ECE 533: VLSI Design

8-Bit Calculator

I certify that this work is original and not a product of anyone's work but my own.

Timothy Chase: _____

Submitted: Dec 08, 2020

Contents

List of Figures

List of Tables

Abstract

This report is an overview and detailed listing of an 8-bit calculator project. Over the term an 8-bit alu and supporting logic was developed and integrated to become a usable calculator. To do this verilog code was written to describe the hardware necessary for an 8-bit ALU, seven segment controller, VGA controller, screen controller and video memory. All of this logic was then connected together in a top-level controller that also maps logic to physical IO. The project was synthesised and implemented to be programmed to a Nexys-4 FPGA development board. The end goal of having a working hardware calculator in hand was achieved and has been documented in this report. All Verilog code as well as the Vivado Project and bitstream file is available at *github.com/timChase98/533Calculator/*.

1 Introduction

A calculator can been seen as the ideal introduction to hardware design. The project has a small barrier-to-entry while maintaining the opportunity for near limitless complexity. For this implementation the project focused on simple arithmetic and logical operations and the complexity of the user interface was expanded to include video output. An Underlying understanding of seven segment and VGA control theory was used to implement full video output. This includes the current status of the calculator as well as the opcode diagram for the ALU. To increase the usability of the calculator a store and load result was implemented. The user could store the result of an operation into the storage register and later load that result back into either the A or B register.

2 Methods and Procedures

2.1 Design Philosophy

The goal of this project was to design a working calculator that could be programmed onto a Nexys-4 FPGA development board. In order to be able to use the calculator several parts needed to be designed and implemented. Design started by conceptualizing how the calculator should work, followed by laying out the architecture needed, then writing the code necessary for each component. The calculator has been split into several sections, inputs that are all handled by the top level, the ALU, and outputs that are handled by various components including the VGA controller and seven segment controller.

2.2 Component Design

2.2.1 Top Level

The Top Level design consists of a set of interconnections between the five main components of the calculator as well as interfacing with physical inputs and outputs. It takes the input from the 100 MHz crystal on the board and uses a binary counter to generate each of the slower system clocks. Since only the VGA controller is sensitive to timing (and is conveniently 100 MHz / 4), simply using various bits of the binary counter is good enough and more complicated clocking logic like PLL's are not necessary. The position of the clocking bits in the counter were mostly picked arbitrarily. The VGA clk was selected as CLK/4 to generate 25 MHz, the seven segment and screen controller clocks were picked to be fast enough to multiplex the displays without flickering but slow enough to not waste unnecessary power clocking logic that doesn't need to be clocked at such high speeds. The top level also takes switch and button inputs from the board. The lower eight switches generate the inputVal bus (used as the input to the A/B register); the upper four switches act as the operation selector. The 5 buttons are organized to latch inputVal to the A register (left), or B register (right), latch the lower eight bits of the output register into storage (center), or to recall the storage register to the A or B registers (up and down respectively). Outputs include the 5 signals necessary for VGA output (RGB + sync pulses), the anode and cathode lines for driving the 8 x 7-segment display, and a buffered version of the switch inputs that drives LEDs to make it more clear which switches are on when looking at the board. The top layer also includes all the necessary interconnects for the rest of the logic components. See Figure 1 for an RTL schematic of the top level design.

