ECE 385

Spring 2023

Lab 4

Experiment #4

Aleena Majeed & Ali Chaudry
17:50 - 17:55

1. Introduction

- **a. Summarize the basic functionality of the multiplier circuit.**

2. Pre-lab question

- **a. 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.**

3. Written description and diagrams of multiplier circuit

- **a. Summary of operation**
  - **i. Explain in words how operands are loaded, how the multiplier computes its result, how the result is stored, etc.**
- **b. Top Level Block Diagram**
  - **i. This can be generated from the RTL viewer. Please only include the top-level diagram and not the RTL view of every module.**
- **c. Written Description of .sv Modules**
  - **i. List all modules used in a format shown in the appendix of this document.**
  - **ii. You may insert expanded RTL diagrams of each individual module here if it is legible.**
- **d. State Diagram for Control Unit**
  - **i. 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.**

4. Annotated pre-lab 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.**

5. Answers to post-lab questions

- **a. Fill in the table shown in 5.6 with your design's statistics.**
- **b. Answer all the post-lab questions. As usual, they may be in their own section or dispersed into the appropriate sections in the rest of the report.**

6. 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.**

1. Introduction

a. Summarize the basic functionality of the multiplier circuit.

The multiplier circuit performs the task of multiplying two numbers, which can be two's complement numbers as well. This is achieved by using the add shift algorithm, which resembles the traditional pencil and paper multiplication technique. In this method, the multiplicand is added to itself a certain number of times, as determined by the multiplier. The digits of the multiplier are taken one at a time from right to left, and each digit is used to multiply the multiplicand. The resulting intermediate product is then placed in the correct position to the left of the previous result.

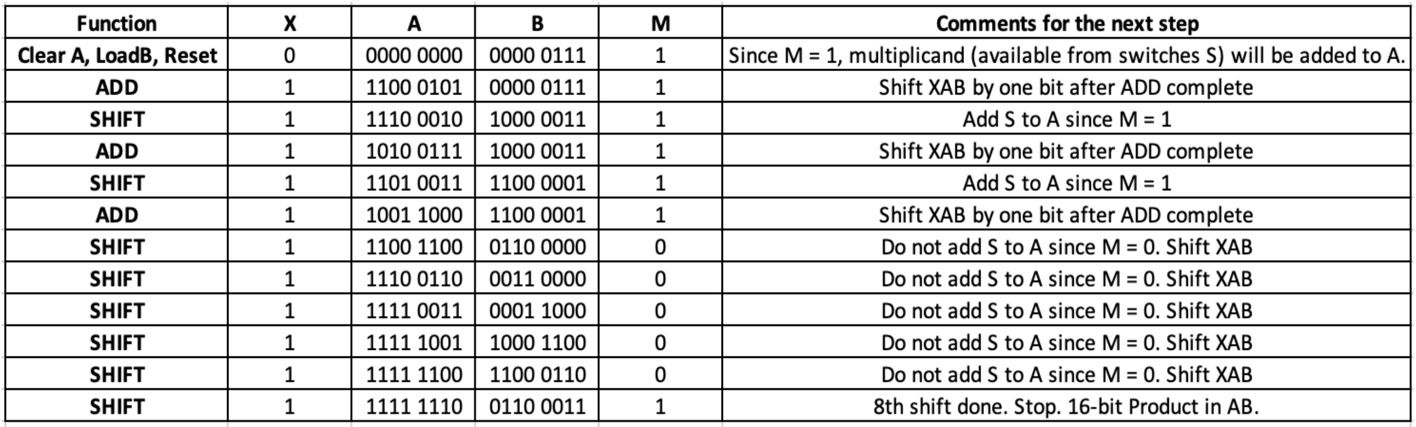
The circuit employs three registers, namely Register X, Register A, and Register B, along with an adder that can perform both addition and subtraction operations, and eight switches. The overall behavior of the circuit is controlled by a dedicated control unit.

To initiate a multiplication operation, the multiplier should be loaded into Register B using the switches on the FPGA board and the Reset_Load_Clear button. This button utilizes logic from the control unit to clear the values in registers A and X and load the multiplier into Register B. Once the switches have been set for the multiplicand, the run button should be pressed, and the circuit should compute the product of the two numbers. The resulting outputs of registers A and B are displayed on the hex displays.

In case of repeated multiplication, the run button must be pressed multiple times, and the reset button should not be pressed unless the user intends to overwrite the value in Register B.

