[64 16] Fixed Point Calculator

ECE 508: Digital Logic and Microprocessor Design – Winter 2017

Santosh R Epuri, Jacob J Morales Argumedo
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: sepuri@oakland.edu, jmoralesargumed@oakland.edu

***Abstract*— The project will focus on creating a fixed point calculator which will take inputs from a keyboard and output the results to an LCD display. The keyboard, LCD and calculator will be connected together by an Artix 7 FPGA development board.**

I. INTRODUCTION

This project will allow the students of ECE 508 to showcase their knowledge of the concepts learned during the course. By combining the PS/2 protocol for reading the keyboard, creating an arithmetic logic unit to do mathematical calculations and outputting these results to an LCD screen, we will have successfully covered and demonstrated our knowledge of building digital circuits using an FPGA.

The main motivation for this project is to explore digital circuit design for a device as simple as a calculator. Though calculators are used and taken for granted every day, our team would like to understand how these simple machines truly work at the digital circuit level.

II. METHODOLOGY

A. Keypad Input

For the keypad input, our team started off by using Dr. Llamocca's PS2 keyboard VHDL code and mapping the constraints in Vivado HDL. We encountered issues with using this code for this application. So, we created a VHDL design using higher level programming.

We were able to find a dedicated keyboard made by Onn that correctly outputs PS2 scan codes. Once connected to the FPGA we were able to see the test code from the PS2 VHDL output the corresponding scan code hex values to the LEDs and 7-segment displays on the Nexys 4 board.

Figure 1: Onn PS2 Keypad

Our team mapped the PS2 scan codes to binary values for each corresponding alphanumeric key on the keyboard (0 to F) while using the scan codes as is for enter, add, subtract, multiply, divide). The method of mapping was simply using a lookup table (LUT).

Alphanumeric Key	PS/2 Code Make	PS/2 Code Break
0	70	F0,70
1	69	F0,69
2	72	F0,72
3	7A	F0,7A
4	6B	F0,6B
5	73	F0,73
6	74	F0,74
7	6C	F0,6C
8	75	F0,75
9	7D	F0,7D
A	1C	F0,1C
B	32	F0,32
C	21	F0,21
D	23	F0,23
E	24	F0,24
F	2B	F0,2B
+	79	F0,79
-	7B	F0,7B
*	7C	F0,7C
/	E0,4A	E0,F0,4A
ENTER	E0,5A	E0,F0,5A

Figure 2: PS/2 Scan Codes

B. 16x2 LCD Screen:

The LCD portion of the project did not go as smooth as expected initially. We used Dr. Llamocca's LCD VHDL code as a starting point, but right away we noticed that the display was behaving oddly. Initially we thought that our LCD's wiring was incorrect since the letters seemed very faint, even after adjusting the contrast setting to the maximum contrast. It turns out that the LCD we are using requires a 5v supply to work correctly with the contrast adjustment. The FPGA board is only capable of outputting 3.3v supply. To get the full range of contrast on this LCD

we must have a full 0v-5v range. This meant that the 0-3.3v range we had connected to the LCD explained the faint letters on the screen. We wired up a negative voltage supply using a 9v battery and a potentiometer to go from -9v to +3.3v. This allowed us to adjust the contrast setting on the LCD to where the text was visible to a normal user's eye.

The second hurtle with the LCD was the initial VHDL code we were using. We noticed that when we powered on the LCD and initialized it using the provided VHDL code, that sometimes the LCD would not show text at all or sometimes it would show strange characters.

We referenced the HD44780U datasheet [2] to try to figure out what could be occurring. It turns out, that if the LCD has less than a 4.5V supply at power on, a manual initialization procedure must be made so that the LCD can work in "low power mode". We modified the state machine for the LCD so that the manual initialization procedure was called during start up. After implementing the changes, the LCD no longer displayed strange characters.