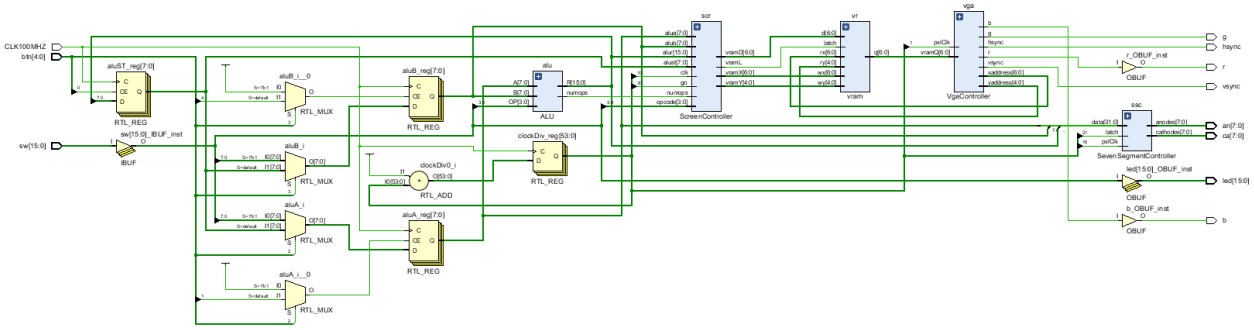

Figure 1: Vivado RTL Top Level

2.2.2 ALU

The ALU takes three inputs A, B and OPCODE, and generates two outputs, R the results signal and numops which is used to tell the display units which of the operands are necessary for a calculation.

Figure 2 for a RTL schematic of ALU. The output is generated by using the opcode signals as the select bits on a 4:16 multiplexer. The inputs of the mux (four of which are currently unused due to lack of creativity in coming up with integer arithmetic). Table 1 shows the currently implemented operations. All operations use the standard Xilinx implementations except the factorial operation that uses a lookup table. As factorial can only go up to eight with a 16-bit output a lookup tables is a more space efficient and faster implementation as this can be ”calculated” in less than a clock cycle. Figure 2 shows the RTL diagram for the ALU.

Opcode bits	R
0000	A + B
0001	A - B
0010	A x B
0011	A / B
0100	A and B
0101	A or B
0110	not A
0111	A xor B
1000	A shift left B
1001	A shift right B
1010	A mod B
1111	A factorial

Table 1: Opcodes

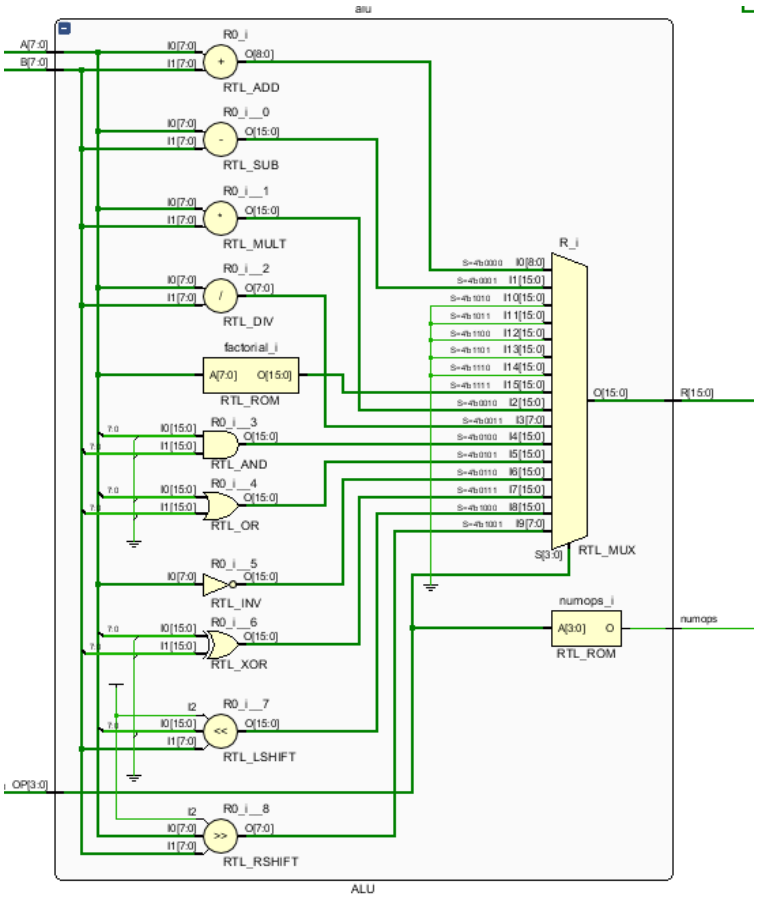

Figure 2: ALU RTL

2.2.3 Seven Segment controller

The seven segment controller is a simplistic multiplexing controller with registered inputs. The controller can be broken down into two sections: the multiplexing logic and the digit decoding logic. Starting with the simpler digit decoding logic we have a 4 bit register with latch. The output of this register drives a ROM decoder acting as a lookup table. The multiplexing logic has eight of these segment decoders; each D input is fed from 4-bits of the 32-bit input to the controller.

