ECE 385 Lab Report 4

Experiment #4: An 8-Bit Multiplier in SystemVerilog

Spring 2022

Trusha Vernekar & Soham Manjrekar

Dohun Jeong (DJ)

tnv2 & sohammm2

Introduction

In this lab, we had to implement a Multiplier unit for two 8-bit 2's complement inputs. Each of the 8-bit inputs was placed in either register A or register B depending on if the input was to be treated as a multiplier or a multiplicand. This design does the simple task of multiplying two 8-bit values with each other, but it does it by a series of additions, subtractions, and bit shiftings of the multiplicand and the multiplier. Consecutive multiplication without pressing reset could also be computed using our design. The final result, instead of stored in two separate 8-bit registers, is stored in a 16-bit register.

Written description and diagrams of multiplier circuit

As mentioned earlier, our multiplier design computes multiplications between two 8-bit 2's complement values. To compute this, we utilized the add-shift multiplication algorithm described in the lab manual. This algorithm computed the multiplication between the value in register B and the switches. Here, register B will be called the multiplier and the switches will be called the multiplicand. The least significant bit of the multiplier called M, and the most significant bit of the 9-bit sign-extended result of the multiplicand, called X, is stored and utilized to check whether the bits need to be added, subtracted, or just shifted to the right. The bit X is stored along with our final result. The final result will be stored in a 17-bit register and will be referred to as XAB. This XAB register consists of bit X at index 16, the register B from the index [7:0], and register A from the index [15:8].

Initially, the value that we want to place in register B as the multiplier is set using the switches. To place this value in register B, the Clear A Load B button must be pressed. Once this is pressed, register A will be cleared and register B will hold the value of the desired multiplier. After this, the switches must be changed to the desired value of the multiplicand. After this, to compute the final result, the Run button must be pressed.

The operations that occur once Run is pressed are as follows. As mentioned earlier, the M bit and the X bit are stored and utilized. To decide if the next operation will be an ADD or a RIGHT SHIFT, the M bit is checked (a least significant bit of register B). If M is 0, then the whole 16-bit register AB must be shifted to the right by 1 in the next operation. If M is 1, then the value of the switches must be added to register A (index [15:8] in our register AB) in the next

operation. The whole 16-bit register AB must then be shifted to the right by 1 in the next operation. The M bit is set every time a shift operation occurs.

A shift operation occurs a total of 8 times, no matter what the values are. After the 7th shift, the next operation is determined on values of both X and M. Since it is after the 7th shift, the value of M signifies the sign of the register B (multiplier). If both M and X are 1, then the value of the switches must be subtracted from the value of register A and then XAB must be shifted to the right for the last time. If this is not the case, then the XAB must only be shifted to the right. As for the bit X which is stored in the most significant bit location in XAB, it is set every time there is an addition or a subtraction operation that occurs.

Our lab required us to have a design that would compute consecutive multiplications. To compute this, we had to clear register A before the run button is pressed once again. Finally, the answer to the multiplication must be stored in register XAB. The LEDs will display the value of AB in the XAB register.

