Fixed-Point Calculator

Robert Kozubiak, Muris Zecevic, Cameron Renny
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
rjkozubiak@oakland.edu, mzecevic2@oakland.edu, cgrenny@oakland.edu

***Abstract*— The goal of this project is to create an unsigned fixed point calculator that takes 2 input numbers with an integer part, fractional part, or both, and perform simple mathematical functions with them, such as addition, multiplication, subtraction, and division. Peripherals utilized include the**

I. INTRODUCTION

This report will cover the specifics on how a fixed-point digital calculator will operate utilizing VHDL coding and the Nexys 4 FPGA board, the peripherals that will be needed, and challenges encountered while in the process of creating a functional end product.

The motivation for this project is to gain more experience on how to write and implement algorithms, in the form of VHDL coding, on computers with practical uses. This is a crucially important topic, as algorithms are the building blocks of every computer program, and this calculator will be a sufficient introduction to how to effectively design these algorithms. This project will reflect many of the topics utilized in Daniel Llamocca's course, Computer Hardware Design, such as fixed-point arithmetic and how to implement peripherals such as a keyboard/keypad or LCD screen. The end-product of this project is a functional calculator that does fixed-point arithmetic on fractional decimal numbers by converting the numbers to binary before operating on them.

This project utilizes around 30 unique components throughout the entire project. The project takes usage of Daniel Llamoca's 'mydebouncer' and 'my_genpulse_sclr' codes for purposes of debouncing the keypad and counters respectively. The designs for the multiply and divide modules, although coded separately, are also designed by Llamocca [1]. The project also reflects philosophy designs found in the Hitachi HD44780U datasheet for the LCD FSM designs [2]. The design also uses popular ideologies on keypad interfaces such as one found on Astro Designs [3]. The end result is a calculator interface that can perform operations on two fixed point numbers, including a range of numbers from 0 to 99, with two decimal values of precision.

II. METHODOLOGY

A. Input Interface

The decimal numbers will be inputted by the user with a keyboard/keypad, and will be passed onto the Nexys-4 DDR board. The keypad should generate an 8-bit binary value. This is done through a keypad controller. This controller contains a FSM. In this keypad FSM, there are 16 states, one for each button. This keypad has four inputs, the rows, and four outputs, the columns. For the first four states, the keypad FSM turns on the leftmost column. Then, if a button is pushed while that column is on, the FSM freezes for as long as the key is pressed, and an 8-bit signal is outputted. After the first four states, the column to right is turned on for four states, and so forth. This causes a rotation of one column being turned on every four of the sixteen states. Again, the end result is an 8-bit signal that is unique for every button pressed.

As mentioned, this keypad controller will produce an 8-bit signal that will pass through what is currently a multiple state input FSM interface, which will allow the user to input any two two-digit numbers, be they decimal or integer values. The FSM is designed as follows.

In State 1, the user will be able to input the first number, Number A. First, the system will check if the counter for A is currently equal to 2. If it is, the user will only be able to input A, B, C, or D, which are the operation buttons. If the counter is not equal to 2, the user will input, along with the operation buttons, the value buttons, 0 to 9, for the first number, A. Number A, along with Number B are both 2-digit decimal numbers, ranging from 0 to 99, including decimal numbers with two decimals of precision.

As mentioned, the system first checks if the user has inputted two numerical values already. This check is done through a counter, CounterA. If this is true and the user has already inputted two numbers for A, the FSM only allows the user to input an operation button. Once the operation button is pressed, the FSM moves into the next state, which is where the user will input number B. If the user hasn't inputted two number values, the user must input either input more numbers, or an arithmetic operation. A check is also placed to make sure a key is being pressed. If a key is being pressed, an 8-bit signal should be sent to the FSM by the input module, the keypad controller. This 8-bit input is then identified by the input FSM.

If the user hasn't inputted two values and the 8-bit input is recognized as a number value, then a 4-bit signal, representing the numbers 0-9, is set. This 4-bit signal is sent to a shift register that shifts in 4 bits at a time. The output of the shift register is 16-bits. These are 16-bit outputs because of the shifting FSM, which lays the foundation on how to

handle decimal placements for A and B, which are discussed further in this report. Also, the counter that is associated with number A, CounterA, is incremented by one. If the user input is a decimal, which is the # button on the keypad, the system goes to State 1dec, which deals with the decimal portion of the number. If the input is an operation input, the system advances to State 2. Just as well, the operator that was inputted determines a 2-bit signal that is saved into a 2-bit register. The identification of each operation can be seen in Figure 1.