2. Pre-lab question

a. 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 for the next step
Clear A, LoadB, Reset	0	0000 0000	0000 0111	1	Since M = 1, multiplicand (available from switches S) will be added 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 S to A since M = 1
ADD	1	1010 0111	1000 0011	1	Shift XAB by one bit after ADD complete
SHIFT	1	1101 0011	1100 0001	1	Add S to A since M = 1
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	1	1110 0110	0011 0000	0	Do not add S to A since M = 0. Shift XAB
SHIFT	1	1111 0011	0001 1000	0	Do not add S to A since M = 0. Shift XAB
SHIFT	1	1111 1001	1000 1100	0	Do not add S to A since M = 0. Shift XAB
SHIFT	1	1111 1100	1100 0110	0	Do not add S to A since M = 0. Shift XAB
SHIFT	1	1111 1110	0110 0011	1	8th shift done. Stop. 16-bit Product in AB.

Above is a table containing the add and shift method to compute the product of -59 (11000101) and 7 (00000111)

3. Written description and diagrams of multiplier circuit

a. Summary of operation

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

How operand is Loaded:

To load the operand, the user must first select the switches to load in the multiplier into register B. Next the button that clears registers X and A and loads register B must be pressed. Once the button is released, registers A and X are cleared and B is loaded with the multiplier using the switches. Next the user sets the multiplicand to the S switches and hits the run button to perform the multiplication operation, while keeping the switches at the same value of the multiplicand throughout the whole operation.

How the multiplier Computes its result:

The multiplier operates using the add shift algorithm. We are able to add at most 7 times, subtract 1 time, and shift 8 times. In the provided example in the lab document, The first addition occurs since the M bit, which is the least significant bit of B, is one and then is followed by a shift operation. This M bit determines whether or not we want to do the operation or shift: if M = 1 we do the operation, and if M = 0 we shift. But, after every add or subtract operation there must be a shift regardless. Additionally, whether to add or subtract is determined by the X bit, which is the most significant bit of A. If X is a 0 then you add, but if X is a 1 then you subtract in order to get the correct sign-bit at the end of the operation, since we're dealing with two's complement. If the M bit is a 0 then you want to shift unconditionally until it becomes a 1, which means you will then need to add or subtract based on the X bit. The shift operation performs an arithmetic right shift on the total 17 bits stored in X, A, and B. The control unit, which has the FSM states, tells the circuit when to stop multiplying, which is when the last shift state has been reached.

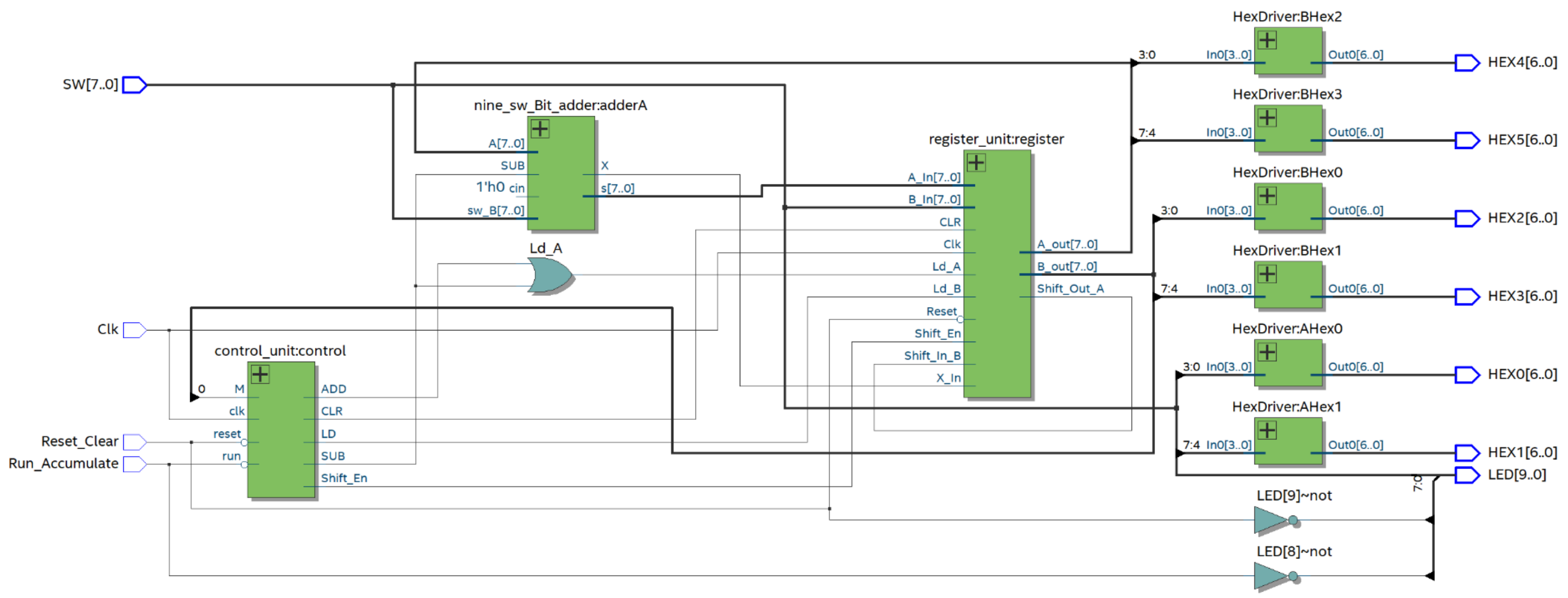
How the result is stored:

