ECE 385

Fall 2021

Experiment #4

An 8-bit Multiplier in System Verilog

Michael Grawe, Matthew Fang
ABE
Abigail Wezelis

Index

Introduction

The focus of Lab 4 was to create a multiplier that takes two 8-bit 2's complement inputs and outputs their 16-bit product. This is done by a shift add method. Switches are used to first load values into the multiplicand register (B) by clicking the Reset_ClearA_LoadB button. Then, the switches become the multiplier. Once Run is clicked, the shift add method begins, and the lowest bit of B determines whether or not an addition occurs between register A, which is initialized to 8'b0, and the multiplier switch values. Regardless of whether an addition occurs, the lowest bit in A is shifted to B, and the lowest bit of B is shifted out. There are 8 total shifts that occur, 7 possible adds, and one possible subtraction. This subtraction takes the place of the last add, since after 7 shifts, the lowest bit of B would be representative of its original sign bit. At the end of operation, A should hold the highest 8 bits of the product, and B should hold the lowest 8.

Pre-Lab

The table below is the desired behavior of our circuit with a multiplicand of 11000101, which comes from the switches, and a multiplier 00000111, stored in B once the ClearA_LoadB_Reset.

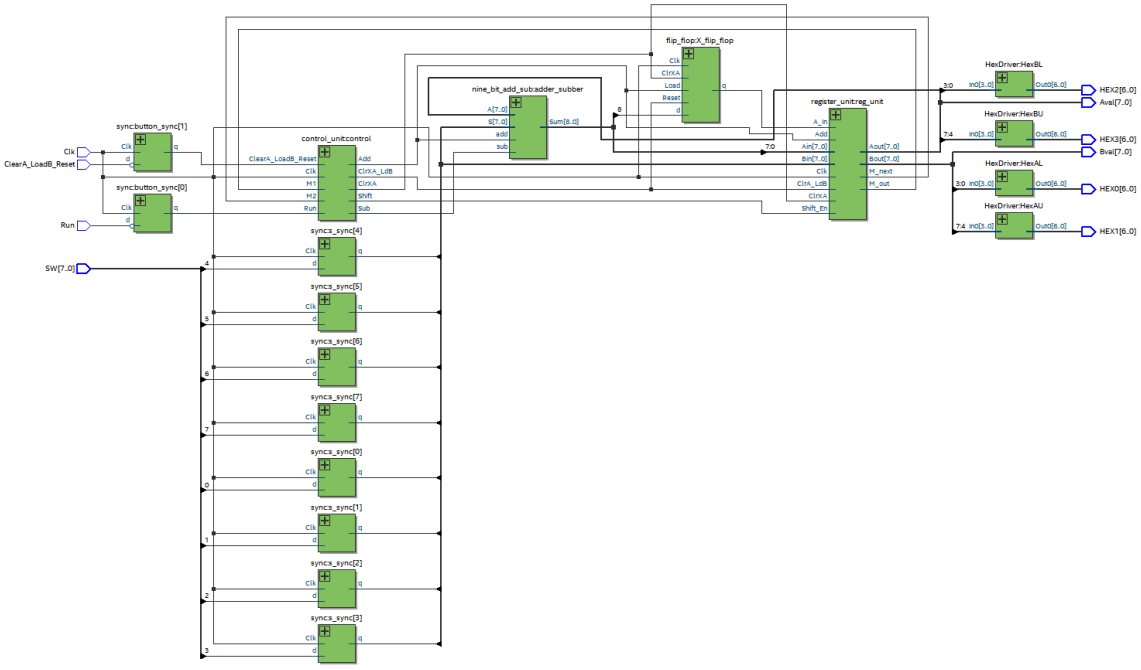
Function	X	A	B	M	Next Step
ClearA_LoadB_Reset X and A get zeroed, B gets the multiplier	0	00000000	00000111	1	M=1, so next step is ADD multiplicand to A
ADD	1	11000101	00000111	1	SHIFT
SHIFT	1	11100010	10000011	1	ADD
ADD	1	10100111	10000011	1	SHIFT
SHIFT	1	11010011	11000001	1	ADD
ADD	1	10011000	11000001	1	SHIFT
SHIFT	1	11001100	01100000	0	SHIFT b/c M=0
SHIFT	1	11100110	00110000	0	SHIFT
SHIFT	1	11110011	00011000	0	SHIFT
SHIFT	1	11111001	10001100	0	SHIFT
SHIFT	1	11111100	11000110	0	SHIFT
SHIFT	1	11111110	01100011	1	DONE 16 bit product in A and B

