ECE 385

Spring 2023

Lab 4

An 8-Bit Multiplier in SystemVerilog

Gustavo Fonseca, Hunter Baisden
TA: Hongshuo Zhang

Introduction

The purpose of this lab was to create an 8-Bit multiplier on the FPGA. The multiplier circuit can multiply two 8-bit numbers together and output a 16-bit result. The multiplier can handle two's complement negative numbers. This means that we can multiply positive with positive, negative with positive, and negative with negative.

Pre-Lab Question

Initial Values: X = 0, A = 00000000, B = 00000111 (achieved using ClearA_LoadB signal), S = 11000101, M is the least significant bit of the multiplier (Register B)

Function	X	A	B	M	Comments for next step
Clear A, Load B, Reset	0	0000 0000	*00000111*	1	Since M = 1, multiplicand (available from switches S) will be added to A.
ADD	1	1100 0101	*00000111*	1	Shift XAB by one bit after ADD complete.
SHIFT	1	1110 0010	1 *0000011*	1	Add S to A since M = 1.
ADD	1	1010 0111	1 *0000011*	1	Shift XAB by one bit after ADD complete.
SHIFT	1	1101 0011	11 *000001*	1	Add S to A since M = 1.
ADD	1	1001 1000	11 *000001*	1	Shift XAB by one bit after ADD complete.
SHIFT	1	1100 1100	011 *00000*	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	00011 *000*	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1111 1001	100011 *00*	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	1	1111 1100	1100011 *0*	0	Do not subtract S to A since M = 0. Shift XAB.
SHIFT	1	1111 1110	01100011	1	8th shift done. Stop. 16-bit Product in AB.

