ECE385
Spring 2019
Experiment #5

An 8-bit Multiplier in System Verilog

Austin Lee (sal3)
Zuling Chen (zulingc2)
Tuesday 3PM ABE
Patrick Wang, Gene Shiue

Introduction

In this experiment we expanded on our knowledge of System Verilog. The objective of the lab was to design an 8-bit multiplier in System Verilog which was then programmed onto a DE2 FPGA board. The multiplier uses eight cycles of two's complement adding and right shifting to complete the operation.

8-bit multiplication example

For the prelab, instead of setting B to -59, we are now setting B to 7 and A to -59. The result in the end is still the same however.

Function	X	A	B	M	Comments for Next Step
CLEAR A, LoadB	0	00000000	00000111	1	Since M=1, switch data is added to A
ADD	1	11000101	00000111	1	SHIFT AFTER ADD
SHIFT	1	11100010	10000011	1	M = 1, switch data is added to A
ADD	1	10100111	10000011	1	SHIFT AFTER ADD
SHIFT	1	11010011	11000001	1	M= 1, switch data is added to A
ADD	1	10011000	11000001	1	SHIFT AFTER ADD
SHIFT	1	11001100	01100000	0	DON'T ADD, M = 0, JUST SHIFT
SHIFT	1	11100110	00110000	0	DON'T ADD, M = 0, JUST SHIFT
SHIFT	1	11110011	00011000	0	DON'T ADD, M = 0, JUST SHIFT
SHIFT	1	11111001	10001100	0	DON'T ADD, M = 0, JUST SHIFT
SHIFT	1	11111100	11000110	0	DON'T ADD, M = 0, JUST SHIFT (7^{th} shift, but M =0, no SUB needed)
SHIFT	1	11111110	01100011	1	8^{th} shift is complete. 16 bit product is now in AB.

Written description(s)

To operate the multiplier, the user has to initially input the 8-bit value using switches. The value is displayed in hexadecimal on the two right LED displays. After inputting the first value, the user presses CLRA_LDB (KEY2) and the data indicated by the switches is stored into register B. At the same time, whatever information was present in registers A and X are cleared, returning both to hold the value zero. The switches could then be changed to indicate the second value to be multiplied and on pressing RUN (KEY3) the algorithm would be applied to the data

stored in register B and to the data currently on the switches. The result would be displayed on all four LED displays in hexadecimal, with an LED indicating register X.

At any time, pressing the RESET button (KEY0) would erase all stored data in all registers. Between consecutive multiplications, registers A and X would be cleared automatically even without the use of RESET or CLRA_LDB.

Module(s)

Like with experiment #4, the adding operation was done with a full-adder. The full-adder module takes two values and a carry-in as inputs and outputs a sum and carry-out.

The module f_add uses the full-adder to perform arithmetic on two 9-bit registers. f_add takes two 8-bit inputs, two logic values indicating whether addition or subtraction is to be performed, and outputs a 9-bit sum/difference. The first step in either addition or subtraction is to sign-extend the two 8-bit inputs into 9-bit values using another module called sext. That helper module simply takes in a signed 8-bit input and outputs a signed 9-bit value. In f_add, the sign-extended values are first summed into a 9-bit intermediate called outadd. Then, s (the sign-extended data in the switches) is inverted and stored in stemp. Bit value 1 is added to stemp in order to create the negative two's complement value of s. This value is summed with the other input (register A) to perform a subtraction-- the result is stored in outsub. If input logic on is 1, then the module's output is outadd. If input logic onsub is 1, then the modules output is outsub. (The module f_sub was designed initially but is never used in the actual implementation as it's function was merged into module f_add).

