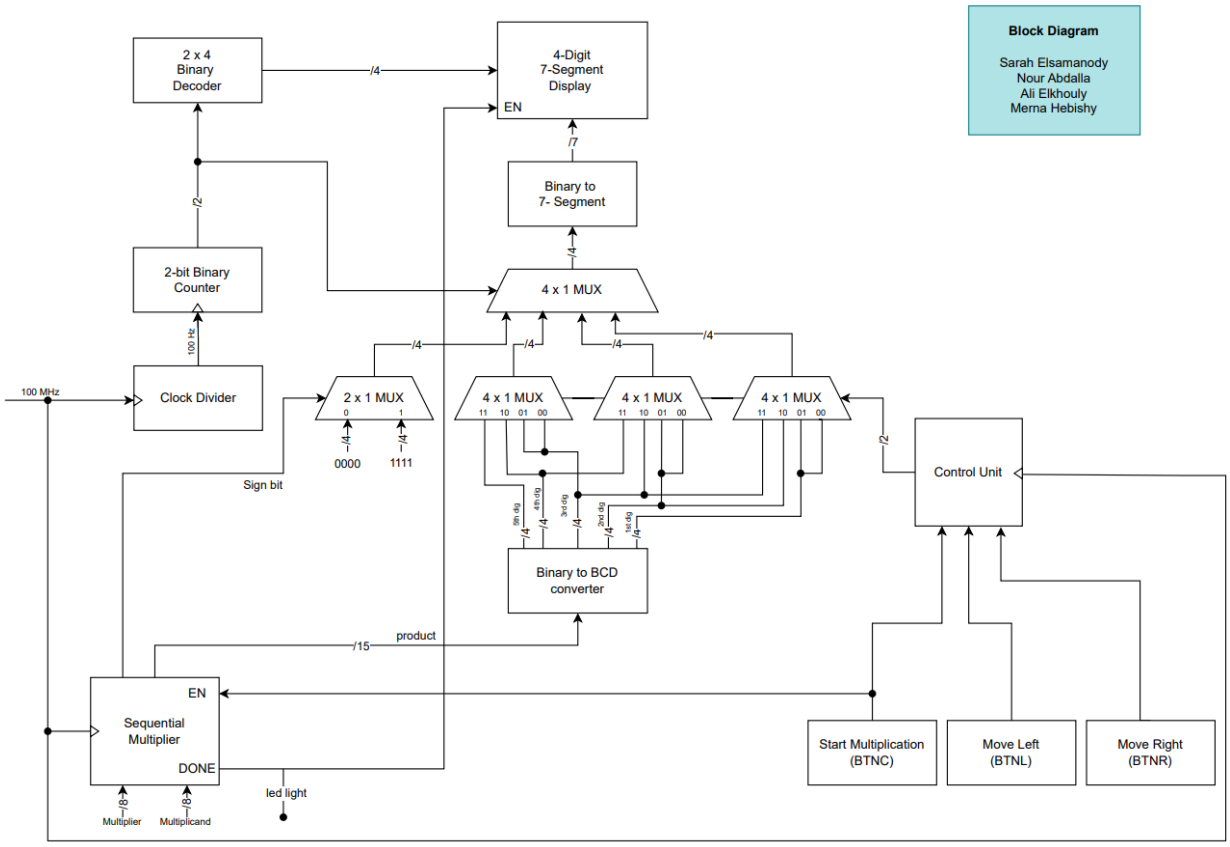
8x8 Signed Multiplier Report
Digital Design I

Ali Elkhouly 900212679
Sarah Elsamanody 900212915
Merna Hebishy 900221976
Nour Abdalla 900213024

Description:

In this project, we designed a sequential signed multiplier that takes 2 signed 8-bit inputs and produces a signed 16-bit product. We used the Basys-3 FPGA board to simulate the calculation, using its components to input and output the product in a suitable format so that any person could use them. We mimic the way we would multiply 2 binary numbers on paper by utilizing adders and shifters in the FPGA. We also divided our work into 3 parts: Block Diagram; Logisim Circuit; Apply to Verilog.

Block Diagram:



IMPORTANT NOTE: Since this step was to check everything new we had to implement in our circuit/modules, we excluded anything we already built and designed in the CSCE2302 course. E.G, the push button detectors. Also this block diagram has been altered as we went on to the next stages, so the final circuit might not 100% resemble this diagram.

