ECE385

Spring 2020

Experiment #5

An 8-Bit Multiplier in SystemVerilog

Yuantao Lu, Yizhen Lu

Section ABD

TA: Nicholas Cebry, Wenjie Pan

Introduction

In this lab, we use SystemVerilog and FPGA to build an 8-bit multiplier. This multiplier can perform multiplications on two 8-bit binary numbers. There is also a sign bit assigned in the multiplier so it also supports multiplications between all combinations of positive and negative operands. The multiplier accomplishes this operation basically by transferring multiplication into adding and shifting bits. The final answer, a 16-bit binary number with an extra sign bit, will be stored in two 8-bit shift registers and displayed on the HEX display segments on the FPGA board with sign bit indicated by an LED.

Pre-Lab Questions

A. Rework 8-bit multiplication example using Multiplier B = 7, and Multiplicand S = -59.

Initial Values: X = 0, A = 00000000, B = 0000011 (achieved using ClearA_LoadB signal), S = 11000101, M is the least significant bit of the multiplier (Register B).

Function	X	A	B	M	Next Step
Clear A, Load B	0	00000000	00000111	1	Since M = 1, multiplicand will be added to A.
ADD	1	11000101	00000111	1	Shift XAB.
SHIFT	1	11100010	10000011	1	Add S to A.
ADD	1	10100111	10000011	1	Shift XAB.
SHIFT	1	11010011	11000001	1	Add S to A.
ADD	1	10011000	11000001	1	Shift XAB.
SHIFT	1	11001100	01100000	0	Shift XAB.

SHIFT	1	11100110	00110000	0	Shift XAB.
SHIFT	1	11110011	00011000	0	Shift XAB.
SHIFT	1	11111001	10001100	0	Shift XAB.
SHIFT	1	11111100	11000110	0	Shift XAB.
SHIFT	1	11111110	01100011	1	Stop. Product in AB.

Written Description And Diagrams of Multiplier Circuit

Summary of Operation

To load operands, we first set switches S to indicate the multiplier and then press the ClearA_LoadB button to load that number into register B, which is an 8-bit shift register. Then, we adjust switches S to represent the multiplicand. These are all preparation steps before the execution.

To perform the operation, we just press the Run button to activate the whole program. The program will go through a state machine designed inside the control unit, which automatically performs adding and shifting bits eight times to complete the multiplication. After the computation, the result will be stored inside two shift registers, A and B, and one extended bit X will be used to indicate the sign of the product.

To clear all data stored in shift registers, just simply press the Reset button, and all data will be wiped out and replaced with zeros.