When the X bit is 1, we subtract and when it is 0, we add the values that were stored in the switches from the value in A before we shifted. The final product is stored into registers A and B as 4 bit hex values, and the answer is stored at four 4-bit hex values.

b. Top Level Block Diagram

i. This can be generated from the RTL viewer. Please only include the top-level

diagram and not the RTL view of every module.



Above is the RTL diagram for our add-shift multiplier, which was generated by Quartus

c. Written Description of .sv Modules

i. List all modules used in a format shown in the appendix of this document.

ii. You may insert expanded RTL diagrams of each individual module here if it

is legible.

- **registers.sv :** The register file includes modules for register X, A, and B, along with a top level module used for instantiating all the registers. Each register has a load enable, data in, data out, and register A and B have a shift in and shift out. Register B is loaded when the ClearAX_LoadB button is pressed and register A is loaded with the data in from the 9 bit adder. Register X takes the most significant bit of A, the 9th bit from the adder, and during a shift, shifts its value into A while register A shifts its least significant bit into register B. Register B's significant bit is M which determines whether to add or subtract. The register_unit module is instantiated in the top level with the control unit, adder, and switches to perform the operations and multiplication correctly.

- **module reg_X()**
  - **input logic Clk, Reset, Shift_In_X, Load_X, Shift_En_X,**
  - **input logic D_X,**
  - **output logic Shift_Out_X,**
  - **output logic Data_Out_X);**

- **module reg_B()**
  - **input logic Clk, Reset, Shift_In_B, Load_B, Shift_En_B,**
  - **input logic [7:0] D_B,**
  - **output logic Shift_Out_B,**
  - **output logic [7:0] Data_Out_B);**

- **module reg_A()**
  - **input logic Clk, Reset, Load_A, Shift_En_A, Shift_In_A,**
  - **input logic [7:0] D_A,**
  - **output logic Shift_Out_A,**
  - **output logic [7:0] Data_Out_A);**

- **module register_unit()**
  - **input logic Clk, Reset, Ld_A, Ld_B, Shift_En, CLR, X_In,**
  - **input logic [7:0] A_In, B_In,**
  - **input logic Shift_In_B,**
  - **output logic [7:0]A_out, B_out,**
  - **output logic Shift_Out_A**

- **processor_toplevel.sv :** The processor_toplevel instantiates all of the components of the multiplier together to make sure the registers and other components are being driven with the correct signals. The top level employs the logic behind the state machine to make sure that the registers A and X are cleared and B is loaded at the same time when the Reset button is pressed. It also ensures that repeated multiplication occurs if Run is repeatedly pressed by the user. The registers are also instantiated with the adder to ensure that register X and Register A can receive the input and outputs of the adder.
  - **module processor_toplevel()**
    - **input Clk, Reset_Clear, Run_Accumulate,**
    - **input [7:0] SW,**
    - **output logic [9:0] LED,**

        - **output logic [6:0]   HEX0, HEX1, HEX2, HEX3, HEX4, HEX5**

- **control_unit.sv :** The control unit includes the finite state machine implemented to control the signals driving the registers and 9-bit adder. It consists of 7 ADD states, 8 Shift States, a SUBTRACT state, a HALT state, a HOLD state, and a CLEAR_LD_STATE. The first state is the CLEAR_LD_STATE, which is responsible for setting the clear and look signal high. The next state is the Hold state which then transitions to the first add state only if the run button is pressed. In the hold state the CLR is set high only if Run is pressed which allows for repeated multiplication. After the first add state is a shift state followed by more add and shift states until the subtract state in which the subtract control signal is set to high. After the last shift state, the FSM reaches the halt state. In the Halt state no signals are high. If run is still not being held down the FSM goes back to the hold state which moves onto the next state if run is pressed again. Additionally the subtract state is and add states are only activated when M, the least significant bit of register B, is a 1.
    - **control_unit()**
        - **input logic clk, reset, run,**
        - **input logic M,**
        - **output logic Shift_En, SUB, ADD, CLR, LD**

- **synchronizer.sv :** The synchronizer unit was provided code in previous labs that we were just able to copy and paste this module and reuse it as a way to eliminate any errors that could come from multiple signals sharing the same clock.

