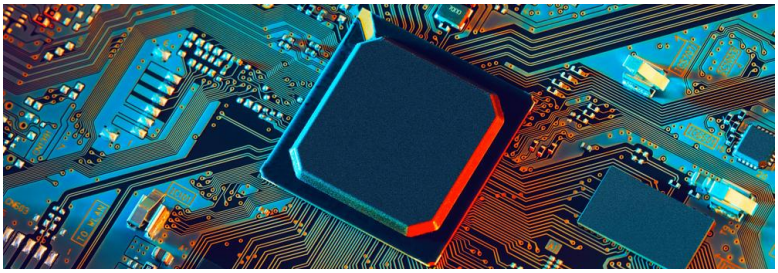
CSCE : Digital Design I

Sequential Multiplier Project



Group 3

Final Report

Submitted by:

Hadj Ahmed	
Raafat El-Saeed	900191080
Thomas	
Mahmoud Nour	900202978

Submitted to:

Dr. Mohamed Shalan

Spring 2023

Introduction:

In this project we were required to design and implement an 8-bit signed sequential multiplier using the famous shift-and-add algorithm and implement it on the Basys 3 FPGA board. This project provided a great learning opportunity for us to implement our knowledge in digital circuit design and Verilog into a real-world application. We faced a lot of challenges during the research, design, and implementation phases of the project. Indeed, these challenges increased our understanding of the material and provided us with a real-life example of trying to manage a project while working with a team during a limited time period. This report basically aims to outline our design details, implementation issues, and validation activities for the project.

A. Functionality

Our design for the signed sequential multiplier was basically implemented in a way to achieve all the following design functionality:

1- The program uses toggle switches on the Basys 3 FPGA board to enter two 8-bit binary signed values.

2- The product of the two numbers is displayed on the 7-segment display

3- The leftmost 7-segment display is used to display the sign bit

4- The 3 right 7-segment displays are used to display the product in decimal format

5- The buttons BTNL and BTNR are used to scroll the product digits (the 3 right 7-segment display) to display the remaining bits of the product as the product can be up to 5 bits.

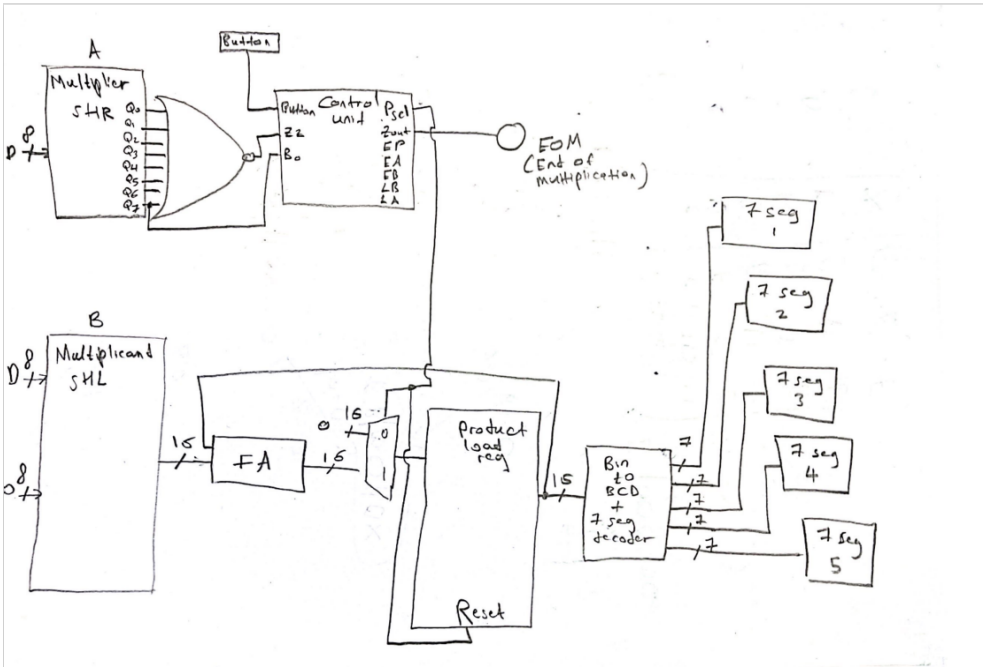
6- The button BTNC is used to start the multiplication.