- ***Control Unit***

  We have a control unit that takes four inputs: the button that signals the start of multiplication (BTNC), the button that moves the digits displayed one digit to the left (BTNL), the button that moves the digits displayed one digit to the right (BTNR), and the clock. The output of the control unit is the state that is used to select the digits displayed through the multiplexers. We needed this since the product of 2 8-bit numbers would produce a 5 digits

decimal number, and we only have 3 places to display the numbers, excluding the sign display, we needed a selection line to decide on which digits are going to be represented.

- Sequential Multiplier

  We also have a block for the sequential multiplier which takes the eight bit multiplier, eight bit multiplicand, BTNC signal from the Control Unit as enable, and the clock. It has three outputs: done, the 15 bit product, and the sign bit. The done changes depending on whether the multiplication is complete or not. Its signal lights up the LED if it's 1. The product is then sent to a binary to BCD converter so that it is split into digits.

- BCD2Binary

  This module converts the binary product from a binary number to a BCD representation so that we could display it on the 7-Segment-Display. It outputs five digits, each four bits and each representing a digit that is the result from the product.

- Multiplexers

  Based on the output of the control unit, only three digits will be selected at any given time, depending on which buttons have been pressed. In state 00, the first three digits from the right will be selected. State 01 is the same as state 00 because it is an unnecessary/don't care state. State 10 selects the second, third, and fourth digit. State 11 selects the third, fourth, and fifth digit. The control unit takes into account the present state and the buttons pressed to choose what the next state should be. The sign bit is used as a select line to choose whether the input will be positive or negative. If it's positive, it will choose 1010; if it's negative, it will choose 1111. The three digits selected and the four bits selected by the sign bit are inserted into another multiplexer, with the output of a two bit binary counter as a select line. The output of the final multiplexer is a four bit binary number that is taken as input by the binary to 7-segment block

- Clock Divider + 2-bit Binary Counter

  This output selects a different digit in the multiplexers with every count. We have a clock divider that divides the clock frequency by a factor so that the display could be visible to the human eye as a stream of numbers. The two bit binary counter takes the divided clock output and outputs two bits used for the 2x4 binary decoder and the multiplexers mentioned above.

- Binary to 7-Segments

  To convert the binary number into seven bit segments, we built this module. It takes the BCD digit and outputs the digit in the 7-segment format. These segments will be placed into the 4-digit 7-segment display so that it appears as digits on the board.

