Lab 4 Report

Noah Conner & Adam Naboulsi
Section ABP

Summary:

Our multiplier circuit computes the product of two signed 8-bit numbers. The product is displayed as a signed 16-bit number. Additionally, the processor supports consecutive multiplication on the previous product, truncated to 8-bit. To interface with the processor, there are eight switches to represent the 8-bit multiplier/multiplicand, a button to load in the multiplier, and a button to compute the product of the multiplier and the multiplicand represented on the switches.

Add-Shift Algorithm:

Initial Values: X = 0, A = 00000000, B = 00000111 (achieved using Reset_Load_Clear button), S = 11000101, M = LSB of B

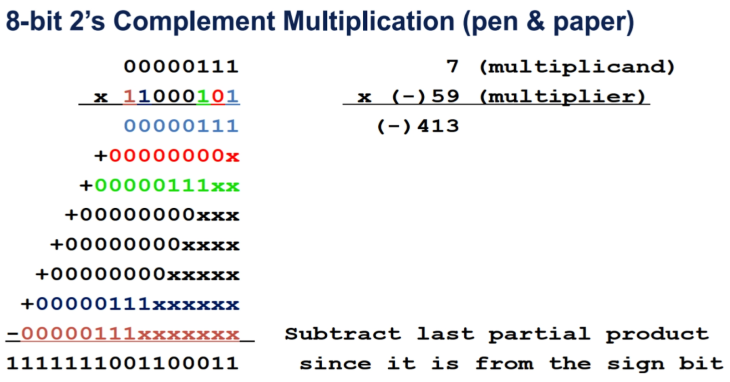
*11000101 * 00000111 = 1111111001100011 (-59 * 7 = -413)*

Function	X	A	B	M	Comments for next step
Clear, Load, Reset	0	00000000	00000111	1	Since M = 1, multiplicand (available from switches S) will be added to A.
Add	1	11000101	00000111	1	Shift XAB by one bit after Add complete
Shift	1	11100010	10000011	1	Add S to A since M = 1.
Add	1	10100111	10000011	1	Shift XAB by one bit after Add complete
Shift	1	11010011	11000001	1	Add S to A since M = 1.
Add	1	10011000	11000001	1	Shift XAB by one bit after Add complete
Shift	1	11001100	01100000	0	Do not add S to A since M = 0. Shift XAB.
Shift	1	11100110	00110000	0	Do not add S to A since M = 0. Shift XAB.
Shift	1	11110011	00011000	0	Do not add S to A since M = 0. Shift XAB.
Shift	1	11111001	10001100	0	Do not add S to A since M = 0. Shift XAB.
Shift	1	11111100	11000110	0	Do not add S to A since M = 0. Shift XAB.
Shift	1	11111110	01100011	1	8th shift done. Stop. 16-bit product in AB.

Operation:

The processor uses a simple add-shift algorithm to compute the product of two signed 8-bit numbers. In comparison to the pencil-paper algorithm, the add-shift algorithm has its advantages and disadvantages. The pencil-paper algorithm, as reflected below, computes a running sum of partial products. The algorithm works by checking each bit of the multiplier, starting at the least significant bit; if the nth bit is 1, we shift the multiplicand n times to the left, increasing the bit length by n, and add the result to the running sum, and if the bit is 0, zero is added to the running sum. If the most significant bit is 1, the n-shifted multiplicand is subtracted from the running sum. After all bits have been iterated through, the running sum will hold the product of the multiplication. This pencil-paper algorithm does not apply nicely to logic due to the fact that we have to add many partial products together while keeping track of what bit we are at. To implement this literally, we would need an adder with twice the number of bits as inputs; thus, the algorithm would be inefficient in terms of its logic implementation. The add-shift algorithm allows us to instead shift the entire running sum to the right and add the multiplicand. The advantage of this is that our adder only needs to be 9-bits wide whereas the pencil-paper multiplier would require a 16-bit adder. When it comes to disadvantages, the add-shift algorithm requires an extra bit to store the sign extension of the running sum (X in our design). This X bit aids in the signed shifting process, explained in further detail below.