Summary of Operation

The multiplicand is loaded into register B using the switches and the reset_load_clear button on the FPGA. The multiplier can simply be loaded into the switches. When run is pressed, the least significant bit of B is checked. If it is '1,' The values of the switches are added to register A, which would have been cleared when run was pressed. Registers A and B are then shifted left with the least significant bit of A being shifted into the most significant bit of B while A is simply sign extended. If the least

significant bit was ‘0,’ registers A and B are just shifted. This continues for the first seven bits of B. When we evaluate the 8th B bit, if it ‘1,’ we subtract, instead of add, the value saved in the switches from the value saved in A before shifting. If it is ‘0,’ we only shift again. After this final shift operation, the product of the two numbers will be saved in A and B as a 4 bit hex value.

Top Level Block Diagram



Description of all Modules

Module: register_unit
Inputs: Clk, ClrA_LdB, A_in, Shift_En, Add, ClrXA,
[7:0] Ain, Bin
Outputs: M_out, M_next,
[7:0] Aout, Bout

Description: This module instantializes our registers A and B. This module takes in inputs, like the ClrA_LdB signal, resends it to the correct register. For example, the clrA_ldB goes into the load input for register B and the clr input for register A.

Purpose: The purpose of this module is to act as a buffer between the processor and the actual registers.

Module: Reg_8
Inputs: Clk, Clr, Shift_In, Load, Shift_En,

[7:0] D
Outputs: Shift_out, Shift_out2,
[7:0] Data_Out

Description: This module controls the shifting, clearing, and loading of the different registers. It also outputs its two least significant bits to the control unit so the program knows whether or not it needs to shift or add.

Purpose: This module controls the state of the different registers.

Module: Nine_bit_add_sub
Inputs: [7:0] A, S,
sub, add
Outputs: [8:0] Sum,
cout

Description: This module takes in the 8-bit values from register A and the switches and sign extends them to 9 bits. The sub signal decides whether or not the switch values are subtracted from the value in the A register and the add signal decides whether or not that sum is passed or the original sign extended A is passed.

Purpose: The purpose of this module is to do the addition and subtraction calculations that make up the multiplication.

Module: Four_bit_adder
Inputs: [3:0] A4, B4,
C4in
Outputs: [3:0] S4,
C4out

Description: This module takes in two 4-bit values (A4 and B4), as well as a carry-in bit (c4in), and uses the ripple connection of four full adders to output the 4-bit sum (S4) and carry out (c4out).

Purpose: This and the Five_bit_adder act as building blocks to create the Nine_bit_add_sub adder.

Module: Five_bit_adder
Inputs: [4:0] A5, B5,
C5in
Outputs: [4:0] S5,
C5out

Description: This module takes in two 5-bit values (A5 and B5), as well as a carry-in bit (C5in), and uses the ripple connection of four full adders to output the 5-bit sum (S5) and carry out (C5out).

Purpose: This and the Four_bit_adder act as building blocks to create the Nine_bit_add_sub adder.

Module: Full_adder
Inputs: ain, bin, ci
Outputs: sout, co

Description: This module takes in two 1-bit values (ain and bin), as well as a carry-in bit (ci), and implements logic to output the binary sum of the three through the sum and carry out (south and co respectively.

Purpose: This module acts as a building block for the Four and Five bit adders. This is done by rippling multiple full adders together.

Module: control_unit
Inputs: Clk, Run, ClearA_LoadB_Reset, M1, M2
Outputs: ClrXA_LdB, Shift, Add, Sub, ClrXA

Description: This module is a finite state machine with 19 states. There are seven add states, one subtract state, eight shift states, one start state, one init state, and one finish state. The first state, start, waits for the Run signal before moving to the init state. This is also where register B can be loaded and the switches can be set. In the init state, the flip-flop and register A are cleared and the least significant bit of register B, M1, is checked to decide whether or not our next state adds or skips. If M1 is high, the values in the switches are added to register A and then both register A and register B are shifted. If M1 is low, registers A and B are just shifted. From the second shift state to the last shift state, the second least significant bit of register B, M2, is checked instead. This helps avoid timing errors if we were to use M1. The eighth add state is replaced with a subtract state to account for multiplication with negative numbers. Finally, the finish state is used as a halt state so users can view the answer before resetting or doing another calculation.