The decoded Q outputs from the segment decoders is stored in a 8x8 memory array. Depending on which digit is currently being displayed an 8-bit row is selected and sent to the cathodes output. The index of the current digit is then used as the input of a standard 3:8 decoder and inverted to be sent to the anodes output. Figure 3 shows the RTL schematic for the seven segment controller. As the digit decoders are simply an 8-bit register and a 4x8-bit ROM the RTL is not shown.

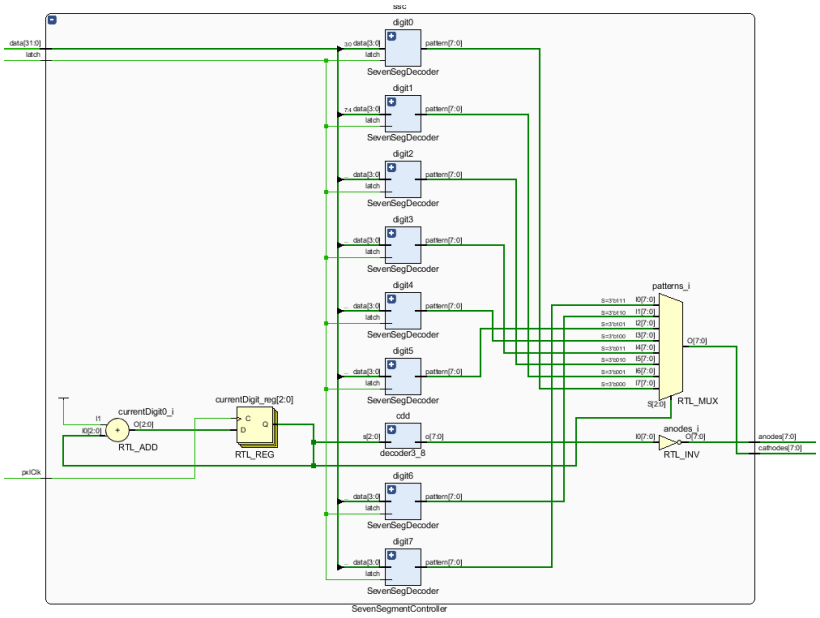

Figure 3: Seven Segment Decoder RTL

2.2.4 Vga Controller

The VGA controller consists of three parts the sync pulse/pixel driver logic, the Video RAM memory management logic and the font ROM. Starting from the simplest component, the font ROM (written in VHDL as it is reused from an ECE 368 project from last semester) is simply a 2048x127 ROM of 16x8 character maps. The maps, each 16 rows tall by 8 pixels wide consist of a slightly modified version of the original MS-DOS font, with some character swaps to include custom symbols needed for a few of the ALU operations. The 7 MSBs of the address select the character and the 4 LSBs select which line is currently being used. The 640x480p resolution allows for an 80 character by 30 line display. As we didn't need the full 80 characters and using a 64 character line allows for much more efficient bit packing each character is address by a 6-bit X address and a 5-bit Y address stored in VRAM and decoded into pixels as the lines are drawn to save on memory. The font ROM is addressed by the 4 LSBs of the line counter concatenated to the LSB end of the 7-bit vramQ input from the video memory controller. The 5 MSBs of the line counter and the 7 MSBs of the horizontal counter are used passed to the X and Y read inputs to the VRAM via

the top level design. Two counters and several binary compactors are used to generate the various phases of the VGA control signals that are passed back to the top level design to be connected to the VGA connector on the NEXYS board. Figure 4 shows the RTL for the VGA controller. You can see that the signals are only driving the green line currently (to get the classic green text on black screen aesthetic); this could easily be modified to match any font and background desired.

