ECE 385

Spring 2019

Experiment #5

An 8 Bit Multiplier in System Verilog

Rob Audino and Soham Karanjikar
Section ABJ, Friday 2:00-4:50
TA: David Zhang

Introduction:

In this lab, we were tasked with creating a multiplier for two 8-bit 2's complement numbers. We would have to implement it through a similar method that we learned as children: long multiplication. Given numbers A and B, this meant that we would have to take the least significant bit of B, multiply it by A, and then add the result to a running sum. This would repeat, with each result being shifted left by another bit before being added to the next running sum. However, if one of the numbers was negative, one would have to add the two's complement of the positive number to the running sum. In order to complete this assignment, we had to implement a Moore State Machine with 19 states. These 19 states accounted for every possible iteration of add, shift, execute, hold, clear, reset, or any other functions that might occur within the multiplier. This helped us account for the many cases the multiplier was equipped to handle: positive times positive, positive times negative, negative times positive, negative times negative, and repeated/continuous multiplication. The final result was stored as a signed 16 bit 2's complement number using both of the 8 bit registers in combination.

Pre-lab Questions:

1. Q:Rework the multiplication example on page 5.2 of the lab manual, as in compute 00000111 * 11000101 in a table similar to the example

A:

Function	X	A	B	M	Comments for the next step
Clear A, Load B	0	00000000	00000111	1	Since M = 1, multiplicand (available from switches S) will be added to A.
ADD	1	11000101	00000111	1	First shift moving all X, A and B.
SHIFT	1	11100010	10000011	1	Second Add into A, so add S to A.
ADD	1	10100111	10000011	1	Second Shift.
SHIFT	1	11010011	11000001	1	Third add since M is = 1. S+A->A
ADD	1	10011000	11000001	1	Since M=0, shift without adding S+A.
SHIFT	1	11001100	01100000	0	Since M=0, shift without adding S+A.
SHIFT	1	11100110	00110000	0	Since M=0, shift without adding S+A.
SHIFT	1	11110011	00011000	0	Since M=0, shift without adding S+A.

SHIFT	1	11111001	10001100	0	Since M=0, shift without adding S+A.
SHIFT	1	11111100	11000110	0	Eight shift so computation cycle is over, no add even though M=1 as operation is done
SHIFT	1	11111110	01100011	1	8th shift done. Stop. 16 bit product in AB.

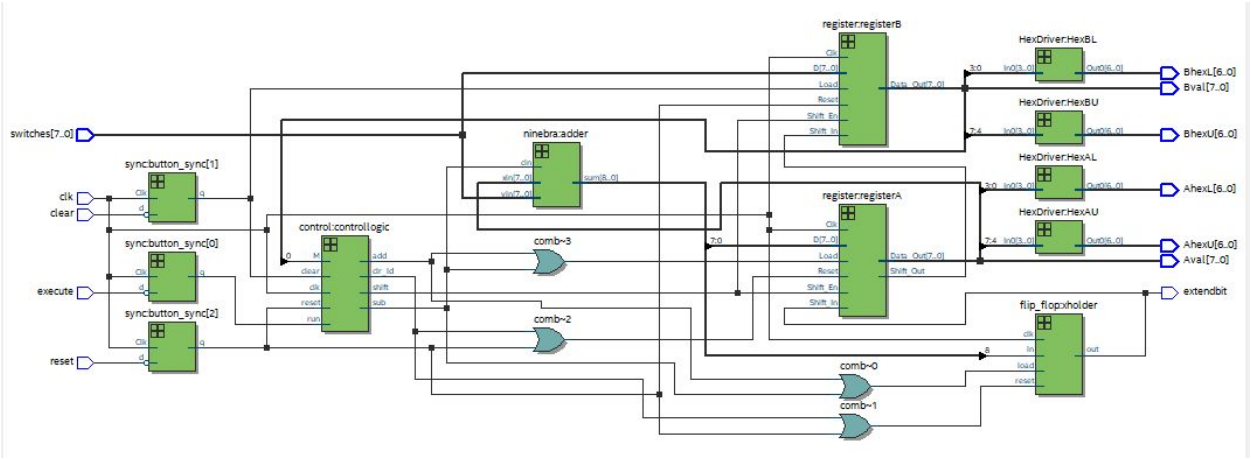
Written description and diagrams of multiplier circuit

Summary of operation:

The operands are first loaded into the switches using the onboard sliders. Then, the ClearA_LoadB button is pressed, which loads the data from switches into B (an 8 bit shift register). After one of the operands is loaded into B, the switches are then changed to another value to be multiplied by. Once the switches have been set, we hit execute, which does the multiplication using the method displayed in the table above using a FSM (in our case a Moore State Machine). The data from the switches is then added to another 8 bit shift register (register A in the diagram above). Using the Eight ADD states and Eight SHIFT, we make sure that we go through every route, and depending on if M=1 or M=0, we either add or do not add. A 9 bit carry ripple adder is used, as an extra bit is needed for the sign extend bit. Furthermore, since we knew that we were at most going to multiply two eight bit numbers, we only needed a 16 bit holder with a single sign bit.