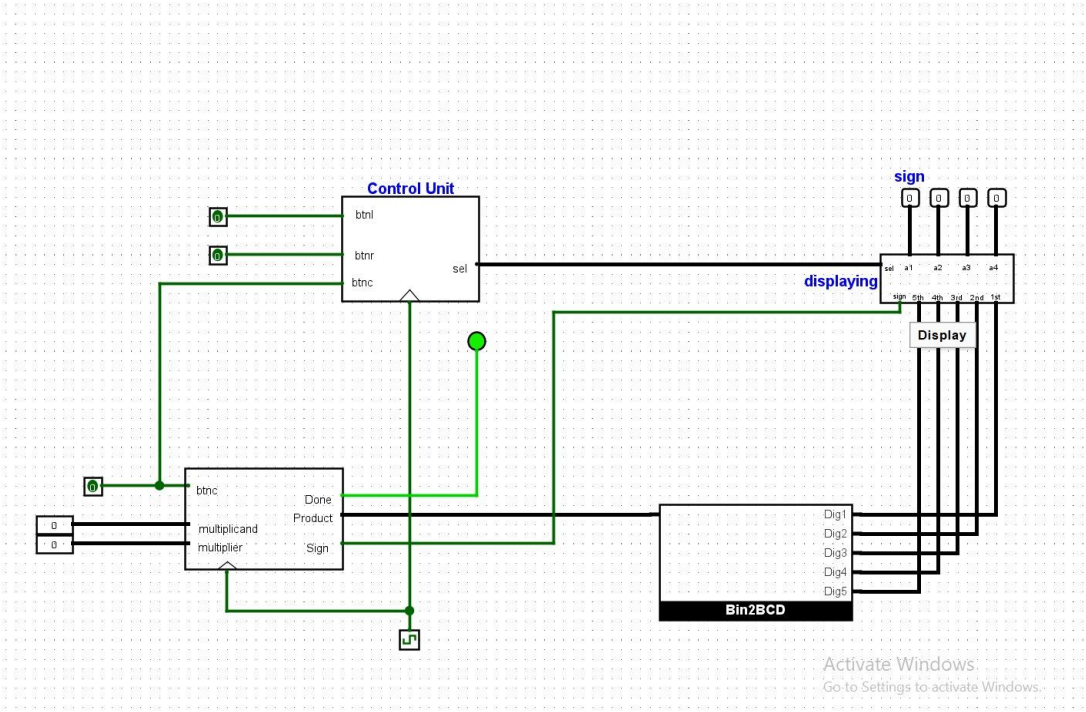
- 4-Digit 7-Segment Display

  This module takes the digit to be displayed from the Binary to 7-Segments module, also takes a 4-bit from a Decoder to decide on which digit will be visible. To enable the 4-digit 7-segment display, the done signal from the multiplier must be 1.

- 2x4 Binary Decoder

  A decoder that takes the counter from the 2-bit binary counter as input, and depending on it outputs a 4-bit binary value that decides on which display on the 7-Segment Display will appear.

Logisim Circuit



This diagram shows the top module of the circuit we implemented on Logisim including all necessary sub components needed to implement the signed multiplication.

How to Run the Circuit on Logism:

In order to run the 8x8 signed multiplier on Logisim we need the following steps:

1. Download & Open our circuit
2. Write in the input pins on the bottom left the multiplicand and multiplier that you wish to multiply
3. Change the input pin of the BTNC to start multiplication
4. Manually press on the clock twice to pass one clock cycle then close BTNC again
5. Then, flip the clock constantly until the red light, done, turns green
6. Since only three digits are displayed, the user can use the two pins BTNL and BTNR to scroll left and right through the output.
7. The most significant number is the sign which is 0 if the product is positive and 1 if the product is negative.

To get a closer look at our sub circuits, the user can press on the desired circuit and the circuit's file will open up, previewing the circuit selected.

Obstacles faced in Logisim:

1. Building the Binary to BCD converter.

2. Connecting the lines without overlapping.
3. Windows & Mac compatibility.
4. Sending files and working simultaneously.

How we overcome these problems:

1. Building the Binary to BCD:
    a. Had to research and learn an algorithm to build this block.
    b. After learning the Double Dabble algorithm, we had trouble implementing it on Logisim, as it also needed a comparator that would add 3 to a number larger than 4.
    c. Built a block in Logisim that would do the latter algorithm and just repeat & use it.
2. Connecting the lines:
    a. Whenever we connected the lines, some would overlap, and some outputs would get wrapped up wrongly.
    b. Checked in the simulation the input and output of each component, and then fixed each wiring and output position in each black box accordingly.
3. Windows & Mac compatibility:
    a. This one was difficult since we couldn't also use the Lab's computer since installing Logisim would've meant needing the access of an Admin.
    b. We had to share one of the laptops and use Zoom Remote Control to the computers.
4. Sending files and working simultaneously:
    a. Had trouble simulating the files we sent to each other, since some files would be missing and would not work properly. This would've also meant working simultaneously would be more difficult.
    b. We utilized the Libraries option, and made a different library with the components needed for the whole circuit and sent it so we could implement it on the whole project.

Verilog Components:

Clkdiv

This module is the clock divider. It takes in a specified clock frequency/input and returns a clock signal that is the divided frequency of the input. In the module, it calls a counter and uses n parameter as the modulo for the counter. The clock won't change signals until it reaches the required frequency.

Bin_counter_nbits

This module is a counter for any number of bits. It only increments when reset is 0 and enable is 1. When the counter reaches n-1, it restarts and begins from 0 again.

Debouncer

This module is the debouncer. Its purpose is to filter out glitches associated with switch transitions. It makes sure that the signal is stable before returning the output z. If reset is 1, it will return 0. Otherwise, the input x is placed into q1 which is then placed into q2 in the next clock cycle and then placed into q3. When all the q's are 1, z will be 1.

Synchronizer

The synchronizer receives an asynchronous input sig and produces an output sig1 within a restricted amount of time.

Bin2bcd

The bin2bcd module converts a binary number to a binary coded decimal using the double dabble algorithm. This means that the binary number will be split into 5 binary equivalents of the digits of the decimal number.

DigCounter/DigSel

This module is a combination of the clock divider and the 2-bit binary counter modules. It produces a variable called dig that is used to select which digit will be chosen from the multiplexer. The dig could be 00, 01, 10, or 11. This will be the select line of the multiplexer and the input for the decoder that decides which 7-segment display will be turned on from the 4 displays.

