multiply two 8-bit 2's complement numbers and display the output on our FPGA board's 7-segment HEX LED displays. We can do this by setting the switches to what we want to load in B and press down on our `Clear_Load_Reset` button. We then change the switches to what we want to set in A and push down the run button to display the result of $A * B$ in the LEDs. As we attempted to get this to work we ran into a few bugs that we had to fix. One of the first problems we ran into was that the `Clear_Load_Reset` signal would not properly load B or clear A. We had to resolve this in our control unit by splitting the clear and load signals to two input logic signals as there were certain states that we did not want to clear A and only load B or vice-versa. By fixing this, our multiplier now also could do continuous multiplication. Our last bug was that we were unable to display any of our input or output logic onto the DE10 FPGA LED displays but after some time we were able to assign the right values to each hex driver fixing our last bug and completing our lab.