Lab Report #4

Aryan Shah (aryans5), Dev Patel (devdp2)

University of Illinois at Urbana Champaign

ECE 385 - Digital Systems Laboratory

Professor Zuofu Cheng, T.A. Gene Lee

October 2, 2023

Introduction

Summarize the basic functionality of the multiplier circuit

The purpose of our circuit is to be able to display the computation of two 8-bit 2's complement numbers on the Urbana FPGA. Using the method explained to us in the lab document, we developed an add shift algorithm to compute the multiplication. We begin by introducing three registers: A, B, and X. We also make a module for an adder that is capable of addition and subtraction. The registers and computation are described by inputs to 8 of our FPGA's switches. The Reset_Load_Clear button allows for us to simultaneously clear the value of the X and A registers while loading in our multiplier in the B register. The reason for this is to allow for repeated multiplication. First, the user flips switches accordingly to input the desired number into our B register. Then, once the Reset_Load_Clear button is pressed, this value is loaded in as our multiplier and the other registers are cleared. The user can then flip switches to specify their multiplicand. Once their multiplicand is specified, toggling the Run button will compute and display the final product. The user can then press the Run button as many times as desired to perform repeated multiplication, while keeping the multiplier constant and adjusting the multiplicand as necessary.

Pre-lab Question

Compute 11000101 * 00000111 in a table

Function	X	A	B	M	Comments for the next step
Clear A, Load B, 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 one bit after ADD operation
SHIFT	1	1110 0010	1000 0011	1	Since M = 1, ADD S to A
ADD	1	1010 0111	1000 0011	1	Shift XAB one bit after ADD operation
SHIFT	1	1101 0011	1100 0001	1	Since M=1, ADD S to A
ADD	1	1001 1000	1100 0001	1	Shift XAB one bit after ADD operation
SHIFT	1	1100 1100	0110 0000	0	M = 0, Shift XAB, do not add S
SHIFT	1	1110 0110	0011 0000	0	M = 0, Shift XAB, do not add S
SHIFT	1	1111 0011	0001 1000	0	M = 0, Shift XAB, do not add S
SHIFT	1	1111 1001	1000 1100	0	M = 0, Shift XAB, do not add S
SHIFT	1	1111 1100	1100 0110	0	M = 0, Shift XAB, do not add S
SHIFT	1	1111 1110	0110 0011	1	Halt, 8th shift complete, display 16-bit product in AB

Written Description and Diagrams of Multiplier Circuit

Summary of operation

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

The operand is loaded by the user inputting the desired number using the switches. This value is loaded into our B register. Next, we press the button to clear the registers X and A while simultaneously loading our register B with the appropriate multiplier value. The user then sets the multiplicand to the S switches, and can hit the run button as many times as desired to compute a multiplication operation. The switches of the multiplicand can be adjusted in between each multiplication operation.