The purpose of the X register is twofold: firstly, it stores the sign of the digits currently stored in the multiplicand, and secondly, it is utilized in order to determine whether to add two corresponding bits between the multiplier and multiplicand. Continuous multiplications present an issue when the result of the multiplication exceeds the capacity of the 16 bit output. In this case, when trying to multiply a number that is too big to be stored in the AB register, the data in B will not have all of the information of the whole number (seeing as the result is too big to be stored in just B), and will be lacking in bits. Lastly, the multiplication algorithm that we used has several advantages over the “pencil and paper method”. Firstly, it requires less extend bits to be stored at any given time, as the “pencil and paper method” has to store all of the extend bits, even if they happen to be a zero. Secondly, it takes less time than the “pencil and paper method”, since the “pencil and paper method” requires that each bit in the multiplicand be multiplied by each bit in the multiplier. However, the advantage that the “pencil and paper method” has over the algorithm used in this lab is that it is much simpler to implement, only requiring a few adders and not having to concern itself with any subtraction.

Top Level Block Diagram:



Written Description of .sv Modules:

Module: ninebra
Inputs: [7:0] xin, [7:0] yin, cin
Outputs: [8:0] sum
Description: This module creates a nine bit ripple carry adder.
Purpose: This module is used for all of the addition in the circuit, so it is indirectly responsible for the multiplication.

Module: fulladder
Inputs: x, y, cin
Outputs: s, cout
Description: This module creates a one bit ripple carry adder.
Purpose: This module is used to create the nine bit ripple carry adder.

Module: control
Inputs: run, clear, clk, reset, M
Outputs: clr_ld, shift, add, sub
Description: This module uses a moore state machine and
Purpose:

Module: testbench_8
Inputs: none
Outputs:none
Description: This module creates instances of ++, +-, -+, --, and repeated multiplication.
Purpose: This module is used to test the functionality of the circuit.

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

Outputs: Shift_Out, [7:0] Data Out
Description: This is an 8 bit shift register directly copied from Lab 4.
Purpose: This register is used to store A and B for the multiplication.

Module: sync
Inputs: Clk, d
Outputs: q
Description: This module creates a synchronizer with no reset.
Purpose: This module is useful for switches and buttons (typically asynchronous signals).

Module: sync_r0
Inputs: Clk, Reset, d
Outputs: q
Description: This module creates a synchronizer with a reset to 0.
Purpose: This module is useful for helping to bring asynchronous signals into the FPGA.

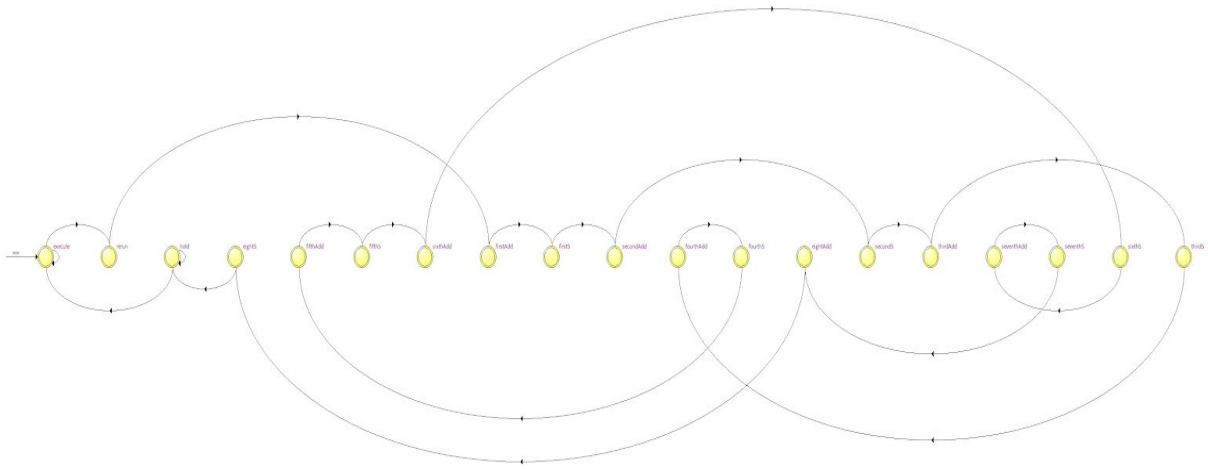
Module: sync_r1
Inputs: Clk, Reset, d
Outputs: q
Description: This module creates a synchronizer with a reset to 1.
Purpose: This module is useful for helping to bring asynchronous signals into the FPGA.

