ECE 385

Fall 2016

Experiment 5

An 8-Bit Multiplier in SystemVerilog

Jacob Brown, Michael Olson
ABD Tuesday 8am
Po-Han Huang, Xinyang Wang

Introduction

In this experiment, we designed a multiplier for two 8-bit 2's compliment numbers, then ran that multiplier on the DE2 FPGA board. We used an add-shift algorithm, similar to the pencil-and-paper method of multiplication, except the final step for 2's complement numbers depends on the sign bit. Consider the following example to calculate 8-bit 00000111 (7) x 11000101 (-59):

```
   00000111                7 (multiplicand)
 x 11000101          x (-)59 (multiplier)
 --------            --------
   00000111            (-)413
  +00000000x
 +00000111xx
+00000000xxx
+00000000xxxx
+00000000xxxxx
+00000111xxxxxx
-00000111xxxxxxx  Subtract (or Add 2's comp of 00000111)
1111111001100011 (2's comp of result=0000000110011101=413)
```

Figure 1: Multiplication Example

The individual steps of the add-shift algorithm can be broken down as seen in Table 1 of the PreLab. The contents of Register B and 16 input-switches are multiplied, leaving the result in registers AB. A ClearA_LoadB signal is used to prepare register A for a new computation, and load the data (multiplicand) from the switches into register B (in this case, 11000101).

When the least significant bit M of register B is equal to 0, the ADD function can be skipped. In addition, since we are using a 2's complement representation, we need to consider negative numbers. If A is negative, then XA will contain the correct partial sum and the sign will be preserved since the shift operation will perform an arithmetic shift on XAB. If B is negative (the most significant bit = 1), then M will be 1 after the seventh shift (see the example above). In that case a subtract operation is performed since the 8th bit of B has negative weight with 2's complement representation.

Our design will execute only one multiply operation when the Run signal is switched from low to high. The 9-bit adder/subtractor was designed using Full Adder primitives, rather than SystemVerilog's arithmetic operators, in order to get a deeper understanding of the underlying hardware.

Pre-Lab

The example mentioned in the introduction can be reworked with it's individual steps tabulated below. This is the computation of 00000111 * 11000101:

Function	X	A	B	M	Comments
ClearA_LoadB	0	0000 0000	1100 0101	1	Clears the first register and loads B with the 2nd value to be multiplied (the 1st value comes from switches). Since M =1, the next op is ADD
ADD	0	0000 0111	1100 0101	1	Shift the X-A-B combined result after adding.
SHIFT	0	0000 0011	1 110 0010	0	Since M!=1, don't add, just shift again.
SHIFT	0	0000 0001	11 11 0001	1	Since M =1, the next op will be ADD
ADD	0	0000 1000	11 110001	1	Shift XAB by one bit after ADD complete
SHIFT	0	0000 0100	011 11000	0	Since M!=1, don't add, just shift again.
SHIFT	0	0000 0010	0011 1100	0	Since M!=1, don't add, just shift again.
SHIFT	0	0000 0001	00011 110	0	Since M!=1, don't add, just shift again.
SHIFT	0	0000 0000	100011 11	1	Since M =1, the next op will be ADD
ADD	0	0000 0111	100011 11	1	Shift XAB by one bit after ADD complete
SHIFT	0	0000 0011	1100011 1	1	Since M =1 and this is the final step which depends on the sign bit (8th bit = M), the next op will be SUBTRACT.
SUBTRACT	1	1111 1100	1100011 1	1	Shift XABby one bit after SUB complete
SHIFT	1	1111 1110	01100011	1	8th shift done. Stop. 16-bit Product in AB.

