ECE 385

Fall 2022

Experiment #4

An 8-Bit Multiplier in SystemVerilog

Ravi Thakkar, Siraj Khogeer

TA: Gavin Wu

10-1-2022

Introduction

In this lab, we use the knowledge gained from Lab 3, where we worked on creating different types of adder circuits, and we implement a two's-complement, sixteen bit multiplier circuit that is capable of storing sixteen bits for the product. We use two eight-bit registers A and B, one one-bit register X and a nine-bit adder circuit to implement an add and shift algorithm which computes partial multiplications using the adder circuit. The algorithm consists of seven adds, eight shifts, and a subtract step to compute the final product.

Pre-Lab Question

-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	Comments on next step
Reset	0	0000 0000	0000 0111	1	Add SW to A
Add	1	1100 0101	0000 0111	1	Shift XAB by one bit after ADD complete
Shift	1	1110 0010	1000 0011	1	Add SW to A
Add	1	1010 0111	1000 0011	1	Shift XAB by one bit after ADD complete
Shift	1	1101 0011	1100 0001	1	Add SW to A
Add	1	1001 1000	1100 0001	1	Shift XAB by one bit after ADD complete
Shift	1	1100 1100	0110 0000	0	Do not add S to A since M = 0. Shift XAB.
Shift 5 times	1	1111 1110	0110 0011	1	Do not add S to A since M = 0. Shift XAB.

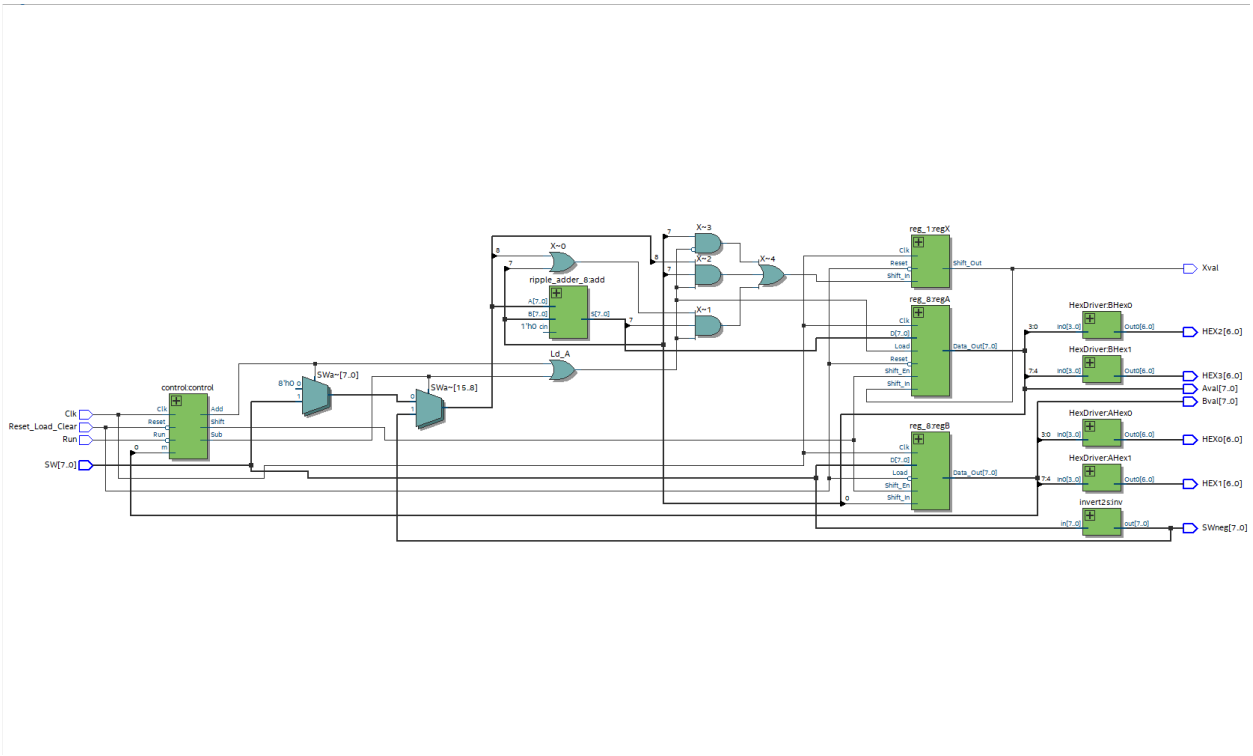
Written Description: Summary of Operation

