**ECE 385**

Spring 2023

Experiment #4

**8-Bit Multiplier In SystemVerilog**

Noah DuVal, Jason Wright

JZ / Friday 3:00 pm

Jason Zhu

**Introduction**

The goal of this experiment was to create an 8-bit multiplier using SystemVerilog. It was accomplished by implementing a simple add-shift algorithm which mimics the way one would do written multiplication. However, it varies as this multiplier deals with two’s complement representation thus there is an additional subtraction step to account for this. The multiplier is able to compute through multiple iterations of both positive and negative inputs.

**Pre-Lab Question**

In the table below is a representation of how the multiplier works in terms of adding and shifting. The computation is 11000101 \* 00000111 (-59 \* 7 = -413) with the final result shown in registers AB.

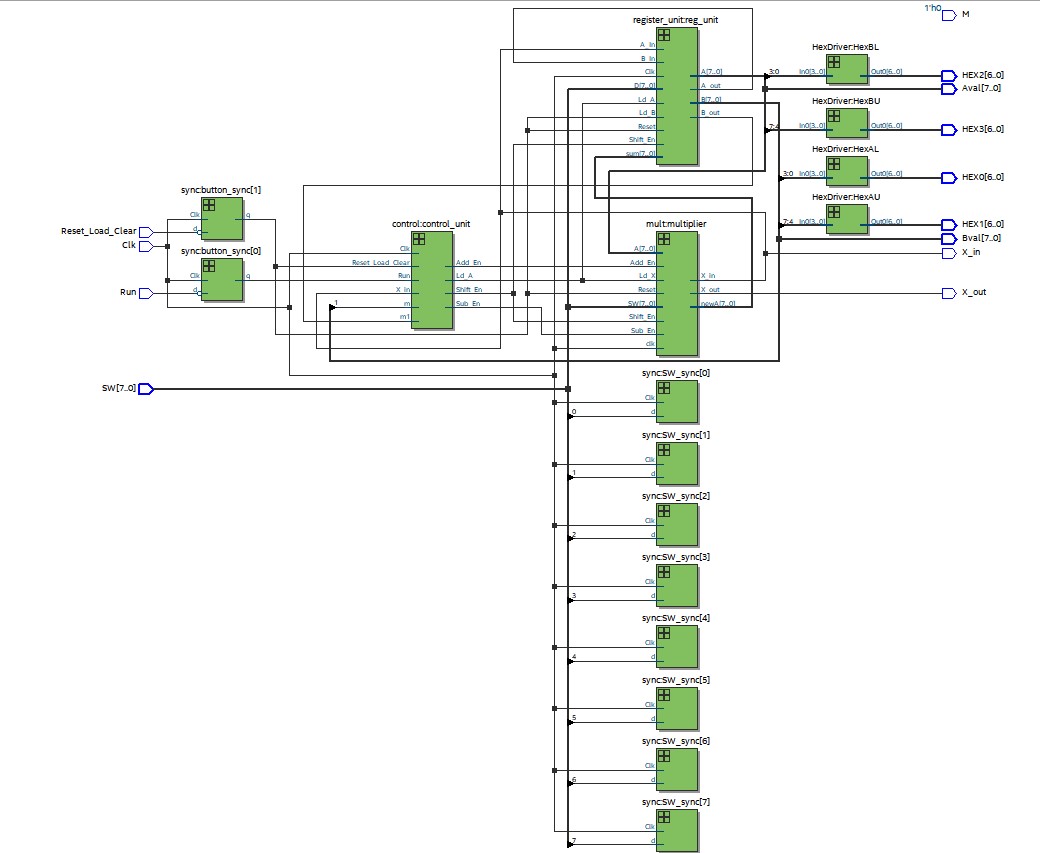
| **Function** | **X** | **A** | **B** | **M** | **Comments For Next Step** |
| --- | --- | --- | --- | --- | --- |
| Clear A, Load B, Reset | 0 | 0000 0000 | 00000111 | 1 | Since M = 1, Multiplicand will be added to A |
| ADD | 1 | 1100 0101 | 00000111 | 1 | Shift XAB by one bit after ADD Complete |
| SHIFT | 1 | 1110 0010 | 1 0000011 | 1 | Add S to A since M = 1 |
| ADD | 1 | 1010 0111 | 1 0000011 | 1 | Shift XAB by one bit after ADD Complete |
| SHIFT | 1 | 1101 0011 | 11 000001 | 1 | Add S to A since M = 1 |
| ADD | 1 | 1001 1000 | 11 000001 | 1 | Shift XAB by one bit after ADD Complete |
| SHIFT | 1 | 1100 1100 | 011 00000 | 0 | Do not add S to A since M = 0. Shift XAB |
| SHIFT | 1 | 1110 0110 | 0011 0000 | 0 | Do not add S to A since M = 0. Shift XAB |
| SHIFT | 1 | 1111 0011 | 00011 000 | 0 | Do not add S to A since M = 0. Shift XAB |
| SHIFT | 1 | 1111 1001 | 100011 00 | 0 | Do not add S to A since M = 0. Shift XAB |
| SHIFT | 1 | 1111 1100 | 1100011 0 | 0 | Do not add S to A since M = 0. Shift XAB |
| SHIFT | 1 | 1111 1110 | 01100011 | 1 | 8th shift done. Stop. 16-bit product in AB. |