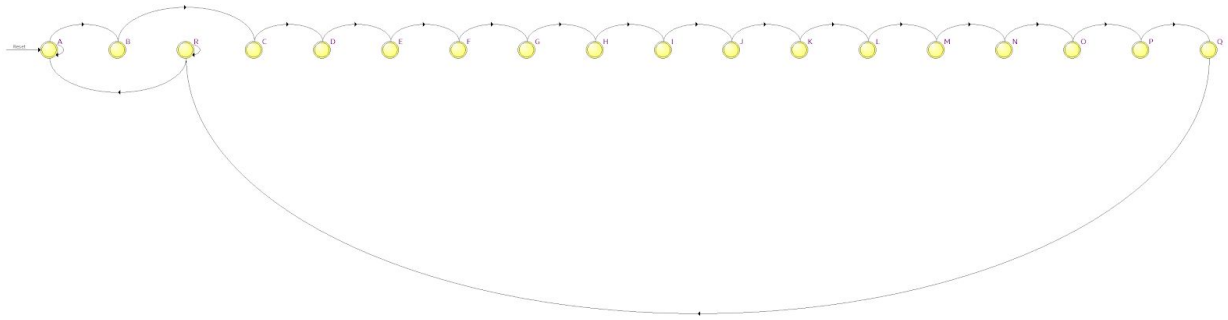
The module control instructs the algorithm sequence to the multiplier. It takes in inputs Clk, Reset, LoadB (and clear A), Run--all from the FPGA board-- and the least significant bit of register B. It outputs logic values Shift_En, LdB, Add, Sub, and resetA. The function of module control is that it dictates the state diagram through which the multiplier runs. There's a total of 19 states including eight add states, eight shift states, a starting state (A), ending state (X), and waiting state (R). In the first seven add states, if the least significant bit input is a 1, then the output of signal add is 1. Otherwise, all output signals are 0. On the eighth state, the least significant bit indicates whether a subtraction will occur (sub = 1) and if not all signals are again 0. In each shift state, the Shift_En output is 1 while all other signals remain 0. The start state, where the multiplier stays when not operating, outputs 1 for resetA and LdB is equal to LoadB. This way, register A and X are cleared even if the LoadB button is not pressed. Once RUN is pressed on the board, the start state moves to the next state and through the add/shift process until it reaches the waiting state. In the waiting state, all signals are low except LdB which depends on the button. This state was included in order to preserve the values of register A and X

until the RUN button was released, at which it enters end state X where all signals are 0 and then returns to the start state.

The top-level module mult is the datapath in which all instances of other component modules are instantiated and inputs/outputs are connected. It takes all the inputs from the board and outputs the 8-bit values of A and B as well as output signals to the LED displays. Three registers A, B and X are created with the former two being 8-bit registers and the last one being 1-bit. The f_add module is also used to create an addition/subtraction function and the control module is called to provide all the correct signal inputs to the other components.

The other modules (synchronizers, Hexdrivers, reg_8) are replicas of given/edited files from the previous experiments.