Explain in words how operands are loaded, how the multiplier computes its result, how the result is stored, etc.

The multiplier works by essentially adding the multiplicand, multiplier number of times, or adding A B times. The way this is done is by first adding the SW into A whenever the LSB of B is 1, and storing the sign of A into X. The module then arithmetically right shifts XAB and then looks at the LSB, and if it's 1, then it adds SW into A and shifts again. This is repeated for the first 7 bits of B. When the last bit of B is reached, if it is 1, then the circuit subtracts SW from A. This is done by first inverting all the bits of SW and then adding 1, which is the same as negative in 2's complement. This is then added to A to get the final result which is stored in AB. Register X has two main uses, the first is to store the sign of A, and the second is to conserve the sign of A+SW. The issue here is that A+SW can sometimes overflow, which occurs when two negative numbers give a positive result, or two positive numbers give a negative result. This is solved by hardwiring X to be 1 if both A and SW are negative, and 0 if both are positive. If they are of different signs, then X = A[7]+SW[7] taking into account the carry. This solves the overflow problem of the circuit. Overall the circuit works by looking at the LSB of B, and adding the SW into A and then right shifting XAB, and when the last bit of B is reached, it subtracts if the LSB is 1, which follows the 2's complement, as if B[7] is 1, then that is equal to -2^6.

Top Level Block Diagram

This can be generated from the RTL viewer. Please only include the top-level diagram and not the RTL view of every module.



Written Description of .sv Modules

Module: **HexDriver** in HexDriver.sv

Inputs: In0[3:0]

Outputs: Out0[6:0]

Description: HexDriver used to map the inputs to outputs that can be seen on the hex display.

Purpose: Maps the input to outputs that can be displayed on the HEX display.

Module: **reg_8** in reg_8.sv

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

Outputs: Shift_Out, Data_Out[7:0]

Description: This module is a simple eight-bit shift register. At the positive edge of a clock cycle, if the Reset signal is high, the register is initialized with zeros. If the Load signal is high, Data_Out is loaded with D, the eight-bit input to the register. If Shift_En is high, then a right shift is performed where the MSB is assigned to be Shift_In and the LSB is assigned to Shift_Out. The rest of the bits simply shift to the right.

Purpose: The purpose of this module is that our design, as recommended by the Lab 4 manual, uses eight bit shift registers A and B to store one of the inputs and the output. The right shift capability of this register is because the algorithm that we've implemented for two's complement multiplication makes it necessary for that sort of capability in our registers.

Module: **register_unit** in register_unit.sv

Inputs: Clk, Reset, A_In, B_In, Ld_A, Ld_B, Shift_En, D[7:0]

Outputs: A_Out, B_Out

Description: This module creates two eight-bit shift registers through the reg_8 module with inputs for load input as well as one shift enable for both register A and B.

Purpose: We decided to interface with the two registers A and B through a register unit module instead of directly calling the reg_8 module so we are able to organize what we are doing with the register unit as a whole.

Module: **reg_1** in reg_1.sv

Inputs: Clk, Reset, Shift_In

Outputs: Shift_Out

Description: 1-bit shift register, that has data inputs Shift_In and output Shift_Out. Synchronous register that if reset is high, then it clears to 0.