Operation	Meaning	Value
A	Addition	00
B	Subtraction	01
C	Multiplication	10
D	Division	11

Figure 1 – The identification for each operation

If the input is anything other than an arithmetic operation, the decimal input, or a number value input, then nothing happens. Likewise, if the user has already inputted two numbers and tries to input another number, nothing happens. Instead, the user remains in the first state of the FSM.

State 1dec works in a similar manner to State 1, with one key difference. If the input is a number, the number is added to the shift register for A, and not only CounterA, but also CounterAdec is incremented by one. CounterAdec serves to keep track of the number of decimal places in the number. If the input is an operator, the system advances to State 2, and the operator that was inputted generates the 2-bit signal to be saved into the register with the same values as shown above.

State 2 and 2dec are the same as State 1 and 1dec where the number B is inputted and the same functions that were done on number A are also done on number B, with some differences. First, if the input is an arithmetic operation, then nothing happens. Another difference being that if the input is the 'equals-sign' input, the system moves onto state 3. This is where the input FSM ends for now, until the entire process is finished. Once the output is given, the user can then reset the entire project with the press of any button.

The input portion of this calculator design can be seen in Figure 2. It is worth mentioning that this figure is only a portion of the top file diagram used in this calculator design project. Because so many components are utilized in this project, it is absolutely necessary to divide the top diagram up into smaller portions, such as the one in Figure 2.

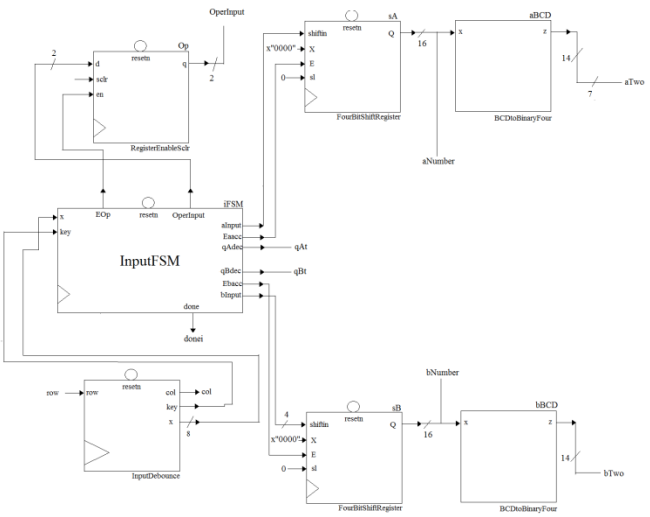

Figure 2 – The input portion of the top file diagram

As mentioned, the shift registers each output a 16-bit sized std_logic_vector value that represents each input, A and B. These are BCD binary numbers, and must be converted into binary values. This is done by the BCDtoBinary module. This module will check the four most significant bits of the 16-bit input. It determines what value, again from 0-9, these four bits are. It then sets a signal, z4, to a binary number value. Then, the program moves on the next four significant bits, and sets z3, and so on for z2 and z1. The output, z, is an addition of all four signals, z4, z3, z2, and z1. So, if the 16-bit input is a hexidecmial value of '9999,' z4 is set to '10001100101000,' z3 is set to '00001110000100,' z2 is set to '00000001011010' and z1 is set to '00000000001001.' The end result is the 14-bit sized output, z, which is a value of '10011100001111,' which is the binary representation for 9999.

Because the user can only input two digits for each number, the initial values of Number A and Number B can be no larger than 99, which is a 7-bit binary number. Because of this, the initial 14-bit values can be cut down to 7-bit values, leaving the user with two binary values, bTwo and aTwo. These will be used later in the design for the multiplication module.