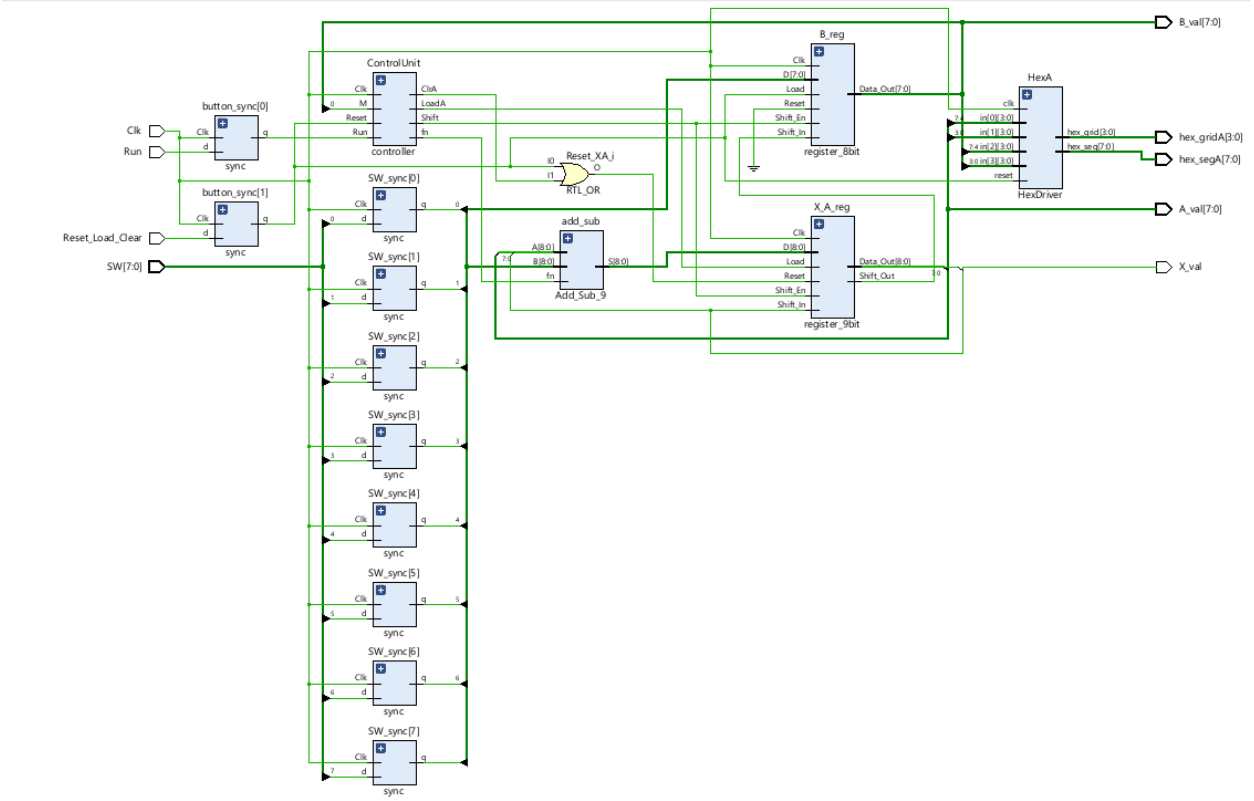
The multiplier computes its result as discussed in class utilizing an add shift algorithm. In the case of our value in register B being its maximum at 0xFF, we find that the maximum operation states we need are 7 ADDs, 1 SUB, and 8 SHIFT. The first addition followed by a shift operation occurs due to the value of the M bit, our least significant bit value of our multiplier. In the

following operations, our M bit specifies whether we want to compute the add or subtract (M = 1) or shift (M=0). It is important to note that even with an add or subtract, there will always be a shift to allow for each digit of the multiplier to be used to multiply the multiplicand. The logic of whether to add or subtract is further identified by the bit in our X register, the most significant bit in our A value. If X is 0, we perform and add operation and vice versa, subtracting if X=1 to account for inputting the correct sign value for our product. Since we are working with 2's complement numbers, attention to signs are important and thus must be accounted for in our final result. Our control unit is responsible for indicating that our final state has been reached and therefore the result at our last state is the final computation. Its logic is specified by an FSM, whose states were defined according to our lab document.

When it comes to storing our final result, we ensure that our last computation is either a subtract or add based upon our X bit. This ensures that the result holds the correct sign value in case of being positive or negative. The final product is loaded into our A and B registers as 4-bit hex values and then displayed on our FPGA as four 4-bit hex values.

Top Level Block Diagram

Generated from the RTL viewer, only top-level (not RTL view of every module)



Written Description of .SV Modules

You may insert expanded RTL diagrams of each individual module here if legible

Processor.sv - This is our top-level module file. It instantiates the necessary input/output signals as well as connections that will be used in our high level design. It defines the 9bit X and A register, 8 bit B register, 9 bit add/subtract adder, and control unit. Finally, it defines the HexDriver for A as copied from our previous labs and synchronizes the switches and buttons we used on our FPGA.

nine_bit_register.sv - The 9bit register module was instantiated for our Registers X and A. The register takes a clock, reset, shift_in, load, shift_en, and data as inputs. It outputs a shift_out and data_out accordingly. Register X is created from the most significant bit of A, the 9th bit from

our full adder, and shifts its values into register A accordingly. We make this register 9 bits rather than 8 to hold the sign bit, which allows us to determine whether the final result is positive or negative.