Table 1: Algorithm Example Worked Out

Waveforms

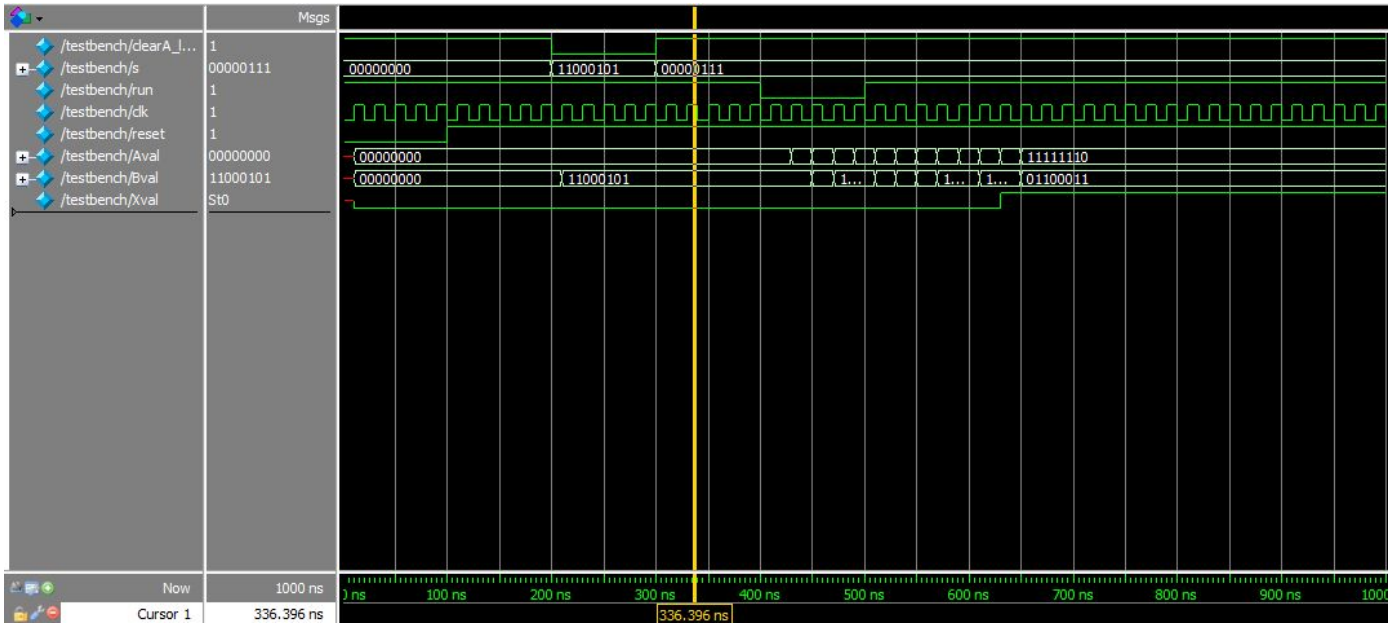


Figure 2. Waveforms of 7 (00000111 in s) times -59 (11000101 in Bval) resulting in -392 (11111110 01111000 in Aval, Bval)