One more aspect of the input FSM worth noting is that if a valid key is pressed, the FSM actually goes into a intersect phase. In this phase, nothing happens until 20 ms have passed. If the key is still being pressed, then another 20 ms must pass. Otherwise, the key has been registered as being depressed, and the FSM can continue as normal. This is a manner of "debouncing" the incoming signal from the keypad. Figure 2 shows part of the FSM algorithmic state diagram for the input FSM.

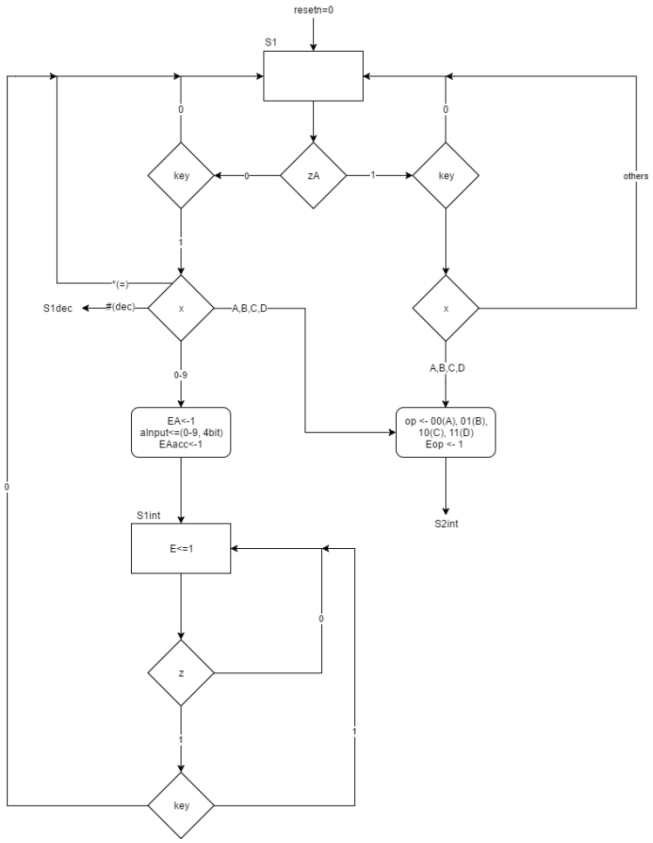

Figure 3 – A portion of the input FSM. State 1dec and 2 work in similar fashion

B. Shifting Interface

As mentioned, once the numbers A and B are inputted, the program should move into the next state, which is properly aligning the two numbers for addition, subtraction and division, based on how many decimal inputs were conducted. For example, if the numbers "3.3" and ".33" are inputted, the shift register values will be, in hexadecimal BCD format, "0330" and "0033". The decimal counters will be 1 and 2, respectively. This shifting FSM will adjust these values so that the decimal points properly align, so the new shift register values are '0330' and '0033.' This is the reasoning for the size of the 16-bit shift registers. If the shift registers were 8-bit sized, then the project design could not account for decimal values. With 16-bit shift registers, the program can properly account for any decimal number between 0 and 99.

Once the system is initially finished with the input FSM, The system then moves to the shifting FSM. This FSM doesn't start until the input FSM is finished, as the first state checks to make sure the input FSM has sent its 'done' signal to the shifting FSM. The first actual process of this FSM is to load the 16-bit BCD contents of Number A and Number B onto new 16-bit shift registers. These registers will serve as the shifted values for Number A and Number B.

Next, the system checks the operation and the value of both decimal counters for Number A and B, qAdec and qBdec, to determine how to arrange and align the numbers.

If the operation is multiplication, the result's decimal count, qR, is (qAdec + qBdec), and the shifting FSM is done. No shifting is needed for multiplication, as the multiply module deals with the 7-bit non-shifted numbers discussed earlier.

If the operation is addition/subtraction, the system compares the values of qAdec and qBdec. If qAdec is larger than qBdec, then qR is equal to qAdec, and Number B is shifted by a number of zeros equal to (qAdec - qBdec). If qBdec is larger than qAdec, then qR is equal to qBdec, and Number A is shifted by a number of zeros equal to (qBdec - qAdec). If they are equal, no shifting is needed, and qR is equal to qBdec, which is equal to qAdec.