Top-Level Block Diagram

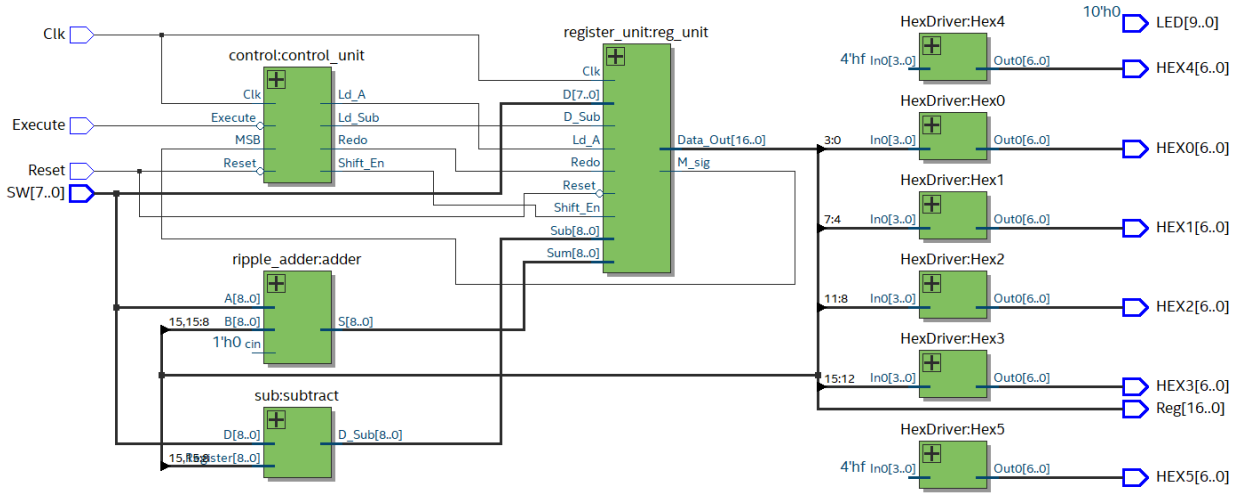

Figure 1: Block Diagram of the Multiplier

State Diagram for Control Unit

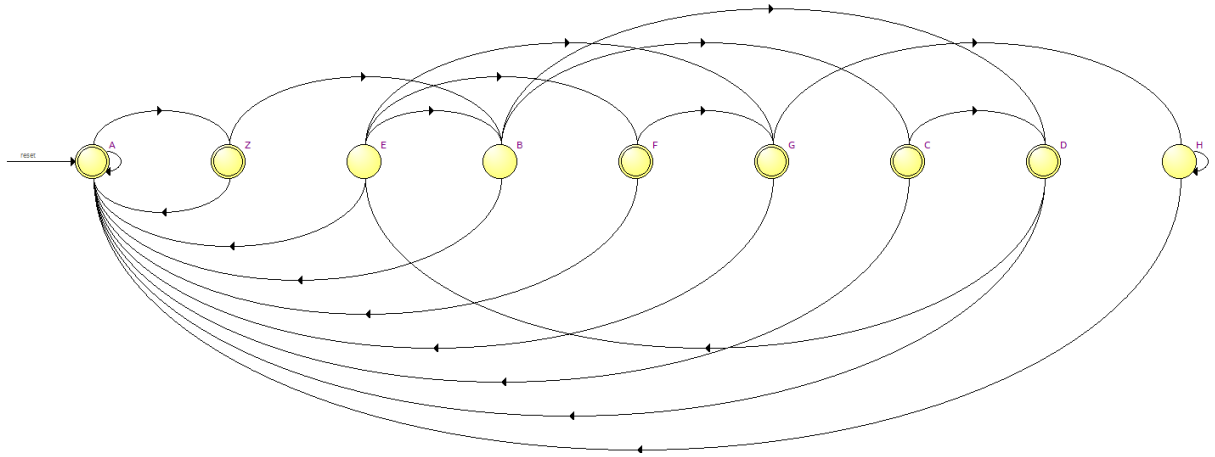

Figure 2: State Machine diagram for the control unit of the multiplier

- A: Reset State, this is the state between executions of the multiplier and where it is returned to once the Reset_Load_Clr button is pressed
- Z: Clear Register A (necessary for continuous multiplication)
- B: Checks M value if the value is 1 sent to state C if 0 sent to state D
- C: Add state, moves to state D
- D: Shift State moves to state E
- E: Counter Check state, if 7 shifts have occurred then move on to the last subtract and shift state, otherwise return to step B
- F: Subtract State moves to state G
- G: Shift State moves to state H
- H: Checks if Execute is not triggered, and returns to state A

Pre-Lab question

a. 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

Function	X	A	B	M
Clear A, LoadB, Reset	0	00000000	00000111	1
ADD1	1	11000101	00000111	1
SHIFT1	1	11100010	1 0000011	1
ADD2	1	10100111	1 0000011	1
SHIFT2	1	11010011	11 000001	1
ADD3	1	10011000	11 000001	1
SHIFT3	1	11001100	011 00000	0
SHIFT4	1	11100110	0011 0000	0
SHIFT5	1	11110011	00011 000	0
SHIFT6	1	11111001	100011 00	0
SHIFT7	1	11111100	1100011 0	0
SHIFT8	1	11111110	01100011	0

Table 1: Multiplication example of 11000101 * 00000111

Annotated pre-lab simulation waveforms.

a. Must show 4 operations where operands have signs (+*+), (+*-), (-*+) and (-*-)