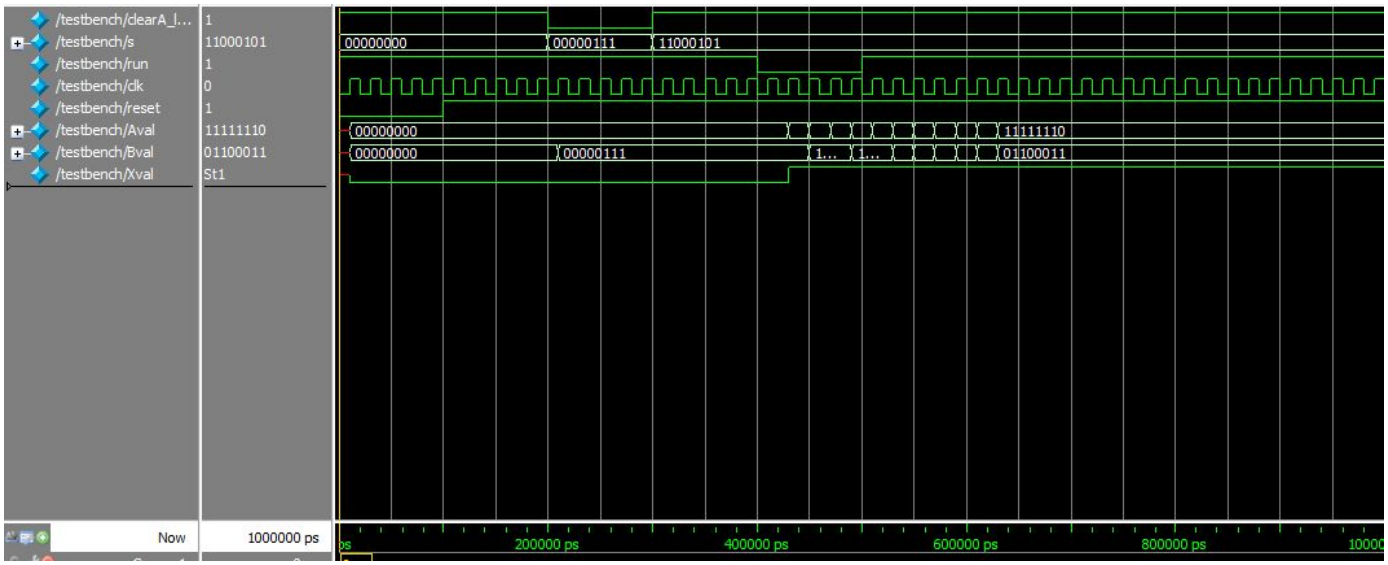


Figure 2. Waveforms of -59 (11000101 in s) times 7 (00000111 in Bval) resulting in -392 (11111110 01111000 in Aval, Bval)