If the operation is division, then the numbers are shifted the same way as they are if the operation is addition or subtraction. However, qR will always be set to 0. This is because the division module will take two numbers and give a quotient and a remainder. Only the quotient will be considered, so shifting is done to get the most out of the division module in the simplest manner possible. Figure 4 shows the FSM diagram for the shifting interface.

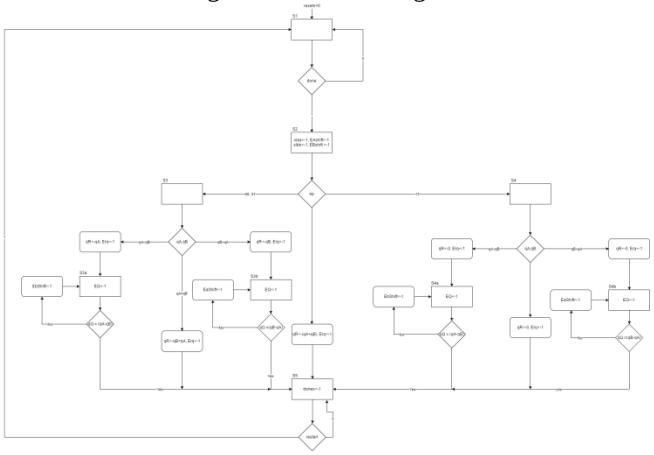

Figure 4– The FSM diagram for the shifting interface

After the shifting FSM is finished shifting and setting qR, then the shifting FSM is finished, for now. The shifting FSM, much like the input FSM, can be reset once the entire process is complete, if the user inputs any button on the keypad afterwards. Figure 5 shows the shifting portion of the top file diagram. Notice that some elements of Figure 2 are tied into Figure 5.

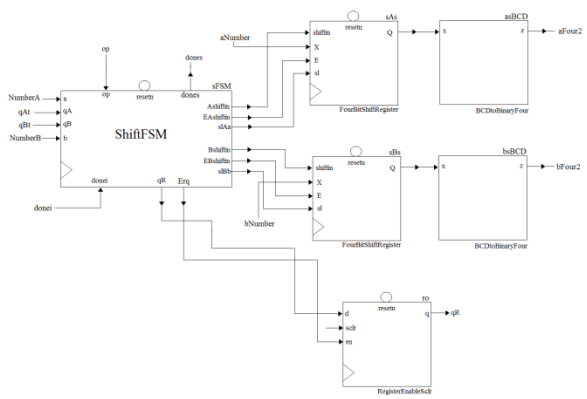

Figure 5– The shifting portion of the top file diagram

C. Arithmetic

Once the shifting FSM is initially finished, the system will then move to the arithmetic portion of the design. This is where the mathematical procedure is done on the numbers, depending on what operation was chosen. A number of procedures are done. First, the 2-bit operation signal is sent to a two-to-four decoder with an enable. The enable is the signal that gets sent once the shifting FSM is finished. The representation for this decoder can be seen in Figure 6. This will be utilized in deciding when to not only perform the arithmetic operations, but also when to output the result onto the LCD display.

Operation	Meaning	DMSA
00	Add A and B	0001
01	Subtract B from A	0010
10	Multiply A and B	0100
11	Divide B from A	1000

Figure 6 – The representation for the two-to-four decoder

There will be two signals each for both A and B. There will be a 7-bit non-shifted signal, as mentioned will come from the input FSM, and a 14-bit shifted signal will come from the shifting FSM. Two of these four signals will be used depending on what operation is being done. These operations are done in unsigned fashion. The addition and subtraction operations will take the shifted 14-bit inputs for A and B, and generate a 15-bit output, to account for overflow. The output will then be cut back down to 14 bits, ending with the addi and subi signals. The multiplication operation will take the two 7-bit non-shifted inputs for A and B and generate a 14-bit output, ending with the multi signal. The division operation will take the two 14-bit shifted inputs and generate a 14-bit quotient and a 14-bit remainder. Only the quotient will be considered in the result, which is the divi signal.

The division and multiplication operations will require a clock signal. This is one of the reasons as to why the two-to-four decoder is utilized. The multiply and divide modules require a starting signal to being their arithmetic processes. The multiply module only begins when both the shifting FSM is finished and the DMSA signal's third most significant bit is high. Likewise, the divide module only begins when both the shifting FSM is finished and the DMSA signal's most significant bit is high.