Unfortunately, even after these changes, we still had a 3rd problem. Sometimes the input sent to the LCD would not do anything or sometimes the LCD would get "stuck". After troubleshooting the issue more, it turns we had two problems. One being that the LCD state machine had a bug where it allowed it to skip S3, S4 and S5. This was fixed. However, it still seemed that sometimes the input would not get sent to the LCD. We were able to use our past experience with this LCD to determine that the timing was too fast. When we increased the time between states for the state machine by the LCD began to work flawlessly. We believe this may be due to our LCD running at 3.3V (instead of 5v) and thus needing the pins asserted HIGH for slightly longer period of time for the LCD to register the assertion.

To wrap up this topic, we also incorporated "scroll left and scroll right" features for the LCD so that we will be able to see all the precision bits from our calculation. The LCD can actually show up to 80 characters per line; however, only 16 are visible at a time, thus the need for scrolling left and right through the screen to see all the printed characters.

C. Calculator results conversion: binary to BCD to ASCII:

For this portion of the project, we found pre-made code online as coming up with the algorithm to convert binary to BCD would have been past the scope of the project. Due to troubleshooting issues, we actually attempted to implement several BCD converters thinking that there were issues with the algorithm itself. The first algorithm from duolos.com [4] used the double dabble method of BCD conversion. It was implemented into our circuit successfully, however, the result always seemed to be truncated or rounded near the LSB values. After struggling with this, our second attempt was another double dabble BCD algorithm from nandland.com [3]. We tried to implement the code as is from the website. It simulated just fine, however, when we tried to synthesize the code in Vivado, it would throw an error. Using Dr. Llamocca's assistance, he helped us turn un-synthesizable code into Vivado friendly synthesizable code. However, even after doing this, we seemed to get the same strange truncation/rounding of the LSB values. After contacting the professor and informing him of our troubles, he found that the issue was actually the MATLAB quantizer feature we were using to convert from decimal to [64 16] binary values. As explained by the professor, it turns out that the fixed point values we were requesting from the quantizer feature were in fact too precise for the quantizer feature to handle. Thus it would round off our input decimal values during the binary conversion. In hindsight, had we known about the limitation of the MATLAB quantizer feature, we would have stuck with the first binary to BCD converted and finished this part of the project sooner.

After debugging all the issues, the binary to BCD converter was successfully implemented. We shift in each BCD value to a 4 bit register and add x"30" to it to convert it to a numeric ASCII character. This then gets sent to the LCD to display. We shift in all the integers, when complete, place the decimal point, and finally place the fractional ASCII digits in the LCD using the same shifting method as the integers.

Figure 3: Vivado test bench of Binary to BCD conversion

Figure 4: Overview of state machine used for LCD display and BCD conversion

D. Calculator for addition, subtraction, multiplication and division.

64-bit signed addition, subtraction, multiplication and division has been designed using Vivado and verified using the test bench. The output is [112 16] because multiplication of two 64-bit inputs needs 128 bits. We have discarded 16 LSBs from the output because 32 bits for decimal number is a little excessive. The input to the LCD is the sign bit and unsigned representation of the math output which is signed.

