ECE 385

Spring 2024

Experiment 4 – Multiplier

Lab 4 Report

Ziad AlDohaim (ziada2)
Mohammed Alnemer (mta8)

Table of Contents

Make sure to include units in Design Statistics, your frequency should be in MHz and power should be in mW

-0.25 for Multiplication Example: Second X should be 1 as X gets set only during an add state.

1. Introduction

This Lab we were tasked to design and implement an 8-bit multiplier using SystemVerilog, aimed at multiplying two 8-bit 2's complement numbers. This design employs an add-shift algorithm similar to the traditional pencil-and-paper method for multiplication, with special considerations for 2's complement numbers. Since the issue with the traditional pencil-and-paper method is that you need to store all partial products and at the end you have to add them all together, which is extremely inefficient. The updated process involves logical operations and control of the data path through a specifically designed control unit, which we will go over in this lab report in further detail.

How does it work?
The initial values for multiplication are set through switches, and the circuit progresses through a sequence of operations including clear, load, reset, add, and shift actions to compute the product of the two numbers. The add and shift operations are executed based on the least significant bit of the multiplier, M-bit, with the entire process managed by a finite state machine. This setup ensures that after the final step, the correct 16-bit product of the multiplication is obtained (stored in both Registers A and B). The circuit supports consecutive multiplications without the need to reset between operations, provided that the initial conditions are properly reestablished for each new multiplication. The Illustration below gives a rough idea of the design (Figure 1)

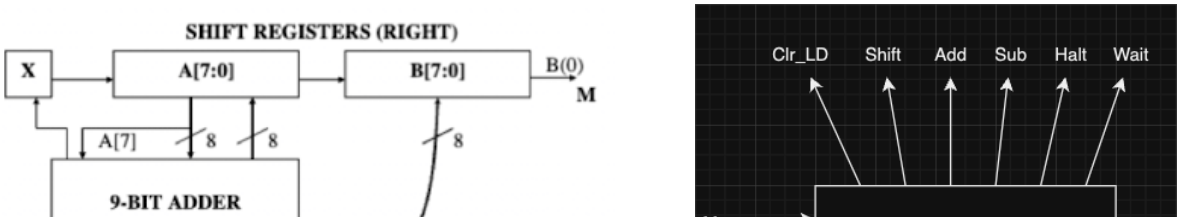


Figure 1: Block Diagram for understanding purposes.

2. Pre-lab question

Function	X	A	B	M	Comments for the next step
Clear A, LoadB, Reset	0	0000 0000	0000 011 1	1	Since M = 1, multiplicand (available from switches S) will be added to A.
ADD	0	1100 0101	0000 011 1	1	Shift XAB by one bit after ADD complete.
SHIFT	1	0110 0010	1000 001 1	1	Add S to A since M = 1.
ADD	1	1010 0111	1000 001 1	1	Shift XAB by one bit after ADD complete.
SHIFT	1	1101 0011	1100 000 1	1	Add S to A since M = 1.
ADD	1	1001 1000	1100 000 1	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1100 1100	0110 000 0	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1110 0110	0011 000 0	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1111 0011	0001 100 0	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1111 1001	1000 110 0	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1111 1100	1100 011 0	0	Do not add or sub S to A since M = 0. Shift XAB.
SHIFT	1	1111 1110	0110 001 1	1	8th shift done. Stop. 16-bit Product in AB.

Figure 2: Table of 1100 0101 * 0000 0111, as requested by pre lab questions

We get the same result when inputting it on the simulation in Vivado through our testbench.