Regardless of what operation is done, all four signals for each operation will be 14-bit signals. These four signals are sent to a four-to-one multiplexer with an enable. The enable is high only when the shifting FSM is finished, and either the first bit of the DMSA signal is high, the second bit of the DMSA signal is high, the divide module is done, or the multiply module is done. The selecting bits of this multiplexer is the two-bit operation signal. This will result in a 14-bit result binary signal, which then gets converted back into a 16-bit BCD signal, resultBCD. This is done in a somewhat reversed process in terms of converting the 16-bit BCD signal to a 14-bit signal. This resultBCD signal, along with qR from the shifting FSM, are then outputted onto the LCD. Figure 7 shows the arithmetic portion of the top file diagram.

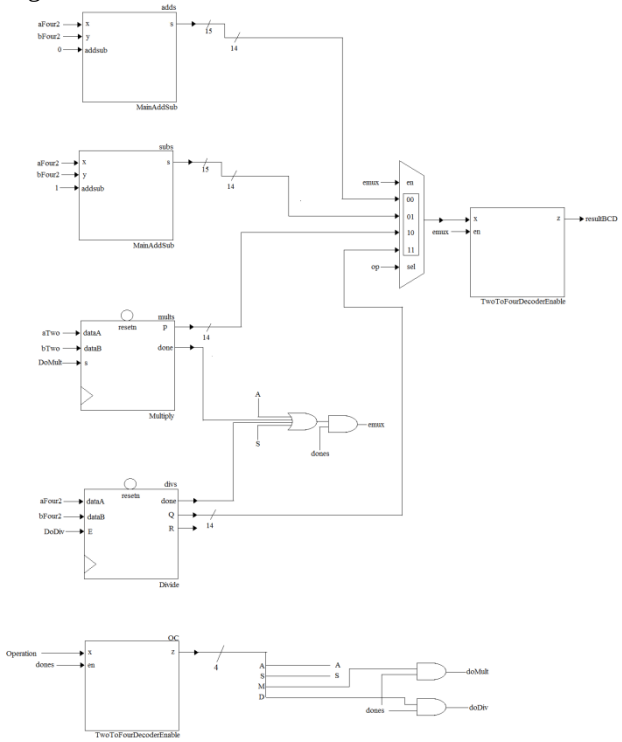

Figure 7– The arithmetic portion of the top file diagram

D. LCD Module

When the arithmetic is done, the result will be outputted to the user thorough an LCD screen connected to the Nexys 4 DDR board. The result is displayed on the LCD once the arithmetic procedure ends, but the LCD, or rather, the LCD FSM controlling the LCD, also communicates with the input FSM and outputs according to what the user is inputting. For instance, when inputting Number A, the LCD outputs "Enter A." Likewise, when inputting Number B, the

LCD outputs “Enter B.” Finally, the LCD outputs the result, including the result number and the decimal point exactly where it should be placed. This is done due to the LCD only outputting one character at a time.

The result is four characters long. Before each character is outputted to the LCD, the LCD FSM checks if it is appropriate to output the decimal number. This is done by checking qR and comparing it to how many numbers have been outputted to the LCD. If it’s appropriate to put the decimal point onto the LCD, the LCD FSM outputs the decimal point, then moves on to outputting the rest of the result.

For example, if the result is supposed to be “34.56,” the resultBCD is, in hex, “3456” and the decimal counter, qR, is 2. The LCD FSM outputs ‘3,’ then ‘4,’ then determines that the decimal point is to be put down, and then it finishes outputting the number.

The LCD FSM starts by first initializing the LCD. This is done by first powering on the LCD, then waiting 15 ms. The LCD takes in 8 bits of data, along with an enable bit and an RS bit, that tells the LCD if an instruction is being sent or if data is being sent. After the 15 ms of waiting, then a low signal is sent to the RS port, along with the instruction setting interface length, the Function set instruction. After 4.1 ms, the same Function set instruction is sent again. After 100 us, the same instruction is sent again. Then, the same instruction is sent a final time. Then, the clear display instruction is sent, and the FSM waits for 1.5 ms. Then, the LCD FSM turns the display on, and sets the entry mode. After the initialization, the LCD FSM can then write data onto the LCD. As mentioned, this is done character by character, the data sent is an 8-bit signal that is decoded into a character that is outputted onto the LCD.

