8-Bit Signed Calculator with Keyboard Input and Hexadecimal Display

EGR 2700 Final Project

Evan Kornhaus, David Kosa, France Zaytouna, Giuseppe Rizzo
Electrical and Computer Engineering Department School of Engineering and Computer Science Oakland University, Rochester, MI
evankornhaus@oakland.edu, dkosa@oakland.edu, fzaytouna@oakland.edu, grizzo@oakland.edu

Abstract

This project is an 8-bit signed calculator that is capable of four operations: addition, subtraction, multiplication, and division. A USB keyboard is used to allow the user to input data easily, and a 7-segment display will output the result for the user to view. The resulting answer will appear on the Nexys A7 FPGA board as a 16-bit, two's complement signed number in hexadecimal form. The inputs and outputs can be both positive and negative.

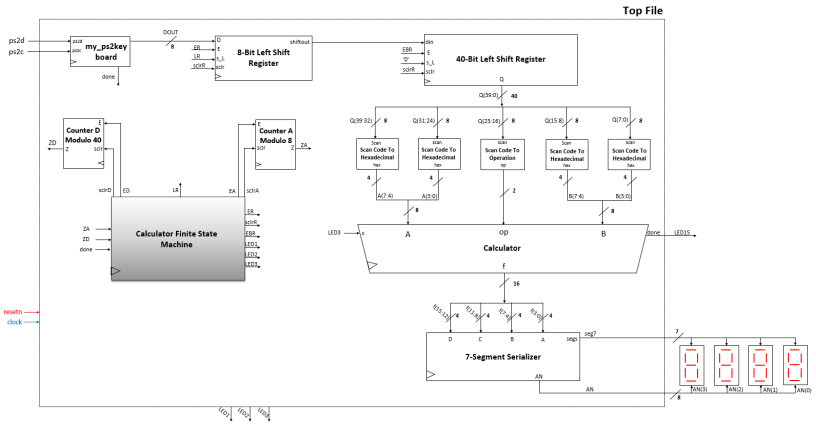

Figure 1: Top File design of the 8-Bit Signed Calculator.

I. Introduction

The purpose of this project is to create an 8-bit signed calculator that can perform addition, subtraction, multiplication, and division. Many of the topics covered in previous labs have helped to prepare for the creation of this calculator. Specifically, labs 2 and 3 helped to build the arithmetic circuits for subtraction and multiplication, as the designs presented in those labs can be extended quite easily to 8-bits. Lab 4 is very similar to the design of the calculator architecture, in that it is composed of four separate circuits connected through a multiplexor (MUX). The calculator takes five inputs which are entered by a USB keyboard; the outputs are displayed as four hexadecimal numbers on a 7-segment display.

II. Methodology

As mentioned previously, the basic outline of the calculator is composed of four arithmetic circuits, one for each of the aforementioned operations, all of which are connected through a MUX. A standard USB keyboard was used to allow the user to input particular values and operations. This required the use of the PS/2 Interface code that was made available by Dr. Llamocca in the Unit 7 notes [1]. A scan code to hexadecimal encoder as well as a scan code to operator encoder were created to convert the selected keyboard values used for calculations (0-F, +, -, ., /). The 7-segment display on the FPGA was used to display the signed hexadecimal result. To display each hexadecimal place at the same time required the use of the 7-Segment Serializer code, which was also made available by Dr. Llamocca in the Unit 7 notes [1]. A finite state machine (FSM) was created to allow control of the datapath circuit through the use of a USB keyboard. The datapath circuit contains all circuits mentioned prior, as well as two counters and two parallel access shift registers to allow calculations to take place solely through the input of a keyboard.

A. 8-bit Signed Adder

The 8 bit signed adder that was used was taken directly from the second lab of this class as it was compatible for this project. However one change that had to be made for this adder is that its inputs needed to be extended to 8 bits as opposed to 5 bits to properly work for this calculator. The inside of the adder simply contained 9 full adders and a sign extender which is what allowed for the output to have 16 bits. The top file of the adder is shown below in Figure 2.

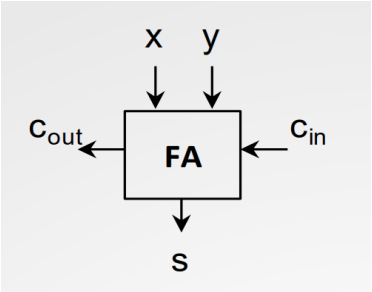

Figure 2: Block design of the 8-bit Signed Adder.