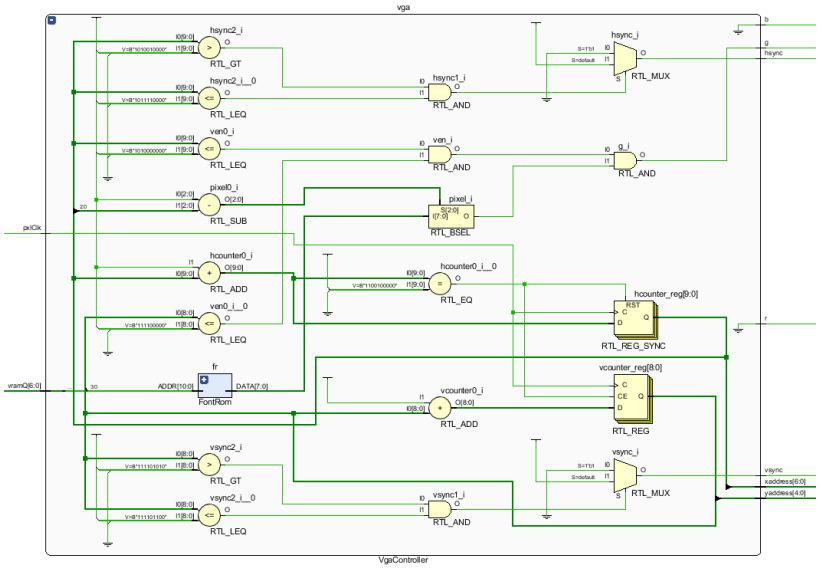

Figure 4: Register File RTL

2.2.5 Screen Controller

The screen controller is the second of the three parts of the video output. Similar to how the VGA controller reads from VRAM; the screen controller is used to write the current status of the calculator into VRAM. The screen controller takes a pixel clock input, the ALU signals, and outputs signals to drive the VRAM write section. It should be noted that the VRAM is latched on the falling edge of the screen controllers pixel clock. This is so that the screen controller can write one character per clock cycle by having the signals stabilize during the high period of the clock; the VRAM signals can then stabilize during the low period of the clock and the VGA controller can update on the rising edge of the next clock. The current screen controller only writes to the top line of the screen (because this was all that was necessary for this project) but has the capability to write to any line if the user interface needs this feature. The screen controller has a char counter that keeps track of the X address currently being written to. As you can see in Figure 5 the aluA, aluB and aluR signals are split into 4-bit nibbles and passed to the input of a 4:16 MUX. The aluA and aluR signals are passed directly to the MUX, however as the B operand is only used on some calculations it should only be displayed when part of a calculation. The numOps signal from the

ALU is used to make this selection. The opcode is also passed to a ROM acting as a LUT to decode an opcode into the appropriate character to be displayed. We also need to shift the output of the MUX up by 0x30 or 0x40 to convert from the number 0x0 and 0xA to the character '0' and 'A' respectively.