7- The LED LD0 is used to indicate the end of the multiplication

B. Project Phases

Our final product that was developed for this project was basically the outcome of different phases outlined as follows:

Phase 1- Block Diagram: Instead of going directly to start with implementing a Logisim design or writing a Verilog module, we started by drawing a simple block diagram for the sequential multiplier as a means of mapping out the different components & their functions.

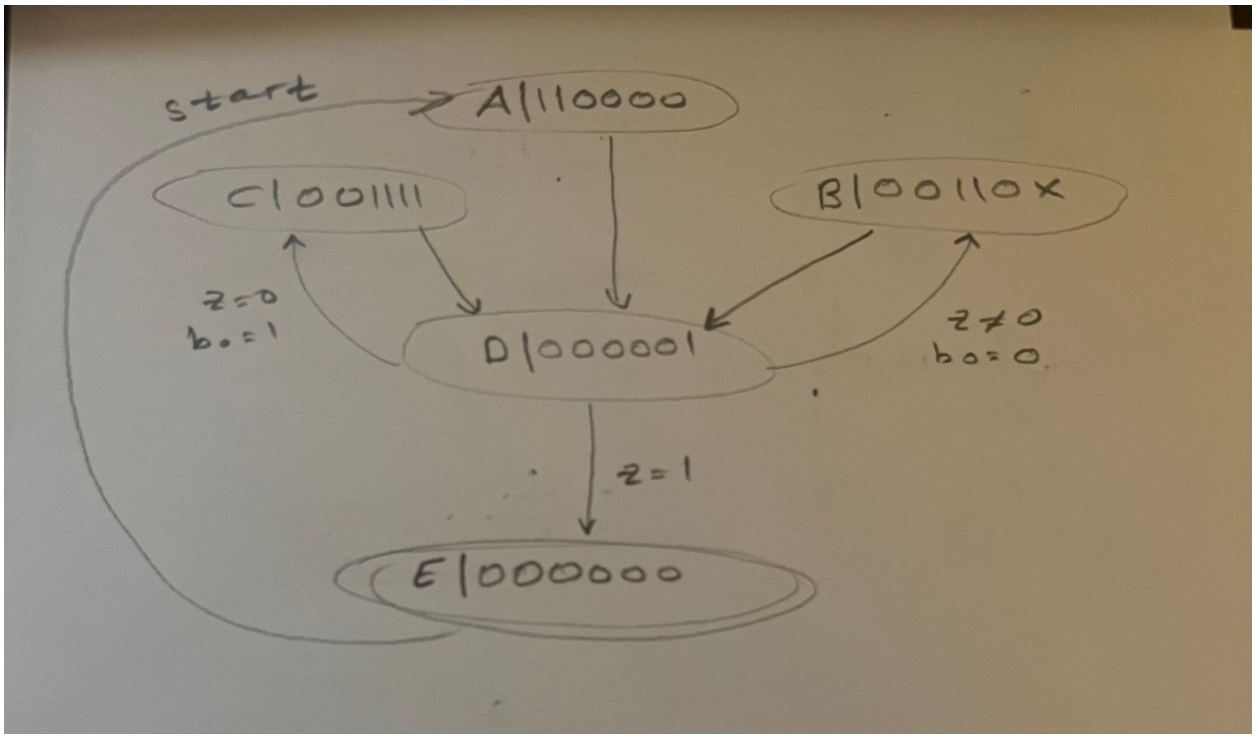


As represented in the image above, the block diagram includes a design of the datapath showing the registers for the multiplier, the multiplicand and the product as well as showing the control unit and a Binary to BCD converter to convert the product into a format that can be displayed on the 5 7-segment display. Since this was just a basic initial design, this initial block diagram did not account for the scrolling of the 7-segment display and instead was represented as shown above as 5 different 7-segment displays. However, this was later enhanced as the project evolved and was accounted for in the Verilog module which will be discussed below.