Purpose: This register module is used to store the value of register X, which holds the sign bit of SW+A. A register is needed because it has to be syncronys. Shift out is connected to the Shift_In of A.

Module: **invert2s** in inverts.sv

Inputs: [7:0]in

Outputs: [7:0]out

Description: This module turns a 2's complements output into the negative version of it. It does this by inverting all the bits but XOR it with 1, then passing it through a 8-bit ripple adder with input 1, and carry 0.

Purpose: This module is used to turn the SW to negative to be used in the subtract operation. Actual negative number is stored in another variable.

Module: **ripple_adder_8** in ripple_adder.sv

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

Outputs: S[7:0], cout

Description: This module takes in two eight-bit inputs (A and B) as well as a carry-in bit and returns an eight bit sum (S) and a carry-out bit. Essentially, this module is a 8 bit ripple carry adder, a smaller version of the 16 bit ripple carry adder module that we created. It uses eight full adders chained together through their carry-out and carry-in signals.

Purpose: This module is used in conjunction with the invert module to give a negative version of an input number. Used in the process of flipping the sign of 2's complement number.

Module: **full_adder** within ripple_adder.sv

Inputs: a, b, c_in

Outputs: s, c_out

Description: This module is the full adder that takes in the inputs a, b, and c_in and adds them, generating the outputs s and c_out. This is done through the formulas $s = a \oplus b \oplus c\_in$, and c_out = (a&b) | (a&c_in) | (b&c_in). Below is a truth table that displays how these functions are just adding these inputs.

a	b	c_in	s	c_out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Purpose: The purpose of this module is to be a building block for the ripple carry. Since they work with only one bit they are easy to construct and comprehend.

Module: **control** in control.sv

Inputs: Clk, Reset, Run, m

Outputs: Clr_Ld, Shift, Add, Sub

Description: The control module has a total of 4 inputs, and 4 outputs. The control system has 18 states. 8 shifts states, 7 add states, and 1 subtract state. The control signal sets Clr_Ld to 1 if Reset = 1, this is redundant. The control circuit starts in the halt state, where all signals are set to 0. When run is pressed, the system starts and goes to the next state at each clock signal. The states begin with an Add state, which sets Add = 1 if m = 1, and 0 for the rest of the signals. The next state is a shift state which sets Shift = 1, and the rest of the signals to 0. The next state is then an add signal, and this pattern is repeated until the 7th Shift state, after that state, if m = 1 the control module sets Sub = 1, and the rest to 0. The control then goes into the final shift state and then goes into the end state, which only returns to the halt state if Run = 0.

Purpose: The control module essentially has 16 states not including the rest and end states. The states start by seeing if m(B[0]) = 1, and if it is it sets the Add value to 1, and then moves to the shift state, this is repeated 7 times, and then on the last bit of B, if m is 1 it tells the rest of the circuit to subtract. It then waits until Run is unpressed.