B. 8-bit Signed Subtractor

The 8 bit signed subtractor is very similar to that of the adder as it was also taken from a past lab and then manipulated to allow for 8 bits instead of 5. The carry in value is set to '1' for the subtractor. This subtractor also has a built-in sign extender for the exact same reason as the adder, to allow for a 16 bit output. The block diagram of the subtractor can be seen in Figure 3.

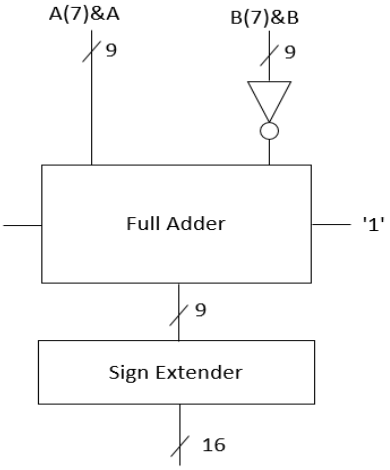

Figure 3: Block design of the 8-bit Signed Subtractor.

C. 8-bit Signed Multiplier

The 8-bit signed multiplier was originally implemented as unsigned. To convert the multiplier circuit from unsigned to signed, a custom component was designed. The component has two inputs, the primary input and an enable. If the enable is active, the component outputs the two's complement of the primary input. If the enable isn't active, the component will output the primary input. A and B are both put through this custom component (with the enable of each being driven by the most significant bit (MSB) of each respective input) before being sent to the multiplier. This forces A and B to be positive before entering the unsigned multiplier component. The output of the multiplier is run through this component again (now with enable being driven by A(7) XOR B(7)) to correct the sign of the output of the circuit to its intended value. The output is then sent to the input of the MUX through a 16-bit bus. The block diagram of the 8-bit signed multiplier is shown in Figure 4.

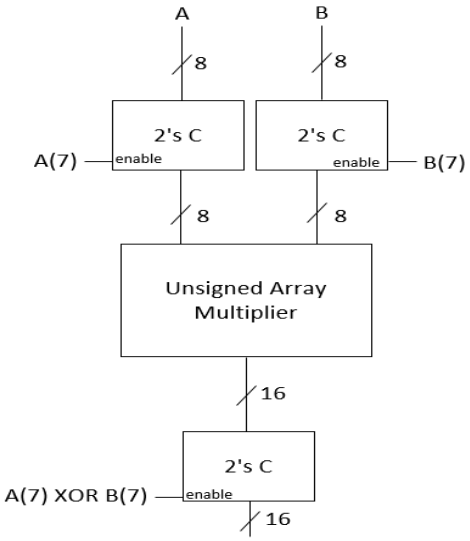

Figure 4: Block design of the 8-Bit Signed Multiplier.

There is a disparity between the number of bits leaving the multiplier and the other arithmetic circuits. In order for the MUX design to function properly, the outputs of the adder, subtractor, and divider all had to be sign-extended to 16-bits to have the same number of bits as the output of the multiplier. From here the 16-bit signals are input to the MUX, where the 's' input selects which operation to output, as can be seen below in Figure 6.

D. 8-bit Signed Divider

The 8-bit signed divider is made up of a few different components. The main component is the unsigned iterative divider, as supplied by Dr. Llamocca from his library of digital system components [2]. The design of the iterative divider, shown below in Figure 5, consists of an FSM, a regular register, two parallel access shift registers, a counter, and a subtractor. The algorithmic state machine (ASM) is shown to the right of the design of the divider, also in Figure 5. The design was modified in order to accommodate 8-bit unsigned division. This was done by increasing the size of multiple buses, the registers, the subtractor and the counter in the architecture. The output was sign extended to 16-bits, in order to match the input bits of the MUX.

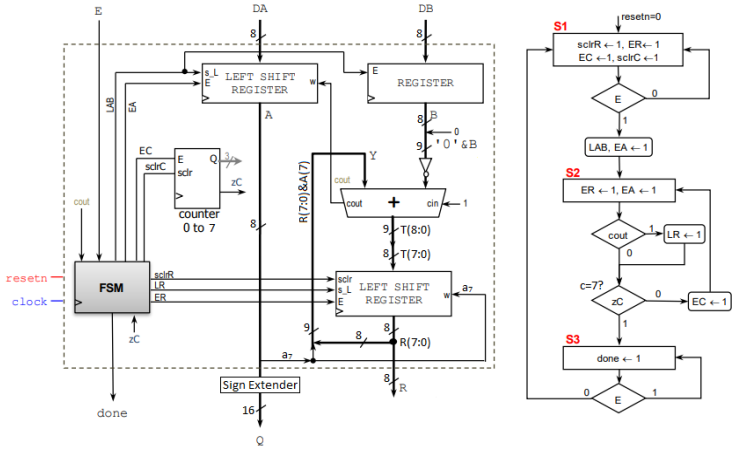

Figure 5: Design and ASM of the Iterative Divider, Modified [2].