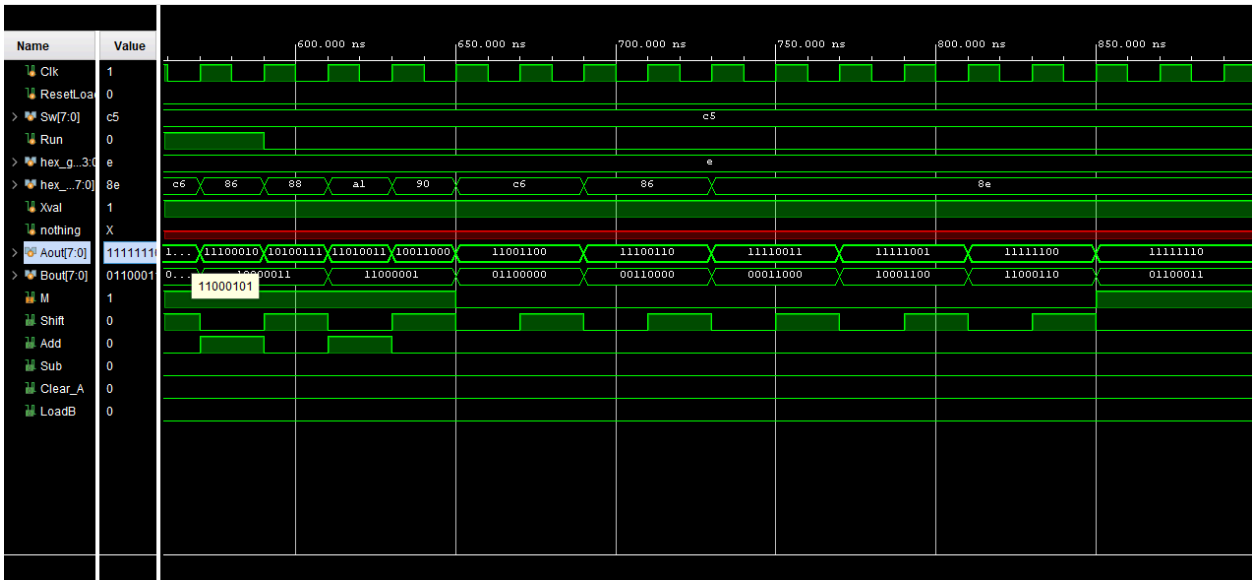


Figure 3: Vivado Simulation showing 1100 0101 * 0000 0111, verifying functionality of experiment.

3. Description and Diagrams of Multiplier Circuit

Loading Operands

The process begins with setting the multiplier value in Register B by adjusting the switches on the board and pressing the Reset_Load_Clear button. This action also clears the X and A registers.

After setting the multiplier, the switches are adjusted to represent the multiplicand. The Run button is then pressed to start the multiplication process. The Reset_Load_Clear button should be released before pressing the Run button to ensure proper operation.
Essentially, we are using the switches as a register of its own, this is much more efficient than having another register to hold the multiplicand.

Computing the Result

The multiplier employs an add-shift algorithm that takes into account 2's complement numbers. This involves a series of add and shift operations based on the least significant bit (LSB) of the multiplier, this bit is referred to as M. If M is 1, the multiplicand (from switches S) is added to A.

The contents of X, A, and B are then shifted arithmetically to the right by one bit. This process repeats for each bit of the multiplier, with the addition or subtraction (in the case of the final bit being negative) of the multiplicand based on the value of M. To visualize this look at Figure X.

The adder works by utilizing a design we implemented for a previous lab, the Carry Ripple Adder (CRA). The CRA works by chaining 9 full adders, where each full adder's Cout is connected to the next full adder's Cin creating a ripple effect, hence the name.

The arithmetic right shifts ensure that the sign of the product is correctly handled, maintaining the 2's complement representation throughout the operation.