Phase 2- Logisim Simulation: Secondly, after drawing the initial basic block diagram, we aimed to simulate the datapath and the details of the black boxes we have and the inner components on Logisim before the development of the Verilog module. Also, please note that in the Logisim simulation we implemented the datapath, the control unit, and the binary to BCD and seven segment display. The Logisim simulation implemented unsigned multiplication and did not include the scrolling for the 7-segment displays as of this phase. Instead, as mentioned above, we outputted the result on 5 seven segment displays using the module provided by Logisim. Attached below are screenshots of the different components that were designed on Logisim for this phase. In this phase, our main goal was to confirm that our multiplication algorithm, the control unit, and the binary to BCD modules work as expected. In the next phase, the Verilog implementation, we add more features as required, namely the scrolling and the signed multiplication. Please find below a brief description of the implementation details of the control unit, the datapath, and the binary to BCD module.

Control Unit:

We implemented the control unit using FSM. The state diagram is attached below.



This is a Moore machine. The output depends on the present state only. The transition between states depends on the two inputs z (this tells whether the multiplier is 0) and b0(the least significant bit of the multiplier). The output of the FSM is the control signals. The output corresponds to this order: LB LA EB EA EP Psel.

State E: This is the initial state, and all the control signals are 0.

On the press of the start button, we begin our multiplication

State A: The only active control signals are LB and LA that load the values of the multiplier and multiplicand in their registers.

State B: This state is reached when we haven't yet reached the end of multiplication (z flag is still 0) and the least significant bit (b0) is 0. This is the state where the shifting of the multiplier and multiplicand occurs, but the product value is not updated.

State C: This state is reached when we haven't yet reached the end of multiplication (z flag is still 0) and the least significant bit (b0) is 1. In This state shifting of the multiplier and multiplicand occurs, and the shifted multiplicand is added to the product.

Note that the shifting was made in one clock cycle, and the control unit state registers took also one clock cycle to update. The next state depends on b0 and z. We initially transitioned from the

C to the B state directly depending on the z and b0 values. This however caused problems. Every clock cycle the control unit outputs the control signals, including EB and EA responsible for shifting the register. However, the registers don't shift immediately, but shift with the next clock cycle. This does not give time for the correct inputs (the z flag and b0) to reach the control unit. To solve this problem, whenever transitioning from C to B, we pass to a static state , where the control signals are all 0. This gives time for the shifting to occur and for the correct values of b0 and z to reach the control unit.

Using the state diagram above, we construct the following state table.

	Present State			Next State											
				b0=0, z = 0			b0=0, z=1			b0=1, z=0			b0=1, z=1		
	y2	y1	y0	Y2	Y1	Y0	Y2	Y1	Y0	Y2	Y1	Y0	Y2	Y1	Y0
A	0	0	0	0	1	1	0	1	1	0	1	1	0	1	1
B	0	0	1	0	1	1	0	1	1	0	1	1	0	1	1
C	0	1	0	0	1	1	0	1	1	0	1	1	0	1	1
D	0	1	1	0	0	1	1	0	0	0	1	0	1	0	0
E	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
	1	0	1	X	X	X	X	X	X	X	X	X	X	X	X
	1	1	0	X	X	X	X	X	X	X	X	X	X	X	X
	1	1	1	X	X	X	X	X	X	X	X	X	X	X	X

	Present State			Output					
	y2	y1	y0	LB	LA	EB	EA	EP	Psel
A	0	0	0	1	1	0	0	0	X
B	0	0	1	0	0	1	1	0	X
C	0	1	0	0	0	1	1	1	1
D	0	1	1	0	0	0	0	0	1
E	1	0	0	0	0	0	0	0	X
	1	0	1	X	X	X	X	X	X
	1	1	0	X	X	X	X	X	X
	1	1	1	X	X	X	X	X	X

We divided the state table into two to accommodate for the page size.