**Table 1: Multiplier Steps**

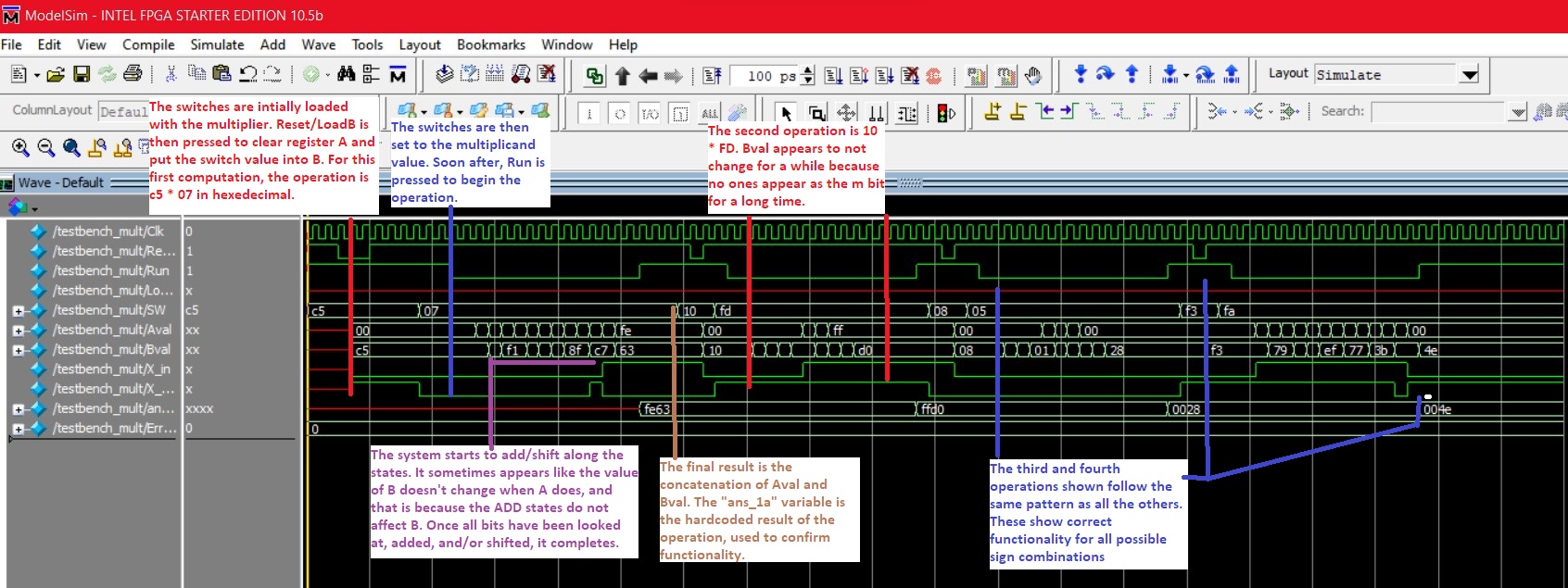
**Summary of Operation**

To initiate the system the multiplier is loaded into Register B by using the on board switches (S) and then pressing the Reset\_Load\_Clear button. This will clear registers X and A while loading the multiplier into register B. The switches are then changed to represent the multiplicand and the Run button is pressed to initiate the system. The system will then look at the least significant bit in register B (M) and will decide what to do. If M is 1 then S will be added to A with X holding the sign bit followed by a shift of XAB. If M is 0 then the system will not add and shift XAB. The system will perform eight arithmetic right shifts then display the result on the hex displays. However, if the eighth bit in register B is 1 then it will subtract S from A to account for two’s complement representation. Finally, the user has the option to either clear the registers and start over with new numbers or press run again to perform continuous multiplications. If run is pressed without a reset, the first add state is skipped in order to accommodate no longer needed to initially load A.