    - **module sync()**
        - **input logic Clk, d,**
        - **output logic q**
    - **module sync_r0()**
        - **input logic Clk, Reset, d,**
        - **output logic q**
    - **module sync_r1()**
        - **input logic Clk, Reset, d,**
        - **output logic q**

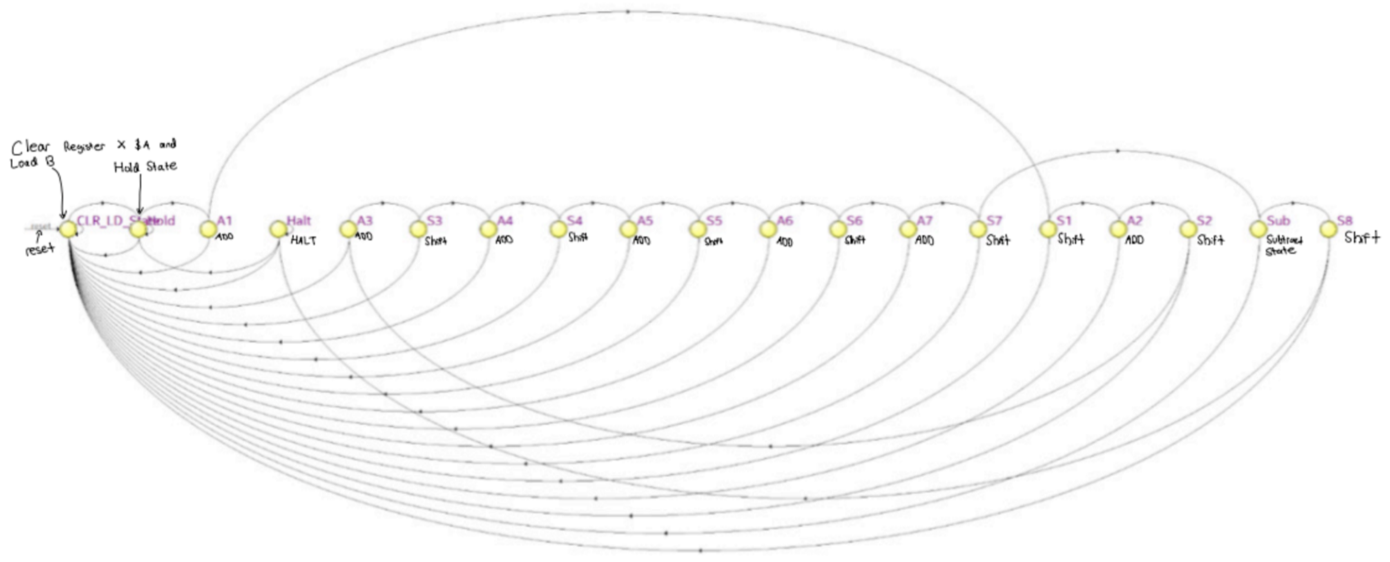
- **9_bit_adder.sv :** The 9-bit-adder is constructed by using a full 1 bit full adder module which is then instantiated in a 9-bit adder module that cascades 9 of these 1 -bit full adders. The sum of the last adder is designated to X which is the most significant bit of register A. Additionally for our 9-bit adder our initial carry-in is 0, but if we want to subtract we check if the subtract signal from the control unit is high and if it is high, then the carry in is set to a 1 and the switch inputs are inverted in order to account for adding a negative number which the same and subtraction in two's complement.
    - **module full_adder()**
        - **input A,**
        - **input sw_B,**
        - **input cin,**
        - **output logic s,**
        - **output logic cout**
    - **module nine_sw_Bit_adder()**
        - **input [7:0] A,**
        - **input [7:0] sw_B,**
        - **input SUB,**
        - **input cin,**
        - **output logic [7:0] s,**
        - **output logic X**
- **Hex_driver.sv :** This module simply contains if statements to apply the correct input/output logic so that the hex drivers of the FPGA get the correct values depending on which LED values are high.
    - **module HexDriver()**
        - **input logic [3:0] In0,**
        - **output logic [6:0] Out0);**

- **testbench.sv :** This last module is the testbench file that allows us to test our code in the ModelSim software. This is done by testing values for the inputs, switches, sums, outputs of the registers, etc. so that we are able to see the outputs of the multiplication on the simulator.