Using the state table we construct truth tables for the next state flip flops and the outputs.

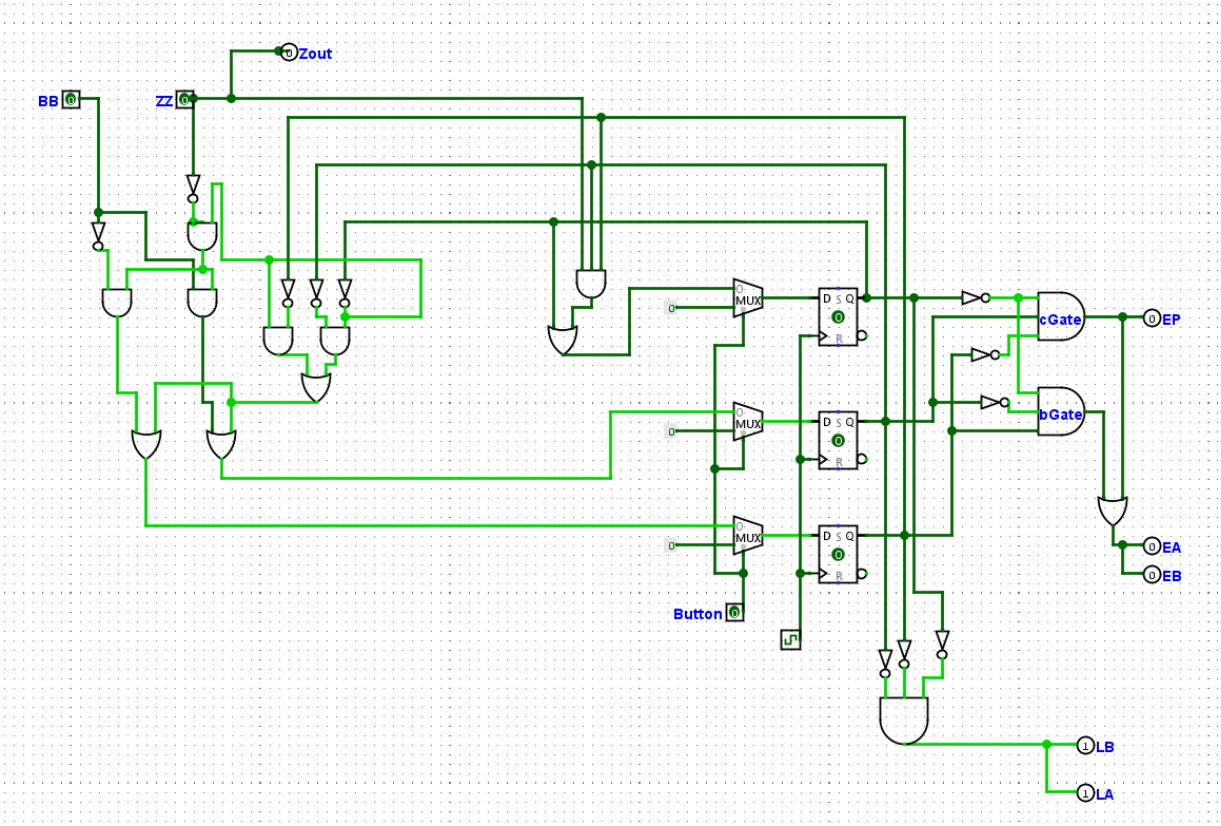
With the help of the Quine McCluskey algorithm, we get the following minimized logic expressions:

$Y2 = y2 + y1*y0*z$ $Y1 = \bar{y}2*\bar{y}1 + \bar{y}2*\bar{y}0 + b0*\bar{z}*\bar{y}2$ $Y0 = \bar{y}2*\bar{y}1 + \bar{y}2*\bar{y}0 + \bar{b}*\bar{z}*\bar{y}2$

$Psel = 1$ $LB = \bar{y}2*\bar{y}1*\bar{y}0$ $LA = \bar{y}2*\bar{y}1*\bar{y}0$ $EB = \bar{y}2*\bar{y}1*y0 + \bar{y}2*y1*\bar{y}0$

$EA = \bar{y}2*\bar{y}1*y0 + \bar{y}2*y1*\bar{y}0$ $EP = \bar{y}2*y1*\bar{y}0$

Finally, using 3 flip flops and gates for the combinational logic, we can implement the control unit. The implementation of the control unit in Logisim is found below.