Top Level Block Diagram

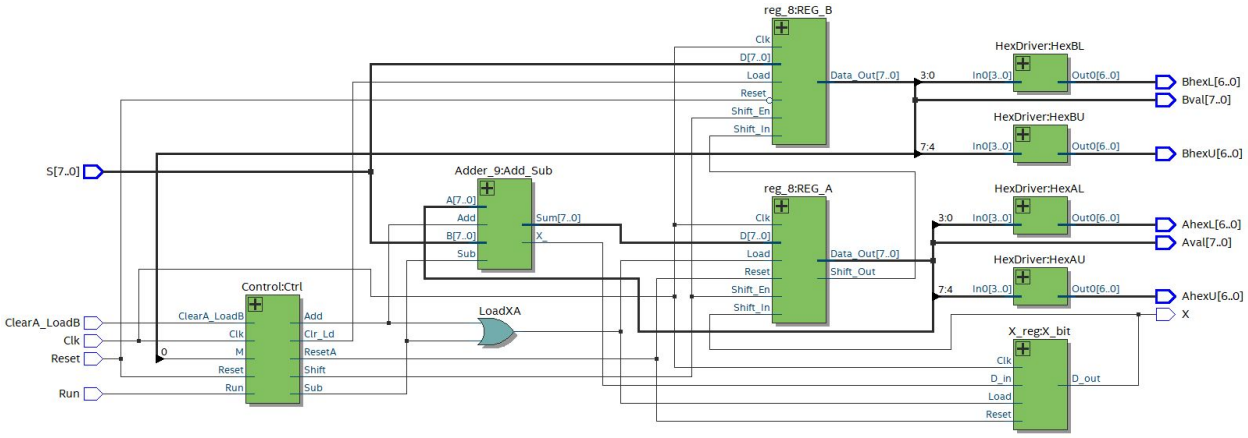

Figure 1. Top Level Diagram From RTL Viewer

Written Description of .sv Modules

Module: *multiplier.sv*

Inputs: *Clk, Reset, Run, ClearA_LoadB, [7:0] S*

Outputs: *X, [7:0] Aval, Bval,*

[6:0] AhexU, AhexL, BhexU, BhexL

Description: This is the 8-bit multiplier which computes the product of two binary numbers(either positive or negative) by loading in operands through *S* and *ClearA_LoadB*, then turn on *Run*. To clear data, turn on *Reset*, and all data stored in the registers would be removed and replaced with 0s.

Purpose: This is the top-level module of the multiplier, which organizes all other modular units in the project together according to the desired data flow.

Module: *Control.sv*

Inputs: *Clk, Reset, Run, ClearA_LoadB, M*

Outputs: *Clr_Ld, Shift, Add, Sub, ResetA*

Description: This is the control unit which sends out critical control signals like

Shift, Add, Sub to the entire system to inform the current configuration the multiplier should be at, or which operation/function the multiplier is about to perform. It updates its output value at each rising edge of the *Clk* signal. The unit has an inside state machine, which is responsible for providing the output of the control unit and would be activated by *Run* signal. When we hit *Run*, the control will continuously send out one group of output signals at each of the 17 following clock cycles to command the rest of the systems(register unit and adder unit especially) to perform the adding and shifting operations to calculate the product. *Clr_Ld* is held at Low during the entire operation procedures to prevent undesired inputs to interrupt the current running operations.

Purpose: This is the "heart" of the multiplier, which sends out control signals to tell the system which operation to take in the current state and prohibits inappropriate behavior of the system by restricting the *Clr_Ld* signal during the entire operation cycle.

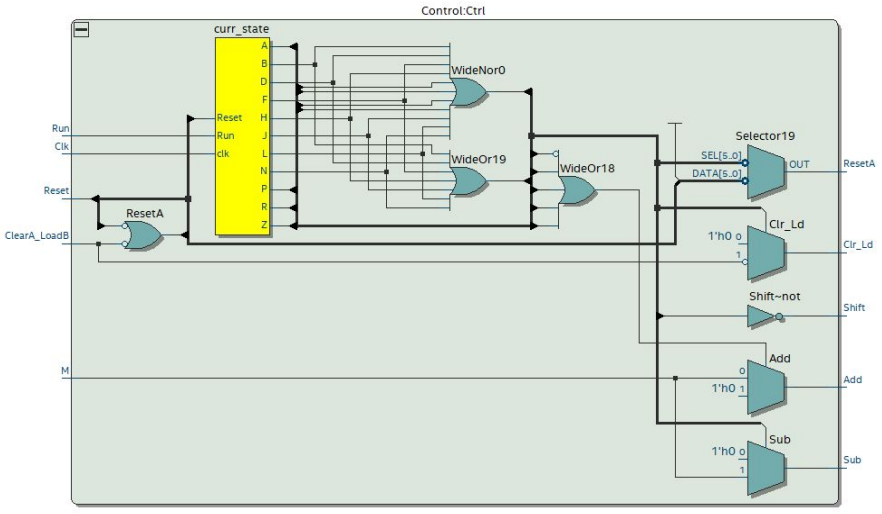

Figure 2. RTL Diagram for Control Module

Module: *reg_8.sv*

Inputs: *Clk, Reset, Shift_In, Load, Shift_En,*

[7:0] D

Outputs: *Shift_Out, [7:0] Data_Out*

Description: This is a positive-edge triggered 8-bit shift register with synchronous reset and load. When *Load* is High, *D* will be loaded into the register at the positive edge of *Clk*. When *Shift_En* is high, *Shift_In* will be loaded into the most significant bit of the register and all data in the register will be shifted rightward one bit and the least significant bit data will become the *Shift_Out* output at the rising edge of *Clk.*

Purpose: This is the shift register unit which is used to store data of each step of the calculation and shift the data serially to the right when needed.