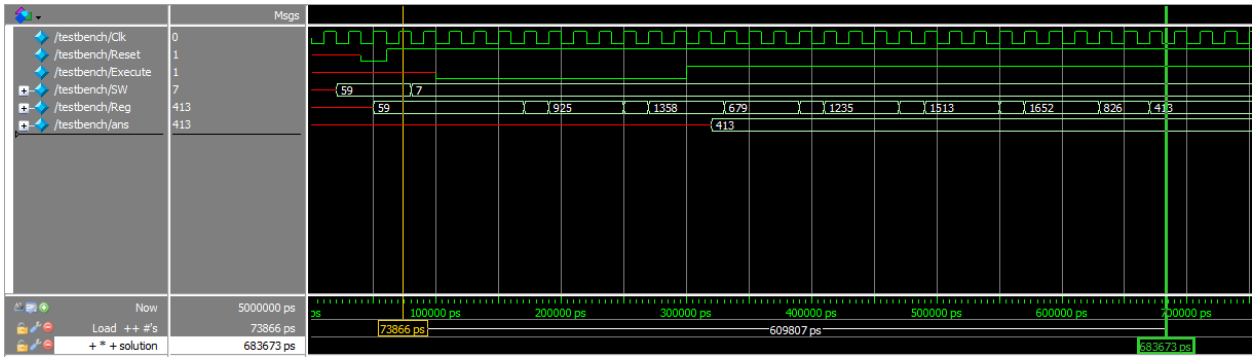

Figure 3: Simulation for (+*+)

- At the yellow cursor, positive +59 is loaded into register B, then the multiplication process of adding and shifting begins, with +7 as the multiplier.
- After all 8 shifts, we see that the hardcoded answer using arithmetic operators stored in the ans register matches our calculated answer of +413.

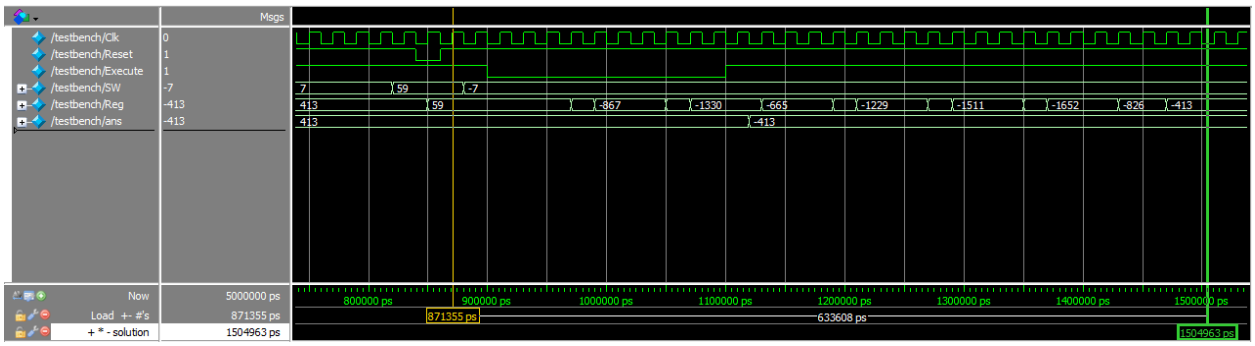

Figure 4: Simulation for (-*+)

- At the yellow cursor, positive +59 is loaded into register B, then the multiplication process of adding and shifting begins, with -7 as the multiplier.
- After all 8 shifts, we see that the hardcoded answer using arithmetic operators stored in the ans register matches our calculated answer of -413.

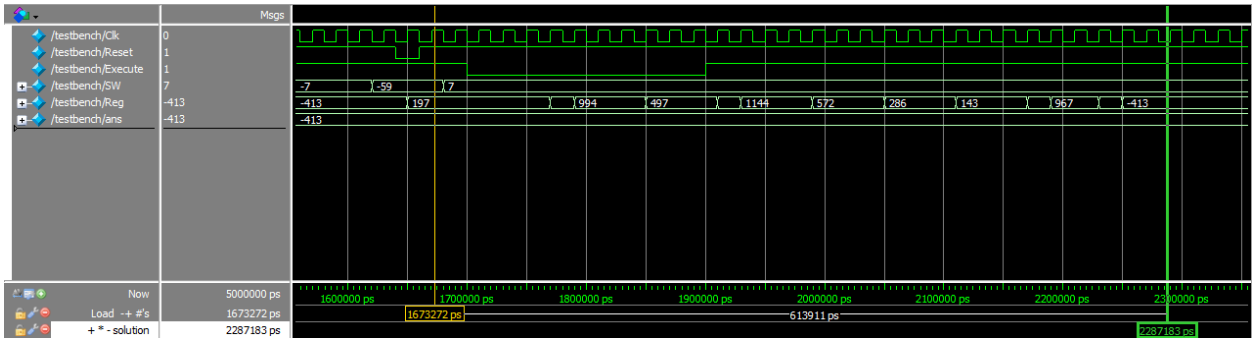

Figure 5: Simulation for (+*-)