Binary to BCD:

After getting the output in the product register, we needed to display it, but we were faced with the challenge of segmenting the bits into units, tens, hundreds, thousands, and ten thousands. We thought of multiple ideas, but they all seemed hard to implement. Until we found the infamous shift add 3 algorithms (double dabble). The algorithm converts binary representations of a number into a binary-coded decimal representation. It works using a shift and add method. Firstly, we consider each four digits as a representation of units, tens, hundreds and so on. In the final output of the shift add 3 methods from right to left. We then shift our binary digit one bit to the left, so we now have one bit in the digits row. Now the algorithm goes as follows for every

row: if the 4-bits in that row are greater than or equal five we add three to the digits and then we proceed with shifting left one bit, if they bits don't represent a number greater than or equal to five then we shift to the left one bit regardless. If we exceeded four bits because of our shifting left operations, then we are now in the tens row. We now examine the bits in the tens row. Of course one or two bits in the tens row will never be equal to exceed five and thus we shift until the tens row contains at least three bits and now we examine the bits, if they represent a number greater than or equal five (from left to right) we add three and shift, if they don't represent a number greater than five we shift to the left one bit regardless.

We continue in this fashion until we have one digit in the ten-thousands row for the 16-bit input or in other words until we have done exactly 15 shifts to the product represented in binary. Thus, we now have in every row our digits for the units, tens, hundreds, thousands, and ten-thousands. We had to choose between two ways to implement this algorithm, either using a sequential circuit with shift registers or a combinational circuit with add three circuits that decides if each four bits are greater than or equal to five or not. We chose the combinational option as it was easy to implement and debug. Moreover, after we get our output from our Binary to BCD converter, we then display each relevant digit (depending on the scrolling feature, which will be discussed below) onto a seven seven segment display using a BCD to Seven segment decoder, which will be discussed later as well.

Datapath:

Our first step in this project was to design the DataPath. The Datapath was designed similar to the unsigned shift and add sequential multiplier but with the added feature of detecting whether the product will be negative or positive. The multiplier and multiplicand are first loaded into their respective shift registers. The som button (start of multiplication) gives us the signal to start our algorithm. If the button was pressed, we check the most significant bit of both the multiplier and the multiplicand. If it is one, then the loaded number is negative. We then proceed to get their two's complement and load them into separate registers. We also perform an xor operation on the most significant bit of the multiplier and the multiplicand before getting the two's complement. If the result from the xor operation is one, then we know our product must be negative since we have a negative number multiplied by a positive one or vice versa. Thus, if the sign flag is one, we make sure to output a negative sign next to the product at the end of multiplication. To sum up, we eventually compute the absolute value of the product of the multiplier and multiplicand and then decide on the sign when outputting the product.

After finishing the loading phase of both the multiplier and multiplicand, we proceed to our shift and add algorithm. We have the product register initialized with zeros. Noting that the product register is 16 bits because the product of multiplying 8-bits by 8-bits can reach a number that is only representable by 16-bit. We first examine the least significant bit of the multiplier register if it is one then we add the product with the multiplicand and save the result in the product register, this gives us our first partial product. Afterwards, we shift the multiplier to the right one-bit and

the multiplicand to the left one-bit, and examine the least significant bit of the multiplier if it is one we add the multiplicand to the previous partial product stored in the product register, else if the least significant bit is zero we don't add anything to the product register. The same principle applies when we calculate our first partial product.