    - **module testbench()**

d. State Diagram for Control Unit

i. 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.



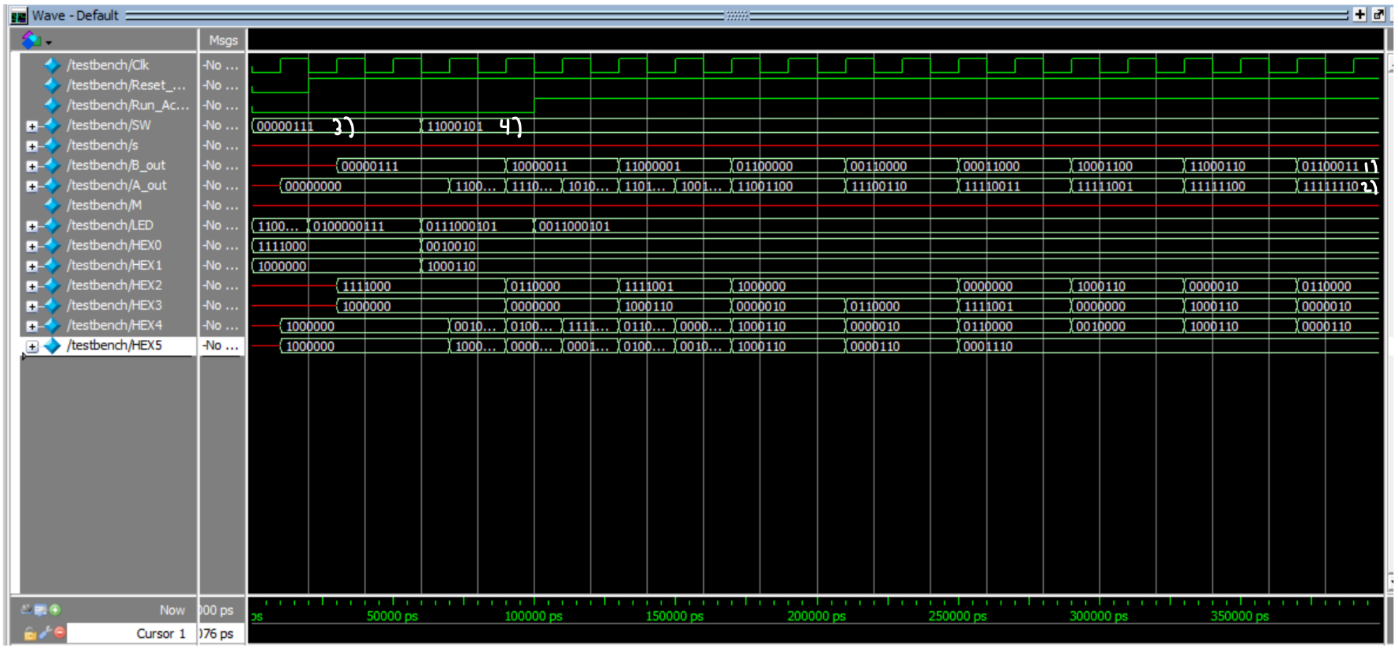
Above is the FSM that was implemented in our shift-add multiplier for lab 4. This FSM contains the 7 adds, 1 subtract, 8 shifts, Clear XA/Load B, and our hold state