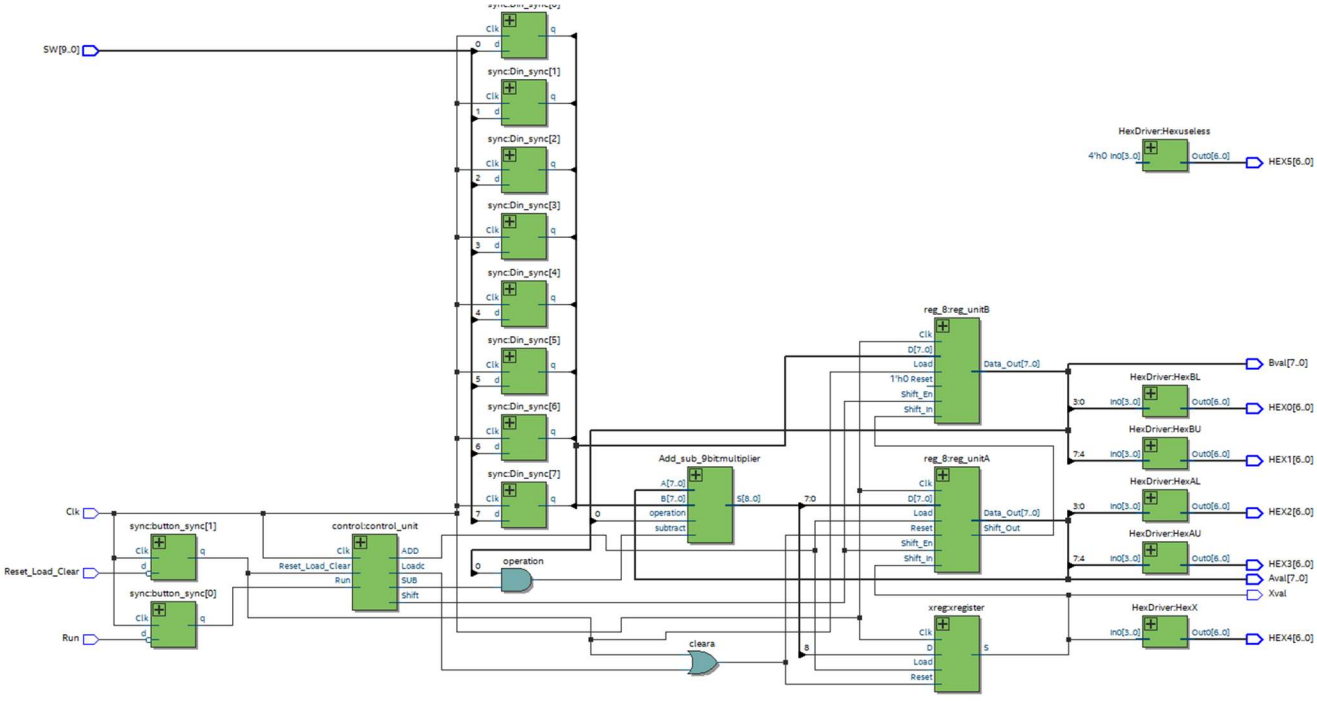
Summary of Operation

This circuit works by using 3 registers, A (8-Bit), B (8-Bit), and X (1-Bit). We will use them during the multiplication operation and to display the result. We are also using an adder that changes which values will be added together based on certain inputs. This allows us to have the control unit not to have values from the registers as input. Whenever the least significant bit (LSB) of Register B is 0, then the value in Register A will be added with zero. If the LSB of Register B is 1, the adder will then check if we are performing a subtraction, if we are, then the value within Register A will be added with the two's complement of the value within Register B. If we are performing addition, then the value within Register A will be added with the value within Register B. The adder will then sign-extend the most significant bit of the 8-Bit addition output, resulting in the adder outputting a 9-Bit value.

There are two main steps to the operation of our multiplier circuit. The first is to load Register B. To do this, set the switches to the desired value to be stored in Register B. Then press the Reset_Load_Clear button. This will clear the values of Register A and Register X, and load

the value of the switches into Register B. The next step is to run the actual operation. To do this, set the switches to the desired value to be multiplied with the value in Register B. Once the run button is pressed, the circuit will begin to go through each of our states to achieve the desired result. There are 19 total states within our design. The first state is used to wait for the run command to come from pressing the button. The second state is to send the reset signal to Register A. The next 14 signals will alternate from an ADD state and a SHIFT state. The first signal, ADD, will signal to Register A that it needs to load the new value from the adder module output. Whether the output of the adder will be the value of Register A added with zero or the value of the switches, will depend on the LSB of Register B. The value of 9th bit within the adder output will be stored within Register X. The SHIFT state will then signal to both Register A and Register B to perform a right shift. The incoming bit for Register A will be the value of Register X, and the incoming bit for Register B will be the bit shifted out of Register A. The 17th state is the SUB state. This state will again signal for the adder module's output to be loaded into Register A. The main difference with the SUB state is that it will signal to the adder module that we are subtracting. Thus, depending on the LSB of Register B, the output of the adder will either be the value of Register A with zero or the two's Complement of B. Then there will be one more SHIFT state that will be performed. The multiplication is now complete. The final state is just a wait state where no values are changed and we can see the output of the multiplication on the FPGA's HEX display, with Register B being HEX0 and HEX1 and Register A being HEX2 and HEX3.

Top Level Block Diagram



Above is a block diagram of the top-level for this design. Each module declared in the top-level is present here: the 3 register units that hold the values, the HexDrivers to display the values, and the ripple adder to add/subtract the values.

.SV Modules

Module: Processor.sv

Inputs: Clk, Reset_Load_Clear, Run, [9:0] SW

Outputs: [6:0] HEX(0-6), [7:0] Aval, [7:0] Bval, Xval

Description: This module is the top level of this circuit and is used to instantiate all the modules within it. This includes the control unit, mux, registers, the adder, and the hex drivers.

Purpose: This module takes the input signals from the FPGA, directs the signals to perform the necessary operations, and displays the outputting signals on the hex displays. This is displayed in the top-level diagram above.