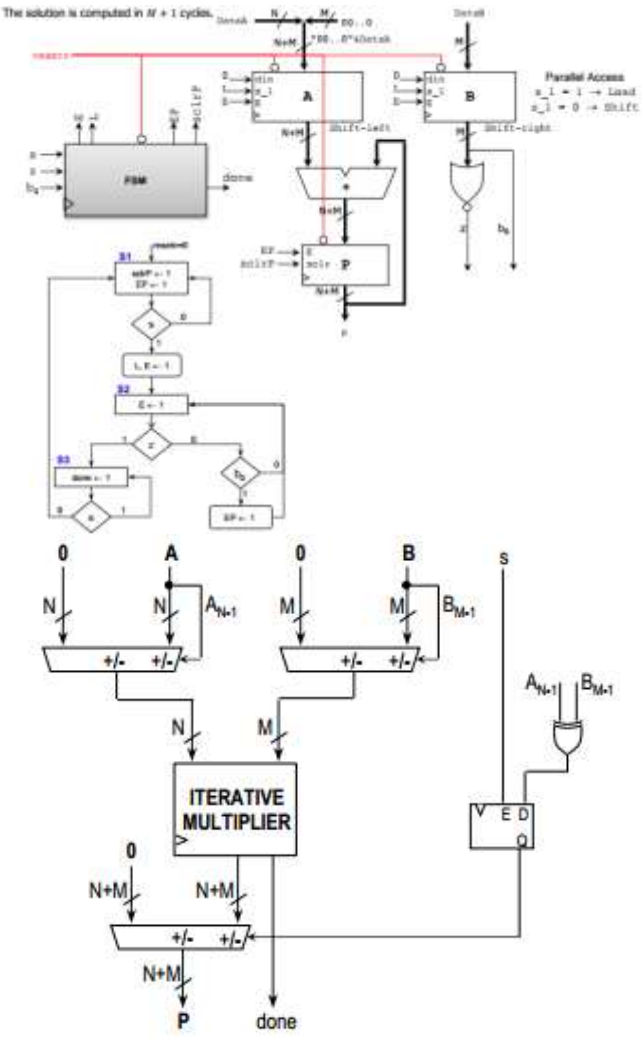
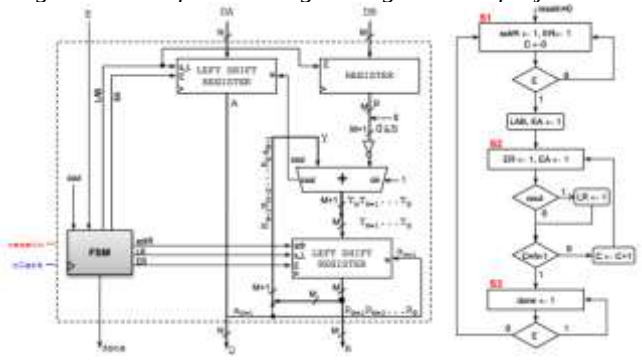


Figure 5: Multiplication logic design used in project



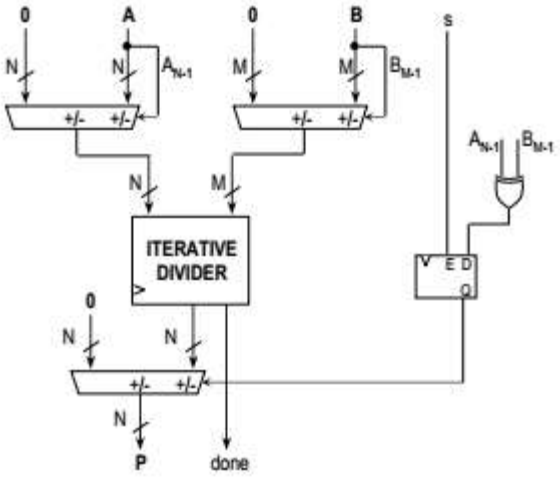


Figure 6: Division logic design used in project

E. Main state machine for entire system

The FPGA has been programmed to read the user entry from key board. The finite state machine shown in *Figure 7* has been designed to parse the user entries into operands and the operation. 16 x 2 hexadecimal characters for both operands were shifted into a 128-bit logic vector.