Pen-Paper Algorithm



8-bit 2's Complement Multiplication (pen & paper)

```
        00000111              7  (multiplicand)
     x  11000101          x (-)59  (multiplier)
        00000111           (-)413
       +00000000x
      +00000111xx
     +00000000xxx
    +00000000xxxx
   +00000000xxxxx
  +00000111xxxxxx
 -00000111xxxxxxx   Subtract last partial product
 1111111001100011    since it is from the sign bit
```

Throughout the computation, a 17-bit register, XAB, is used to store the concatenated value of X (1-bit), A (8-bit) and B (8-bit), respectively, where X is the sign extension of A. In order to begin the process, the user will set the signed 8-bit multiplier on the eight switches of the FPGA development board. After which, the Reset_Load_Clear button can be pressed, loading the multiplier into B and clearing {X, A}[1]. The switches can then be set to the multiplicand and will remain the same throughout the computation. The 9-bit sign extended multiplicand is represented by S (9-bit). Additionally, the least significant bit of B is represented by M (1-bit). At the end of the operation, the product will be stored in {A, B}.

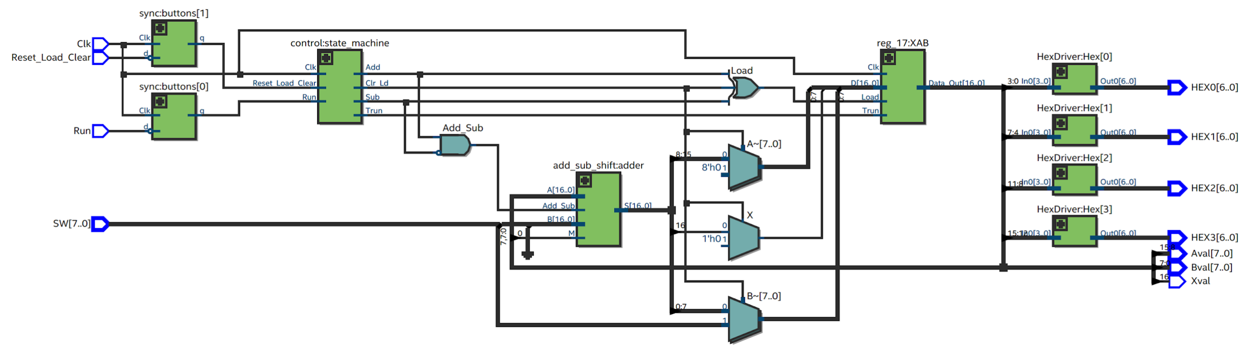
In order to keep track of the various operations, we have implemented a state machine with eight shift states, an idle state, a truncate state, a wait state and a reset/load/clear state. The state machine module has 1-bit outputs Clr_Ld, Add, Sub and Trun. When the Reset_Load_Clear button is pressed it enters the reset/load/clear state and sets Clr_Ld to high. It is important to note that in any state, only one of the outputs is allowed to be high. In the top level module, this tells the register to load the multiplier into B and clear {X, A}. Once this process is finished and the button is released, the machine will return to the idle state. From which, the Run button can be pressed, triggering the start of a multiplication. The machine then enters the truncate state, setting Trun to high. This tells the top level module to clear {X, A}, truncating {X, A, B} to 8-bits. Then, the machine will enter into the first shift state and synchronously move to the next shift state until the last one is reached. At this point, the machine will enter a wait state, only returning to idle when the Run button is released. In the first seven shift states the output Add is high and in the final shift state the output Sub is high. Now that we know how the state machine works, we will take a look at the top level module and see how it makes use of the Add and Sub signals. When Add is high there are two things that could happen, depending on the value of M. M controls whether S is to be added or subtracted to {X, A}. If M and Add are high, the sum of {S, 00000000} and {X, A, B} is shifted right once (by concatenation) and stored back into register XAB. However, if M is low and Add (or Sub) is high, the sum of zero and {X, A, B} is shifted and stored into register XAB. If instead M and Sub are high, the two's complement of S will be added. Finally, if neither M, Add nor Sub are high, no add or shift operation will be performed. After the state machine returns to idle, the multiplication will be complete and the product will be stored in {A, B}. The table above illustrates this process.