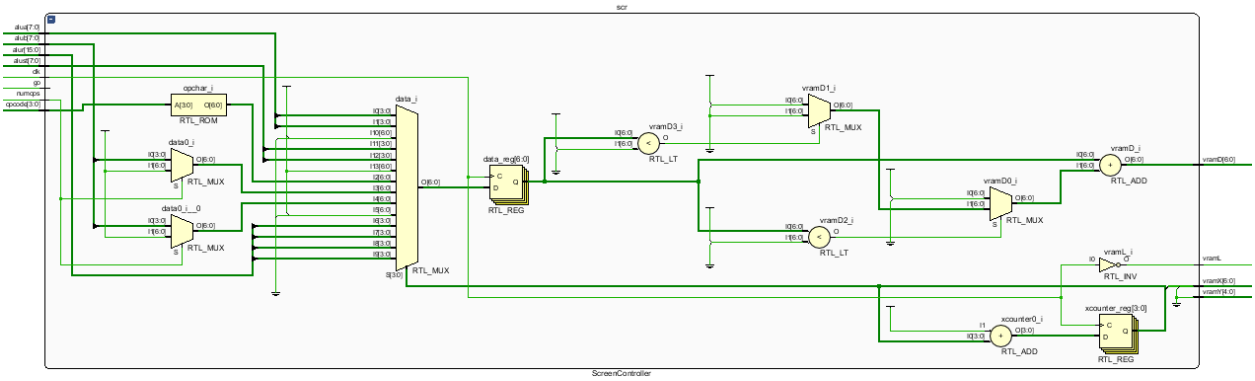

Figure 5: Screen Controller RTL

2.2.6 Video RAM

The Video RAM is a simultaneous read/write two dimensional memory array. The video memory is initialized on reset to be all 0's, except for the bytes necessary to display the onscreen instructions. As nothing overwrites these characters they can be loaded into the initial RAM image instead of being continuously redrawn. This both reduces the number of RAM writes necessary and also decreases the necessary complexity of the screen controller. While is is possible to use the IP library to generate a block ram, the decision was made to create ram on the fabric of the FPGA as the block ram IP didn't match the desired timing diagrams of having a read take less than a clock cycle. Because the VRAM array is so large the RTL diagram is not readable (due to the 128x32x6 array of registers and almost 12 thousand wires on a single screen) and is thus not included.

2.3 Simulation

A test bench was created for the ALU. This bench stepped through all of the pre-programmed opcodes and verified the results of the calculation. To do this the operands for each calculation were chosen and the operation was done by hand to be programmed into the expected result register. A task was written to compare the output of the ALU to the expected results register and keep track of errors. Results of the test runs are discussed in Section 3.1. The remainder of testing and validation was done by hand as visual inspection of seven segment displays and VGA outputs are much easier to do than a test bench for these controllers would have been.

3 Experimental Results

Results are split into two sections. Outputs of the test bench and a demo of the working calculator.

3.1 Simulation Results

The results of the simulation can be seen in Figure 6 and Figure 7 for the waveform and console output respectively. You can see in the both the console and waveform that all output was as expected and no errors occoured.

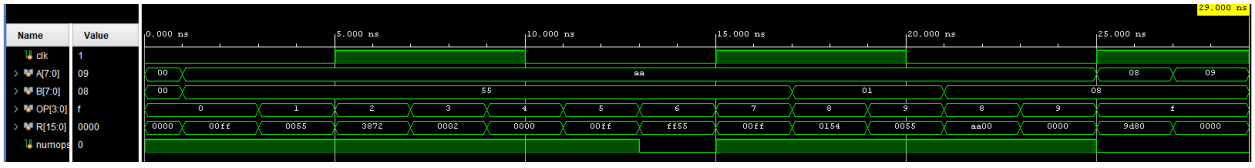

Figure 6: ALU Simulation Waveforms

```
run all
At 2:    op = 0000   a = aa      b = 55  R = 00ff    result = 00ff
At 4:    op = 0001   a = aa      b = 55  R = 0055    result = 0055
At 6:    op = 0010   a = aa      b = 55  R = 3872    result = 3872
At 8:    op = 0011   a = aa      b = 55  R = 0002    result = 0002
At 10:   op = 0100   a = aa      b = 55  R = 0000    result = 0000
At 12:   op = 0101   a = aa      b = 55  R = 00ff    result = 00ff
At 14:   op = 0110   a = aa      b = 55  R = ff55    result = ff55
At 16:   op = 0111   a = aa      b = 55  R = 00ff    result = 00ff
At 18:   op = 1000   a = aa      b = 01  R = 0154    result = 0154
At 20:   op = 1001   a = aa      b = 01  R = 0055    result = 0055
At 22:   op = 1000   a = aa      b = 08  R = aa00    result = aa00
At 24:   op = 1001   a = aa      b = 08  R = 0000    result = 0000
At 26:   op = 1111   a = 08      b = 08  R = 9d80    result = 9d80
At 28:   op = 1111   a = 09      b = 08  R = 0000    result = 0000
Ended with 0 errrors
$finish called at time : 29 ns : File "C:/Users/tim_c/Documents/Vivado/533/533.srcs/sim_1/new/aluTB.v" Line 84
```