eight_bit_register.sv - The 8bit register module was instantiated for our B register. It contains a clock, reset, shift_in, load, shift_en, and data as inputs. It further outputs a shift_out and data_out. The least significant bits from register A our loaded into register B. Its most significant bit is our M bit, deciding whether to add or subtract our values. The register B is loaded as our multiplier when we press the Reset_Load_Clear button and its value is not changed by the clear button as to allow for repeated multiplication.

nine_bit_adder.sv - To allow for our adding and subtracting operations, we constructed a 9-bit adder using 1-bit full adders. The sum of our last adder is fed into register X, the most significant bit for our A register. We initialize our first carry-in as 0, which can change depending on if our control unit signifies that we must subtract our values. If the values are to be subtracted, our carry-in value becomes 1 and our inputted bit values are inverted so that we can “subtract” (add the inverted binary number) our numbers to account for two’s complement computation. The file also instantiates the 1-bit full adder, with 3 inputs and 2 outputs. It calculates the sum based on the XOR of the 3 inputs, and also calculates the carry-out value. It was used in the design of our 9-bit adder/subtractor.

Control_unit.sv - This was our main control unit. It defines the different states that we have in accordance with our designed FPGA. In total, it held a maximum of 7 ADD states, 1 SUB state, 8 SHIFT states, a HALT state, a HOLD state, and a clear/load state. The state begins in the clear/load state. Once the appropriate multiplier is loaded in and the multiplicand is indicated, the run button is pressed then transitions into the hold state to indicate our first add operation. The M

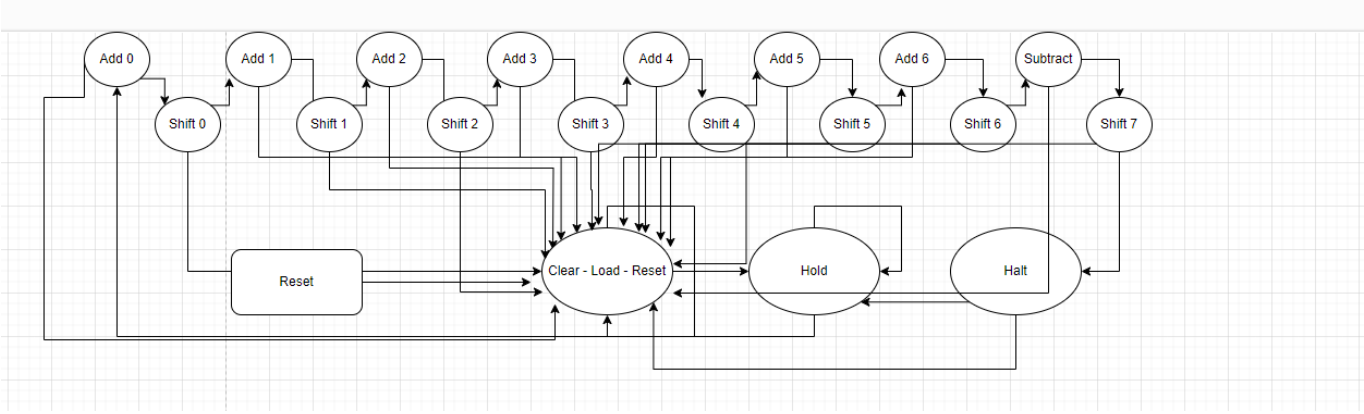
bit determines whether our following computations will be an add or subtract. After our final shift, the FSM reaches a halt state where all the signals are set to low. If the run button is no longer pressed following this, the FSM transitions to the HOLD state. If we indicate repeated multiplication, the clear state is set to high once the run button is pressed repeatedly.

HexDriver.sv - This file was not modified, rather copied from given files of prior labs. It contains the input/output logic so that our hex drivers are displaying correctly on the FPGA based on our LEDs.

Synchronizers.sv - This file was not modified, rather copied from given files of prior labs. This file brings our asynchronous signals such as clock and reset into the FPGA's clock domain. It avoids any timing issues that can result from our different modules sharing the same Clk.