After repeating this algorithm several times, we need to stop, and according to our design we stop when the multiplier register is empty, i.e. fully loaded with zeroes. We set the flag eom (end of multiplication) to one when our multiplier register is empty and this flag is set to the output of all the bits of the multiplier register going through a nor gate, thus the eom flag will only be one if the multiplier register is zero. Of course, what controls these operations of when to stop, when to add, when to shift, and when to load is the control unit and it will be discussed thoroughly in the next section.

Phase 3 - Development of Verilog Module & FPGA Implementation: After having presented the demo using Logisim and received constructive feedback on how to enhance and improve the design, we started accounting for all missing components & including them in our design for the Verilog module. Please note that in the Verilog module, we accounted for both signed multiplication as well as the scrolling of the 7-segment displays which were missing from the design on Logisim. This following section will aim to explain the Verilog implementation of our project.

Attached here is the link to our github which includes all developed Verilog module files: https://github.com/HadjAhmed2003/DD_Project

In what follows we'll go through the top module, named "circuit", and explain the function of each block, or each module, and explain how each module works with the main circuit. If needed we'll dive into one of the instantiated modules to explain its contribution in the bigger picture.

```
22  module circuit(
23      clk, rst, multiplier, multiplicand, seg, an, btnc, eom, btnl, btnr
24      );
25      input clk, rst, btnc, btnl, btnr;
26      input [7:0]multiplier, multiplicand;
27      output [6:0] seg;
28      output [3:0] an;
29      output eom;
30      wire som, s1, sr, isNegative;
31      wire [7:0] abs_multiplier, abs_multiplicand;
32      wire [15:0] product;
33      wire [3:0] units,tenth,hundredth,thousandth,thousdandth2;
34      button start(btnc, clk, rst, som);
35      button shift_left(btnl, clk, rst, s1);
36      button shift_right(btnr, clk, rst, sr);
37      absolute abs(multiplier, multiplicand, abs_multiplier, abs_multiplicand, isNegative,som, clk, rst);
38      UM multiply(clk, rst, abs_multiplier, abs_multiplicand, product, som, eom);
39      binary_to_digits btd(product,units,tenth,hundredth,thousandth,thousdandth2);
40      bi_bcd bcd(units,tenth,hundredth,thousandth,thousdandth2,seg,an,clk,rst,s1,sr,som,eom,isNegative);
41  endmodule
```

Following the declaration of the inputs and outputs, we declare wires to carry:

1- The three button values

2- The sign of the product

3-The absolute values of the multiplier and multiplicand

4-The values of the units, tenth, hundredth, thousands, and ten thousandth, which will be extracted from the product binary representation using the bcd module.

Next, we instantiate three buttons, one for the start of multiplication, and two for the scrolling. We then use the absolute module to store the absolute values of the multiplicand and multiplier in their respective wires declared above. The module also, using an xor operation, determines the sign of the product, and stores it in the isNegative wire.

Then we have the algorithm itself, the shift and add algorithm, which uses the registers of the unsigned multiplier and multiplication and does unsigned multiplication. The module starts only on the push of the button start, and outputs 1 on eom to indicate end of multiplication.

Next in the code, we have the binary_to_digits module, which uses the double dabble algorithm (explained above in the Binary to BCD section) to extract from the 16 bit binary value of the product the values of the units, tenth, hundredth, thousands, and ten thousandth and stores them in their respective wires declared above.

Finally we have the display module, which is responsible for outputting the final result on the seven segment display. The module takes as input the BCD of the product through the 5 wires of the units, tenth, hundredth, thousands, and ten thousandth, the value of the scrolling buttons, and the sign of the product. The module uses the 4 following 4 modules as helper modules:

ClockDivider: Used as an input to the shifter and counter. Divides the frequency such as to allow for the counter to change the digit value at a suitable speed, and used as the clock for the scroller, which faced problems when it used the normal clock speed of the FPGA.

Shifter: The shifter is used for scrolling. It takes as an input the units, tenth, hundredth, thousands, and ten thousandth, and the values of the scroll left and scroll right buttons, and it outputs three wires named first, second, and third which hold the values that should be currently outputted on the screen.