b. Top Level Block Diagram

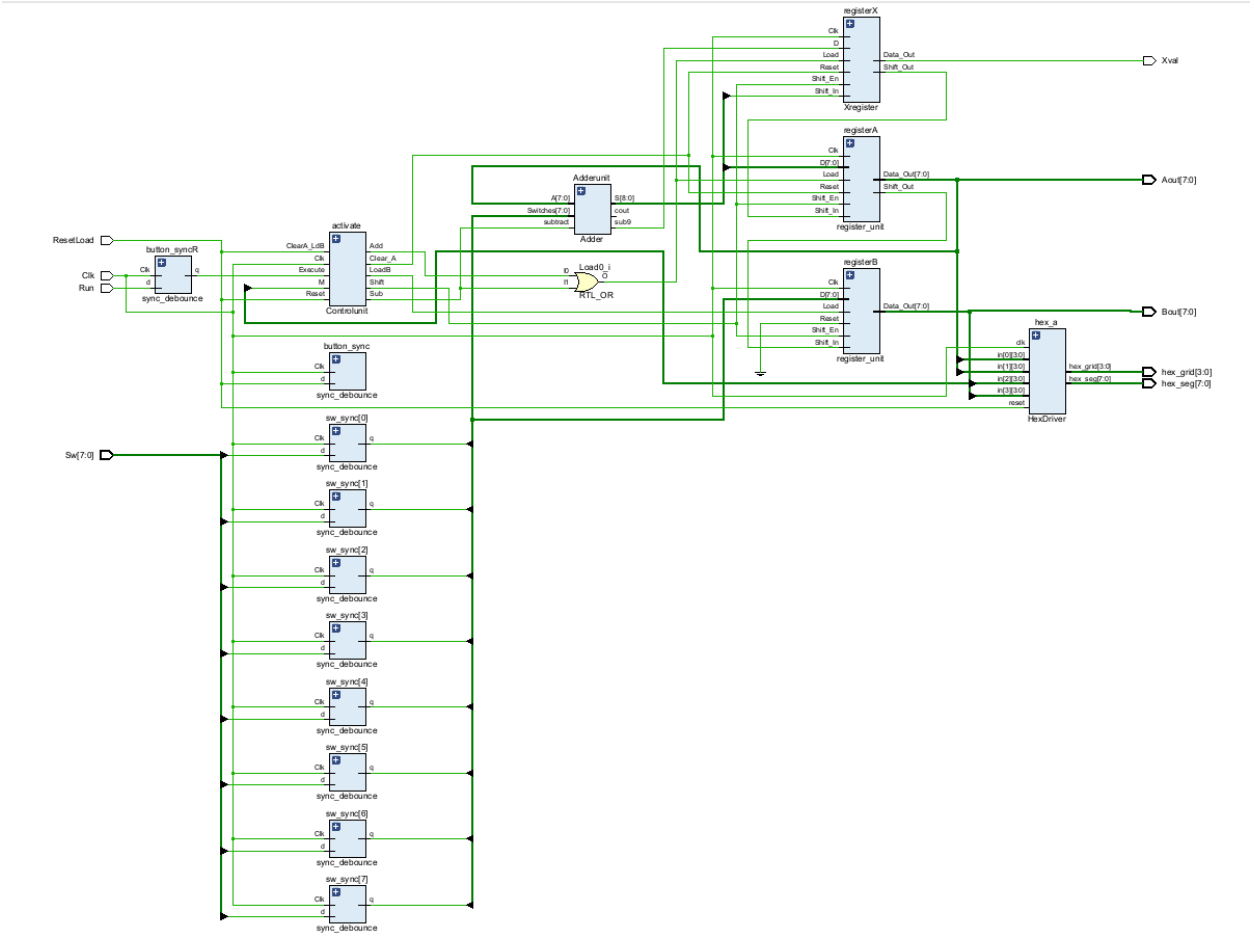


Figure 4: RTL of top-level diagram

.SV Modules

c. Written Description of SystemVerilog Modules

i. List all modules used in the format shown in the appendix of the Lab 2.2 document.

ii. Descriptions for modules which you have already written may be copy/pasted from (your own) previous lab reports.

Multiplier_toplevel.sv

Input:

Clk, ResetLoad, [7:0] Sw, Run.

Output:
[3:0] hex_grid,[7:0] hex_seg,Xval, [7:0] Aout, [7:0] Bout.

Description: This module is the central hub. Within this module, there is the instantiation of RegA, RegB, RegX, the Control Unit, Adders, HexDrivers, and Synchronizers. This file ensures they all coordinate flawlessly to execute the multiplication process.

Purpose: As the Top Level, it utilizes all the modules which will be described below and this module constructs the actual program.
Changes:
This was made from scratch, however, the experiment file gave us some indication on what we needed as inputs and outputs.

Controlunit.sv
Inputs:
M, Reset, Clk, Execute, ClearA_LdB,
Outputs:
Clear_A, LoadB, Shift, Add, Sub

Description: This module issues precise commands to the adders and registers, dictating the appropriate times for operations such as ADD, SHIFT, LOAD,and CLEAR. We also had to include the integration of two additional states, namely WAIT and HALT, to facilitate multiple multiplication cycles and to signal the end of machine operations respectively.

The Finite State Machine (FSM) of our design utilizes the Moore FSM, and has a total of 20 states: RST, ClearXA, LD_B, ADD1,ADD2,ADD3,ADD4,ADD5,ADD6,ADD7, SHIFT1,SHIFT2,SHIFT3,SHIFT4,SHIFT5,SHIFT6, SHIFT7, SHIFT8, SUB1, HALT, and WAIT. These states are controlled by the inputs Run, Reset, and M.Which can be seen below (Figure 5).

We implemented the use of M bit by using if else functions within the ADD and SUB states, when M was high we would add, if not then we wouldn't, similar if else statements for the SUB state.