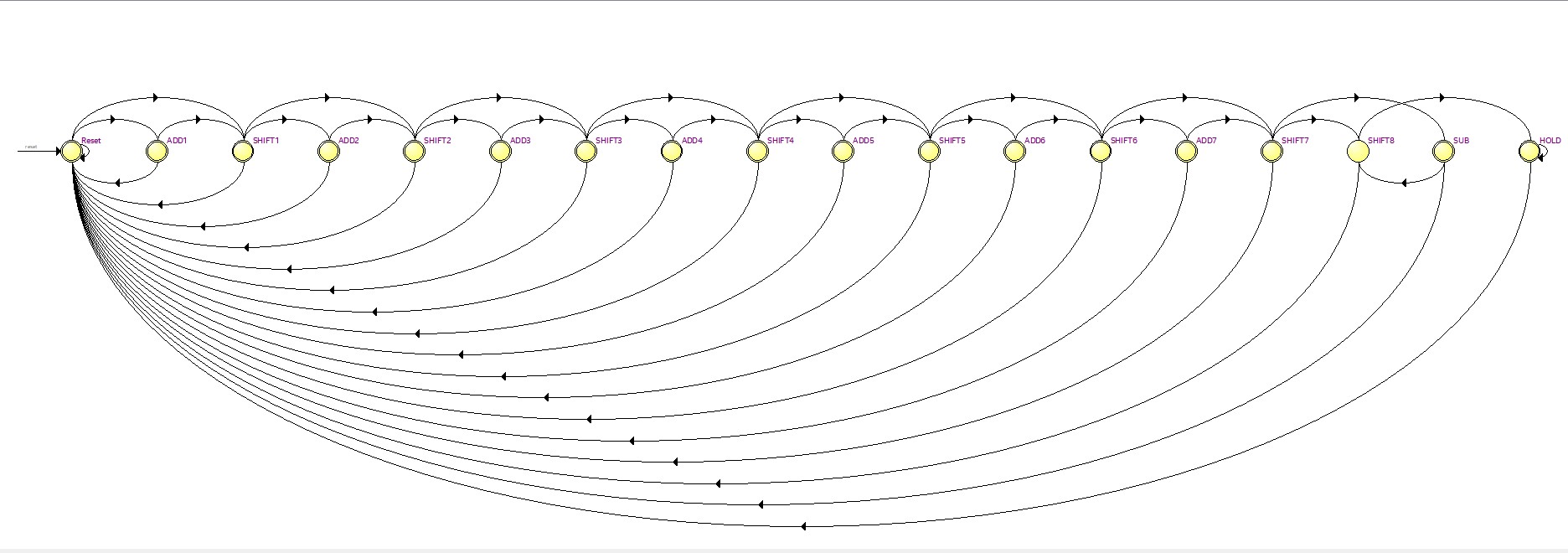
**Diagrams**

****

**Figure 1:** RTL Block Diagram



**Figure 2:** Annotated simulation



**Figure 3:** State Diagram

**State Diagram Description:**

The above state diagram (Figure 3) shows the 18 states and their connections. The first state, reset, is where the system stays before run is pressed. The final state, hold, is where the signal remains until run is not pressed, to account for the run button being held. Each state has a path back to reset that occurs if the reset button is pressed. The other states are each ADD or SHIFT state. If the “m” bit of register B is a 1, the path to the next add (or subtract in one case) is taken, otherwise the system skips that state to move to the next shift state.

**Module Descriptions**

Module: Control

Inputs: logic Clk, Reset\_Load\_Clear, Run, m,

Outputs: logic Shift\_En, Ld\_A, Add\_En, Sub\_En

Description: This module uses positive edge triggered flip flop logic to move between operating states given only the current state and the input buttons.

Purpose: This is how the system’s state machine is created. It uses a combination of shift states, where the registers are told to shift, and optional add/subtract states that tell the machine to do addition and put the result into register A.

Module: HexDriver

Inputs: logic [3:0] In0,

Outputs: logic [6:0] Out0

Description: This is the standard HexDriver module we have already used many times previously. It encodes the transformations needed to transform a 4 bit input to be displayed properly on the hex displays.

Purpose: This allows our registers to be displayed on the FPGA’s hex display.

Module: Multiplier

Inputs: logic [7:0] A, SW, Shift\_En, Add\_En, Sub\_En, clk, Reset, Ld\_X

Outputs: logic [7:0] newA, X\_in,

Description: This module contains the majority of the combinational logic needed to implement the multiplication function. It contains a 9 bit ripple adder, multiple multiplexers, and the connection logic between them. It uses the multiplexers to optionally add or subtract register A with the switch value depending on what state the machine is in.

Purpose: This is the module where the majority of the machine is implemented. The physical operation is done here controlled by the state machine.