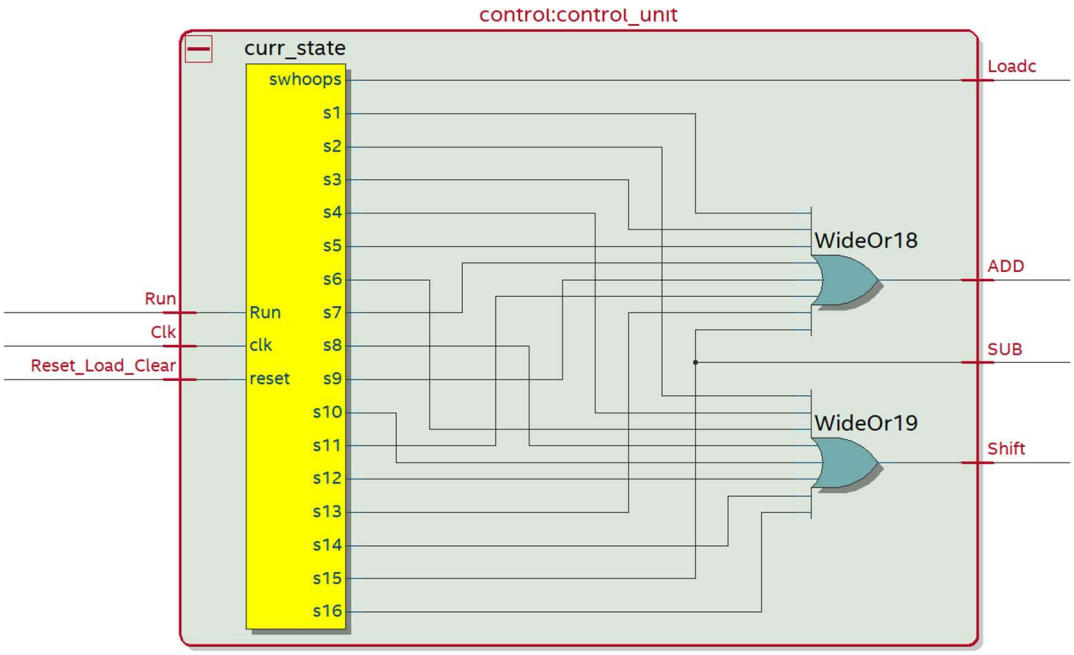
Module: control.sv

Inputs: Clk, Reset_Load_Clear, Run

Outputs: Loadc, ADD, SUB, Shift

Description: This module is the control unit for the design. By steeping through the states when called upon, it performs the correct operations in the right order. Loadc is high when we need to clear register A and load register B. ADD is high when we want to add. SUB is high when we want to subtract, and shift is high when we want to shift. The control unit follows the basic steps of ADD -> SHIFT->ADD->SHIFT until the end, where ADD and SUB are high to account for subtraction.

Purpose: This module starts running when the Run signal turns active low. It then steps through 20 states, alternating between shifting and adding to perform multiplication of 2 numbers.



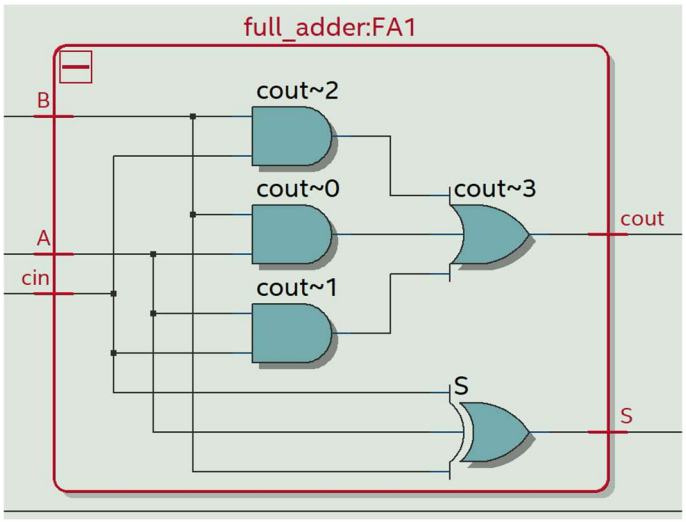
Module: full_adder.sv

Inputs: A, B, cin

Outputs: S, cout

Description: This is the baseline one-bit full adder. It is composed of 3 inputs, A, B and cin that are all XORed (*A ^ B ^ cin*) together to obtain S. The operation *(A & B) | (A & cin) | (B & cin)* is used to obtain cout.

Purpose: This module is the basis of all the adders above it. To scale the adders to 16 bits, a 4x4 hierarchy of the full adder is used.