- At the yellow cursor, positive -59 is loaded into register B, then the multiplication process of adding and shifting begins, with +7 as the multiplier.
- After all 8 shifts, we see that the hardcoded answer using arithmetic operators stored in the ans register matches our calculated answer of -413.

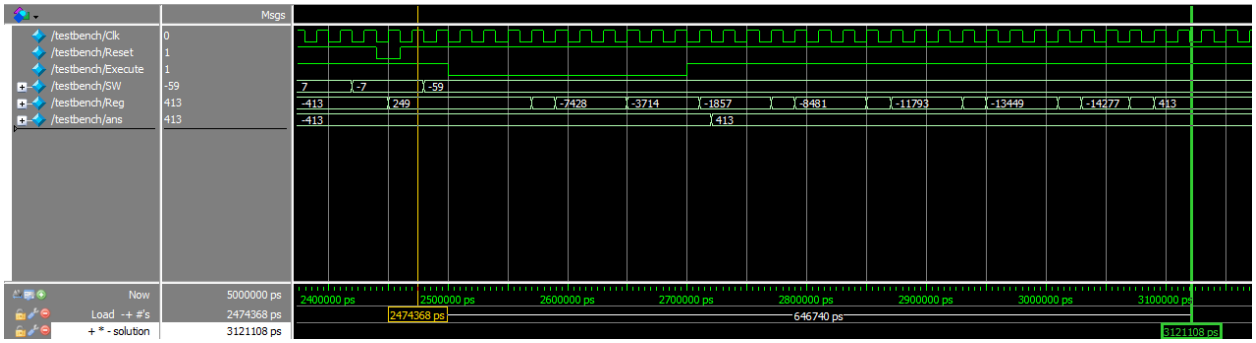

Figure 6: Simulation for (-*-)

- At the yellow cursor, positive -59 is loaded into register B, then the multiplication process of adding and shifting begins, with -7 as the multiplier.
- After all 8 shifts, we see that the hardcoded answer using arithmetic operators stored in the ans register matches our calculated answer of +413.

Post-Lab Questions

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

LUT	131
DSP	0
Memory (BRAM)	0
Flip-Flop	37
Frequency	240.04 MHz
Static Power	89.94 mW
Dynamic Power	0.00 mW
Total Power	98.77 mW

Come up with a few ideas on how you might optimize your design to decrease the total gate count and/or to increase maximum frequency by changing your code for the design.

Answer:

One way we optimized in the first place was by having an optimized fsm, instead of listing out each and every possible state we utilized a counter. Another potential optimization that we implemented was using fewer registers, we combined our A and B registers into one large 17-bit wide register.

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

Answer: The purpose of the X register is to hold the most significant bit of the 9-bit sign-extended value of register A (the multiplicand). This bit signifies if the number in register A is positive or negative. As for a 2's complement value, the most significant bit signifies the sign of the value. This bit is saved in register X. While computing the multiplication operation, this register is checked to determine if the next operation is going to be addition, subtraction, or just a bit shifting. This register is set every time there is an addition or a subtraction operation. It is clear at the start of the computation.

3) 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?

Answer: If we set register X as the carry-out of an 8-bit adder when adding two numbers with the most significant bit is 1, the adder will output a 0 as the most significant bit. According to our algorithm, the X bit holds the sign of register A. In the case mentioned above, X would be 1 instead of 0 (0 suggesting that the number is positive). Due to cases like these, we would essentially lose the sign of register A. Hence, we use the 9th bit of the sign-extended value of register A as bit X instead.

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

Answer: The point after which continuous multiplications would not work would be when the result of any of the operations exceeds 16 bits. The maximum value that can be represented using 16 bits is 32767. So, if the result of any of the prior multiplications exceeds this value, our algorithm would fail for any further multiplications.