Module: HexDriver
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This module drives the LED display on the FPGA board.
Purpose: The purpose of this module is to ensure that we can use our FPGA board to test the functionality of our multiplier.

Module: Processor
Inputs: clk, reset, execute, clear, [7:0] switches
Outputs: extendbit, [7:0] Aval, [7:0] Bval, [6:0] AhexL, [6:0] AhexU, [6:0] BhexL, [6:0] BhexU
Description: This module is the top level module for the multiplier.
Purpose: This module instantiates all of the other modules in the multiplier.

Module: flip_flop
Inputs: clk, load, reset, in
Outputs: out
Description: This module creates a D Flip-Flop
Purpose: This module is used to store the extend bit.

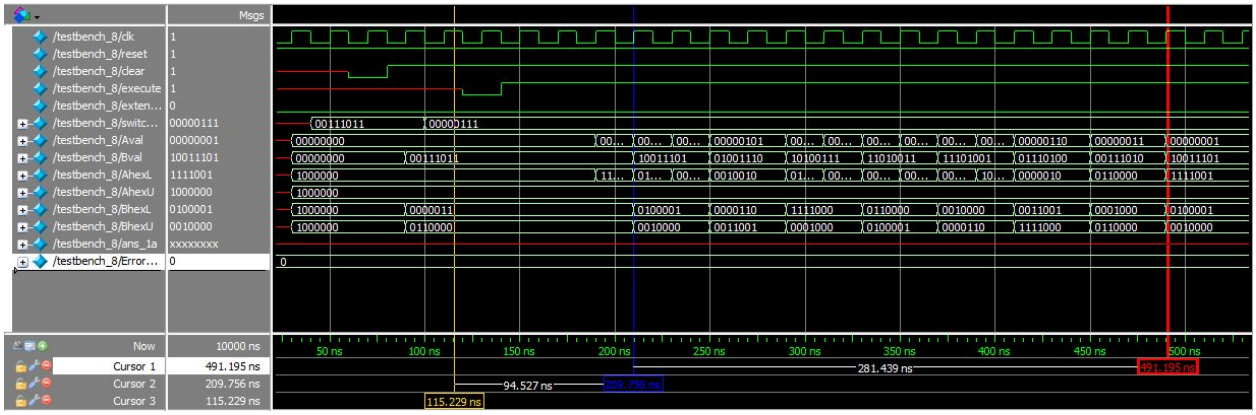
State Diagram for Control Unit:



Since this is a Moore State Machine, the transitions correspond to the clock.

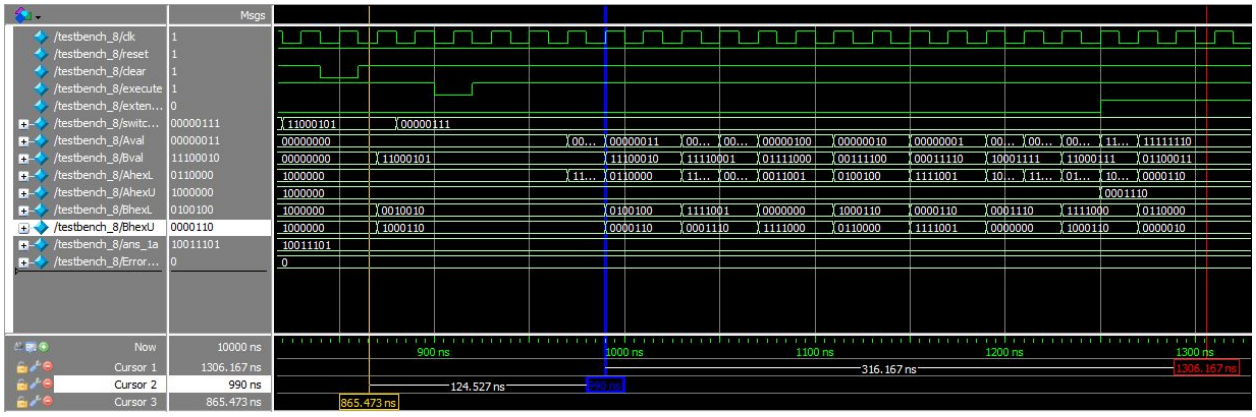
Annotated pre-lab simulation waveforms:

Positive Positive Multiplication:



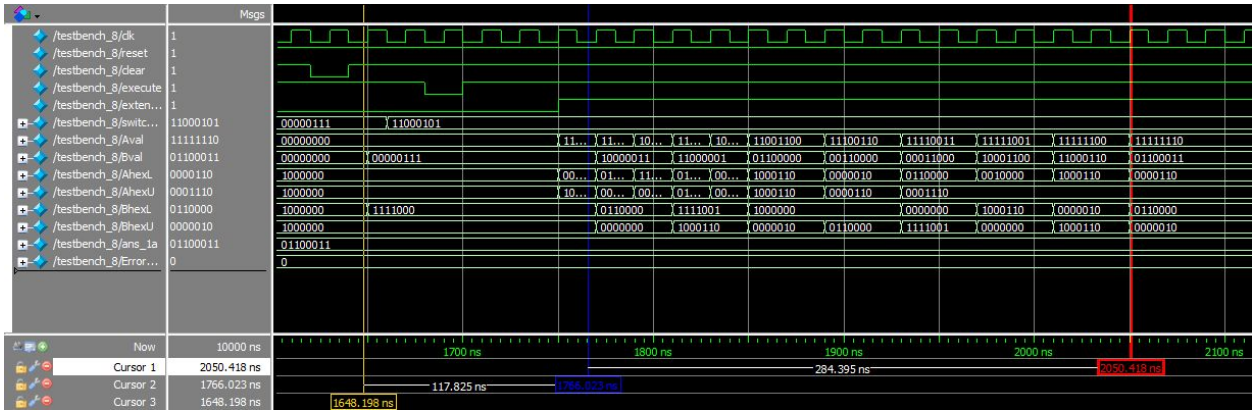
Line Color	Event Occurring
Yellow	Register B has been loaded with 59 and 7 is loaded into the switches.
Blue	The multiplication process begins.
Red	The multiplication operation is over, and the result is loaded into register B (BVal). The carry out bit is zero

Negative times Positive:



Line Color	Event Occurring
Yellow	X, A and B have been cleared, register B has been loaded with -59, and 7 is loaded into the switches.
Blue	The multiplication process begins.
Red	The multiplication operation is over, and the result is loaded into register B (BVal). The carry out bit is one.

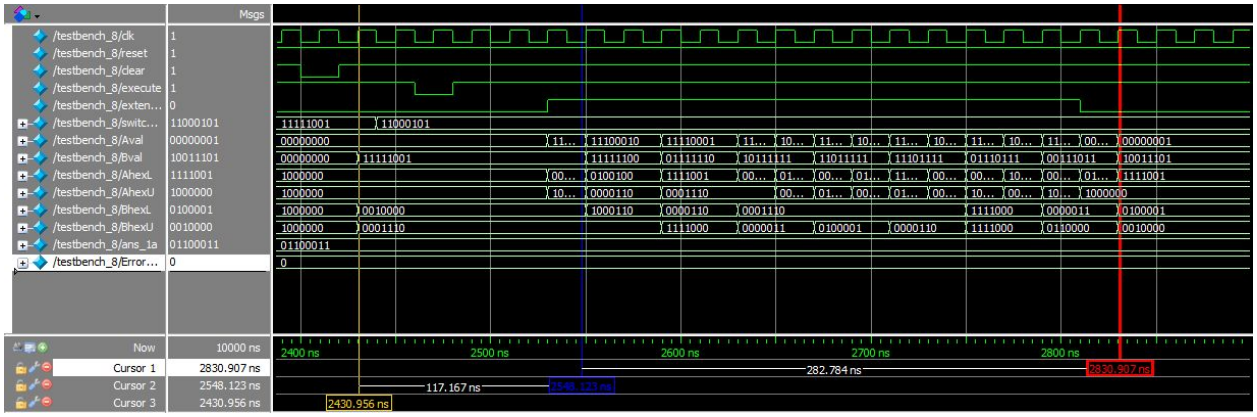
Positive times Negative:



Line Color	Event Occurring
Yellow	X, A and B have been cleared, register B has been loaded with 7, and -59 is loaded into the switches.