Counter: This is used in the sensitive list of an always block to output the three digits of the product and the product sign at the same time.

SevenSegDec: This takes the binary value of a digit and outputs it on the seven segment display.

We're now done with the general description of the Verilog implementation. You can find below and in the source files of every module a more detailed description of each module specifically.

Verilog Modules Details:

Below is a list of all Verilog modules we created to achieve the desired functionality of the project with all its features. Each module will be discussed in detail below.

Module 1 - Circuit

Module 2 - UM (Unsigned Multiplication)

Module 3 - Absolute

Module 4 - Binary_to_BCD

Module 5 - Display

Module 6 - Button

Module 7 - ClockDivider

Module 8 - Debouncer

Module 9 - Synchronizer

Module 10 - Edge_Detector

Module 11 - SevenSegDec

Module 12 - Scroller

Module 13 - Counter

Module 1 - Circuit

This is the top module which executes the entire circuit. The module takes as input the clock and reset signals as well as three inputs btnc, btnl and btnr representing the 3 buttons. It also takes as input two 8-bit signed variables, the multiplicand and the multiplier. The output of this module is a 7 bit array representing the 7 segment display and a 4 bit array for the anode signals as well as an output signal eom representing end of multiplication as explained above. The circuit initiates the process by instantiating the module Button firstly when the user presses on the start button (BTNC) on the FPGA. This triggers the start of the multiplication. The circuit module also

instantiates 2 other instances of the button module; one for when the user presses on the BTNL button on the FPGA for the scrolling left and one for when the user presses on the BTNC button on the FPGA for the scrolling right. The circuit module also instantiates the module Absolute which returns the absolute of the multiplicand & the multiplier while keeping track of the output sign. Moreover, the circuit module instantiates the UM (Unsigned Multiplier) module and passes to it the absolute multiplicand & the absolute multiplier to conduct unsigned multiplication. The circuit module also instantiates the module Binary_to_Digits to convert the product into Binary Coded Decimal and it also instantiates the module Bi_BCD which is responsible for the final output on the FPGA. Accordingly, one can see that this circuit module is the main module controlling the flow of the circuit.

Module 2 - Unsigned Multiplication

This module is responsible for calculating the unsigned multiplication of two numbers. It receives as input the clock and reset and start of multiplication (som) signals as well as the multiplier and the multiplicand and it generates as output the product of the multiplication and an end of multiplication (eom) signal. If the reset signal is active, then all registers and variables are reset to their initial values. If the som signal is active, the multiplication process is initiated. The multiplicand register and multiplier register are loaded with the values of multiplicand and multiplier respectively. The product is set to 0, and eom is set to 0 to indicate that the multiplication is in progress. If the multiplier register is not zero, the multiplication process continues. If the least significant bit of multiplier register is 1, indicating a non-zero value, the product is updated by adding the multiplicand register value to it. Then, both the multiplicand register and multiplier register are shifted right and left respectively by 1 bit. If the multiplier register is zero, it means that the multiplication process is complete.

Module 3 - Absolute

This is a basic module that calculates the absolute value of two 8-bit numbers. If the reset signal is active, the sign variable is set to 0, ensuring a predictable initial state. If the som signal is active, indicating the multiplication has started, the sign variable is determined by the XOR operation between the most significant bit of the multiplier and the most significant bit of the multiplicand. This calculates whether the two numbers have the same sign or different signs. If neither the reset nor the som is active, the sign variable is not updated.

Module 4 - Binary_to_BCD

This is a basic module that basically implements the double dabble algorithm as explained above in the previous section in details.