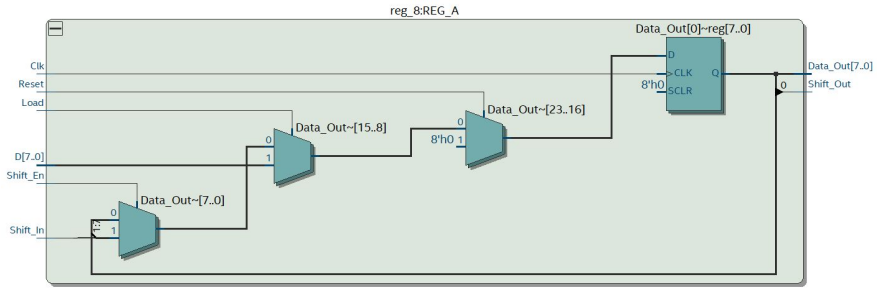

Figure3. RTL Diagram for Register Module

Module: *X_reg.sv*

Inputs: *D_in, Reset, Clk, Load*

Outputs: *D_out*

Description: This is a positive-edge triggered 1-bit register/flip-flop with reset and load functions at the rising edge of the *Clk* signal. It loads data *D_in* into the register when *Load* is on and clears data when *Reset* is on.

Purpose: This is a 1-bit register to store values for X, the sign bit of the product,

which is used as one of the inputs for the arithmetic shift operations for the 8-bit registers, and applied to the LED display for the final result.

Module: *Adder_9.sv*

Inputs: *[7:0] A, B,*

Add, Sub

Outputs: *X_, [7:0] Sum*

Description: This is an 8-bit adder/subtractor which performs summation or subtraction on two 2's complement 8-bit binary numbers. When *Add* is high, the module performs addition, whereas when *Sub* is high, the module performs subtraction. The operation is completed using 9 full adders inside the module. The output of the operation is sent out through *Sum* signal and *X_* is the extended sign bit to indicate the sign of the final answer.

Purpose: This is the adder/subtractor which performs addition and subtraction essential in the multiplication process.

Module: *full_adder.sv*

Inputs: *x, y, z*

Outputs: *s, c*

Description: This adder performs one-bit addition with carry-in and carry-out bits. Signals *x, y* are the two operands of the addition, *z* is the carry-in bit. These signals would produce *s* as the sum of the operands with carry-in bits, while *c* as the carry-out bit.

Purpose: This is the most fundamental addition unit required inside the adder/subtractor module.