Figure 7: ALU Simulation Console Output

3.2 Demo

Attached to this report is a video demo of the completed project working on a Nexus 4 Development board connected to a VGA monitor. The video can also be found at *github.com/timChase98/533Calculator*. Additionally a few images have been included in this report. Figure 8 shows an image of the programmed NEXYS 4 board running the hardware calculator. The seven segment display shows the 16-bit result on the four left-most digits (currently 0x0038 or 56), the next two digits show the A input to the ALU (0xAA or 170) and the two right-most digits show the B input(0x03 or 3). The four left-most switches select the 4-bit opcode

(currently 0b0011) which translates to division. 170 / 3 = 56. (Remember we are using integer division so we truncate the decimal component.) The eight right-most switches act as the integer input (currently set to 0x03). The left button latches the input into the A register, the right does the same to the B register. The center button latches the 8 MSBs into the storage register. Up and down latch the storage register into the A or B registers respectively. Figure 9 is an image of the VGA output from the nexus board. You can see the top line contains the same math as the seven segment display. 0xAA / 0x03 = 0x0038. The onscreen display also shows that the storage register current contains the value 0xFE. The VGA output also includes the opcode table. An up arrow denotes that the switch should be up; the same applies to a down arrow and the switch being down.

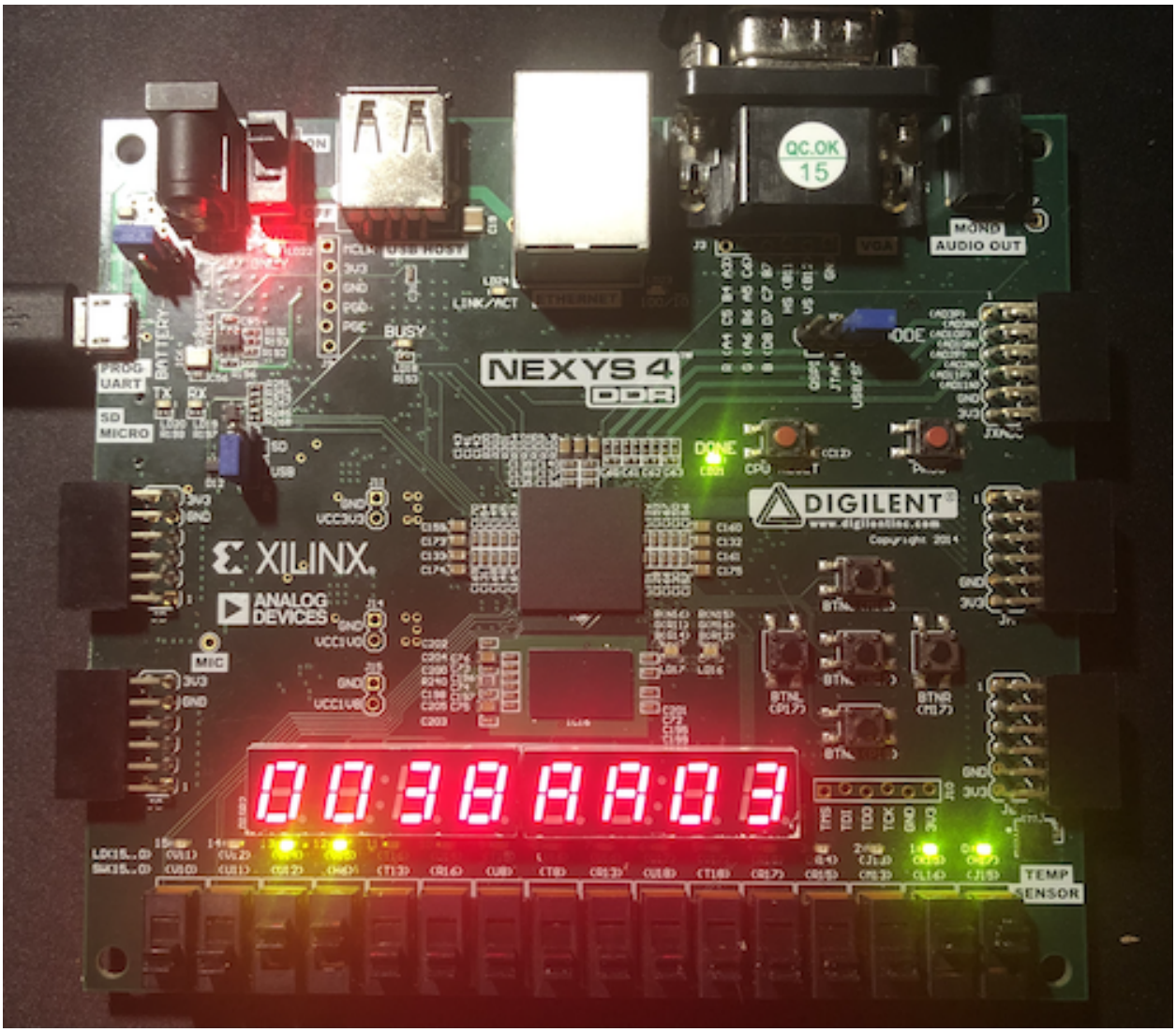

Figure 8: Image of NEXYS 4 board showing the result of 0xAA / 0x03 = 0x38

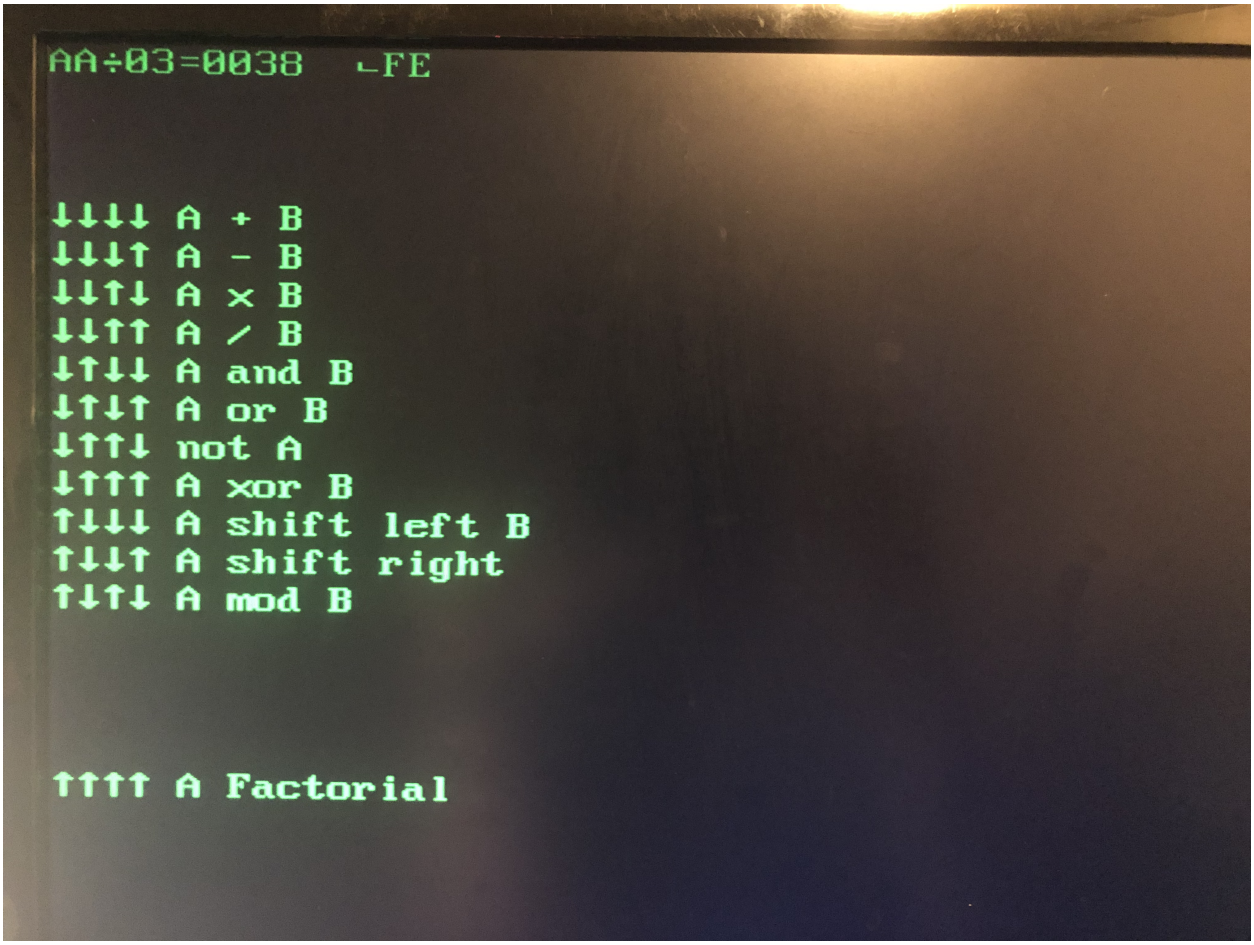

Figure 9: Image of VGA output from NEXYS 4 board showing the result of 0xAA / 0x03 = 0x38, as well as opcode instructions and the contents of the storage register.