Module 5 - Display

The display module combines the scroller and the SevenSegDec module to display the final result. It stores in wires “first”, “second”, and “third” the three digits that should be outputted to

the seven segment display. It also takes as an input the product sign. Using these four values it can instantiate the SevenSegDec module and pass to it a digit value. Then using the fast counter implemented previously it can change the digit value at a pace coordinated with the same pace at which the anode active value changes in the SevenSegDec module implementation such that it will output the three digits as well as the product sign at the same time

Module 6 - Button:

This is a basic module responsible for generating output signals based on the input which is the click of a button on the FPGA be it the start of the multiplication through clicking on BTNC or shifting left or right through BTNL or BTNR. In this module, we instantiate a clock divider, a debouncer, a synchronizer and an edge detector module. The button module is the one coordinating the functionality of the modules instantiated inside it to handle input debouncing and ensure reliable signal detection.

Module 7 - ClockDiver / Module 8: Debouncer / Module 9: Synchronizer:

These are the complementary modules providing the basic functionality as the name of the modules imply. The clock divider module basically produces a slower clock while the debouncer module removes glitches and noise from the input signal generating as output debounced version of the input signal. The synchronizer then synchronizes the input signal to the clock domain and its output represents the synchronized version of the input signal.

Module 10: Edge Detector:

This module implements a FSM that detects an edge

Module 11: SevenSegDec:

This is a basic module similar providing the functionality of displaying to the seven segments as explained above in previous sections in detail

Module 12: Scroller

The module scroller is, as the name implies, used for scrolling through the digits of the product. The module uses a 2 bit wire named sel to indicate which three of the five digits should be displayed. A value of 0 indicates that the units, tenth, and hundredth will be displayed. A value of 1 indicates that the tenth, hundredth, and thousandth will be displayed. A value of 2 indicates that the hundredth, thousandth, and ten thousandth will be displayed. sl (to scroll left) and sr (to scroll right) are the buttons for scrolling and they are two of the inputs of the module. If the scroll right button is pressed we add 1 to the value sel. If the scroll left button is pressed we subtract 1 from the value sel. This change eventually scrolls the digits. Note that we have a maximum value of 2 and a minimum value of 0 for the value sel. That is because after scrolling right from the most left two times, we can’t scroll anymore. Same goes for the left scrolling

Module 13: Counter:

This is simply a modulo 4 counter which counts at a high frequency. The same counter is used in the SevenSegmentDec module for the always block that changes the anode active at a sufficiently high speed. This is also used in the sensitivity list of an always block in the bi_bcd module to change the value that will be outputted on the seven segment display at a high speed, such that all the digits and the product sign are outputted.

C- Implementation issues:

One major issue we faced during implementation was that the buttons wouldn't start the multiplication. We examined the whole code from beginning to end but couldn't quite figure it out. Until we noticed that we hadn't included a reset button in the module that initializes the buttons, this in turn affected our denouncer, synchronizer and edge detector modules. We also faced the issue of the scrolling being too fast, so we had to include a clock divider to the shifter so that it shifts the digits slower.

D - Validation Activities:

One of the validation activities that we implemented is that the scrolling feature does not operate if the user does more than 2 clicks on the same button to shift either shift left or right. Also as part of the validation we conducted, we tried out numerous test cases including multiplying positive multiplicand by a positive multiplier as well as multiplying a negative multiplicand by a negative multiplier and also a negative multiplicand by a positive multiplier and finally a negative multiplicand by a negative multiplier. We ensured that the product for each of the four cases is correct and accounting for the change in sign. Finally, the product register was designed as 16 bits in order to account for the maximum value of the multiplication result and to avoid overflow.

E- Team Members Contributions

For most part of the project, the four of us worked together equally as a team all the way from the brainstorming phase initially to the development of the simulation on Logisim to the final development of the Verilog modules & FPGA implementation. Thomas and Mahmoud mainly worked more on the development of the initial block diagram at the beginning of the project. Raafat and Hadj then worked more on the development of Logisim simulation. Then we all again brainstormed for how the Verilog modules will be developed after receiving the feedback from the demo we presented. For the actual coding, we divided the module amongst ourselves. Hadj and Mahmoud worked more on the development of the modules dealing with the output of the multiplication and the shifting features. Raafat and Thomas worked more on the development of

the modules relating to the input of the buttons and signal processing. The module pertaining to the multiplication itself and the main circuit module was a collaborative team effort.