After the LCD outputs the result, the process is technically done. Once the LCD FSM is done, a signal is sent to the Input FSM that allows it to recognize a single input. When the user presses any key, the system will return to the first state of the input FSM. Likewise, the input FSM sends signals that result every other FSM and component in the system, and the process begins all over again. Figure 8 shows the LCD part of the top diagram.

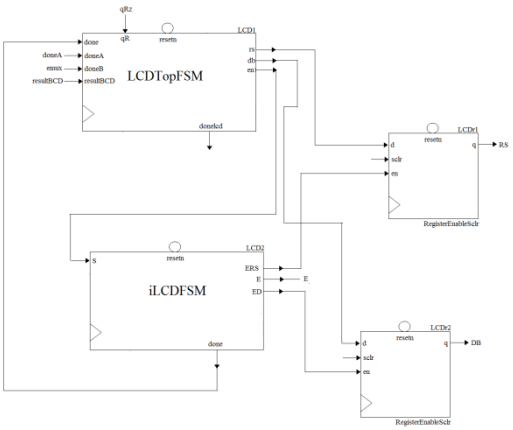

Figure 8– The LCD portion of the top file diagram

III. Experimental Setup

The hardware used for this project is the FPGA board known as the Nexys 4 Artix-7. The FPGA has Pmod ports that will allow for the keypad and LCD display to be connected. Some simulations have also been taken to describe further how some of the FSMs function. Note that, given some FSMs have a rather long wait time, not all simulations can be easily taken. For most of these simulations, time should not be taken account of, as counters have most likely been edited heavily.

Figure 9 shows the timing simulation conducted on the InputFSM, and shows the outputs related to the first state, inputting Number A. Here, the blue signals are the inputs generated from the keypad. The yellow signals show the output signals related to Number A, including the enable for the counter that keeps track of how many digits have been inputted, those being EA, zA and qA, along with the 4-bit signal sent to the 4-bit shift register and the shift register’s enable signal, aInput and EAcc.

Here, it can be seen that “48,” in hex, is first sent to the FSM, immediately sending the FSM to the decimal phase of inputting Number A. Then, a ‘1’ and a ‘9’ are inputted from the ‘keypad,’ thus maximizing the number of inputs that can be made to Number A, as noted by zA, which goes high when that limit is reached. Finally, an operation signal is sent to the FSM, sending the FSM to the state where Number B is inputted. Also note the intersecting states that occur when an input is made, and that the FSM only leaves those states when the key input is set to low, thus ‘depressing’ the key on the keypad.

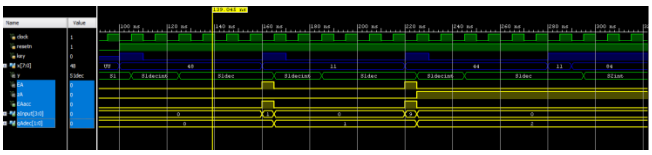

Figure 9 – A simulation done for the input FSM

Figure 10 shows a simulation for the shifting FSM. Here, the signals related to Number A are in blue, the signals related to Number B are in yellow, and the output signals are in purple. Here, A is equal to 0.09, and B is equal to 99. So, qA is higher than qB. Therefore, qR is set to qA, which is 2. Next, B has to be shifted in order for A and B to properly align, hence why B is shifted to "9900," while A remains "0009."