Left_shift_reg

This is a left shift register. It takes in a signal to load, an enable, a clock, and the multiplicand. It outputs the shifted multiplicand. When load is 1, the multiplicand is stored in the shifted multiplicand. When it is enabled, it shifts once to the left and a 0 is added to the right.

Right_shift_reg

This is a right shift register. Similar to the left shift, it takes in load, enable, clock, and multiplier. It outputs the shifted multiplier. The multiplier is loaded into the shifted multiplier when the load is 1. Otherwise, when it is enabled, we add a zero to the left and shift one to the right.

n_bit_register

This is an n-bit register. In this case, it takes 16 bits of data and returns 16 bits of quanta. If reset is 1, we put 0's into quanta. Otherwise, we put the data in quanta for every positive edge of the clock.

Rising_edge_detector

The rising edge detector checks and detects the rising edge of a pressed button. We do this by making a finite state machine. We start at state A where no active high signal has been received yet. If we do receive one, we move to state B. The output z will be 1 if we're in state B and reset is not 1 since we received an active high signal. When we receive an active low signal, we move back to A. If we're still receiving 1's, we go to state C. We stay in C until we receive an active low signal for which we move to state A. This is so that we transform the positive edge of the button only, as humans would press on buttons for a long time and we only need a simple signal. So this transforms a long press into a small one.

Push_button_detector

The push button detector checks if we have pressed a button or not. It takes in clock, reset, and x. It outputs z. We call the clock divider, debouncer, and synchronizer to make sure we're taking a stable

signal. Then we call the rising edge detector to check when we receive an active high signal which indicates that a button has been pressed.

Buttons_control_unit

This module acts as the control unit. We have 4 parameters in this module to represent each state. State R, presented by the binary value 2'b00 which represents the reset state, which means we didn't start the multiplication yet; State A, presented by the binary value 2'b01 which means we just finished the multiplication, and it prints the least significant numbers of the product; State B, presented by the binary value 2'b10, which means we shifted the digits 1 step to the left; State C, presented by the binary value 2'b11, which means we shifted the digits 2 steps to the left.The Control Unit has the clock as an input, as well as three buttons, and outputs the current state. If we click on the start button (btncZ), we move to state A. Otherwise, we stay in R. Once we're in state A, we check if the left button was clicked (btnlZ) while the right button was not clicked (~btnrZ). If that is the case, we go to B. When we are in state B, we check the same conditions as for A and move to C. If we press on the right button and not the left button, we go back to state A. Otherwise, we stay in B. After we're in state C, we check that the right button is pressed and that the left button isn't. If that is the case, we go back to state B. Otherwise, we stay where we are. The default state to be in is R. If we're in state A, we see the first three digits on the right. If we're in B, we see the middle three digits. If we're in C, we see the leftmost three digits. The states change at the positive edge of the clock, and if the start button is pressed, the state becomes A.

Numsel

Ths Numsel module acts as the five multiplexers in the block diagram. It chooses which three digits will be displayed as well as the sign. It takes 5 digits, each 4 bits, sign, state, and seldig. It outputs a 4 bit digit after selection. Based on the state, we assign a digit to each of the three multiplexers. State 00 and 01 have the same assignments: the first three digits from the right. State 10 assigns the three middle digits to the multiplexers, and state 11 assigns the first three digits from the left to the multiplexers. Based on seldig, one of the three digits stored in the multiplexers will be the output, including the sign. In our implementation, we decided to let 1010 represent positive and 1111 represent negative. The digits should be chosen consecutively quickly so that they all appear at the same time.

Signmult

The Signmult module is the multiplier. It takes the eight bit multiplier, eight bit multiplicand, start, state, and the clock. It has three outputs: done, the 15 bit product, and the sign bit. When the BTNC button is pressed, start is 1. First, we either select the multiplicand or the two's complement of the multiplicand based on the first bit from the left of the multiplicand. This choice is placed into the selmultiplicand wire. The same process is repeated for the multiplier and it is placed into a selmultiplier wire. We check the completion of multiplication by seeing whether the output of the right shift register is 0 or not, also that we are not at State R. If it is, then we are done. If there are still 1's in the right shift register, then we are not done. Afterwards, we call instances of the left shift register and the right shift register. We load the left register with the selected multiplicand when the start button has been pressed. The register is enabled when we're still not done with the multiplication. The output, the left shifted multiplicand, is stored in the lshifted wire. The same happens with the right shift register, except it takes