4 Discussion

Designing any complex system can be a daunting task, however with good conceptual design and and an in depth understanding of the necessary components even the most difficult of designs can be realized. This project required understanding of several digital logic design concepts including data path design, control paths, and synthesizable arithmetic. Additionally knowledge of the control theory of seven segment display multiplexing and VGA timings were used to proved more usable output from the calculator. These concepts are not limited to use in a calculator design. These concepts are applicable to any digital design project.

5 Conclusion

A usable calculator was successfully designed and implemented in verilog and was programmed on to a Nexys-4 FPGA development board. Among the features implemented are an ALU capable of preforming twelve operations, a 8x7-segment display, and video output over VGA. A test bench was also designed to evaluate the functionality of the ALU. This design came with many learning experiences as this was our first major project done in Verilog rather than VHDL.

6 Recommendations

Recommendations for this project include a strong emphasis on version control and intelligent use of time. As with any complex system making several changes at once can cause the whole system to break. Using version control where one can easily roll back a change to determine the root cause of the issue would greatly increase productivity. As synthesis and implementation are hugely CPU and memory bound operations ,a fast computer and correct Vivado settings are necessary. With synthesis and implementation both set to incremental(only rebuild changes) and the workflow set to ”run-time optimized” (skipping the optimization step to trade fabric area for faster builds) a one line change may still take up to seven minutes to generate a bitstream. Such frequent changes can lead to huge amounts of time spent waiting for a progress bar (I had to spend about an hour trying to get the '$\sim$' character to look correct in the font ROM).

6.1 Future Work

Future work would involve the creating of additional operations, moving from integer to fixed or floating point, and likely a more intelligent screen controller. Opcodes are currently limited to 4-bits. This limit has no real basis other than a lack of ideas for new operations. Up to an 8-bit opcode could be implemented without needing to make changes to the existing layout of controls. Adding fixed or floating support would allow for instructions including trigonometric and exponential operations. Additionally as the output register is limited to 16-bits many computationally complex algorithms can be implemented via lookup table. The factorial operator was done this way as 9! is $>$16-bits. Thus a lookup table from 0-8 allows calculations to be executed in less than a clock cycle and is very space efficient. The Fibonacci sequence could be implemented via a 5 bit lookup table.

7 Laboratory Reflection

While this project was made significantly more difficult by limited access to on-campus resources and the additional workload associated with learning from home, these challenges were overcome to create a working project. Given more time to work on this I definitely would have looked into adding fixed point calculations, a second line on the VGA output that displayed the math in decimal, as well as HEX as arithmetic calculations are typically more usable in decimal while logical calculations are more usable in HEX or binary.

8 References

[1] D. Rancour, "ECE 533 Design Project," p. 1, 2020.

[2] "Nexys 4 reference manual." [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/nexys-4/reference-manual

9 Appendices