State Diagram for Control Unit

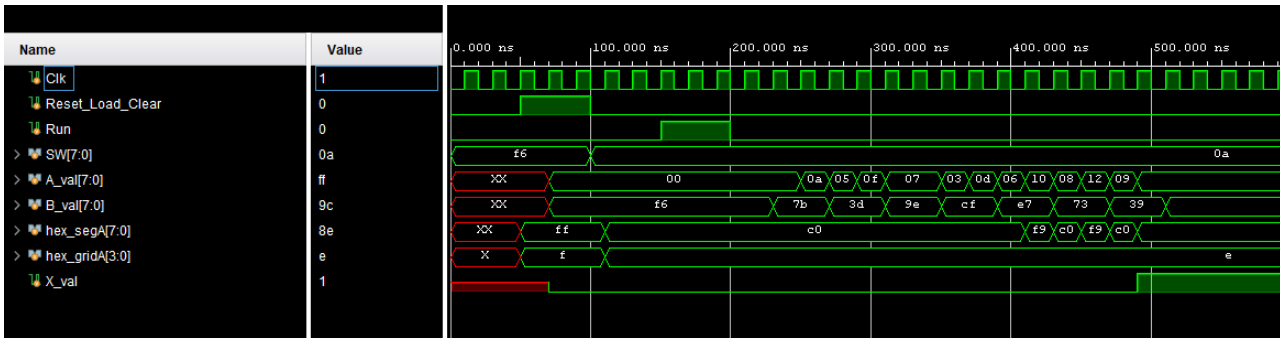
Use an external drawing tool such as Visio, Inkscape, Draw.io



Annotated pre-lab simulation waveforms

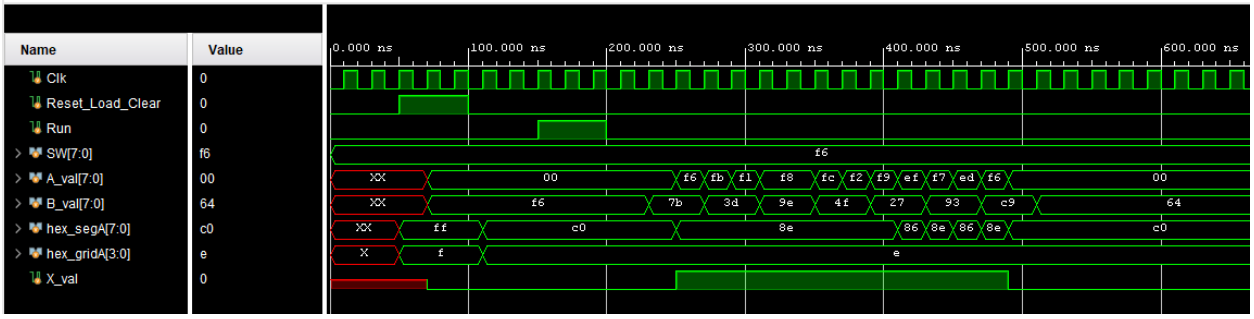
Must show 4 operations (+*+), (+*-), (-*+), (-*-)

+*-:



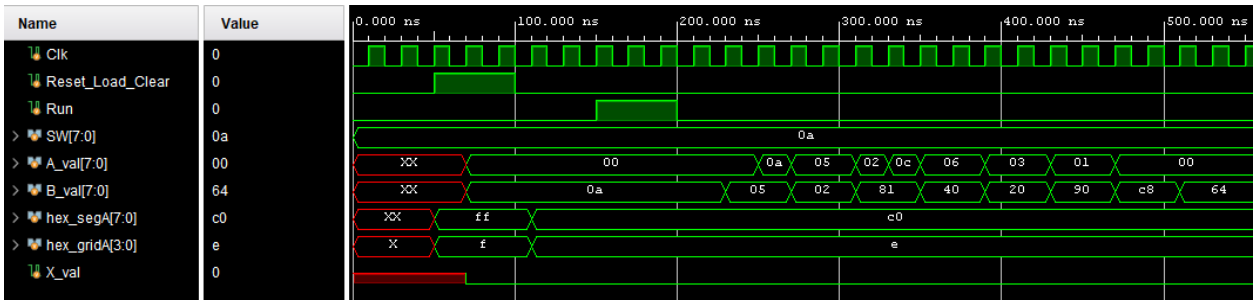
As shown in the waveform, we first loaded the value of the switch with “F6”, which corresponds to the signed value of -10 (in decimal). Then, we set the “reset-load-clear” input to “1” (high), which triggered the FSM to load the switch values into the B_Val register. After setting the load state back to 0, the value in B_Val doesn’t change, as it serves as the base value. We then loaded “0A” into the switches, which corresponds to the decimal value of +10 (in decimal). We set the Run state to 1, and then back to 0 to allow one execution of the run state. This multiplies the two hex values together, and results in a final value of “9C” which is the signed value of -100 (in decimal). This final value is then stored in the B value register, overwriting the original value of F6. It is also split into the two Hex Registers, the larger one containing 8E and the smaller one containing E (which add to 9C). The X_Val (which represents a signed output) is set to 1, which is consistent with our negative final answer. If we were to call the run state again, the SW value would replace the “FF” in A_Val, and the multiplication of a factor of +10 would continue.