4. Annotated pre-lab 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.

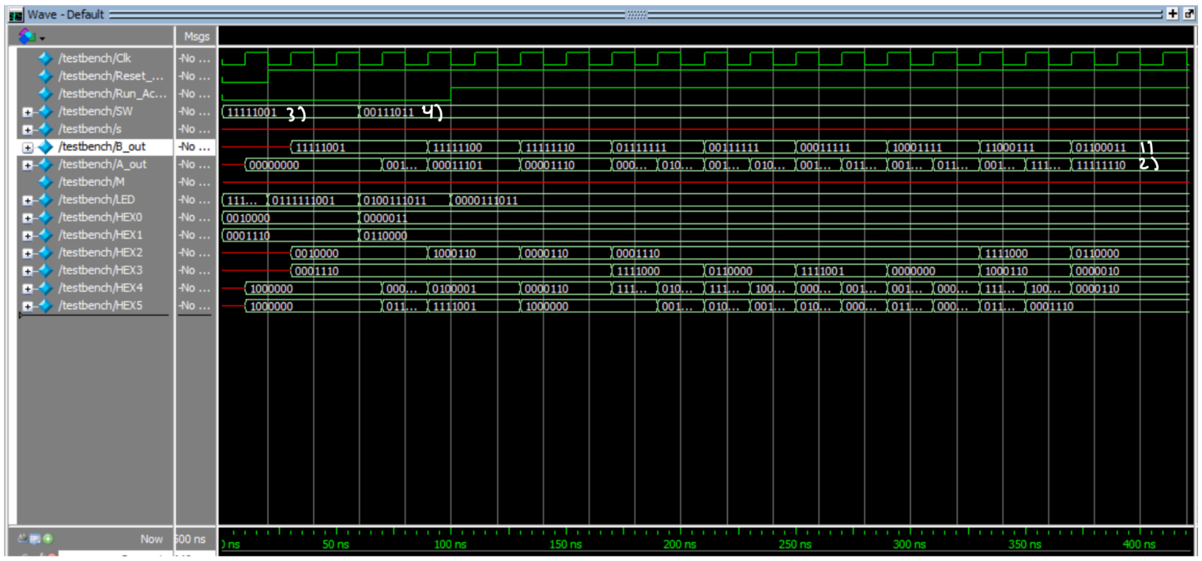
7 * -59 operation (+*-)



Above is the Modelsim waveform for multiplying positive and negative number (7-59) - description of annotation is below*

1) Number 1 shows the Bout value of our multiplier. This represents the lower half of the product of the multiplicand and multiplier stored in register B. In this case, the lower value is x63 in hex
2) Number 2 shows the Aout value of our multiplier. This represents the upper half of the product of the multiplicand and multiplier stored in register A. In this case, the upper value is xFE in hex
3) Number 3 shows the value that is initially to be stored in the B register. This is the multiplicand which will be getting shifted out with the new added values. Initially register B holds -59 (11000101)
4) Number 4 shows the value that is initially to be stored in the A register. This is the multiplier which will be getting added to itself and then shifting into register B. Initially register A holds 7 (00000111)

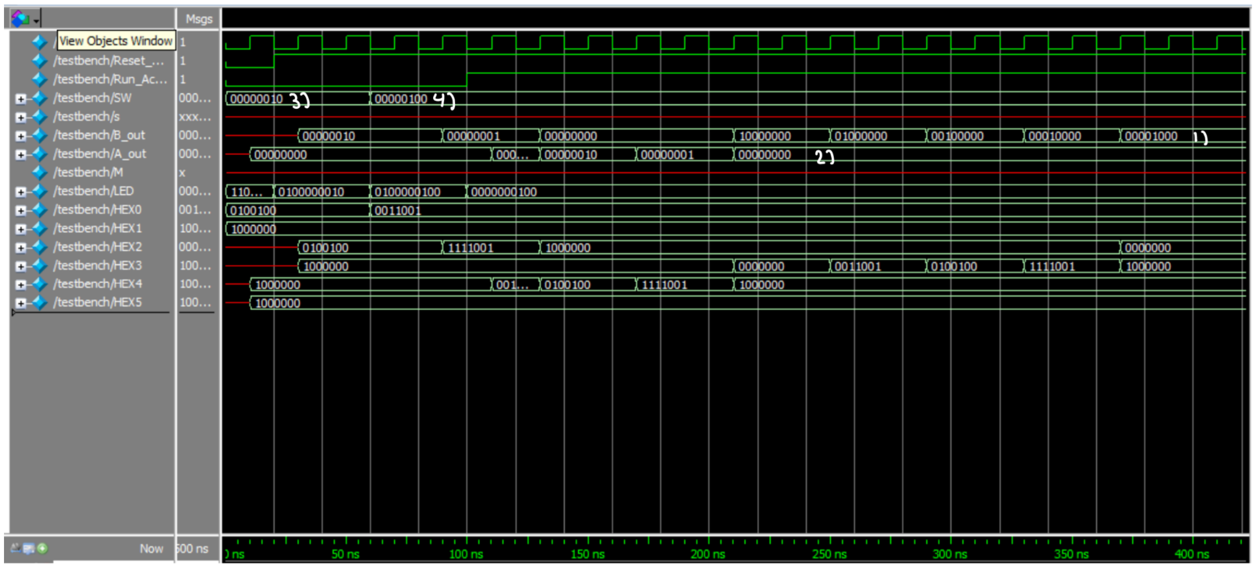
-7 * 59 operation (-*+)