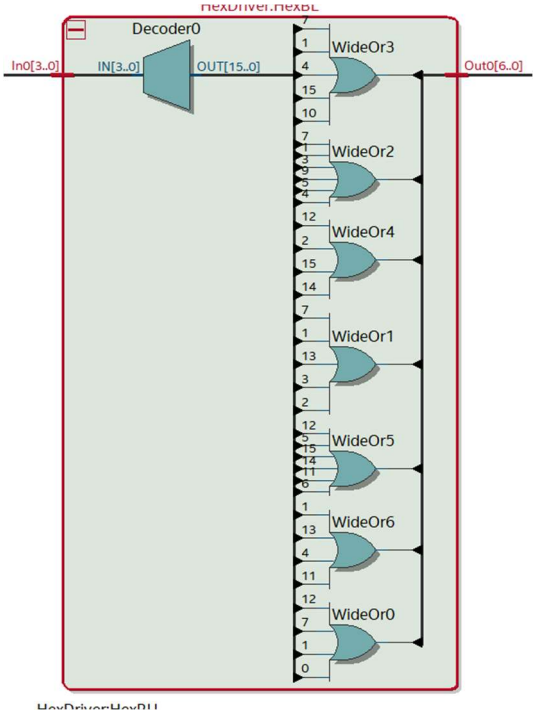
Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This module pairs every unique case of 4-bits (16 in total) into a 7-bit value used to illuminate the hex displays.

Purpose: This was used to output the values we were working with to the hex displays, both for debugging and for the demo itself.



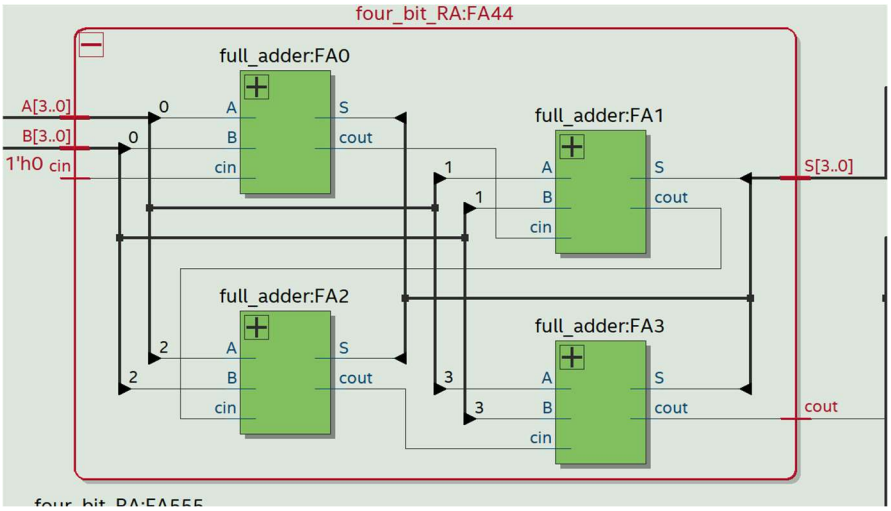
Module: four_bit_RA.sv (in ripple_adder.sv)

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

Outputs: [3:0] S, cout

Description: This module takes in two 4-bit inputs (A and B) as well as a carry in (cin) and adds them together using 4 full adder modules and outputs the 4-bit answer to S and the carry out to cout.

Purpose: To create the 4x4 hierarchy of the ripple adder/subtractor, this module is used to scale the full adder up from 1-bit to 4-bits and as a helper function to the ripple adder module.