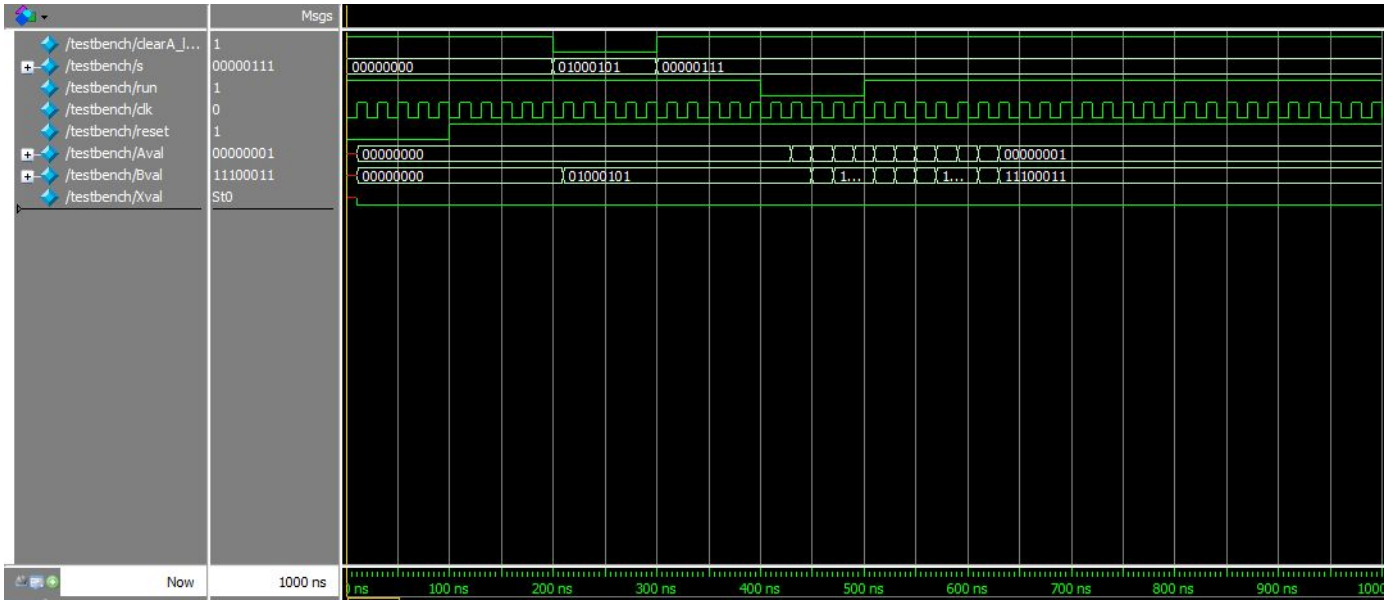


Figure 4. Waveforms of 7 (00000111 in s) times 69 (01000101 in Bval) resulting in 483 (00000001 11100011 in Aval, Bval)

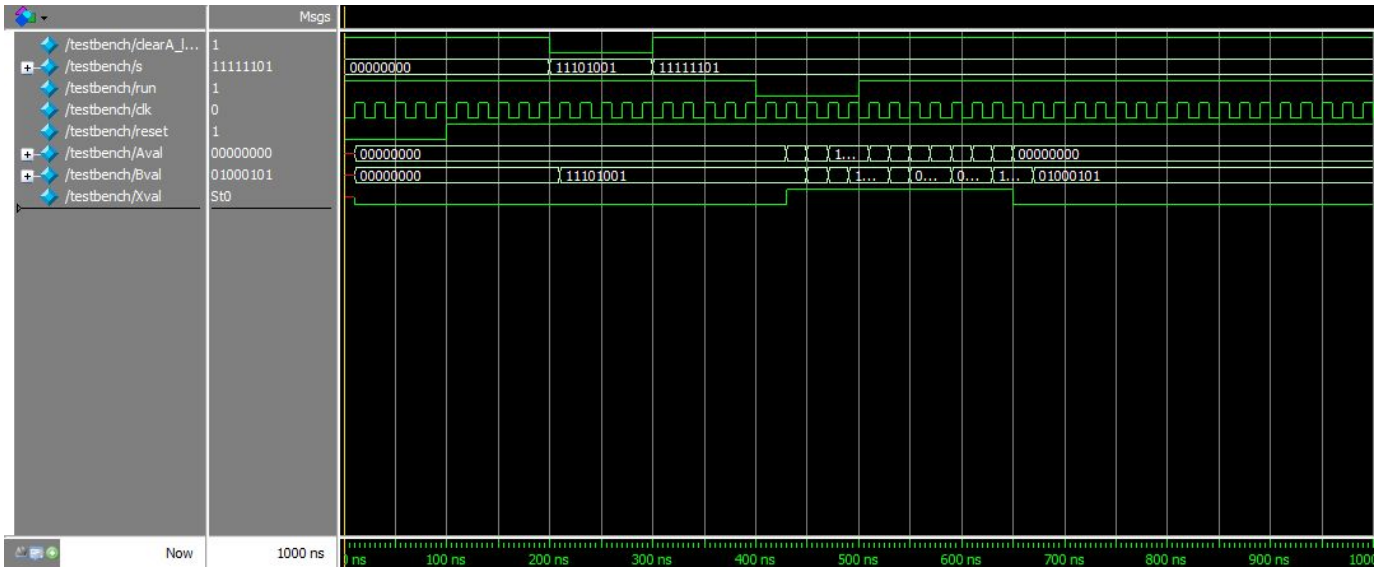


Figure 5. Waveforms of -3 (11111101 in s) times -23 (11101001 in Bval) resulting in 69 (00000000 01000101 in Aval, Bval)