the selected multiplier as input. We check if the first bit from the right of the shifted multiplier is 0. If it is, we assign wire A to be 16 bits of 0 since we're multiplying by 0 (a zero row). Then, we add A to B, which is empty the first time around, and assign the output to wire C. C is inserted into a 16 bit register which outputs B, the accumulated sum. Whenever we click on the start button, this register resets. The product is assigned C when the first bit on the right from the shifted multiplier is 1; otherwise it is assigned 15 bits of 0. The sign bit is the XORing of the most significant bits of the multiplier and multiplicand. Regarding the sign bit, we first check if any number of ours is 0, if they are, then the sign bit states 0 (positive).

SevenSegDecWithEn

The seven segment decoder takes a 4 bit number, 2 bit digit, and an enable as input and outputs 7 bit segments and 4 bit anodes. When it is enabled, it will select which anode will be turned on based on the digit. The number will be used to select which segments should be turned on. To show the sign of the number as well, we chose 1010 to be positive (no segments will be lit up) and 1111 to be negative (only the middle segment will be lit up to act as a minus sign).

Signed_Multiplier_8by8

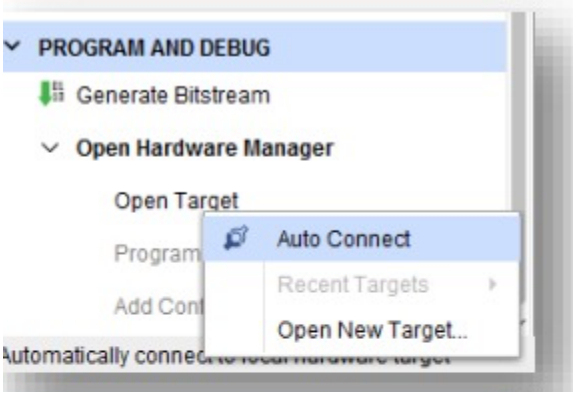
This module combines all the modules to create the desired result on the board. It takes the clock, multiplier, multiplicand, start, scrollL, and scrollR as inputs. The button to scroll left is scrollL and the button to scroll right is scrollR. The outputs are done–which is connected to LD0–, 7 bit segments, and 4 bit anodes. Done lights up the LED when the multiplication is complete. We call three instances of the push_button_detector for each button; btncz is when start is pressed, btnrz is when the right button is pressed, btnlz is when the left button is pressed. Then, we call an instance of the clock divider so that clock out is given to the control unit. The control unit returns the state through a two bit wire named sel. We insert the multiplicand, multiplier, btncz, and clock as inputs and receive done, product, sign, and sel. The product is converted into a binary coded decimal through the bin2bcd module instantiation. We assign each of the 5 digits 4 bits of the BCD output. The numselector chooses the digits based on the state and outputs a digit based on seldig. The DigSel module gives seldig only after being enabled. The DigSel is only enabled when the multiplication is over. Finally, the seven segment display module is called to display the digits on the board.

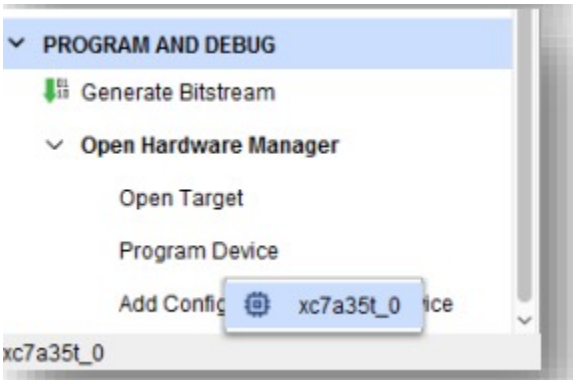
How to Run It:

In order to run the 8x8 signed multiplier we need the following steps:

1. Download our program
2. Open it on Vivado and then generate bitstream

3. After it is done, press auto connect and program your FPGA





4. The input of our multiplier comes from switches 0 to 7
5. The input of our multiplicand comes from switches 8 to 15
6. Both multiplier & multiplicand are signed numbers represented in 2's complement so their range is -127 to 128
7. Then, to start multiplication we press BTNC
8. It then will run multiplication and the 7 segment display will light
9. The most significant digit is responsible for the sign of multiplication
10. The other three show the number and since the maximum value from multiplication of 2 8 bit numbers are 5 digits our program enables scrolling
11. Using BTNL and BTNR, we can view the remaining values of the product on the FPGA
12. After the multiplication is done the least significant led light turns on
13. If you want to run another multiplication, change the values of multiplier and multiplication using the switches, then press BTNC over again to run a new multiplication