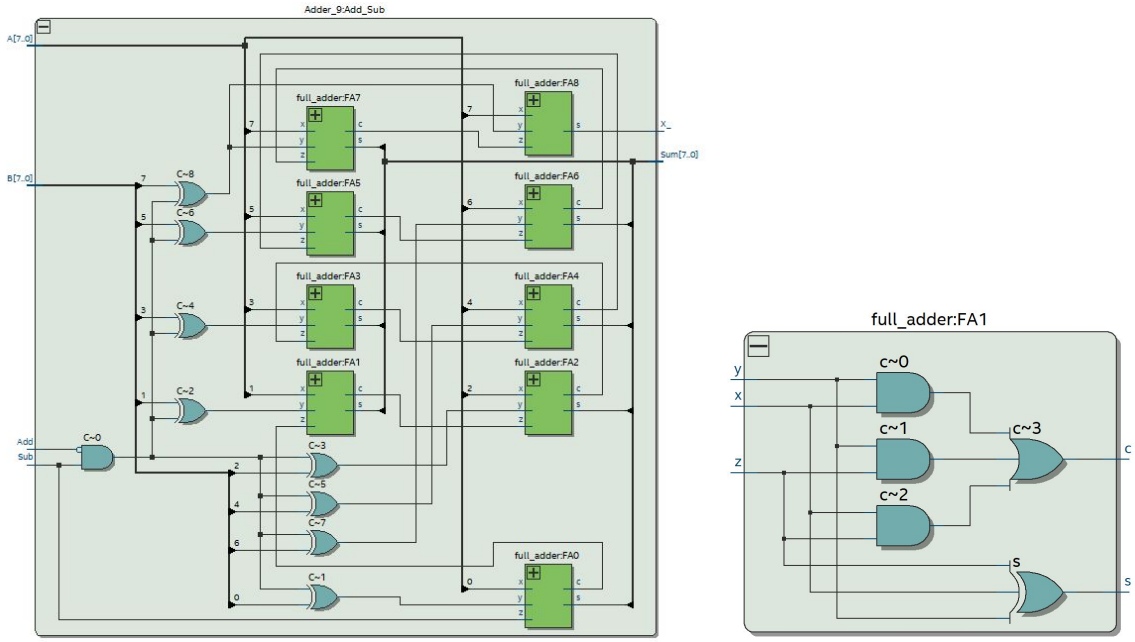

Figure 4,5. RTL Diagram for Adder/Subtractor Module

Module: *HexDriver.sv*

Inputs: *[3:0] In0*

Outputs: *[6:0] Out0*

Description: This module transfers input binary numbers into their corresponding seven-bit binary sequences for LED display.

Purpose: This module helps transfer the data stored in shift registers into Hexadecimal form and into a form that can be displayed on LED and easy to read.