Circuit Description

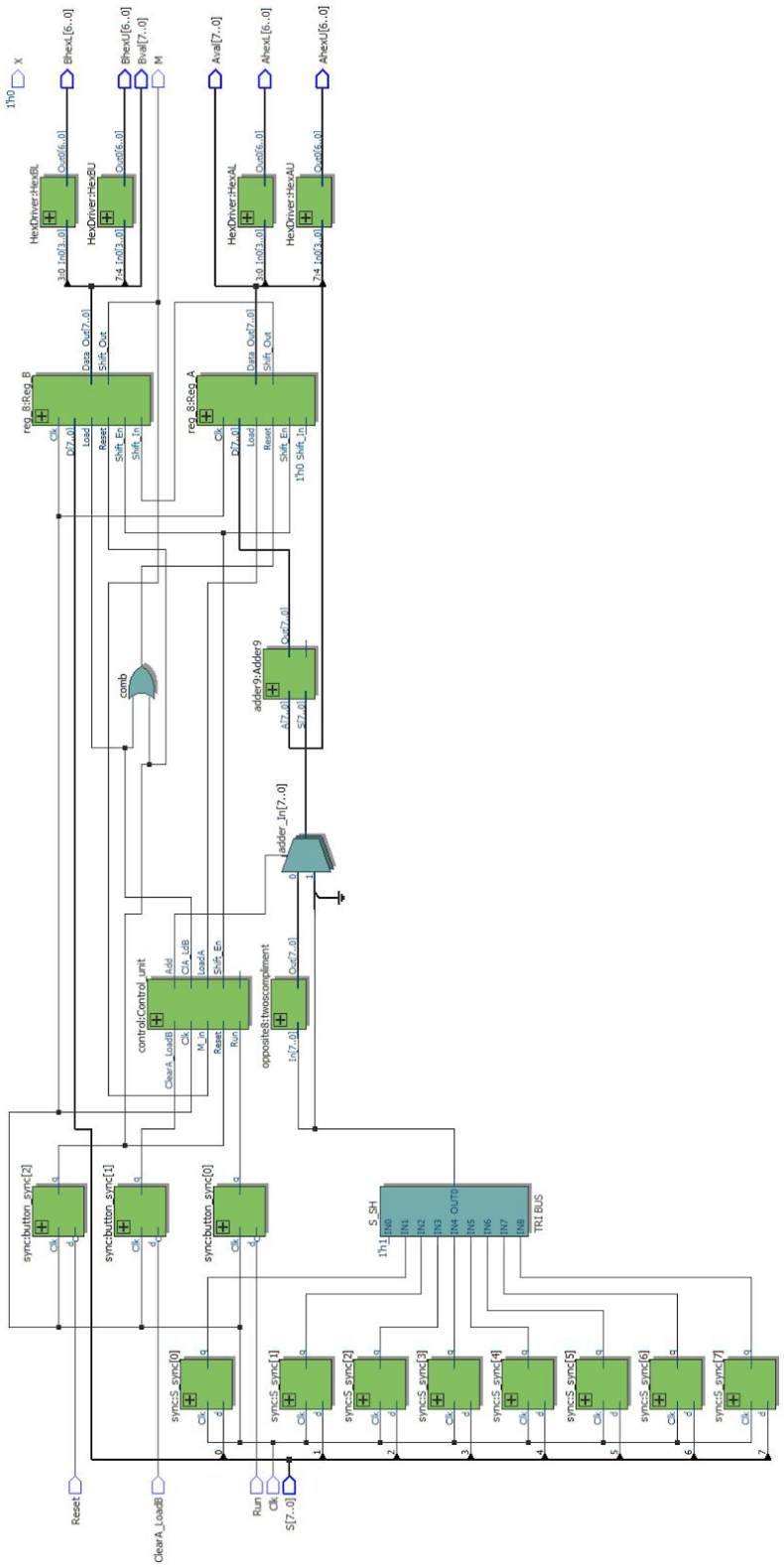
Summary of Operation

When ClearA_LoadB is pressed, the control logic sends a reset signal to shift-register A, while simultaneously sending a parallel load signal to B. The parallel inputs of B are connected directly to the switches, so whatever value the switches are set to at the time the button is pressed, it will be loaded into register B. The Run button begins the computation, proceeding the state machine from a halt-state to the first of 17 more states, alternating between potentially adding/subtracting and shifting, which just shifts the most-significant bit of A into register A, the least significant bit of A into register B, and the least significant bit of B into the M output, which is used to determine if the add or subtract should happen or not.