If a consecutive operation was to be performed, the multiplier would become the value in B, truncating {X, A, B} to 8-bits. If {X, A, B} changes value when truncated i.e. {X, A, B} represents a number greater than 8-bits, the algorithm fails. Consequently, this limits the user to how many consecutive multiplications can be performed before the running total becomes erroneous.

With all that said, it may still be unclear why there is an X bit. This extra bit helps in the signed shifting process. A shift operation is typically performed synchronously by a shift register. A shift register shifts in a given bit into the most significant bit position and pushes out the least significant bit. If a signed shift were to be performed, the shift-in bit would need to be equal to the most significant bit. To accomplish this, X is used to store the sign extended bit of A so that the register can shift in X when performing a signed shift. In our design we did not use a shift register but instead shifted by concatenation. Our design did not actually require the X bit since we could perform a signed shift using concatenation. Another point to clarify is why we needed to sign extend S and A to 9-bits before adding/subtracting. If we instead added/subtracted S and A as 8-bit numbers and set X equal to the carry-out bit, there would be an error. The problem arises when adding a negative number and a positive number. The carry-out of this operation will always result in a carry-out bit opposite in sign of the most significant bit e.g. 1101 + 0110 = (1)0011. Notice how the carry-out bit is 1. If X were to equal 1 and a shift operation was performed, this would not be a signed shift. However in the 9-bit implementation, you would get the correct value to be stored in X, 11101 + 00110 = $\underline{0}$0011.

[1] Concatenation is represented by { } e.g. {01, 1101} = 011101.

Top-Level Diagram:



<u>*Module Descriptions:*</u>

Module: full_adder (within add_sub_shift.sv)

Inputs: A, B, Cin

Outputs: S, Cout

Description: Single-bit adder with carry-out and sum; uses combinational logic to compute cout.

Purpose: Used as the basic building block for the 9-bit Ripple Adder used on our design. Chaining multiple full adders allows for computation with larger numbers (more bits).

Module: add_sub_shift.sv

Inputs: [16:0] A, [16:0] B, Add_Sub, M

Outputs: [16:0] S, Cout

Description: This module instantiates a 9-bit ripple adder which adds two values, B_ (internal 9-bit logic) and A[16:8], and stores the output in S_ (internal 9-bit logic) . Input A holds the value of our 17-bit register XAB and input B holds the value of the sign extended (9-bit) switch values (S) concatenated with eight 0 bits. In an always_comb block, it checks for the value of input M and Add_Sub; if M and Add_Sub are high, it sets B_ = B[16:8] which is the value of the sign-extended switches, and sets cin_ of the ripple adder to 0 (since high Add_Sub equates to an Add operation). If M is high and Add_Sub is low, it sets B_ = 2's complement of B[16:8] and cin_ of the ripple adder to a 1 (since low Add_Sub equates to a subtract operation). If M is low, it sets B_ = 9-bit 0 and cin_ to 0 such that A[16:8] is just passed through the adder (when M is low we do not want to add or subtract). With each operation completed in this module, a shift happens through concatenation where the output S is set to {S_[8], S_, A[7:1]}; S_[8] is the sign extension of S_ which becomes the X of our XAB reg in our design; S_ (9-bit output of our adder) which becomes {A,B[7]} in our XAB reg; and A[7:1] becomes B[6:0] in our XAB reg.

Purpose: The purpose of this module is to add/subtract our multiplicand to our XAB register when appropriate and shift it by one bit while setting the new Xval.

<u>*Module:*</u> Reg_17.sv
Inputs: Clk, Reset, Load, Trun, [16:0] D
Outputs: [16:0] Data_Out

Description: This module represents a 17-bit register which is synchronous with Clk and Reset and sets the output of the register to zeroes when input Reset is high and loads D into the register when the input Load is high. When input Trun (truncate signal) is high, it concatenates (from MSB to LSB) 9 zero bits and Data_Out[7:0] (clears XA and truncates to contain only value B in our design), effectively truncating the register to output the lower 8 significant bits.

Purpose: This module is used to hold the concatenated value {X,A,B} which contains the running sum and the final product in {A,B}.

<u>*Module:*</u> HexDriver.sv
Inputs: [3:0] In0
Outputs: [6:0] Out0

Description: Based on the inputs, a switch statement is used to map the outputs to the corresponding LED segments (7 total) on the HEX display of the FPGA board.