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

Answer: The main advantage of using the implemented multiplication algorithm as compared to the pencil-and-paper method is that it is faster because it skips the ADD state when it is not required to be executed. It simply shifts the bits instead of adding. For example, when bits 0 and 0 need to be added, instead of going to the ADD state, it goes to the SHIFT state instead. A disadvantage of this method is that it has a complicated state machine which is not intuitive to implement.

Conclusion

Our design works for many cases, however, when tested against edge cases like -116 and 116, our multiplier runs into a problem of the top 8 bits being incorrect. We found that our register shifted the correct number of times, however, our calculation of the X register broke when it reached the upper 8 bits. To fix it we would have to reimplement how we calculated X. Currently, it is calculated as the sum of the A register and the switches, but it is not cleared out. We think that is where our implementation breaks.

As mentioned above, our design worked for the most part. It would fail to compute multiple multiplications after a certain number of computations. A few things in the lab manual were difficult to understand at first such as the Block Diagram for our circuit. Since we have never learned about a Multiplier design like the one we implemented before, it was difficult for us to understand what computations were happening with the multiplier and the multiplicand. It took us the most time to understand the significance of register X in the entire design since it was not mentioned in the lab manual.

This lab was a great introduction to this type of multiplier because we had never learned about this design in any of our previous courses. We learned how a series of additions, bit shiftings, and subtractions lead to a multiplication operation. We learned the significance of the X and M bit in our circuit, which was to determine if the next operation must be an addition, subtraction, or a shift. Overall, our goal to create an 8-bit 2's complement multiplier was completed.

Appendix A

Module: hexdriver.sv

Inputs: [3:0] In0,

Outputs: [6:0] Out0

Description: Takes a nibble or 4 bits and converts it to a 7 bit equivalent that is displayable on the HEX# output LEDs of the FPGA.

Purpose: To allow the seven segment displays to display what the input and output of our code are.

Module: processor.sv

Inputs: Clk, Reset, Execute, [7:0] SW,

Outputs: [16:0] Reg, [9:0] LED, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5,

Description: This module calls all of our other modules and creates variables to be passed between modules.

Purpose: This module, depending on the comments, can prime the software for either SignalTap or ModelSim, and instantiates the other modules. Without it, our FPGA or testbench would not do anything.

Module: 8_bit_register_unit.sv

Inputs: Clk, Reset, Ld_A, Ld_B, D_Sub, Shift_En, Redo, [7:0] D, [8:0] Sum, [8:0] Sub,

Outputs: [16:0] Data_out, M_Sig,

Description: This module instantiates our 16 bit register of XAB, defining if the module should add, subtract, shift or reset.

Purpose: This sets up our register, and makes sure that they are receiving all pertinent signals, so that they can act correctly.

Module: control_unit.sv

Inputs: Clk, Reset, Execute, MSB,

Outputs: Shift_En, Ld_A, Ld_Sub, Redo

Description: State Machine of the multiplier ensures that the switches are added to register A, when M is 1 and shifts after each addition or when M is 0, it also makes sure the register shifts 7 times before considering the subtraction operation, the state machine also clears out register A before any subsequent multiplications without the use of the Reset button to support continuous multiplication.
Purpose: To make sure that the multiplier conducts 1 multiplication per button press and that it follows the correct steps of adding and shifting given the M value and X value

Module: adder.sv
Inputs: [8:0] A, [8:0] B, cin
Outputs: [8:0] S, cout
Description: Ripple Carry adder module, simply 9 full adders where the carry is passed through to each subsequent adder.
Purpose: To perform a 9-bit addition for the A register and the switches both of which are sign-extended to calculate X.

Module: sub.sv
Inputs: [8:0] D, [8:0] Register,
Outputs: [8:0] D_Sub
Description: Subtraction module, gets the two's complement of the sign-extended switches then adds using the adder
Purpose: To perform the optional subtraction operation in case the msb of register B is 1.

Module: reg_8.sv
Inputs: Clk, Reset, ADD, Shift_En, SUB, Redo [7:0] D, [8:0] Sum, [8:0] Sub,
Outputs: [16:0] Data_Out, M
Description: 16-bit register whose values are set by the switches in the lower most bits when reset and switches plus the upper 8 bits when adding, this register also shifts when shift_en is 1.
Purpose: To implement a multiplier output in one register rather than utilizing two separate registers.