Module: Processor

Inputs: logic Clk, Reset\_Load\_Clear, Run, [7:0] SW

Outputs: logic [7:0] Aval, Bval, X\_in, M, [6:0] HEX0, HEX1, HEX2, HEX3

Description: This module is the top-level for the circuit, and is the interface between all other modules. It calls the various modules with corresponding inputs/outputs to achieve the expected results. It also contains the logic needed for the intermediate signals and FPGA input switches and buttons.

Purpose: This module is required to interface between the system inputs and the modules responsible for control and operation of the multiplier circuit.

Module: Reg\_8

Inputs: logic Clk, Reset, Shift\_In, Load, Shift\_En, logic [7:0] D,

Outputs: logic Shift\_Out, [7:0] Data\_Out

Description: This is a positive edge triggered 8 bit register. It has synchronous load and asynchronous reset. Shift\_In is needed for all shift states, while Load is needed to update the values of register A after addition, and for the initial loading of register B.

Purpose: This module is used to define registers A and B. It allows them to shift during shift states, and load when needed. Combined, the contain the multiplication result after the full state cycle.

Module: Register\_unit

Inputs: logic Clk, Reset, A\_In, B\_In, Ld\_A, Ld\_B, Shift\_En,[7:0] sum, [7:0] D,

Outputs: logic A\_out, B\_out, [7:0] A,[7:0] B,

Description: This module contains the calls of the Reg\_8 module.

Purpose: This helps keep the registers organized in one place, and makes calling multiple registers easier by reducing the number of repeated calls to the same input.

Module: Synchronizers

Inputs: logic Clk, d,

Outputs: logic q

Description: This module defines positive edge triggered flip flops. These flip flops take in asynchronous inputs and have a synchronous output.

Purpose: This module is needed to map the FPGA button inputs to synchronous input signals to avoid potential errors in timing within the system.

Module: Testbench\_mult

Inputs: N/A

Outputs: N/A

Description: This testbench module allows the user to set various signal values in order to simulate and analyze the system’s behavior.

Purpose: This is needed to simulate and confirm functionality of the finalized system.

**Post-Lab Questions**

| **LUT** | 89 |
| --- | --- |
| **DSP** | 0 |
| **Memory (BRAM)** | 0 |
| **Flip-Flop** | 37 |
| **Frequency** | n/a |
| **Static Power** | 89.94 mW |
| **Dynamic Power** | 0 |
| **Total Power** | 98.67 mW |

**Table 2: Design Statistics**

Optimizing the circuit would most likely have to take place in the multiplier. Our method of multiplication is straightforward to implement, but relies on a lot of additional operations. To increase efficiency, we could use a more efficient addition method. For ease of application, we used a 9 bit ripple adder, but using a 4 bit and 5 bit ripple adder combined with the lookahead method could increase speed.

The X register stores the sign of the value in register A. This is necessary because when A is shifted, it needs to have its sign preserved. It has to be stored in a register because it needs to be preserved for the next clock cycle. It is set every time the value in A changes.

If you were to use an 8 bit adder carry-out instead of a 9 bit adder, the sign would not be preserved after certain edge-case addition operations. This could result in sign errors under the right circumstances. Adding the extra bit prevents this by allowing the carry-out to be trashed.

Continuous multiplications are limited by the number of bits that can be stored in a single register. This is because register A is not preserved once the multiplication starts (except for its sign). If you continue to multiply beyond this point, the overflow will ruin the data stored in A, and the answer will no longer accurately reflect the desired operation.

As discussed with the continuous multiplication, this method is limited by the number of bits that can be stored in register B, which is a disadvantage over the pencil and paper method. For advantages, this method requires fewer registers, as the pencil and paper method would require you to store and shift the multiplicand many times before finally summing several of them together. Our implementation only requires two 8-bit registers, plus a one-bit register for X.

**Conclusion**

Throughout the duration of this experiment there were not too many major problems encountered as once we understood what we needed to do it was fairly clear how to do it. However, once the code was finished being written we were able to gain further understanding of modelsim and debugging techniques as there were multiple little errors producing large effects such as, bit sizing, syntax, and naming. Thus, going forward we will make sure to keep an eye for them in the process to reduce the time it takes to debug. As far as the directions go we feel like it was pretty clear what needed to be done once the manual was read over a few times. The chart which was provided and we had to make for the pre lab helped out a lot to understand what the state machine should look like. The incomplete block diagram was also helpful as it assisted in visualizing the use of the registers in the way that they were separately together as X, A, and B created the final 16-bit result as XAB. All in all, this lab helped further develop our SystemVerilog skills for future use in more difficult experiments.