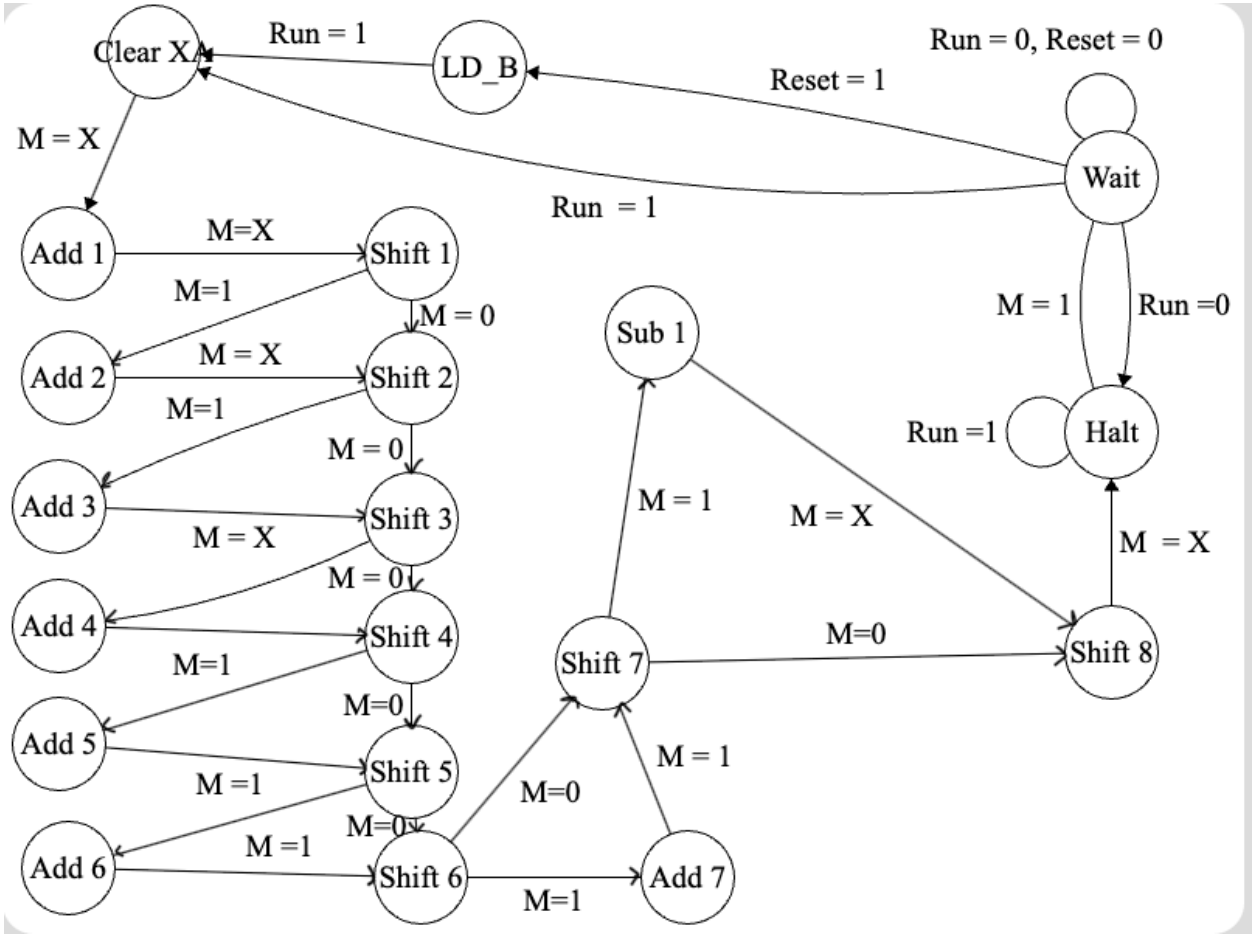


Figure 5: FSM diagram for Control Unit

Purpose: this module utilizes the finite state machine depicted in figure 5. It ensures a progression through the defined states, controlling the sequence of operations in the multiplier circuit.

Changes: Wrote from scratch, took inspiration from lecture and Lab 3 Report.

Registersabx.sv
Inputs:
Clk, Reset, Shift_In, Load, Shift_En, [7:0] D,
Output:
Shift_Out, [7:0] Data_Out.

Description: This module encapsulates two distinct entities, each proposed for storing and manipulating data. It includes an 8-bit register for the A and B inputs and a 1-bit register for the X input. Register B is assigned the value from the Switch upon activation of the Reset signal.

When the Reset signal is high, the registers within the module are initialized to zero. The Load signal enables the capture of the input data 'D' into the respective registers. When Shift_En is asserted, the contents of the registers are shifted to the right by one position, with Shift_In determining the new value of the most significant bit. The Shift_Out signal outputs the least significant bit that is shifted out during this process, and Data_Out presents the current value of the registers post-operation.

When the Reset is set to 1, the registers are cleared, setting their outputs to a default low state. A Shift_In signal triggers a rightward shift operation across the registers, which is reflected on the hex_grid display, with the most significant bit of register A being the input Shift_In.

Purpose: This module is responsible for creating the three registers (Reg A, Reg B, and RegX). See Figure 5 for RTL Design.