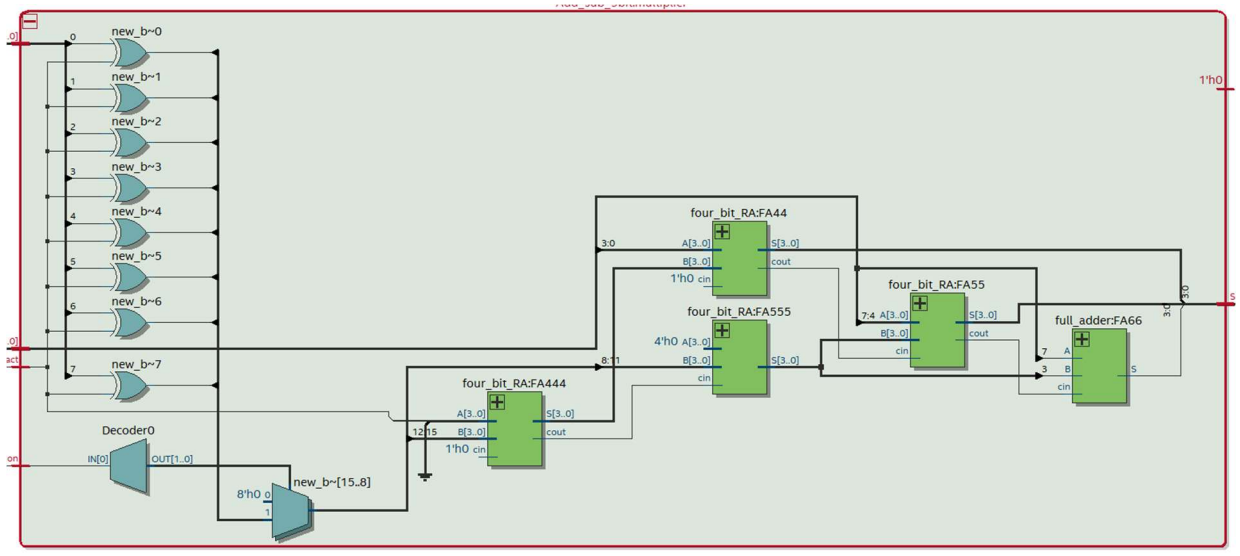
Module: ripple_adder.sv

Inputs: [7:0] A, [7:0] B, operation, subtract

Outputs: [8:0] S, Cout

Description: This is a 9-bit adder that is used to do both addition and subtraction. A and B are 8-bit inputs to be combined. Operation is a 1-bit value that tells the adder whether to add values, or simply 0. Subtract tells the adder whether to do addition or subtraction.

Purpose: This module is called after every shift to properly implement multiplication. The subtract 1-bit input allows it to also do subtraction, and the operation input allows it to add 0 or registers, depending on its value.



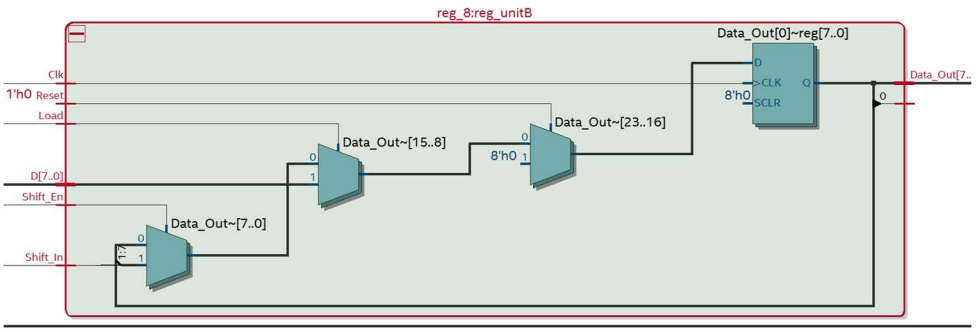
Module: reg_8.sv

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

Outputs: [7:0] Data_Out

Description: This module instantiates an 8-bit register using the 8-bit D signal to load the register when Load is high.

Purpose: This register is instantiated once in the top-level module and is used to store the values of A and B.