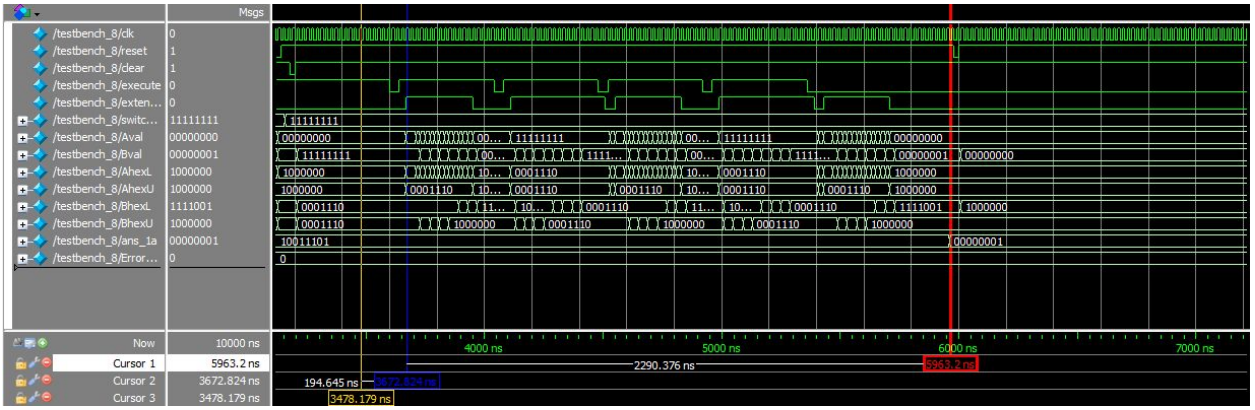
Blue	The multiplication process begins.
Red	The multiplication operation is over, and the result is loaded into register B (BVal). The carry out bit is one.

Negative times Negative:



Line Color	Event Occurring
Yellow	X, A and B have been cleared, register B has been loaded with -7, and -59 is loaded into the switches.
Blue	The multiplication process begins.
Red	The multiplication operation is over, and the result is loaded into register B (BVal). The carry out bit is zero.

Multiple Multiplications:



Line Color	Event Occurring
Yellow	X, A and B have been cleared, register B has been loaded with -7, and -59 is loaded into the switches.
Blue	The multiplication process begins.
Red	The multiplication operations are over, and the result is loaded into register B (BVal). The carry out bit is zero.

Answers to two post-lab questions:

1. Q: Fill in the table shown in 5.6 with your design's statistics

A:

LUT	78
DSP	0
BRAM	0
Flip-Flop	39
Frequency	65.6 MHz
Static Power	98.52 mW
Dynamic Power	2.69 mW
Total Power	149.36 mW

2. Q: Write down several ideas on how the maximum frequency of your design could be increased or the gate count could be decreased

A: Since our design involves using a 9 bit Ripple Carry Adder, we could probably use either the Carry Select or the Carry Lookahead Adder to decrease the time that it takes for additions to be completed, and in doing so, improve the maximum frequency. A Mealy State Machine could have been a more efficient design, as it would cut down on the number of states, but at the same time, this type of design would require a counter and would add complexity to the design. In addition, we could have found a simpler or more efficient multiplication algorithm to reduce the number of gates or improve the frequency respectively.

Conclusion:

Our design ended up working perfectly in the demo, but it took some time for us to get it to that point. The biggest error that we had came up when we were testing our continuous multiplication, specifically with negative numbers. We were testing simply raising a number to a power, so we were loading in -1 or -2 into register B, and then keeping the switches at that same number. We found that when we performed the third consecutive multiplication, our result would end up being one less than it should have been. For example, with -1, our calculation of (-1)^4 produced 0, and (-2)^4 produced 15. We were perplexed for a while, but then realized that our Moore State Machine required another state right after execute, and this state had to clear A for the next multiplication. We named this state re-run.

A smaller error that we encountered later concerned our testbench. We had failed to precede one of our hexadecimal numbers with the letter ‘h’, and so this gave a compiler error several times before we spotted it. The only issue that we had with the given lab instructions was that the post lab question #1 directed us to the wrong page in the IQT document, so we had to ask a TA where to find this information.

All in all, this lab was a good intermediate level lab for system verilog. We weren’t given as much to begin with, and we had to come up with our own state machine. We learned even more about the inner workings and syntax of system verilog, and the things that we learned this week will certainly be relevant in labs to come.