Module: **Lab4test** in Lab4test.sv

Inputs: [7:0]SW, Clk, Reset_Load_Clear, Run,

Outputs:[6:0] HEX1, HEX0, HEX3, HEX2, [7:0] Bval, [7:0] Aval, Xval, SWneg

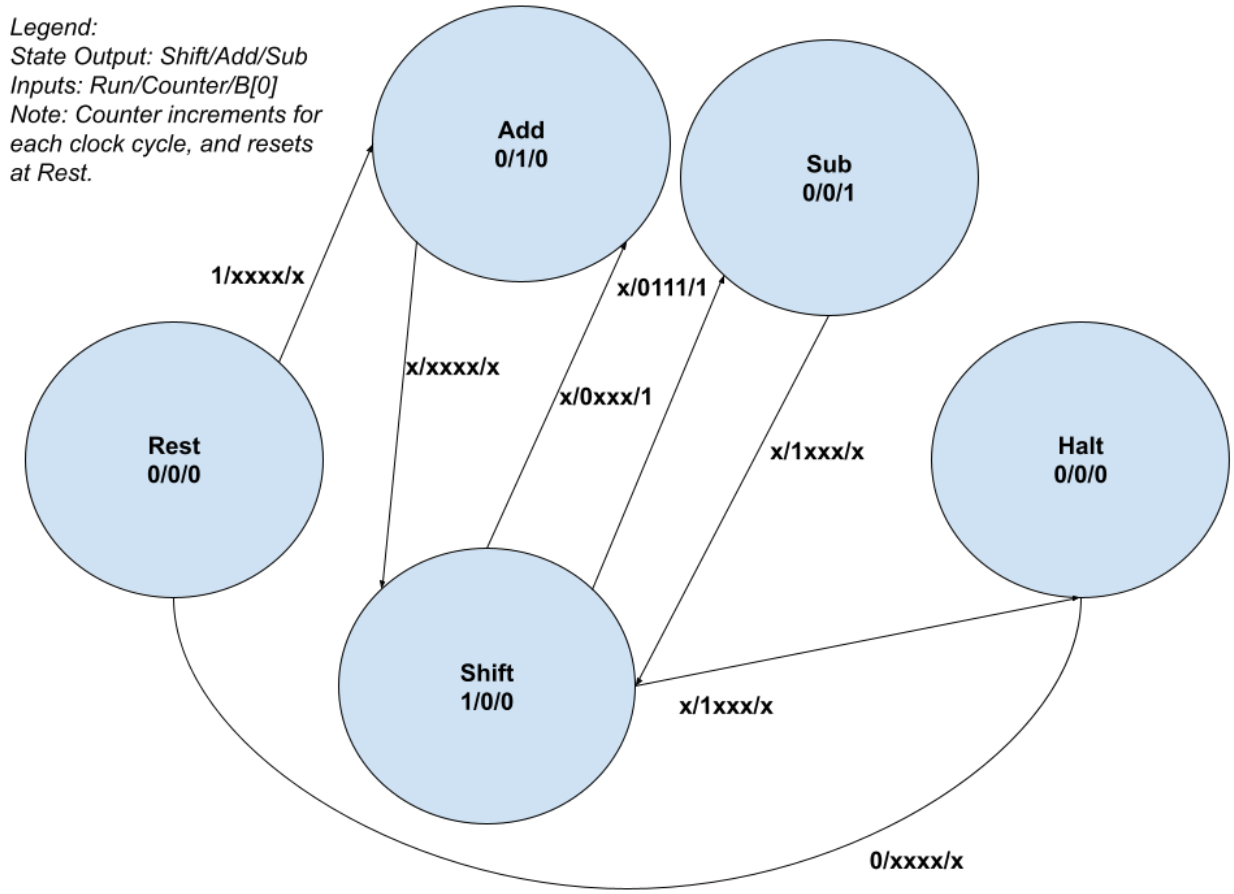
Description: Top level module for the lab. Module takes in the Switches, Clk, Reset, and Run signals and runs the multiplier circuit. The module starts by instantiating Reg A,B and X. The module then declares some temporary variables to hold values, this includes SWneg which stores the output of the negative of the SW which comes from the invert2s module

instantiation. The module then has an if statement with 3 conditions. If Add = 1 then SWa = SW, else if SUB = 1 then SWa = SWneg, else SWa= 0. Add and Sub are control signals from the instantiated control module. A Ld_A signal is then assigned with Add|Sub, which means that A will be loaded whenever an add or subtraction operation is done. A 8-bit ripple adder is then used to add RegA and SWa and which is then stored in A. X is then assigned using the formula X = A[7]&(Aval[7] | SWa[7])&Ld_A | Aval[7]&SWa[7]&Ld_A | Aval[7]&~Ld_A. This is the same as using a 9-bit adder with X being the MSB but this was used to make debugging easier. Function is the same. The X and S are then imputed into RegA and RegX, and at each clock cycle they are loaded. Note that X (input) is set to be equal to Aval[7] during the shift state, this is done as X needs to hold the sign bit when an Add state is skipped. Finally, Aval and Bval go through the HexDriver and are outputted. X was supposed to be connected to a 9-bit adder but was brought out of the adder for debugging purposes.