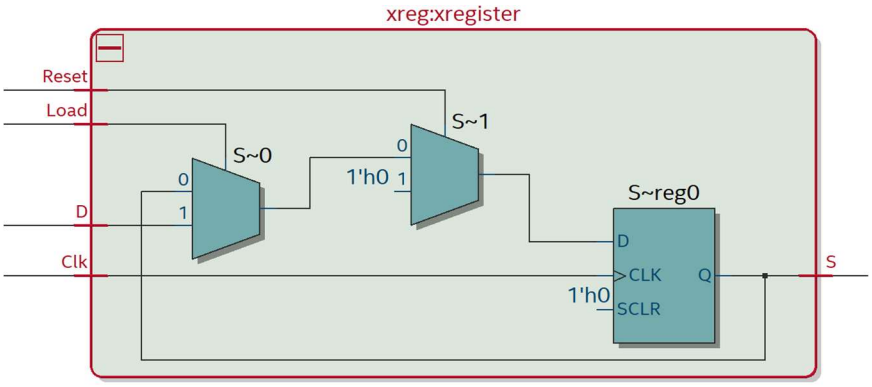
Module: xreg (in reg_4.sv)

Inputs: Clk, Reset, Load, D

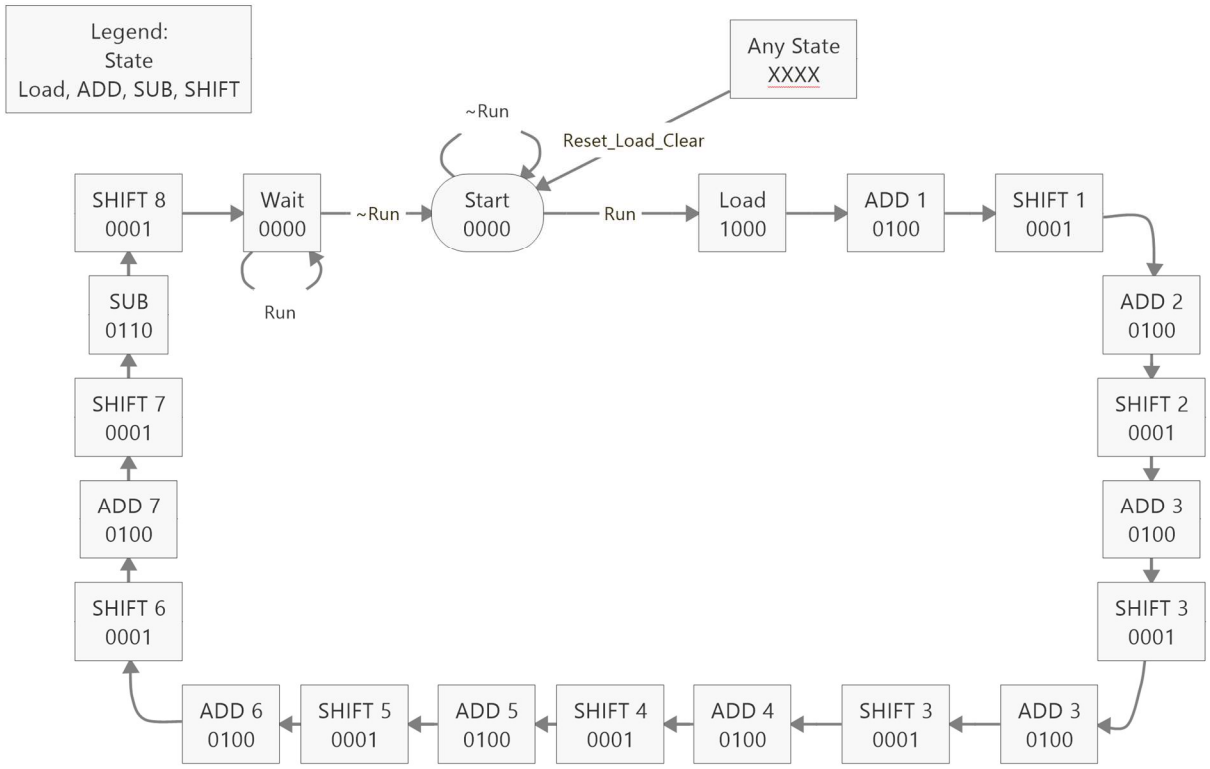
Outputs: Data_Out

Description: This module instantiates a 1-bit register using the 1-bit D signal to load the register when Load is high.

Purpose: This register is instantiated once in the top-level module and is used to store the value of X.