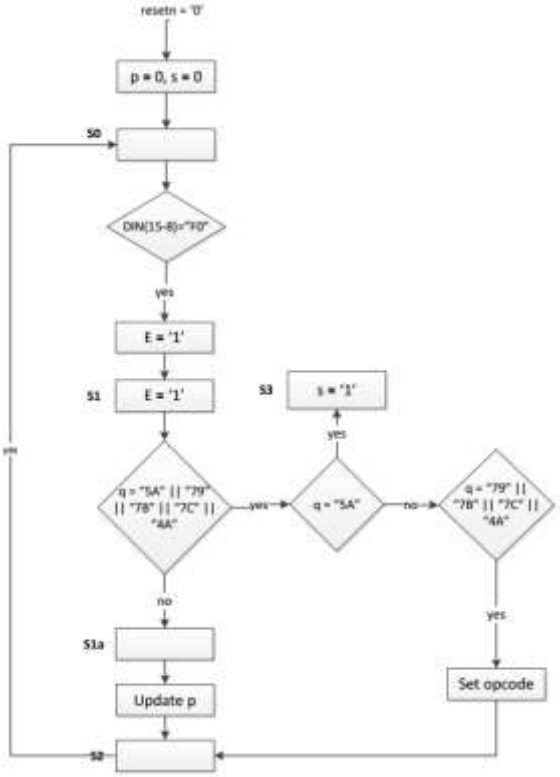


Figure 7: Parsing state machine

III. Experimental Setup

We used two methods of ensuring our hardware is functioning correctly. For the calculator itself, we used the Vivado test bench to ensure that our calculation of addition, subtraction, multiplication and division was functioning correctly. This was done by simply inputting different operands into the ALUs and comparing the output to known good results.

For the LCD portion of the project, we used multiple methods of testing the hardware and FPGA circuit. As mentioned before, we found by visual inspection that the LCD was not initializing correctly. We were able to adjust the timing of the assertion of the data signals to the LCD until the LCD printed characters bug free. This was not something that would be feasible on the test bench as we were using a 5v LCD with a 3.3V supply, thus it needed extra "HIGH" assertion time.

Our Binary to BCD to ACSII conversion circuit of course had to be thoroughly tested using the Vivado test bench functionality. We were able to understand and debug our BCD conversion by observing the values being shifted in and out of registers. This was not something that would have been easily done by looking at the output on the LCD as inspection of each circuit and state machine status needed to be observed.

Finally, for our keyboard input circuit, we had to use a combination of visual confirmation and test bench confirmation to debug and ensure the circuit was fully functional. Through the Vivado test bench, we were able to find issues with our VHDL code and state machine. However even after fixing the code, the actual real world behavior was erratic. For whatever reason, after hours of troubleshooting, we found that the FPGA board needed a power cycle and everything began to function as expected since then.

IV. Results

We were able to successfully implement a [64 16] input fixed point calculator using a keyboard for input, FPGA for signal processing, and LCD for outputting the results. Our output, as mentioned previously is in the format of [112 16] and our LCD successfully converts the binary values to ASCII characters to display on the LCD.

Figure 8: Picture of entire project, with keyboard and LCD working.

Conclusions

Although our project has met all the requirements set forth in the proposal, we would recommend an improvement to make this calculator user friendly. We currently have to input the operands in HEX format (16 hex values * 2 + 1 operation code = 33 characters). This requires the user to first "think" in decimal and convert the decimal values into hex. From there the user must input the values into our FX calculator in HEX. The result comes out in user friendly decimal format thanks to the binary to BCD conversion. In order to allow direct decimal input from the keyboard, we would have to solve the problem of converting BCD input to binary values, which would require additional development time.

References

[1] Nexys 4 DDR Keyboard Demo. Digilent Inc., n.d. Web. 15 Mar. 2017. <https://reference.digilentinc.com/learn/programmable-logic/tutorials/nexys-4-ddr-keyboard-demo/start>

[2] HD44780U (LCD-II). Hitachi, 1999. Web. 15 Mar. 2017. https://www.sparkfun.com/datasheets/LCD/HD44780.pdf

[3] "Convert Binary numbers to BCD in VHDL and Verilog." Convert Binary numbers to BCD in VHDL and Verilog. N.p., n.d. Web. 21 Apr. 2017. https://www.nandland.com/vhdl/modules/double-dabble.html

[4] "Binary To BCD Conversion." Binary To BCD Conversion. N.p., n.d. Web. 21 Apr. 2017. https://www.doulos.com/knowhow/vhdl_designers_guide/models/binary_bcd/