The 9-bit adder is made of an 8-bit adder and some combinational logic that takes in the value from the switches and register A, storing the result in register A. At the end of our add-shift algorithm, the result is stored across XAB - 17 bits representing a 16-bit multiplication result and a sign-bit.

In order to subtract, we made a module called “opposite” that will take in an 8-bit number and return it’s 2’s complement value. If the 15th state (corresponding to 7 previous shifts and 7 previous potential arithmetic operations) requires a subtract, it will send the opposite module result of the value of switches to the adder instead of the switches value itself.

Top Level Block Diagram



Modules

Module: adder9.sv
Inputs: [7:0] A, [7:0] S
Outputs: [7:0] Out, X
Description: Special purpose 9-bit adder for this lab. Takes in data from register A and the switches, sums them, and returns the value back to register A as well as sets X.

Module: adder8.sv
Inputs: [7:0] A, [7:0] B, c_in
Outputs: [7:0] S, c_out
Description: Uses two 4-bit full adders to add 4-bit halves of inputs A and B, producing an 8 bit sum and a carry-out bit.

Module: adder4.sv
Inputs: [3:0] A, [3:0] B, c_in
Outputs: [3:0] S, c_out
Description: Uses four single bit full adders to add each pair of corresponding bits of inputs A and B, producing a 4 bit sum and a carry-out bit.

Module: full_adder.sv
Inputs: A, B, c_in
Outputs: S, c_out
Description: Uses logic to determine the sum of A and B, outputting to S with carry-out bit c_out.

Module: opposite.sv
Inputs: [7:0] In
Outputs: [7:0] Out
Description: Uses the 8-bit adder to add NOT(In) with all zeros and a carry-in bit of 1, thus outputting the 2's complement opposite of In.

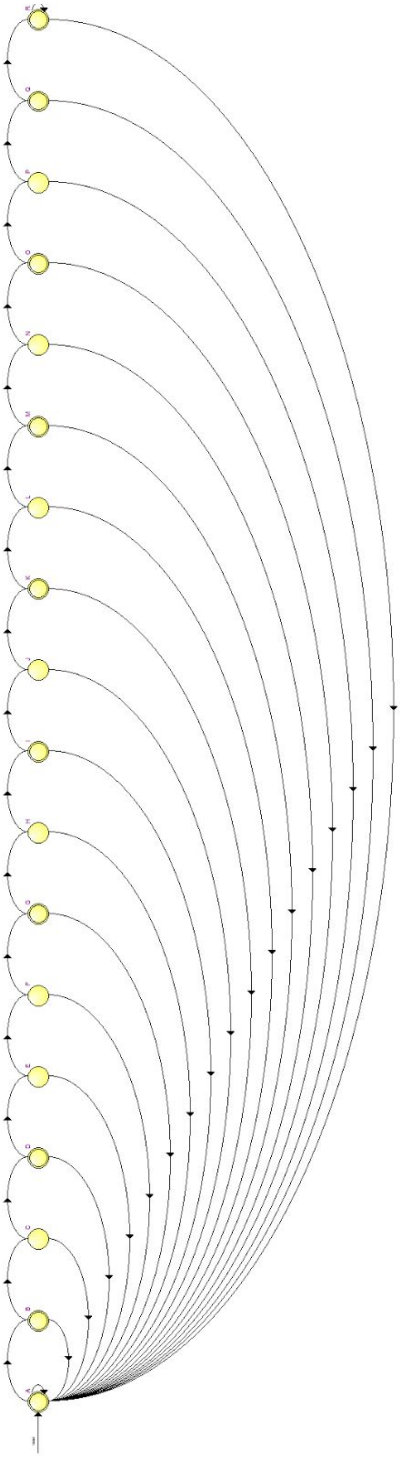
Module: Reg_8.sv
Inputs: Clk, Reset, Shift_In, Load, Shift_En, [7:0] D
Outputs: Shift_out, [7:0] Data_out
Description: Uses an always_ff block to update on the positive edge of Clk. Reset will load all zeros into Data_out, Load will store the data in (D), Shift_En will shift all of the bits over one with Shift_On filling the empty bit. Shift_out is always assigned the least significant bit of Data_out.