State Diagram for Control Unit



The State Diagram for the Control Unit only relies on two inputs: Reset_Load_Clear, and Run. If Reset_Load_Clear is pressed at any state during the state machine, the operation will stop there and go back to the Start State. Once the State Machine reaches the wait state, we do not want to start another operation immediately if Run is still high from the initial press. Thus, we will wait for the Run button to be depressed, letting the State Machine to go be to the Start State, making the user press the Run button again to perform another multiplication.

Annotated Simulation Waveforms

Post-Lab

1. **Design Statistics Table for 8 Bit Multiplier**

LUT	113
DSP	0
Memory(B-Ram)	0
Flip-Flop	21
Frequency	50 Mhz
Static Power	89.94 mW
Dynamic Power	0.00 mW
Total Power	98.73 mW

2. **Post-Lab Questions**
   a. What is the purpose of the X register? When does the X register get set/cleared?
   The purpose of the X register is to save the most significant bit (MSB) of Register A, which will be input into the MSB of Register A when the SHIFT state occurs. Without the X register, we would have to either have a 0 or 1 hardcoded into the input, which would only be correct for one possible sign. The X register is cleared whenever Reset_Load_Clear button is pressed. The X register is set whenever we are in the ADD state. After performing the 8-Bit addition, the MSB of the output is sign extended by one bit and saved into the X register.
   b. What would happen if you used the carry out of an 8-bit adder instead of output of a 9-bit adder for X?
   There would be certain cases where using the carry out of the adder would result in the incorrect value of X. In the cases where the carry out bit differed from the MSB of Register A, the value shifted into the MSB of Register A during the SHIFT state would be incorrect, leading to the resulting multiplication to be incorrect.
   c. What are the limitations of continuous multiplications? Under what circumstances will the implemented algorithm fail?
   The main limitation of continuous multiplication is the range at which continuous multiplication is possible. Even though the output of the multiplication is a 16-Bit signed number, because Register A must be cleared before another multiplication can occur, the following multiplication will only consider the 8 least significant bits of the output, which are stored in Register B. To have successful continuous multiplication, we need to make sure that the result of the previous multiplication can be expressed solely within Register B. This means that the range of values that you can perform continuous multiplication is -128 to 127. Trying to perform continuous multiplication with a number outside this range will result in the algorithm failing.
   d. What are the advantages (and disadvantages?) of the implemented multiplication algorithm over the pencil-and-paper method discussed in the introduction?

The main advantage of the implemented multiplication algorithm is that we do not need to save the multiple values to perform the multiplication. In the pencil-and-paper method, we would need to save 8 different values to be used during the operation, each needing registers to save their value. In addition, we would need to save the output within a 16-Bit Register. In the implementation we used, we only need to use 3 registers to perform the correct multiplication. Additionally, the amount of addition necessary for the pencil-and-paper method is also large. There will have to be 8 additions performed simultaneously with the 16-Bit register. Since each value to be added is 8-Bits, we would need to perform the addition on the specific portion of the 16-Bit register. Also, we would have to develop a way to shift those values the appropriate amount. With the implemented algorithm, we only need to perform addition between two 8-Bit registers.

Conclusion

1. **Functionality**
   Our design had multiple bugs within development. The first main bug was with the implementation of the adder module logic. The way we set up which values should be added became very convoluted in the end, resulting in us accidentally switching the logic for whether addition should be performed at all or if the addition was between Register B or the two's complement of Register B. Another main bug was with our implementation of two's complement within the adder module. This bug occurred at specific values resulting in the output being one value off the correct value. We realized that this bug was caused by us not saving the "plus one" value performed in two's complement. We only added this value as the carry-in bit into the adder. Thus, in certain cases, this value made a difference in the final output. We fixed this by adding one to the value of the negated value of Register B before adding it to the value of Register A, also remembering to change the carry-in bit of this addition to zero. Once we fixed these two issues, our design was successful and capable of full functionality as described in the requirements.
2. **Improvements**
   Overall, this lab was presented very well. One thing that we found kind of confusing is the incomplete block diagram. While we understand that it is our job to finish, in the current state it was confusing that the control unit was not connected to the addition aspect at all. If it was explained more clearly that we had to connect these together, then it would be easier to implement the final design. In contrast, we found the multiplication example very helpful for understanding the logic of the multiplication method as it had never been introduced to us prior to this lab.