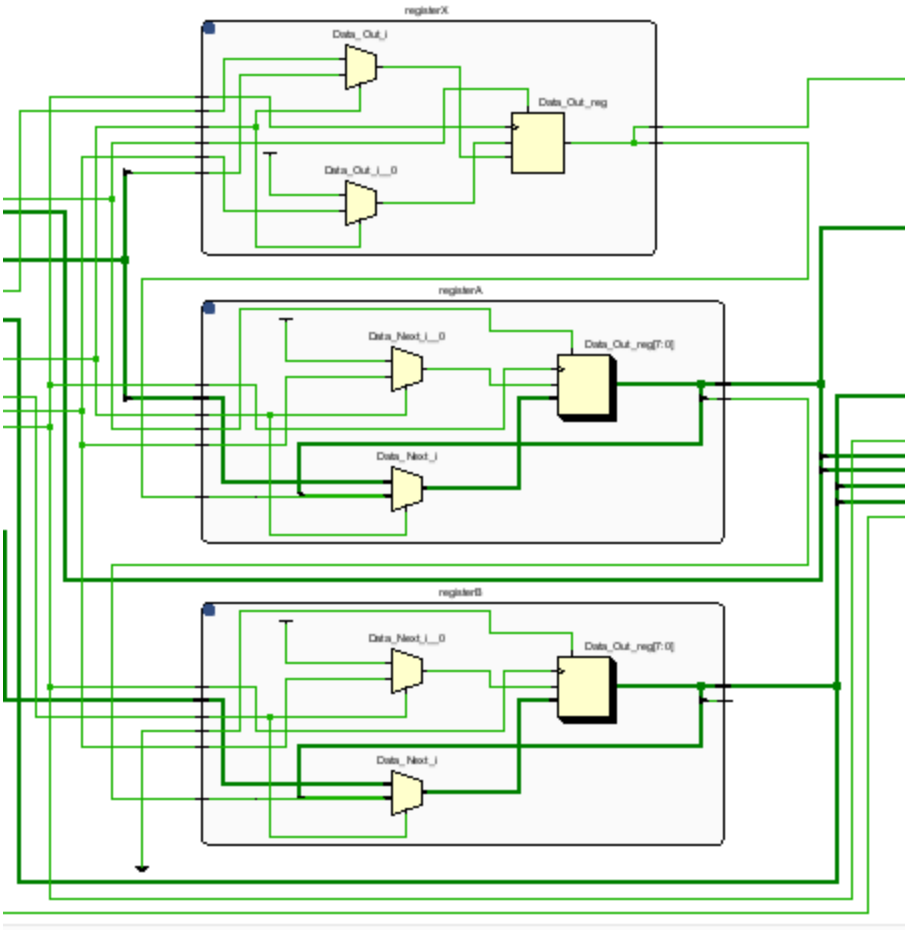


Figure 6: RTL Design of the three different Registers (A, B, and X)

Changes: Bulk of code taken from lab 3 and from lecture slides, modified one module to fit Register X by simply changing 8 to 1.

Adder.sv
Inputs:
A[7:0], B[7:0],Cin
Outputs:
S [8:0], Cout

Description: The Adder.sv module encapsulates a refined Carry Ripple Adder (CRA) mechanism, which includes nine full adders to address both addition and subtraction operations within the digital multiplier circuit. This configuration processes two 8-bit inputs, A and B, alongside a carry-in (Cin), to yield a 9-bit sum output. The essence of this design lies in its dual-mode functionality, where the first 8 bits represent the sum of the inputs, and the ninth bit, indicative of overflow or additional carry, is specifically managed to be stored in register X before being outputted.

To modify the adders to perform subtraction, we added a new signal, as illustrated in Figure 2, y = (A XOR Sub), with Sub being a control input that shows the operation mode (addition or subtraction). Furthermore, the carry-in (Cin) of the 9-bit adder is directly linked to

Sub is simply NOT ADD. This modification ensures the adder's adaptability in handling both arithmetic addition and subtraction by toggling the

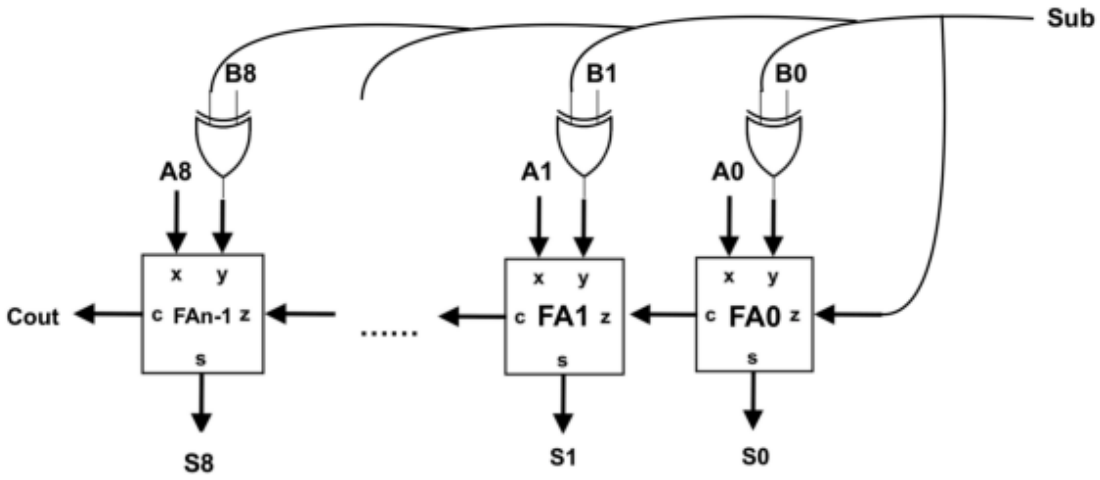


Figure 7: Block diagram for 9-bit adder with subtraction feature using the input *Sub* which switches the operation to subtraction when it is logic High.

Purpose:
The primary function of this module is to perform arithmetic addition, which is a fundamental operation within the broader context of the digital multiplier circuit. It enables the precise calculation of sum outputs and carry-out values, which are critical for the accurate execution of the multiplication algorithm.