State Diagram



Schematic Block Diagram

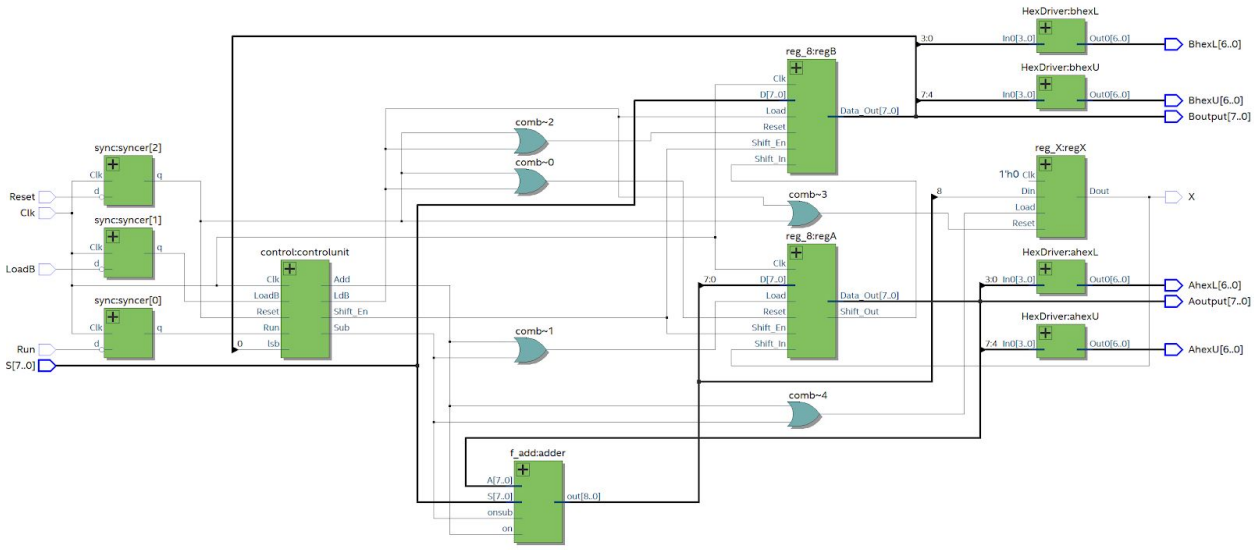


Figure 1: Top-level (mult) block diagram

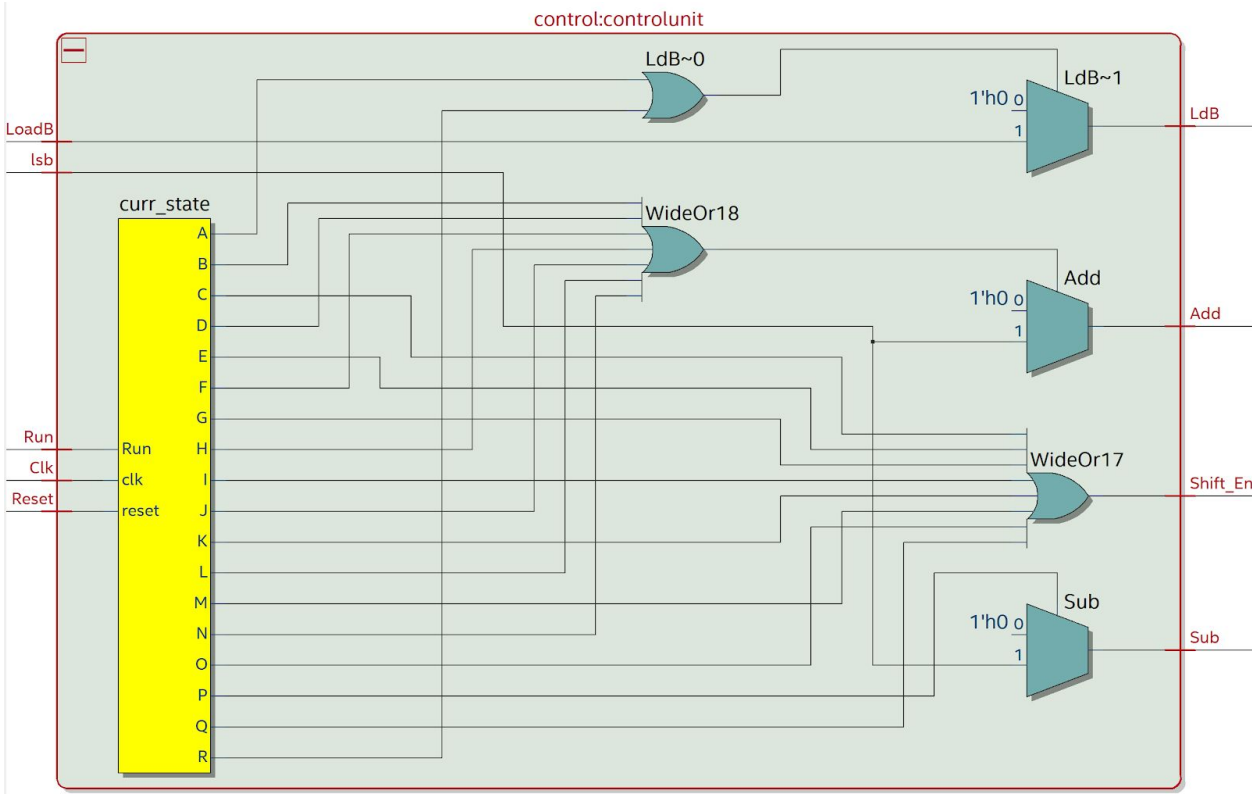