-*-:



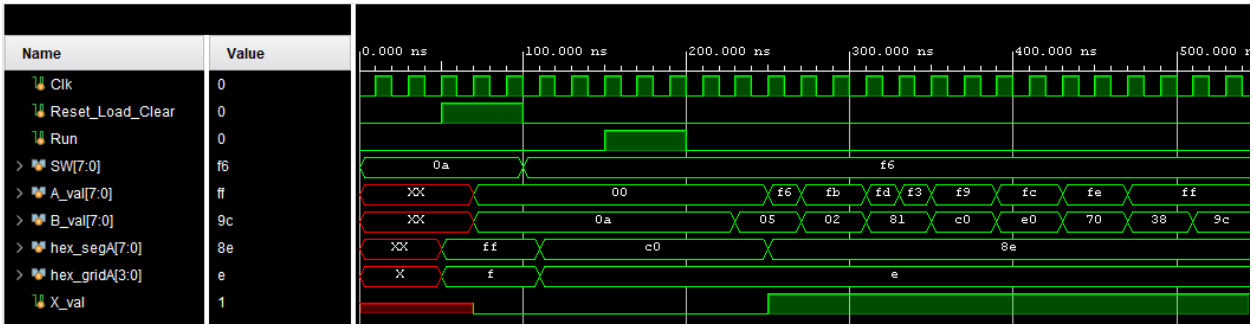
As shown in the waveform, we first loaded the value of the switch with “F6”, which corresponds to the signed value of -10 (in decimal). Then, we set the “reset-load-clear” input to “1” (high), which triggered the FSM to load the switch values into the B_Val register. After setting the load state back to 0, the value in B_Val doesn’t change, as it serves as the base value. We then loaded “F6” into the switches, which corresponds to the decimal value of -10 (in decimal). We set the Run state to 1, and then back to 0 to allow one execution of the run state. This multiplies the two hex values together, and results in a final value of “64” which is the signed value of +100 (in decimal). This final value is then stored in the B value register, overwriting the original value of F6. It is also split into the two Hex Registers, the larger one containing C0 and the smaller one containing E (which add to 64). The X_Val (which represents a signed output) is set to 0, which is consistent with our positive final answer. If we were to call the run state again, the SW value would replace the “00” in A_Val, and the multiplication of a factor of -10 would continue.

+*+:



As shown in the waveform, we first loaded the value of the switch with “0A”, which corresponds to the signed value of +10 (in decimal). Then, we set the “reset-load-clear” input to “1” (high), which triggered the FSM to load the switch values into the B_Val register. After setting the load state back to 0, the value in B_Val doesn’t change, as it serves as the base value. We then loaded “0A” into the switches, which corresponds to the decimal value of +10 (in decimal). We set the Run state to 1, and then back to 0 to allow one execution of the run state. This multiplies the two hex values together, and results in a final value of “64” which is the signed value of +100 (in decimal). This final value is then stored in the B value register, overwriting the original value of F6. It is also split into the two Hex Registers, the larger one containing C0 and the smaller one containing E (which add to 64). The X_Val (which represents a signed output) is set to 0, which is consistent with our positive final answer. If we were to call the run state again, the SW value would replace the “00” in A_Val, and the multiplication of a factor of +10 would continue.