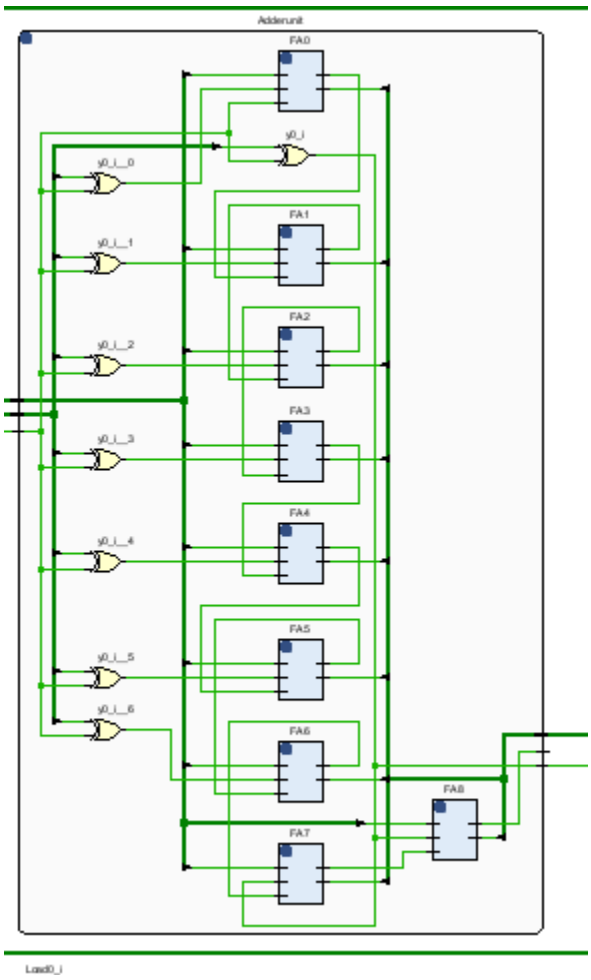


Figure 8: RTL Design for Adder

Changes:
This module was adapted from its initial design used in Lab 3, with specific enhancements made to optimize its integration and performance within the current lab's multiplier circuit. In other words, we added more full adders to change the CRA from lab 3 from an 8 bit adder to a 9 bit adder. We modified the y input of each full adder, allowing for seamless transition between addition and subtraction modes through the Sub control input by XORing A and SUB.

HexDriver.sv:
Inputs:
in[3:0], ln(), and reset.
Outputs:
hex_grid[3:0] and hex_seg[7:0]

Description:
This module takes in an 4 or 8 bit sequence and formats it to be displayed on the hexadecimal LEDs to show the values in hexadecimal.

Purpose: This functionality is particularly useful in debugging and demonstration contexts, where quick and clear visibility of system states or values is necessary.

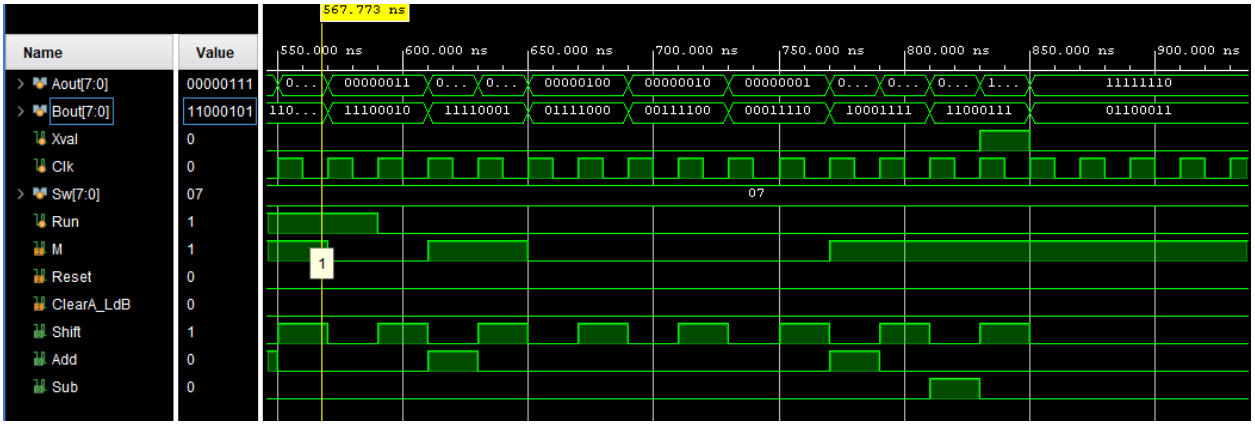
Changes:
We did not have to change anything.

Syncrhonizers.sv
Input:
Clk, d,
Output:
Q
Description:
The purpose of the Synchronizers.sv module is to provide a reliable mechanism for handling signals that cross between different clock domains within a digital system. By synchronizing these signals to the system's clock, the module helps to prevent metastability, ensuring that the system's operation remains stable and predictable.

Changes:
We did not change anything, taken from previous labs.

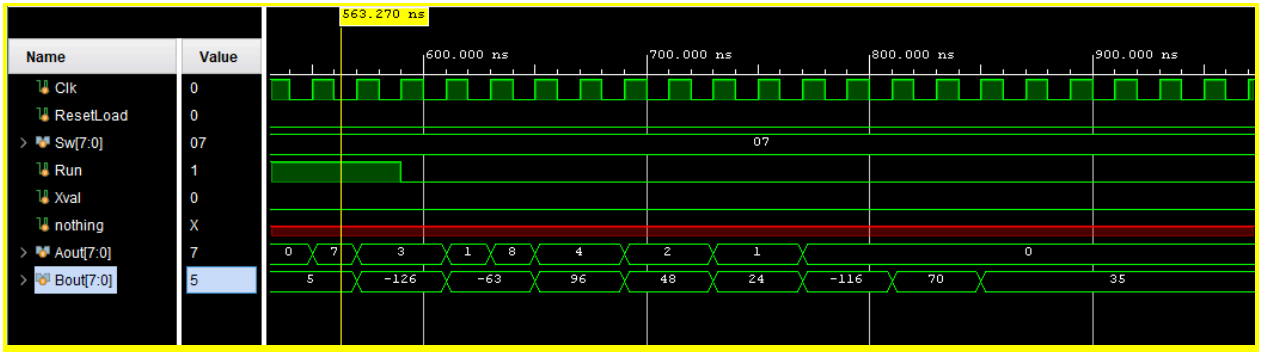
4. Annotated pre-lab simulation waveforms.