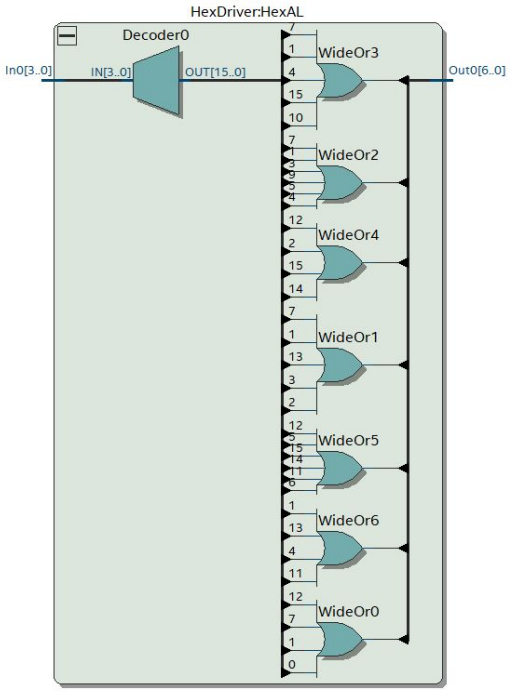

Figure 6. RTL Diagram for HexDriver Module

State Diagram for Control Unit

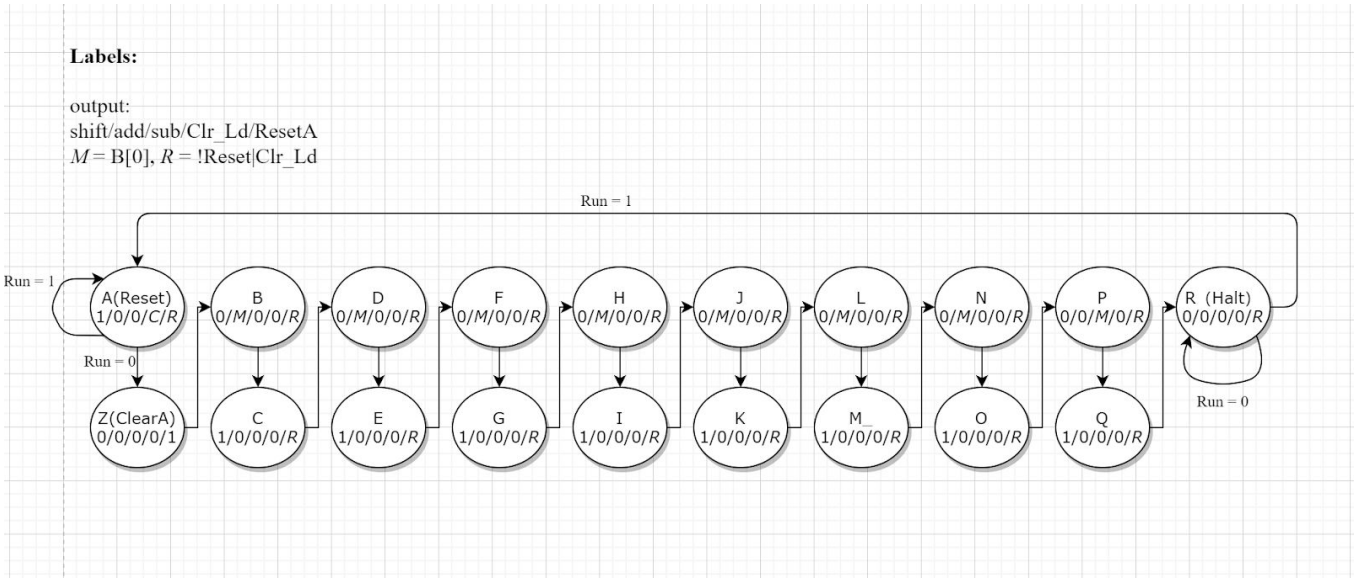

Figure 7. State Diagram for Control Unit

This Moore State Machine only takes *Run* as input, which is an active-low signal. However, it has several outputs. *shift* decides whether the registers need to be shifted; *add*/*sub* decides do we need to subtract or add *S* to register *A*; *Clr_Ld* means clear *A* and Load *B*, which is also an active low signal; *ResetA* resets register *A* and *X*'s value to 0; *Reset* is a signal passed into Control Unit that represents the reset signal from user.