-*+:



As shown in the waveform, we first loaded the value of the switch with “0A”, which corresponds to the signed value of +10 (in decimal). Then, we set the “reset-load-clear” input to “1” (high), which triggered the FSM to load the switch values into the B_Val register. After setting the load state back to 0, the value in B_Val doesn’t change, as it serves as the base value. We then loaded “F6” into the switches, which corresponds to the decimal value of -10 (in decimal). We set the Run state to 1, and then back to 0 to allow one execution of the run state. This multiplies the two hex values together, and results in a final value of “9C” which is the signed value of -100 (in decimal). This final value is then stored in the B value register, overwriting the original value of 0A. It is also split into the two Hex Registers, the larger one containing 8E and the smaller one containing E (which add to 64). The X_Val (which represents a signed output) is set to 1, which is consistent with our negative final answer. If we were to call the run state again, the SW value would replace the “FF” in A_Val, and the multiplication of a factor of +10 would continue.

Answers to post-lab questions

Table 5.6

LUT	67
DSP	0
Memory (BRAM)	0
Flip-Flop	71
Latches	0
Frequency	229 MHZ
Static Power	0.074W
Dynamic Power	0.023W
Total Power	0.097W

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

The primary goal of our X register is to indicate whether we are multiplying positive or negative numbers during our operation. Based on whether you are multiplying a positive or negative value, we can then decide whether we need to indicate an add/subtract operation for our final state. If our X value is 1, then we subtract so that our final result contains the correct sign bit to indicate a negative result. On the other hand, while our X value is 0, we continue an ADD operation as necessary. The value of X gets set by our most significant bit of our A register and the output of our 9-bit adder/subtractor. It gets cleared along with register A when the Reset_Load_Clear button is pressed and a new multiplicand is to be set.

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

The process for multiplying two 8-bit 2's complement numbers requires a 9-bit adder to be accurate. If an 8-bit adder were used instead, the most significant bit that would have been designated for dealing with negative numbers would be lost. Therefore, using the 8-bit adder you would ignore the MSB value and thus perform only addition during the steps.

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

In our design, the continuous multiplication failed when our result was over 16 bits. This is due to overflow, as the final result cannot be stored properly given the space in our X_A and B registers. Usually, this occurs when two larger numbers are repeatedly being multiplied that overflow over 16 bits, requiring a higher bit output to display the correct result.

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

The advantages of utilizing our implemented multiplication algorithm include the reduction of logic gates and necessary registers. It severely simplified our design rather than the high amount of logic needed to implement the pencil-and-paper method. A disadvantage of using the implemented multiplication algorithm is speed. Using the pencil-and-paper method would account for only 8 clock cycles for 7 ADD and 1 SUBTRACT state(s) rather than 16 clock cycles to account for the 7 ADD and 8 SHIFT state(s) necessary in the implemented algorithm. This increases the complexity of our design and the speed at which it can compute, especially at higher input values.

Conclusion

Discuss functionality of design, what didn't work and what could have been done to fix it

In order to implement a two 8-bit 2's complement multiplier, our high-level functionality included the design of an add-shift algorithm. We designated a 9-bit register to hold the 8-bit value of A as well as its most significant bit representing the value of X, to determine our add/subtract operation. A control unit was designed implementing an FSM to instantiate our state logic for shifting, adding, subtracting, holding, loading/clearing, and halting. Finally, we designed a 9-bit adder/subtractor composed of 1-bit full-adders from a previous lab design.

Due to the specificity of our lab materials, no errors outside of issues with code syntax and improperly understanding the design were encountered. Above all, the control unit was the most difficult to design. Upon examining the table provided to us in the lab document, our first attempt at making this module had errors of improperly defining the states accurately. This caused bugs in our design and upon testing, repeated multiplication would not work and rather simply hold the value of register B without performing the operation on the multiplicand. This error was rectified after reviewing the FSM logic more thoroughly and altering the code to make it correct.

Ambiguity or difficulty in lab materials for next semester

N/A, the lab materials were sufficient to carry out this lab to completion.