Figure 2: Control module

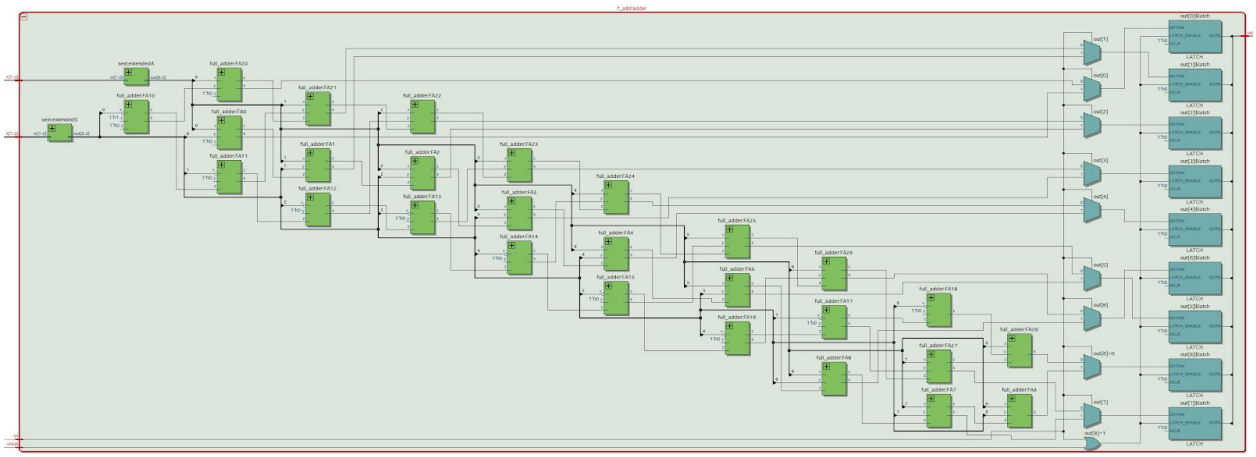
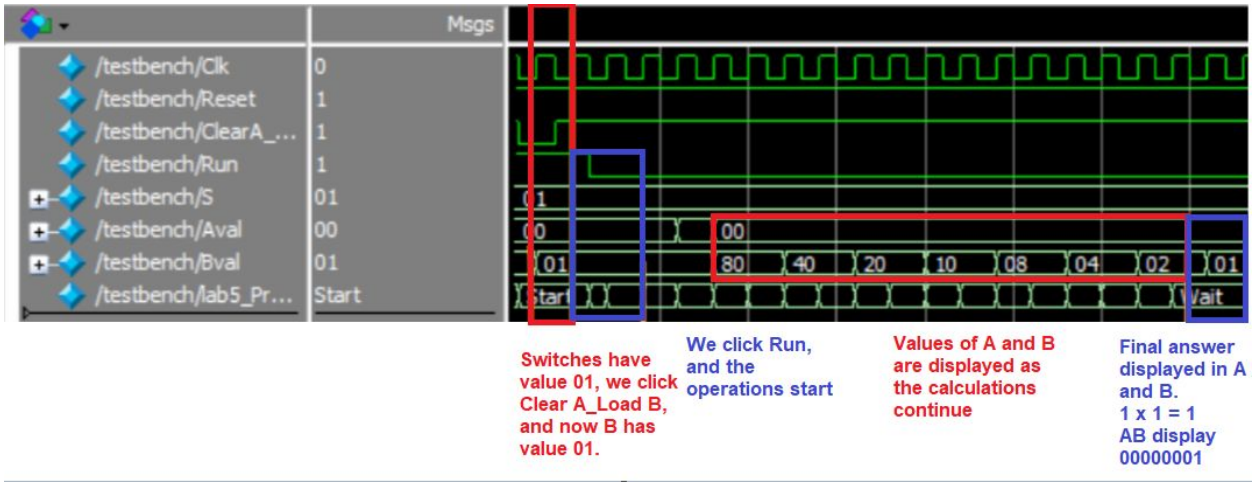


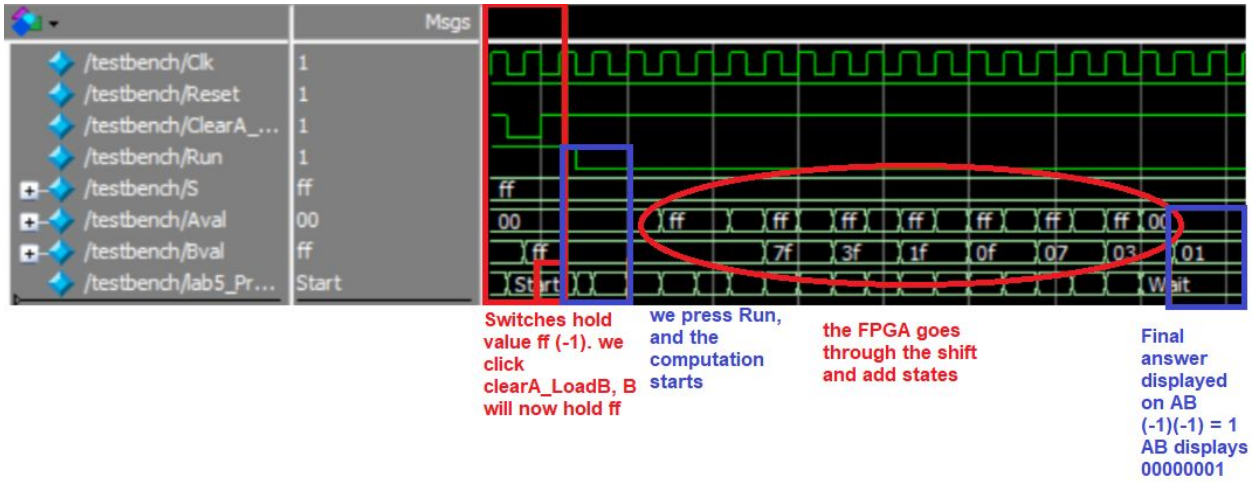
Figure 3: f_add module using full adders