*Above is the Modelsim waveform for multiplying negative and positive number (-7*59) - description of annotation is below*

1. Number 1 shows the Bout value of our multiplier. This represents the lower half of the product of the multiplicand and multiplier stored in register B. In this case, the lower value is x63 in hex
2. Number 2 shows the Aout value of our multiplier. This represents the upper half of the product of the multiplicand and multiplier stored in register A. In this case, the upper value is xFE in hex
3. Number 3 shows the value that is initially to be stored in the B register. This is the multiplicand which will be getting shifted out with the new added values. Initially register B holds -7 (11111001)
4. Number 4 shows the value that is initially to be stored in the A register. This is the multiplier which will be getting added to itself and then shifting into register B. Initially register A holds 59 (00111011)

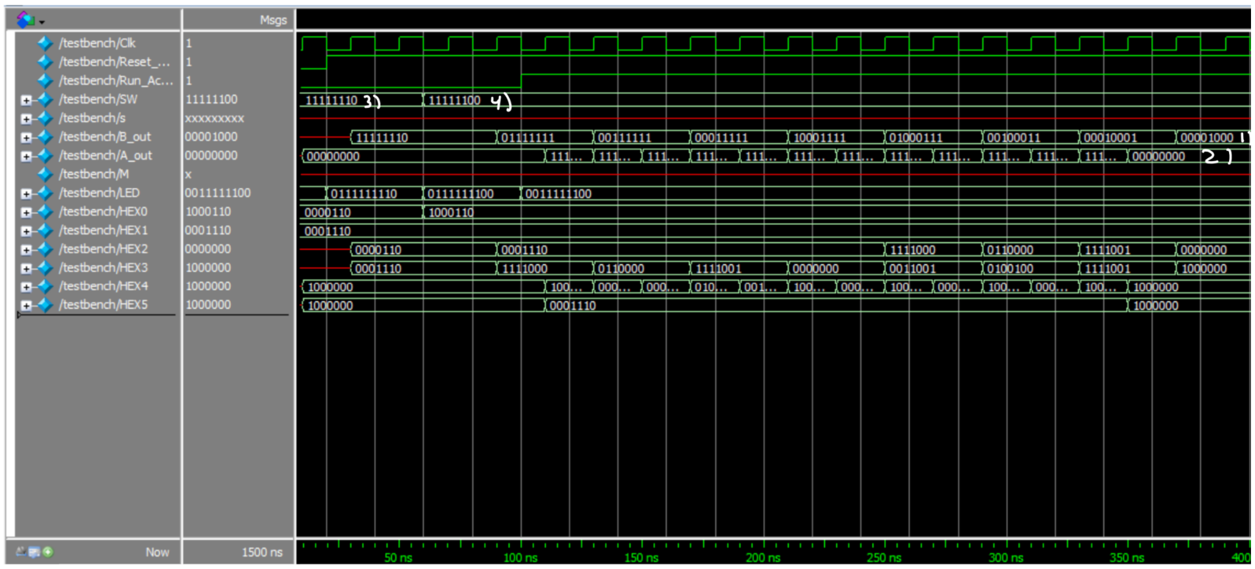
2 * 4 operation (+*+)



*Above is the Modelsim waveform for multiplying two positive numbers (2*4) - description of annotation is below*

1. Number 1 shows the Bout value of our multiplier. This represents the lower half of the product of the multiplicand and multiplier stored in register B. In this case, the lower value is x08 in hex
2. Number 2 shows the Aout value of our multiplier. This represents the upper half of the product of the multiplicand and multiplier stored in register A. In this case, the upper value is x00 in hex
3. Number 3 shows the value that is initially to be stored in the B register. This is the multiplicand which will be getting shifted out with the new added values. Initially register B holds 2 (00000010)
4. Number 4 shows the value that is initially to be stored in the A register. This is the multiplier which will be getting added to itself and then shifting into register B. Initially register A holds 4 (00000100)

-2 * -4 (-*-)