Purpose: Top Level for the lab, needed to complete the assignment. Used to instantiate all other modules and serve as the main circuit. Used for simulation.

State Diagram for Control Unit

This can be done in a program like Visio, but if the Quartus state diagram generator is used, you must label the states and transitions. By default, the tool does not generate a very legible state diagram.

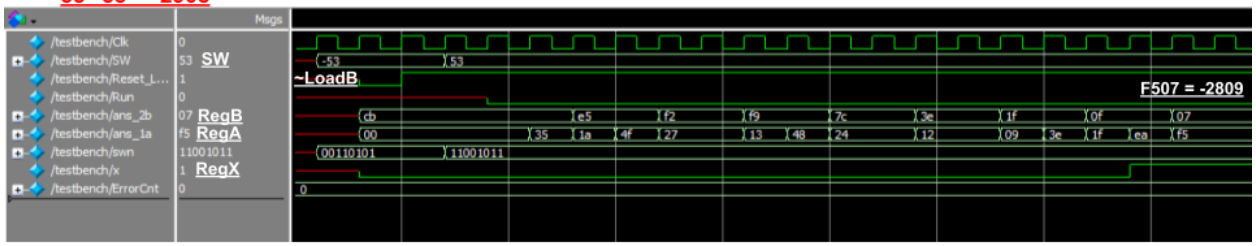


Annotated Pre-Simulation Waveforms

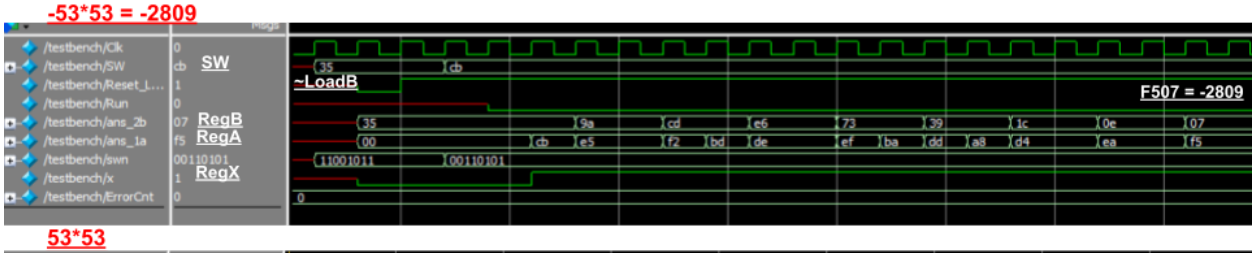
a. Must show 4 operations where operands have signs (+*+), (+*-), (-*+) and (-*-)

b. Waveform must have notes that clearly show the operands as well as the result, etc