Purpose: Used to output bits of registers in the register unit to the HEX displays on the FPGA board.

<u>*Module:*</u> control.sv
Inputs: Clk, Reset_Load_Clear, Run
Outputs: Clr_Ld, Add, Sub, Trun

Description: The control module consists of 10 states – RLC, WAIT, IDLE, TRUN, A-H – where transitions to the next state are synchronous with the clock and outputs Clr_Ld, Add, Sub, and Trun are assigned based on the current state (only one can be high for each state). It also takes in two signals: Run which tells the processor when to transition through the states which carry out the multiplication operation (A-H), and Reset_Load_Clr which tells the processor to return to the initial state (RLC). In the initial state RLC (reset/loadB/clearA), Clr_Ld is the only output set to high; the next state is set to IDLE if input Reset_Load_Clear is low. In the IDLE state, all of the outputs are low (set according to default case in case statement) since we want the machine to hold its current position (no desired changes); the next state is set to TRUN if input Run is high. In the TRUN state, Trun is the only output set to high; the next state is set to A regardless of the input values (want to truncate XAB before the consecutive run of states A-H which carry out the operation). For states A-G, Add is the only output set to high (when the 7 consecutive shifts and possible adds occur); the next state is set to B-H correspondingly regardless of the input values. In state H, only the Sub output is set to high (when the final shift and possible subtraction occurs); the next state is set to WAIT regardless of the input values. In the WAIT state, all outputs are set to low; the purpose of this state is allow for consecutive multiplication operations; the next state is set to IDLE if input Run is low, thus once it returns to the IDLE state and Run is toggled, the consecutive multiplication operation (without reset) takes place.

Purpose: The control module controls and acts as the state machine of the processor, assigning outputs based on the current state and signaling transitions between states based on the inputs.

Module: Synchronizers.sv
Inputs: Clk, d
Outputs: q

Description: The synchronizer module outputs a clock-synchronized signal of an asynchronous input signal. Includes 3 sync modules, one with no reset for switches and buttons, one with reset to 0, and one with reset to 1.

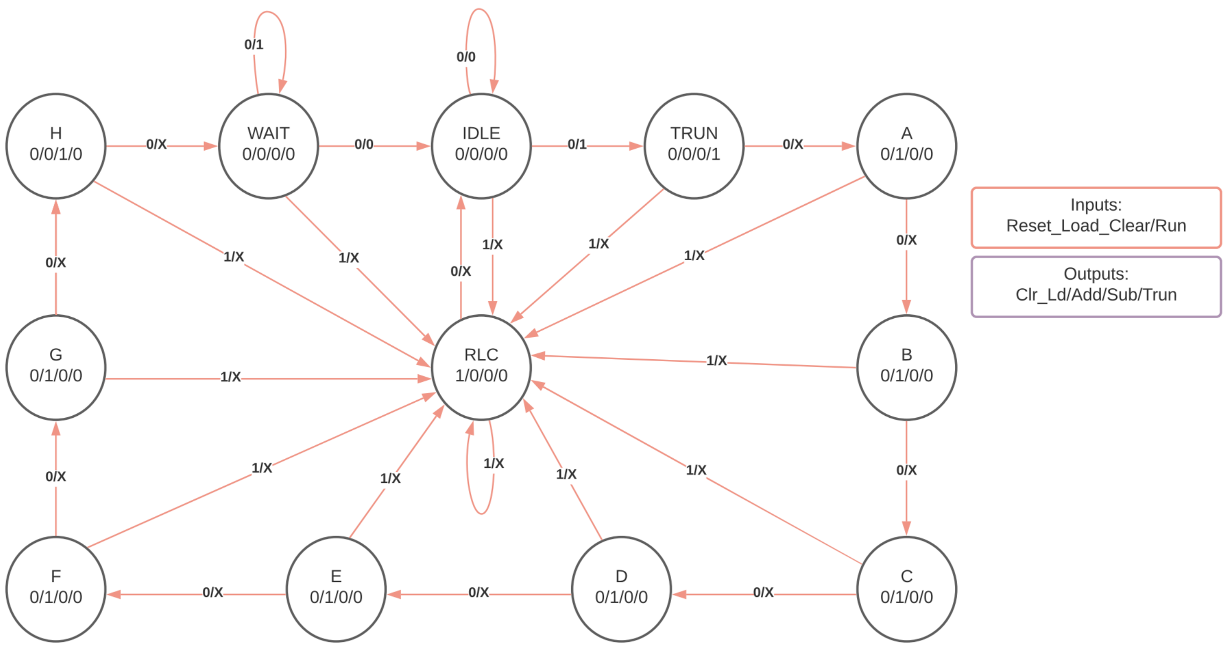
Purpose: This is used to synchronize all of the user-inputted signals.