Purpose: This module is important for controlling when we add and subtract and when we shift in order to provide the correct answer.

Module: Flip_flop
Inputs: Clk, Reset, Load, ClrXA, d
Outputs: q

Description: This module takes in the most significant bit from the nine_bit_add_sub module and stores it for one clock cycle before shifting it into register A.

Purpose: This module is used to sign extend A when the registers are shifted.

Module: Processor
Inputs: Clk, ClearA_LoadB_Reset, Run
[7:0] SW
Outputs: [7:0] Aval, Bval,
[6:0] HEX0, HEX1, HEX2 , HEX3

Description: This module is the top level for this lab. It takes all the user inputs and acts as a central hub for all the other modules and instantializes all the other modules.

Purpose: This connects all our modules together so they can communicate together and send and receive inputs and outputs to one another.

Module: Synchronizer
Inputs: Clk, d
Outputs: q

Description: This module syncs our buttons with clock cycles.

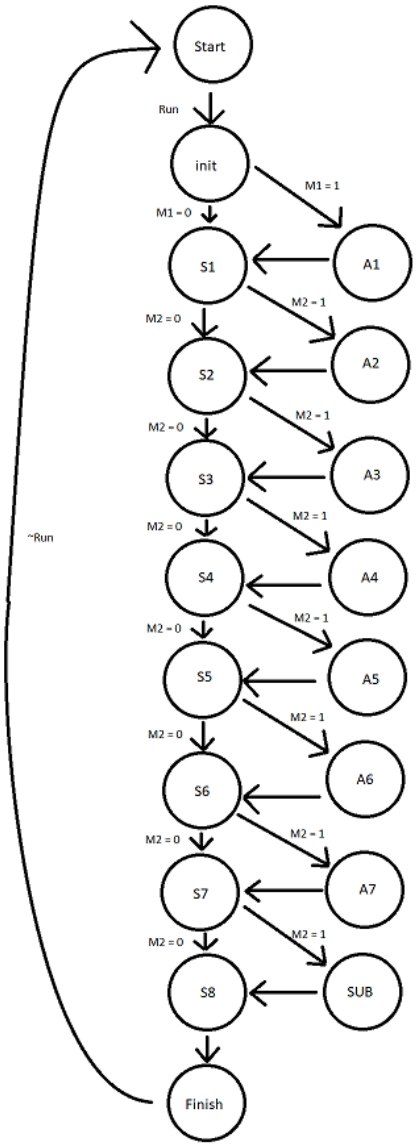
Purpose: This is used to guarantee our button presses are registered. If the buttons were not synced, there could be cases where our button signal would be between clock cycles and missed.

Module: Hexdriver
Inputs: [3:0] In0
Outputs: [6:0] Out0

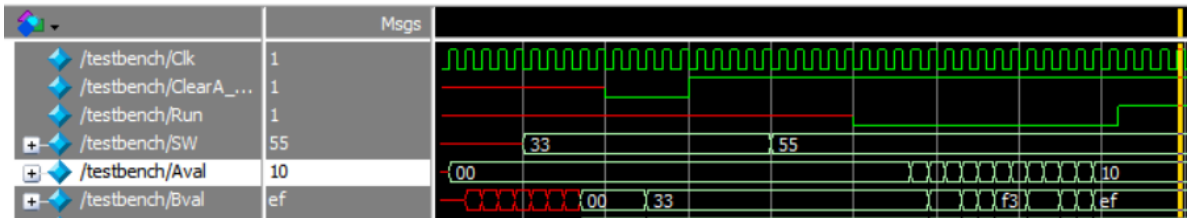
Description: This module takes the 4-bit input In0 and converts it to the corresponding 7 bit representation of its HEX value in Out0.

Purpose: This module is used to map the contents of the registers into Hex values so that they can be displayed on the Hex display on the DE-10.