Annotated simulation waveforms

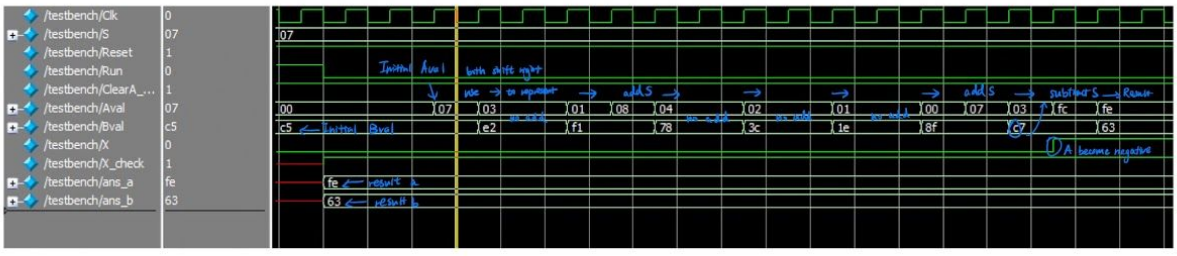

Figure 8. Waveform of operation +-*

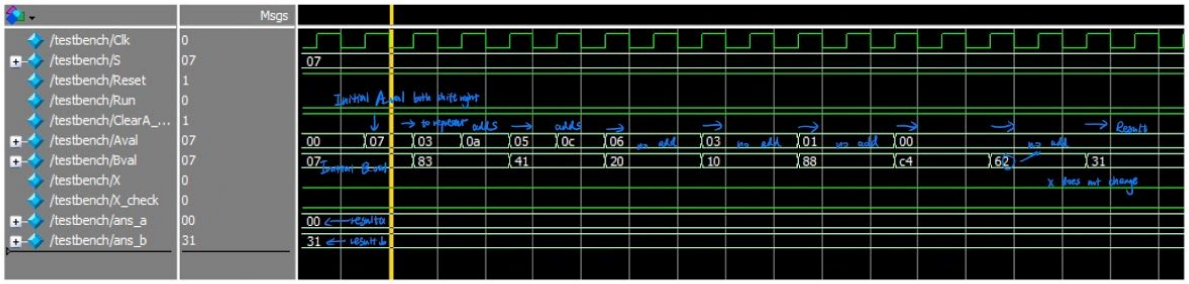

Figure 9. Waveform of operation ++*

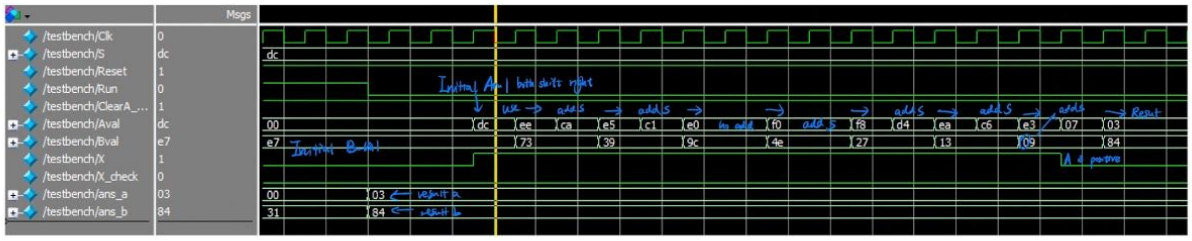

Figure 10. Waveform of operation --*

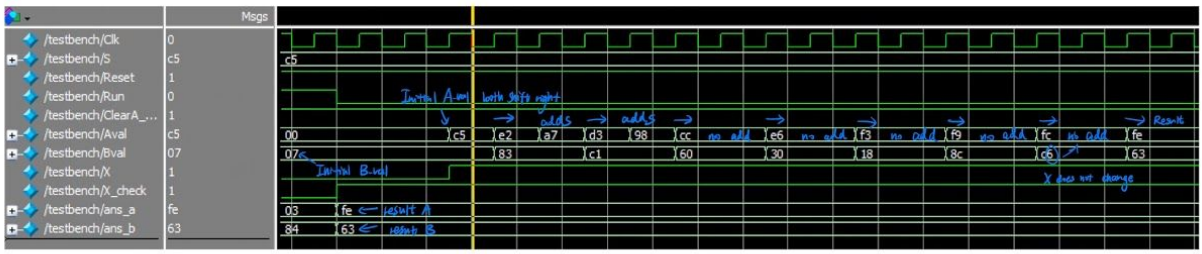

Figure 11. Waveform of operation -+*

Post-Lab Questions

1). Design Resources and Statistics Table

LUT	96
DSP	No Blocks Used
Memory(BRAM)	0 bit
Flip-Flop	36
Frequency	65.29 MHz
Static Power	98.52 mW

Dynamic Power	2.61 mW
I/O Thermal Power	48.14 mW
Total Power	149.27 mW

Possible measures to enhance performance of the multiplier:

- Using Carry-Select Adders instead of Carry-Ripple Adders in the adder/subtractor unit since Carry-Select Adders are generally more time efficient than the Carry-Ripple Adders, which could allow a higher frequency.
- The *Reset* and *Load* signals for X register and register A share the same logic gates respectively. We can use the same signal inputs to those ends to reduce the number of gates used.

2).

- X register is used to represent the sign of the final product. The X register gets set/cleared basically when register A gets set/cleared. To be more specific, X is cleared either when *Reset* or *ClearA_LoadB* is activated. X is set when either *Add* or *Sub* signal is High. Namely, every time register A parallelly loads the value from the adder/subtractor, the X value would also be updated.
- Continuous multiplication is another function that this multiplier can sometimes achieve. Since the algorithm is based on 8-bit times 8-bit multiplication, as long as the previous product exceeds this constraint, it will fail to provide the correct answer. To be more specific, if the previous product is not between -128 and 127(decimal), namely the most significant bit of register B is no longer a sign indicator, the algorithm will not be able

to give the correct result. This limits all continuous multiplications to be within this pretty small interval.

- The algorithm used in the multiplier is basically identical as the pencil-and-paper method. The advantage is that at the final stage for subtraction, the multiplier changes it to adding the number's 2's complement. However by hand, it is relatively harder to deal with binary subtractions. Also, it is more efficient because it only takes less than 20 clock cycles, which is about 400ns. In addition, comparing to the naive algorithm for multiplication(a*b = a + a*(b-1)), it is faster. Let the length of the inputs be *n*, our algorithm will take at most *n* n-bit add/subtract operations to complete, however, the naive algorithm will take at most 2^n times of n-bit additions to complete.

  As for disadvantages, say if the multiplier is a small number, like 7, it only has three digits of 1, then we can simply stop when these three bits are added. However, the programmed multiplier will still go through all bits to complete the result. It still has potential to be improved.

Conclusion

This multiplier is able to perform multiplications between either positive or negative 8-bit binary numbers(2's complement). It would also support continuous multiplication as long as the previous product is within the tolerance range. (-128~127 in decimal) In the process, we initially found that the code works perfectly fine in simulation, but doesn't work at all when loaded into the FPGA. Then we figured out that it is because all the keys on the FPGA board are active low. After we negate these control signals inside the multiplier, it finally works. So

I would really recommend emphasizing this point on the manual so that students wouldn’t waste more time debugging because of this mistake, while their code could actually be valid.