Simulation waveform(s)

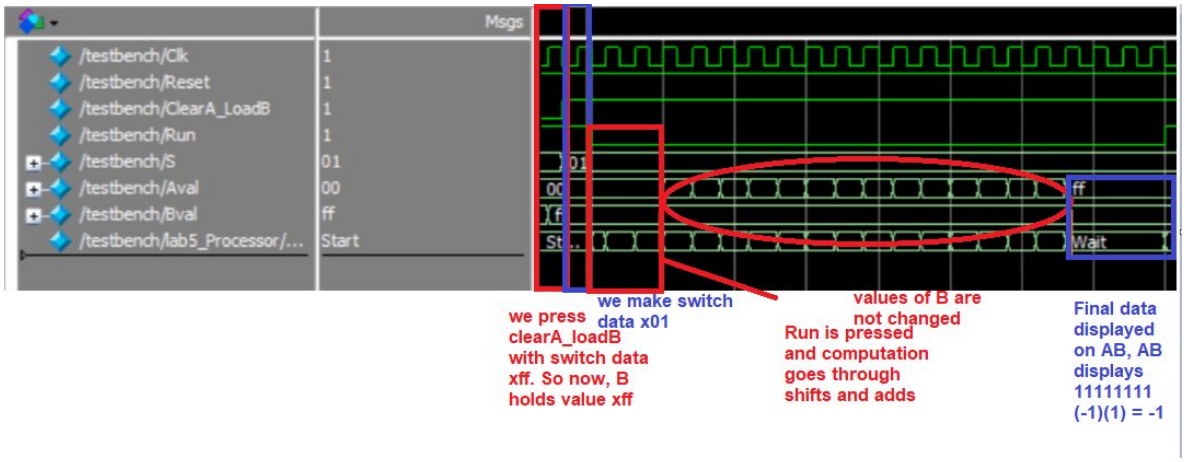
(+)(+), Testing 1x1



(-)(-), Testing -1 * -1



(-)(+), Testing -1 * +1



(+)(-) Testing +1 * -1

Post-lab

LUT	108
DSP	0
BRAM	0
Flip-Flop	55
Frequency	71.76 MHz
Static Power	98.52 mW
Dynamic Power	5.15 mW
Total Power	103.67 mW

For our design, we made use of over 16 states to implement and count our eight required shifts. We could've boiled this down to essentially 2 states, a shift and add state which would have limited the amount of hardware implementation needed. For our implementation, we essentially visited each shift/add state and then depending on the M-bit, either skipped that step or executed. If we were to reduce this down to just 2 states, it would allow us to also increase the frequency of our design as it would no longer have to go through any unneeded states.

The purpose of the X register in our design is to hold the 9th bit extended from the addition of A and S. We use that data from the X register to shift into A whenever we shift XAB. The X register should also be cleared whenever we clear A and load B.
This algorithm works well and shouldn't really result in any issues. The only limitation that this design has is whenever the numbers/answers can't be represented with a total of 8 bits. For longer numbers, this design doesn't work. We can continuously multiply until we reach a number that is too big to be represented by the number of bits we have. For pencil and paper, you have to do different operations for different combinations of numbers (-,+), (-,-), (+,+), (+,-). With our general algorithm we implemented we can just run through it regardless of the case, but with paper and pencil, for special cases, you don't have to worry about sign extensions (+,+) and (+,-). In hardware implementation through SystemVerilog, all test cases are run the same way, so even when we don't need the extra steps, they're done regardless resulting in slower computation.

Conclusion

Our design and implementation worked in the end for smaller numbers. We were able to demonstrate the ability to consecutively multiply (-1) by itself and get the correct answer. We were also able to demonstrate multiplication with smaller numbers, but our code started breaking after the numbers got too large. During demo, we were unable to get the test bench working properly as well. After looking into our code, we saw that our logic to select the output from the adder was being compiled as latches, so none of the data signals that were sent in were showing up on the simulation. We also had the issue of using one select signal for both Loading A and in using the full adder. At the same time as we were trying to use A's value to be put into the full adder unit, we were also trying to input the full adder unit's data into A. This could've been solved with adding a few addition flip flops to delay the data from the adder into A.