Moore State Diagram

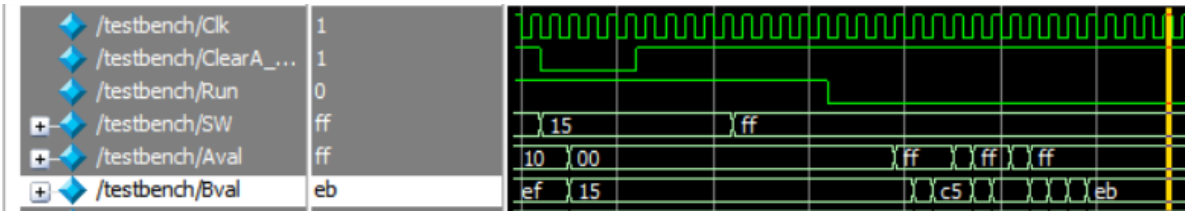


Simulation Waveform

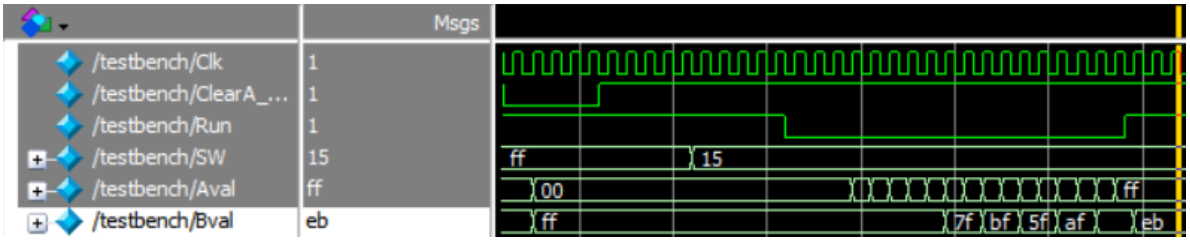


This simulation trace shows two positive numbers, 2'h33, 51, and 2'h55, 85, multiplied together. When ClearA_LdB is pressed, the values stored in the switches, 2'h33, are stored to register B. The switches are then set to 2'h55 and run is pressed. The product

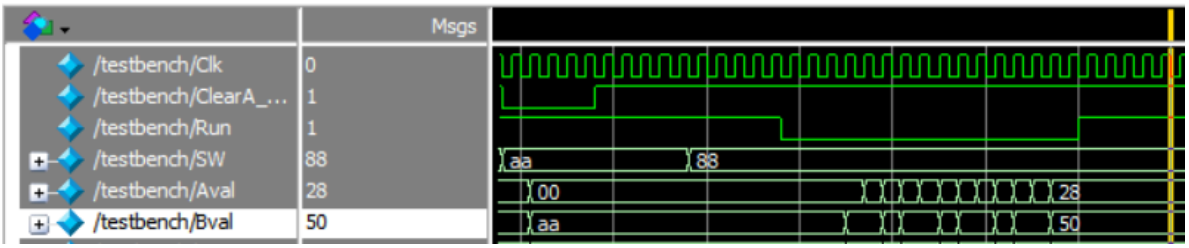
of the two numbers is rapidly calculated as add and shift operations are completed and the final product is stored in the two registers, 2'h10 in A and 2'hef in B to form the product 4'h10ef, 4335.



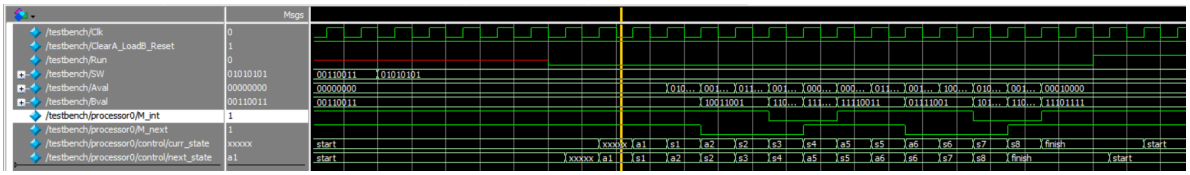
This simulation trace shows one positive number, 2'h15, 21, and one negative number 2'hff, -1, multiplied together. When ClearA_LdB is pressed, the values stored in the switches, 2'h15, are stored to register B. The switches are then set to 2'hff and run is pressed. The product of the two numbers is calculated after a series of add and shift operations and yields 4'hffeb, -21, as the product.