Below are a couple of test case simulations for our system. In all simulations, Aout represents the A register and Bout represents the B register.



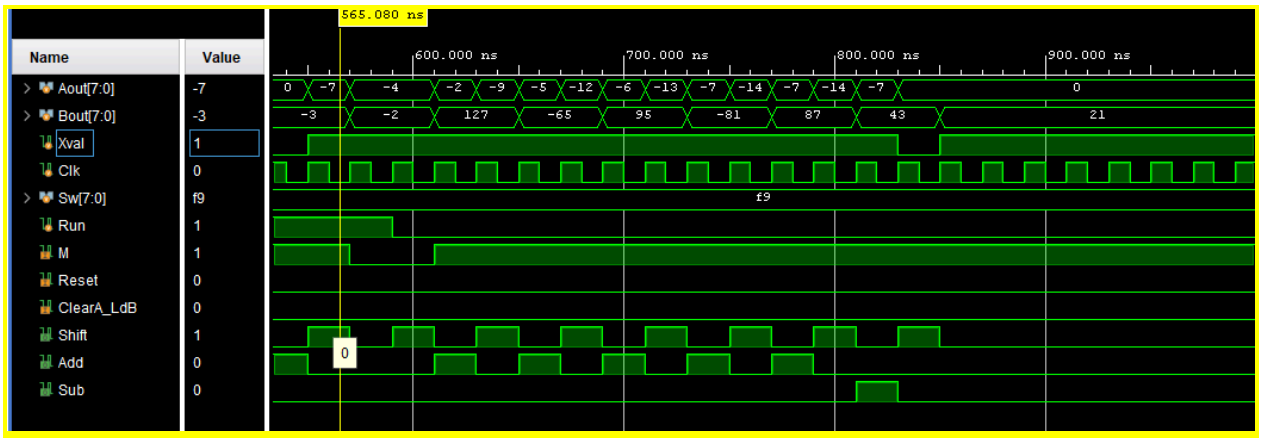
7*-59 (+*-)

Figure 9: Simulation waveform portraying the operation 7*-59 with 7 being the multiplicand loaded into register A and -59 being the multiplier loaded into register B. The results of the operation are computed and present at the end of the waveform being stored in registers A and Bas a 16-bit two's complement value.



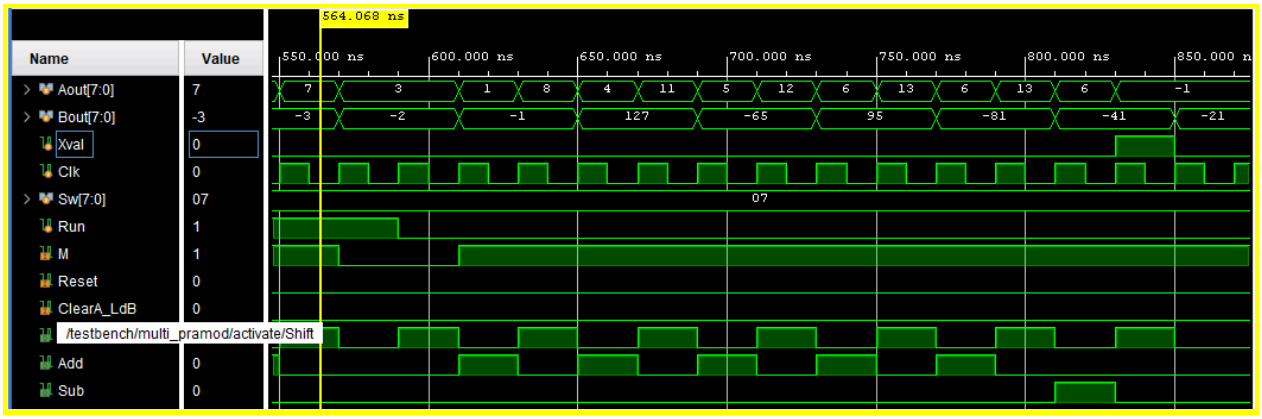
5*7 = 35 (+*+)

Figure 10: Simulation waveform portraying the operation 5*7 with 7 being the multiplicand loaded into register A and 5 being the multiplier loaded into register B. The results of the operation are computed and present at the end of the waveform being stored in registers A and Bas a 16-bit two's complement value.