Module: Register_unit.sv
Inputs: Clk, Reset, A_In, B_In, Ld_A, Ld_B, Shift_En, [7:0] D
Outputs: A_out, B_out, [7:0] A, [7:0] B
Description: Holds two Reg_8 instances (one for register A, the other for register B). The two registers are controlled via their respective inputs, and either will load from D if enabled.

Module: Control.sv
Inputs: Clk, Reset, ClearA_LoadB, Run, M_in
Outputs: Shift_En, ClA_LdB, LoadA, Add, Subtr
Description: Controls state outputs and state transitions.

Module: NewMultiplierTopLevel.sv
Inputs: Clk, Reset, ClearA_LoadB, Run, [7:0] S
Outputs: [7:0] Aval, Bval, [6:0] AhexL, [6:0] AhexU, [6:0] BhexL, [6:0] BhexU
Description: Contains all modules and connects them appropriately. Syncs buttons as well as sets proper hex display values.

State Diagram



States A through R are shown here, where every other state (A, C, E, G, I, K, M, O, and Q) are potential ADD/SUBTRACT operations, and the others are shift states. At any time, RESET can begin the states over again. Reflecting upon this design, we realised it could have been done more efficiently, using only 5 states. The only necessarily unique states would be start, end, shift, add, and subtract. In this implementation, we would've used a counter to keep track of what bit is being operated on, helping determine if the ADD or SUBTRACT state would be the next state. Start and End would be distinctly different in that the output after ClearA_LoadB isn't the same as after a computation, and can be pressed any number of times before pressing Run.

Post-Lab

LUT	89
DSP	0
Memory (BRAM)	0
Flip-Flop	37
Frequency (MHz)	69.25
Static Power (mW)	98.54
Dynamic Power (mW)	1.88
Total Power (mW)	156.35

Q) Come up with a few ideas on how you might optimize your design to decrease the total gate count and/or to increase maximum frequency by changing your code for the design.

A) As apparent from the above state diagram, our design is much too cumbersome compared to what could have handled the same task. After further consideration post-lab, we believe just five states would have sufficed as opposed to eighteen. Such changes could improve the speed of the design.

Concerning total gate count, we've kept our use of unnecessary units low, and don't believe there is much room for improvement. Using fewer states should not have reduced this number.

Conclusion

In the end, our design didn't resolve the proper result, and the complexity of our state diagram made it difficult to debug. The XAB result appeared to be random data, changing even after resetting the values and running the same calculation. We were unable to discern the issue in time, so the cause of the problem isn't known, but the Reset and ClearA_LoadB functionality worked as expected. We would have to carefully trace the block diagrams to find the source of the anomaly.

As far as the given lab materials, we feel the X and M values were a little ambiguous. They could have been either registers to store these values, or simply wires to propagate a signal. There were also two Reset signals going into the control logic in the given incomplete block diagram, which was a little confusing. The add-shift algorithm was also never explicitly stated,

we had to infer based on what was shown in the example. The step-by-step breakdown of the example made this easier, but something along the lines of, “If M=1, add or subtract, then shift. If M=0, just shift. Do this until every bit of B has been shifted out”, would have been very helpful. This statement is, in essence, the add-shift algorithm.

The lab overall was a good introduction to more complex circuits in SystemVerilog, we simply didn’t complete a working circuit on time. It was a learning experience particularly concerning looped functions (looped addition = multiplication).