Obstacles faced in Verilog:

1. Getting familiar with debugging different modules on Vivado.
2. Synchronizing the clock through all modules.
3. Combining modules together and passing the right connections.
4. The push button detectors worked sometimes and sometimes did not work.
5. Integrating the control unit to accommodate all states.
6. The LED LD0 would light up before starting the multiplication, not only after it's done.
7. Multiplying a negative number with 0 would produce a negative 0, which is logically wrong.
8. The Counter and Clock Divider wouldn't produce any values.

How we overcome these problems:

1. Debugging on Vivado:
    a. Kept debugging and training on Vivado and Verilog syntax.

    b. Asked the Teacher Assistant for help whenever needed in syntax or how to implement certain ideas in Verilog
    c. Researched algorithms and methods to implement our ideas in Verilog.
    d. Used Testbenches to test each module and check the input and output of each module after implementing them all together in a single module.
2. Synchronizing the clock:
    a. Tested the clocks by going through the Testbench.
    b. Gave the testbench a realistic initial clock, by calculating the clock of the FPGA into a period time, and making the clock of the Testbench change after each period of that time passes by.
3. Combining modules:
    a. Had errors in the combining stage.
    b. Checked that each module in the Top module had a name and correct connections.
    c. Checked for the sizes of each wire and register used in the Top module and that is being passed around in the modules.
    d. Made sure that the registers and wires that are being passed around are the correct logically and syntax-wise as well.
4. Push Button Detector:
    a. Had problems that it did not always detect when the buttons were pressed.
    b. Used Testbenches to check the input and output of each button and button detector.
    c. Realized that the problem is that when the button is pressed, the positive edge of the clock does not intersect when the button is on, so the Control Unit wouldn't change states.
    d. Decreased the frequency of the Control Unit to make it match with the outputs of each push button detector.
5. Integrating Control Unit for All States:
    a. The Control Unit was giving false states and alternating rapidly.
    b. Checked for the wiring and fixed it by switching the inputs and outputs.
6. LD0 lighting up before starting multiplication:
    a. Checked the wiring and input signals through the Testbench.
    b. Realized that done is technically true (1) because the right register would be 0 before the multiplication.
    c. We sent the state as an input to the multiplier, and made it check that we are not at state 00–which is the signal that says we did not press the start button yet– before changing the done to 1.
7. Negative result when multiplying negative with 0:
    a. Realized the problem would be since XORing the most significant bits would produce a negative sign, which would be sent and make the answer logically wrong.
    b. Made an always block that checks if one of the numbers is 0 or not. If one of them is 0, then the sign bit will always be 0.
    c. Also to change the sign segment from changing whenever we change it in the FPGA's input, we added an extra condition that the user must press the start button for it to change.
8. Clock Divider and Counter wouldn't work:

a. We checked through the Testbench, and realized that since we did not start with a reset, their initial values would be an X (a don't care).
b. We initialized the values of the counter and clk_out inside of each to 0 with an initial block, so now we can increment and change their values.

Dividing up the work:

We want to highlight that we all contributed together and helped each other in our tasks.

1. **Block Diagram**
   All members contributed to the designing of the block diagram.
2. **Logism**
   a. Sarah & Merna : Implemented the sequential multiplier which included: Left shifter, Right shifter & 2's complement was done with the help of Nour & Merna.
   b. Nour: Implemented the displaying unit & CU of the buttons.
   c. Ali: Implemented the double dabble algorithm
   d. We then all contributed in the integration of the separate elements of the circuit
3. **Verilog HDL**
   We decided that each member of the group should work on the module for the parts they did in Logisim to ease up the process.
   a. Sarah & Merna : The multiplier module with the shifters, 2's complement & register file.
   b. Nour: The control unit
   c. Ali & Merna: They gathered all modules needed for displaying
   d. Ali: Worked on the module of binary to BCD
   e. Sarah: Implemented the top module
   f. Nour & Ali: Worked on the debugging of the code after combination

On the whole, there was equal distribution of the work that made the entire process efficient and all team members were open to discussions and explained to one another the missing gaps, so we all grasp the entire concepts within the project.