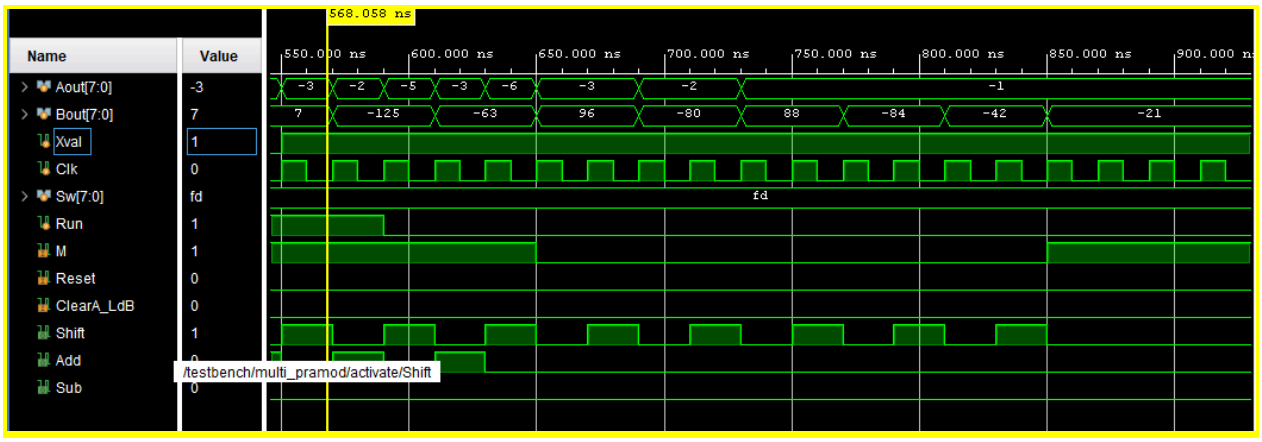
-7*-3 = 21 (-*-)

Figure 11: Simulation waveform portraying the operation -7*-3 with -7 being the multiplicand loaded into register A and -3 being the multiplier loaded into register B. The results of the operation are computed and present at the end of the waveform being stored in registers A and Bas a 16-bit two's complement value.



7*-3 = -21 (+*-)

Figure 12 : Simulation waveform portraying the operation 7*-3 with 7 being the multiplicand loaded into register A and -3 being the multiplier loaded into register B. The results of the operation are computed and present at the end of the waveform being stored in registers A and B as a 16-bit two's complement value.



-3 * 7 = -21 (-*+)

Figure 13: Simulation waveform portraying the operation -3*7 with -3 being the multiplicand loaded into register A and 7 being the multiplier loaded into register B. The result of the operation is computed and present at the end of the waveform being stored in registers A and B as a 16-bit two's complement value.

5. Answers to post-lab questions

A) Design's statistics.

LUT	80
DSP	0
Memory BRAM	0
Flip-Flop	62
Latches	0
Frequency	130.05
Static Power	0.486
Dynamic Power	56.107
Total Power	56.593

B) Post Lab Questions

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

The X register captures and holds the most significant bit (MSB) of register A. Primarily, it functions as a carry-in bit during arithmetic shift operations, which is pivotal for maintaining the sign of A during shifts. This feature is particularly crucial when performing multiplication involving negative numbers, as it ensures the correct arithmetic sign extension of A. The X register's value being aligned with A's MSB allows the system to correctly interpret and manipulate A's sign, facilitating the accurate execution of signed multiplication operations. Whenever A is reset or loaded with a new value, X must also be reset to reflect the new MSB of A.

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

If the carry-out from an 8-bit adder were used, X would not accurately reflect A's MSB in all cases. Instead, it would represent the overflow bit resulting from the addition, which might not always correspond to A's sign bit. This discrepancy would lead to errors in sign extension during shifts, potentially resulting in incorrect multiplication outcomes.

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

Given the constraint of an 8-bit two's complement system, which can represent values from -128 to 127, the main limitation of continuous multiplications is the range of values that can be handled. When you multiply numbers in this system, it can result in overflow or underflow if the result is more than 127 or less than -128 respectively, it won't fit within the 8-bit range, leading to incorrect results.

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

The algorithm is designed to be efficient in a hardware context, making use of shift and add operations that are typically fast. Compared to an implementation of the pencil-and-paper method, which might require multiple registers to hold intermediate results of each step, the implemented algorithm is optimized to minimize the use of additional registers. This is achieved through careful planning of the operation sequence and the reuse of registers for multiple purposes, thus reducing the overall amount of registers used.

6. Conclusion

In this Lab, we built and tested an 8-bit multiplier with SystemVerilog. We used an add-shift method to multiply numbers, including negative ones. Our design worked well for most numbers, showing how digital logic works in real life.

However, our design had trouble with multiplying big negative numbers together. For instance, instead of getting 3529 as a result, we got 3229. We think this happened because we didn't test our design enough after making it.

We also had a tough time with the control unit, especially with setting up the state machine. We missed adding some states because we added them one by one and didn't know we could add all states at once with a single line of code. This mistake actually helped us get better at finding and fixing errors. We used simulations to test different inputs and watched how the states and signals changed. With a lot of testing and help during office hours, we managed to solve the issue.

I also felt the information about states was not clear enough, which made things confusing. It didn't help that we only went over this in class two days before the lab, which wasn't enough time for most of us to understand it fully.