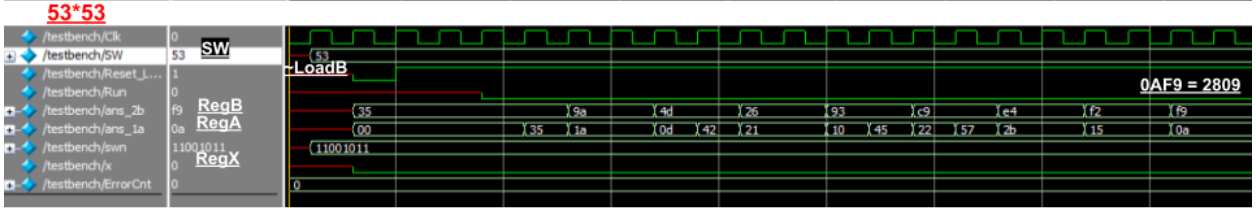
53*-53 = -2908



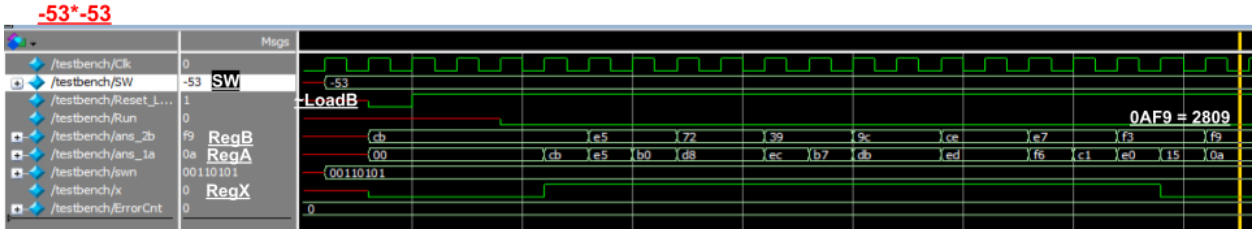
-53*53 = -2809



53*53



-53*-53



Postlab Questions

1. Complete the design statistics table

LUT	93
DSP	0
BRAM	0
Flip-Flop	35
Frequency Mhz	207.13
Static Power mW	89.94
Dynamic Power mW	0
Total Power mW	98.69

2. Make sure your lab report answers at least the following questions:
    a. What is the purpose of the X register? When does the X register get set/cleared?
    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?
    c. What are the limitations of continuous multiplications? Under what circumstances will the implemented algorithm fail?
    d. What are the advantages (and disadvantages?) of the implemented multiplication algorithm over the pencil-and-paper method discussed in the introduction?

a. The purpose of the X register is to represent what the sign of the A register is. Register X is set according to the Most Significant Bit (A[7]) of Register A after the subtraction of A with the switch bits, since the value is two's-complement and we want to conserve that sign.
b. Considering the possibility of overflow, if you used the carry-out of an eight bit adder, you would not represent the correct sign. Consider the example of having a four-bit adder with two's complement inputs A = 1111 (-1 in decimal) and B =0111 (7 in decimal). Adding these two inputs should result in an output of S = 0110 (6 in decimal). The C_out of a four-bit adder would be a 1, which would suggest that the sign of the output would be negative, which is untrue. However, if you used an n+1-bit adder, which is a five bit adder for this scenario, the 5-th bit of the Sum of that adder would be a zero which would indicate the correct sign of the operation. (11111+00111) = (00110), cout = 1.
c. Eventually, the numbers that you would be multiplying would get too large for the system to handle. This would be when the product from a multiplication operation can't be properly implemented with sixteen bits. Another issue is that some numbers can't be truncated to fit into 8-bits.
d. The advantage of the algorithm that we implemented is that we only need 9-bit adders instead of larger adders that we would need for the pen and paper method. However, the pen and paper method would allow all the partial sums to be added together easily without a subtraction operation in the end and a control module to track when a subtraction needs to happen. The tradeoff of this implementation is that it requires more registers to store these values to be added. Our implementation could also be better as it is serialized and easier to interpret.

Conclusion

a. Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it.

b. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it does not get changed.

Overall, the lab was a good continuation of the concepts that we had learned and implemented in the last lab. It was a good experience to be able to create all of the modules including the state machine to be able to manipulate more elementary constructs such as the ripple-carry adder and the shift register to be able to create a two's complement multiplier. It was also a good exercise to be able to implement the algorithm to multiply and using the pen-and-paper algorithm as a basic, more understandable implementation of an adder and then developing a more digital logic friendly algorithm.

The lab manual was pretty straightforward and it was not very difficult to follow. I think that what was done right was introducing the pen-and-paper method before trying to tell us to implement the actual algorithm because we felt like we would have been confused if we didn't get to look at that algorithm first.