Module: multiplier.sv
Inputs: Clk, Reset_Load_Clear, Run, [7:0] SW
Outputs: Xval, [7:0] Aval, [7:0] Bval, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3

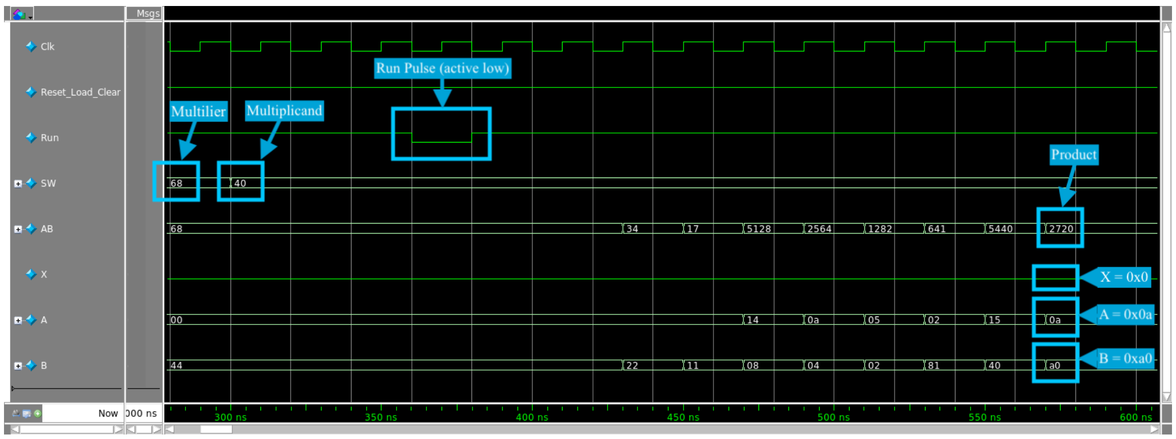
Description: This module connected the pieces of our multiplier; it instantiated the major modules within our processor's design including the control unit (state machine), our 17-bit XAB register, the add_sub_shift module, the Hex Drivers (setting the desired values to output onto our FPGA Hex LEDs), the synchronizers, with the necessary internal logic needed/used by the modules. It is also in charge of setting internal signal Load (high when Add, Sub, or Clr_Ld are high) which is passed in to our register; setting the value of Add_Sub (high when Add is high and low when Sub is high) which is passed into our add_sub_shift module; clearing XA and (reset/clear) and loading B with the switch values, SW, when signal Clr_Ld is high (internal logic that is passed into the control unit); and lastly loading the next values in XAB – {X, A, B} = {X_, A_, B_} – where internal logic X_ holds the Xval, A_ holds Aval, and B_ holds Bval.

Purpose: This module served as the top level of our design, connecting the various modules to form our multiplier and overseeing operation.

State Diagram:

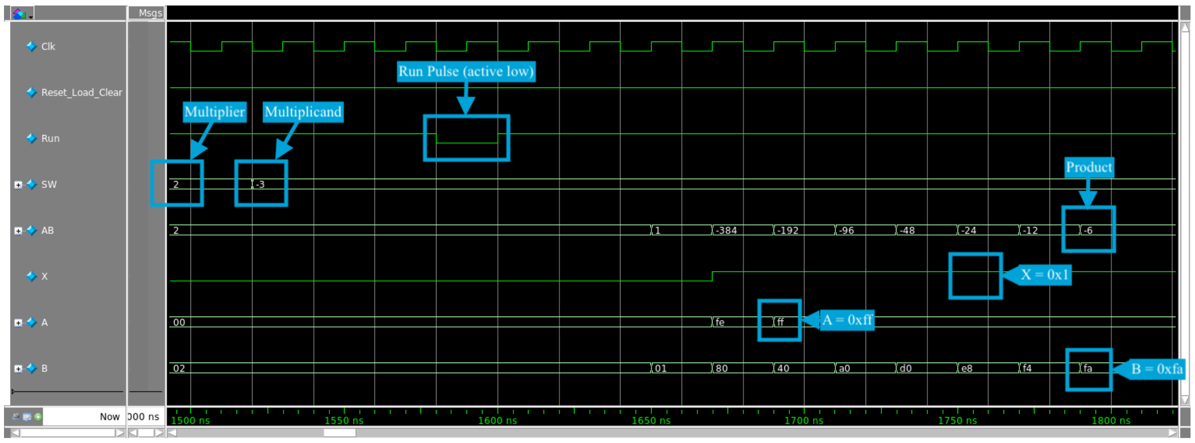


Test 1:
68 × 40 = 2720



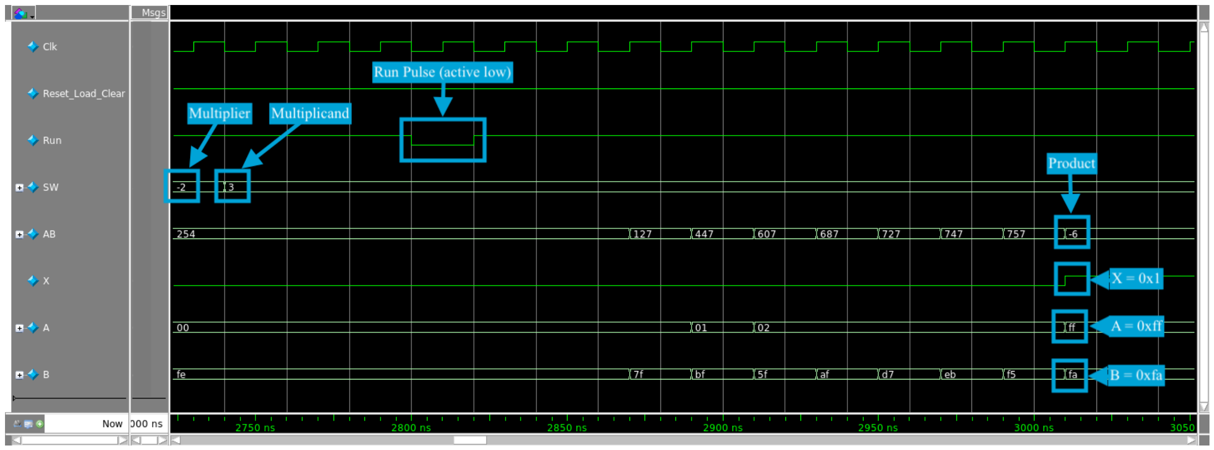
Test 2:

$2 \times -3 = -6$

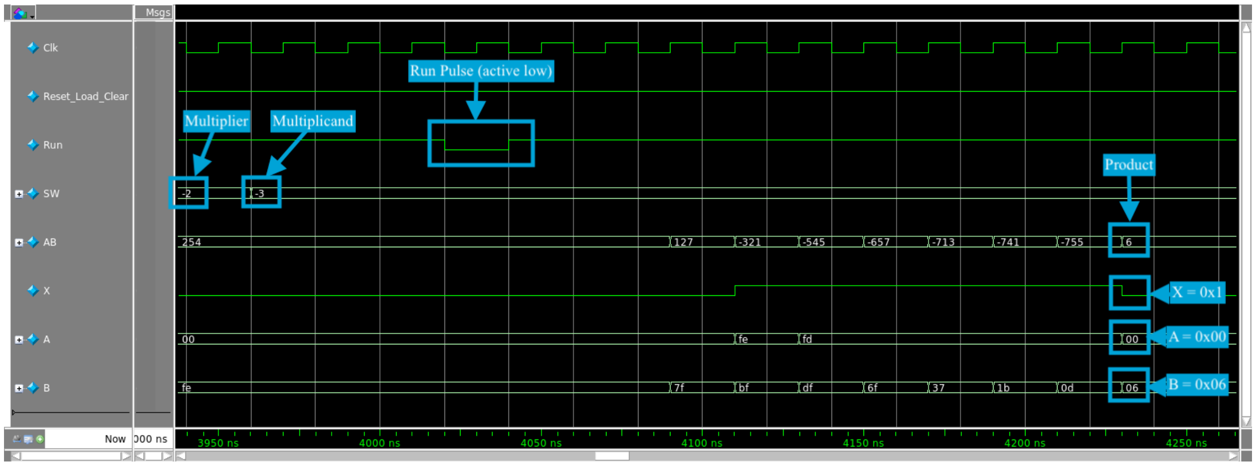


Test 3:

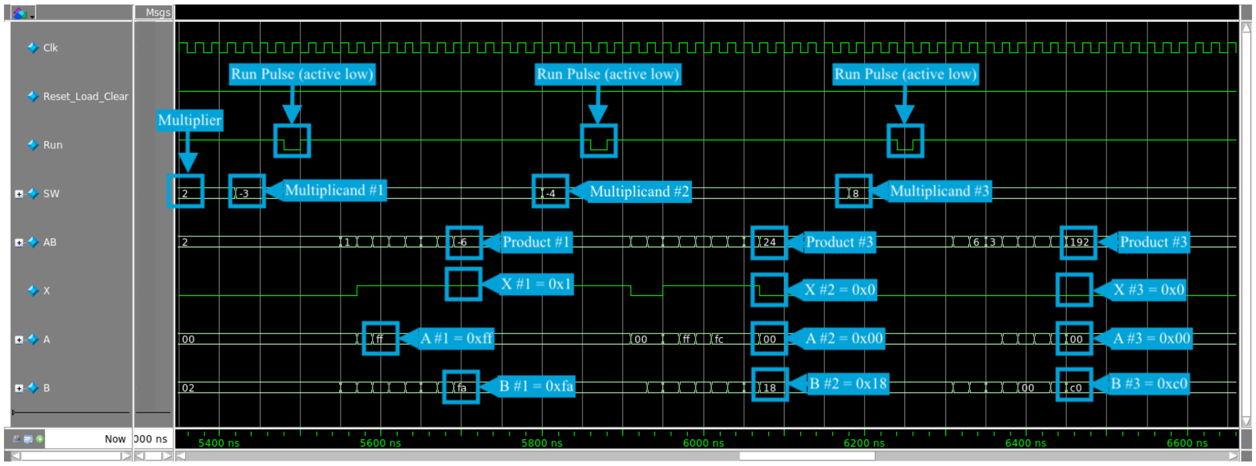
$-2 \times 3 = -6$



Test 4:
-2 × -3 = 6



Test 5:
2 × -3 × -4 × 8 = 192



Design Statistics

LUT	119
DSP	0
Memory (BRAM)	0
Flip-Flop	22
Frequency	96.86 Mhz
Static Power	89.97 mW
Dynamic Power	1.51 mW
Total Power	104.01 mW

Optimizing:

In optimizing our design, we found ways to reduce the total gate count but not without a tradeoff. Since the multiplication process requires the addition and subtraction of values, we needed an adder of some sort. This ripple carry adder is an adder that requires a very small amount of resources but is much slower than other types of adders. Since this design does not need to be very fast, we chose the ripple carry adder for its simplicity. If speed was an issue, we could resort to a carry lookahead adder or carry select adder as they are much faster but require much more resources than the ripple carry adder.

Conclusion:

Our processor allows the user to compute the product of two 8-bit numbers and even supports consecutive multiplication if the previous product does not exceed 8-bits. The switches and buttons located on the FPGA development board make it easy for the user to set the multiplier and multiplicand and run the operation. All parts of our design are fully functional.

All in all, the lab manual and given materials were clear, concise, and useful as a guide for implementing the multiplier. The explanation for how the algorithm works and the provided table outlining the steps for a specific multiplication operation facilitated the implementation process. Similarly to the last lab, in terms of improvements, maybe including a link to an excel sheet with the I/O Pin Assignments would be convenient and more efficient.