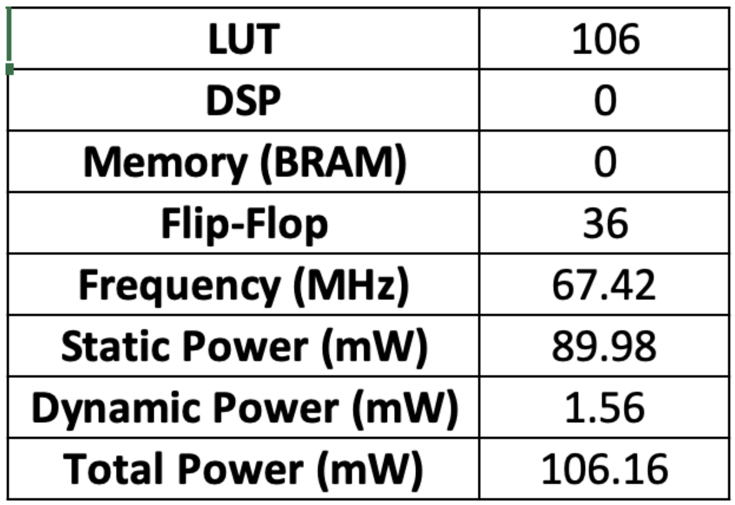


Above is the Modelsim waveform for multiplying two negative numbers (-2-4) - description of annotation is below*

1. Number 1 shows the Bout value of our multiplier. This represents the lower half of the product of the multiplicand and multiplier stored in register B. In this case, the lower value is x08 in hex
2. Number 2 shows the Aout value of our multiplier. This represents the upper half of the product of the multiplicand and multiplier stored in register A. In this case, the upper value is x00 in hex
3. Number 3 shows the value that is initially to be stored in the B register. This is the multiplicand which will be getting shifted out with the new added values. Initially register B holds -2 (11111110)
4. Number 4 shows the value that is initially to be stored in the A register. This is the multiplier which will be getting added to itself and then shifting into register B. Initially register A holds -4 (11111100)

5. Answers to post-lab questions

a. Fill in the table shown in 5.6 with your design's statistics.



LUT	106
DSP	0
Memory (BRAM)	0
Flip-Flop	36
Frequency (MHz)	67.42
Static Power (mW)	89.98
Dynamic Power (mW)	1.56
Total Power (mW)	106.16

b. Answer all the post-lab questions. As usual, they may be in their own section or dispersed into the appropriate sections in the rest of the report.

• What is the purpose of the X register? When does the X register get set/cleared?

The X register is used to hold the most significant bit of register A and is the most significant bit output from the 9-bit adder. Register X is the sign extension of register A that lets you know if you are multiplying negative or positive numbers and while you're doing the multiplication, it is the determining factor of whether you want to add or not. . Register X is cleared at the same time register A is cleared/set which is when you want to multiply new numbers.

• What would happen if you used the carry out of an 8-bit adder instead of output of a 9-bit adder for X?

If you use the carry out of an 8-bit adder instead of the output of a 9-bit adder for register X, the result of the 9-bit addition operation would be truncated to an 8-bit result. This would result in the most significant bit of the 9-bit being lost. Only 8-bit addition operations could be performed.

• What are the limitations of continuous multiplications? Under what circumstances will the implemented algorithm fail?

The limitations of the continuous multiplication is that the implemented algorithm fails when the result of the multiplied numbers is over 16 bits since overflow takes place. As a result, the product gets truncated since it cannot be stored in registers X, A, and B. This failure can occur when the two numbers that are being imputed are too large and require more than 16 bits in order for the output to be accurately represented. In other words, if the output requires 32 bits.

• What are the advantages (and disadvantages?) of the implemented multiplication algorithm over the pencil-and-paper method discussed in the introduction?

The advantages of the implemented multiplication algorithm over the pencil-and-paper method is that it uses less logic gates, elements, and registers. There are some disadvantages such as, the multiplication algorithm also has a lot of steps and is not very intuitive and results in a more complex process. Another disadvantage is that the pencil and paper method would be faster as it would require 8 clock cycles whereas the multiplier would require more than 16 since there are up to 7 ADD and 8 Shift states needed.

6. Conclusion

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

The design functioned as intended and implemented the add-shift algorithm to multiply the number loaded into register B and the switches. A 9-bit adder was used to compute the multiplication and the answer was stored in Register X, register A, and register B.

We had a problem related to repeated multiplication because we did not define our states properly. When we multiplied a negative one repeatedly, we got an answer of zero on the third round of multiplication. To fix it we added a HOLD state after the HALT state so long as the run was not being pressed. In the HOLD state we set all signals to zero except clear which was set to run which means if run is pressed in the hold state, then register X and A are cleared, otherwise they are not.

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.

The add shift algorithm was not adequately explained, leaving me no choice but to seek supplementary resources for a more coherent understanding. Given that these concepts are fundamental to the lab's functionality, I strongly believe that better explanation is essential. It is my hope that such oversights will be corrected to avoid future frustrations.