This simulation trace shows one negative number, 2'hff, -1, and one positive number 2'h15 21, multiplied together. When ClearA_LdB is pressed, the values stored in the switches, 2'hff, are stored to register B. The switches are then set to 2'h15 and run is pressed. The product of the two numbers is calculated after a series of add and shift operations and yields 4'hffeb, -21, as the product.



This simulation trace shows two negative numbers, 2'haa, -86, and 2'h88, -120, multiplied together. When ClearA_LdB is pressed, the values stored in the switches, 2'haa, are stored to register B. The switches are then set to 2'h88 and run is pressed. The product of the two numbers is calculated after a series of add and shift operations and yields 4'2850, 10320, as the product.



Here is a closer look on the series of adds and shifts for the case of two positive numbers multiplied to one another. A similar analysis can be done on the other simulation traces but they are not provided in this report. On the yellow marker, the initial state, M1 is checked. Because it is high, we go straight to the add1 state. After the add1 state, we go straight to the shift one state and look at M_next, the next least significant bit of B to determine whether or not we need to add or just shift. Because B was 2'h33 = 8'b00110011, we see that M_next is high, so we need to enter the add2 state. This logic of checking whether or not the next state is add or shift continues. At the shift7 state, if register B's original value had 1 as the most significant bit, we would have to enter the subtract state instead of an add8 state, but, because the most significant bit was 0, we only shift. From there, we enter the finish state and wait for the run to be released before we enter the start state again where we can do another calculation.

Post-Lab Data

The following table displays the data that comes from our multiplier implementation.

Data Category	Data from Multiplier
LUT	94
DSP	0
Memory(BRAM)	0
Flip-flop	38
Frequency	75.87 MHz
Static Power	90.08 mW
Dynamic Power	1.71mW
Total Power	128.76 mW

Post-Lab Question

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

The X register's purpose is to ensure that the correct bit is shifted into A when the multiplier is in a SHIFT state. In doing this, it ensures that the sign of A is preserved. The X register gets set whenever the circuit is in an ADD state. This is done so that X is only updated whenever an addition is performed between S and A. X register is cleared whenever the ClearA_LoadB_Reset button is pressed, or whenever the multiplier unconditionally enters the INIT state at the beginning of operation. This is done so that A and X are always zeroed out when performing sequential multiplication without hitting the Reset button.

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 we used the carry out of an 8-bit adder, then our sine bit would not be correct. For example, if A holds 11110000 and we want to perform an add with S=00001111, then our X value would be 0 if we used the carry-out of 8-bit addition. This is clearly incorrect, as A is a negative number and should remain negative through this addition. Shifting in that 0 X value will cause A to become positive and switch signs. Therefore, we sign extend A and S and use the 9^{th} bit as X.

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

Our limitations on continuous multiplication lie in the X and A registers being cleared to 0. This is an assumption that we make when performing our multiplication. If this is not able to occur while preserving the numbers being multiplied, then continuous multiplication is impossible. For example, if the 16 bit product takes more than 8 bits to represent in 2's complement, then it cannot be a candidate for continuous multiplication. In other words, if the product is outside the range of (-128,127), it is not a candidate for continuous multiplication. This makes sense, since if we were to clear the registers and try to perform this multiplication, it would be impossible to do with and 8-bit multiplier.

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

The biggest advantage of the implemented algorithm is memory usage. The previous addition values do not need to be stored using the shift method, as they would with the paper and pencil method. The biggest disadvantage of the implemented algorithm is that it can be more complicated to understand what is going on, or why it is going on. The paper and pencil method is more intuitive, and much easier to understand. However, as long as the shift algorithm is implemented correctly, its savings on memory usage outweigh that potential drawback.

Conclusion

The goal of this lab was to create a multiplier able to take in two 8-bit 2's complement inputs and output a 16 bit product from scratch without any prior files. This lab relied heavily on model sim waveform analysis to debug the project and guarantee functionality. One design aspect that was exceptionally difficult was having the least significant bit control whether addition or shifting was required. Instead of using the least significant bit, our implementation used the second lowest bit in order for our multiplier to operate correctly. To control the state of the next state needs to look at the second lowest bit instead of the least significant bit because when we shift and reach the next state, the second lowest bit will become the least significant bit. If we instead always looked at the least significant bit, an undesired next state would be chosen and would result in an incorrect product