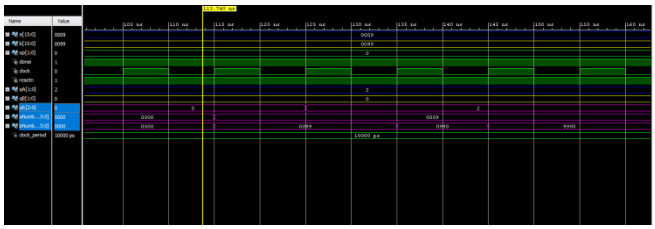

Figure 10– A simulation done for the shifting FSM

Figure 11 shows a simulation for the LCDTop FSM module. This is where the LCD sends instruction and data signals. The red signals are inputs that are sent to the LCD. As mentioned earlier, RS represents if an instruction is being sent or if data is being sent. Note that for every state, enable is turned on for a short period of time. This is because LCDTop FSM is also tied to an inputLCD FSM that controls when the enable is turned to a high signal. In that FSM, enable is set to high for a brief period of time, which allows the data from the 8-bit db port to be sent. Also note that the LCD has to initialize and set itself up before it writes data. It accomplishes all of its instruction commands before sending data. Also note that some initialization phases require waiting after sending the signal. This is, while not seen in the simulation, especially the case with the 'clear display' instruction, which as mentioned, requires roughly 1.5 ms for that instruction to complete before any other instructions or data can be sent.

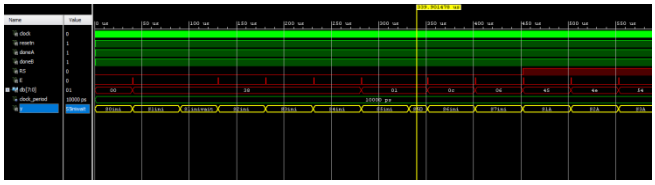

Figure 11 – A simulation done for the LCDTop FSM

IV. Results

This project is programmed onto the Nexys-4 DDR FPGA board. An Arduino 4x4 matrix serves as the keypad, and a 1602A LCD display is utilized. These two components are wired to the Pmod ports on the board.

The biggest hurdle to cross was by far getting the input interface to work. At first, a USB keypad was utilized instead of the 4x4 matrix. However, that proved to be too problematic to use, as the USB keypad would stop functioning sporadically. The USB keypad was then replaced with the Arduino keypad. At first, there were several issues getting the desired signal, and debouncing the signal proved to be rather problematic. However, all issues with the Arduino keypad have been, for the most part, straightened out. There are still glitches every now and then with the keypad, but that is more than likely due to user input than anything else. The keypad is now generally outputting the desired signals consistently.

The LCD interface was also a huge hurdle to pass through. At first, the calculator was, although intended to be designed for the LCD, instead designed with the seven-segment displays and LEDs on the FPGA board in mind. The biggest hurdle for the LCD was getting an understanding the datasheet behind it, or rather, Hitachi's HD44780U datasheet. However, once an understanding was made, the LCD was successfully integrated into the design. The LCD is powered by an Arduino Uno, but the Arduino Uno serves no other purpose. Additionally, the seven-segment display and LED signals, although not discussed at all in this report, remain on the project. Every four bits of the resultBCD signal gets sent to one of the seven-segment displays. Also, the rightmost LEDs on the board show the value for qR. Finally, the leftmost LEDs on the board show the operation value. The end result, regardless, is the proper result with the proper decimal count for the result. The only exception to this is when Number A is subtracted by a Number B that is larger than Number A, in which the result is 9999, when it should be a negative number. The entire calculator composition can be seen in Figure 12.

Figure 12 – The digital calculator

The user will, as mentioned heavily in this report, be able to input two digits for Number A, an operation, and two digits for Number B. In return, the user will receive the proper answer to the calculation made. Afterwards, the user can press any button to do another input.

V. Conclusion

This project, although heavily complex, provides a very solid infrastructure for a digital fixed point calculator. Although limited by only allowing two numerical inputs per number, this design provides an efficient and reasonable basis for a calculator system that can accomplish simplistic mathematical operations such as addition, subtraction, multiplication and division. This project should serve as a reasonable design and philosophy behind designing and developing a decimal point calculator system.

VI. BIBLIOGRAPHY

[1] Llamocca, Daniel. "VHDL Coding for FPGAs." Reconfigurable Computing Research Laboratory. N.p., n.d. Web. 16 Apr. 2017. <http://www.secs.oakland.edu/~llamocca/VHDLforFPGAs.html>.

[2] HD44780U (LCD-II) (n.d.): n. pag. Hitachi. Web. <https://www.sparkfun.com/datasheets/LCD/HD44780.pdf>.

[3] Designs, Astro. "Keypad Scanner." Astro Designs. N.p., 05 Oct. 2015. Web. 16 Apr. 2017. <http://astro-designs.com/pixi_example_no2.html>.