The purpose of the divider, like the other arithmetic circuits, was to perform signed division. In order to do this, the same two's complement components implemented in the signed multiplier were also implemented for the divider. These components were placed before the inputs and after the output of the unsigned divider. The MSB of each input controls if the respective input has two's complement performed on it or not. The inputs are essentially sent through an "absolute value" component, forcing the inputs to act as positive values. For the component on the output, the enable is activated by the boolean expression A(7) XOR B(7). This means that the positive output that comes out of the divider has two's complement performed on it if only one input's MSB is a value of 1 (i.e., (0 XOR 1) or (1 XOR 0)). If the expression returns a 0 (i.e., (0 XOR 0) or (1 XOR 1)), the output remains positive.

E. Calculator/MUX Operation Selection

A multiplexor design was implemented so that the arithmetic operations are easily accessible through the use of a select line. As shown in Figure 6, the four previously described operations act as inputs to the MUX; the select line chooses which operation signal is output by the MUX. This is done by the user through the use of the keyboard. By using one of the four designated operation buttons (+, -, ., /) as the third user input, the 8-bit scan code associated with that input is stored in a 40-bit register which passes a scan code to the operation component. This component encodes the scan code into a 2-bit value which is sent to the select line of the calculator, as can be seen from Figure 1. The architecture of the arithmetic calculator is shown below in Figure 6.

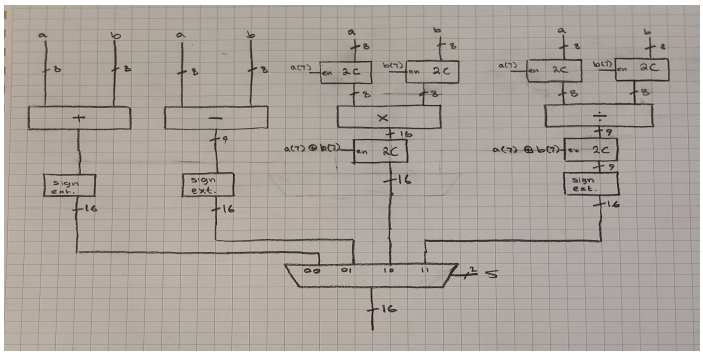

Figure 6: 8-Bit Calculator MUX-Implementation

F. PS/2 Interface for Keyboard

In order for the user to input data to the calculator through the use of a USB keyboard, a PS/2 interface was required. The code for the PS/2 interface was made available by Dr. Llamocca in the Unit 7 notes [1]. The interface sends data transmitted from the keyboard, as well as a clock signal alongside the data. In short, the USB keyboard communicates directly with an auxiliary microcontroller on the FPGA board which emulates a PS/2 bus. These PS/2 bus signals are converted from the USB protocol to the Nexys A7 with the help of the XDC file, allowing the keyboard to connect to the PS/2 interface component as if it were using a PS/2 protocol [3]. The interface reads the start bit, the 8 data bits (least significant bit first), an odd parity bit, and the stop bit; these are read on the falling edge of the PS/2 clock as shown in Figure 7. The start bit is always a '0', the stop bit is always a '1', and the parity bit is determined by the 8 data bits. The parity bit is set to '1' if the data bits are composed of an even number of both '0's and '1's; otherwise, the parity bit is '0'.

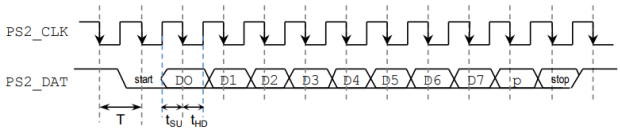

Figure 7: PS/2 clock and data [1].

The 8 data bits represent the scan code, which become stored in the 40-bit register shown in the top file in Figure 1. The scan code that is output by the PS/2 interface is represented by the signal "DOUT", and is output alongside a "done" signal, which lasts for one clock cycle (FPGA clock). The done signal is an important input to the FSM, as it instructs the datapath circuit to store the output scan code in the 40-bit register. The PS/2 interface top file is shown in Figure 8 below.

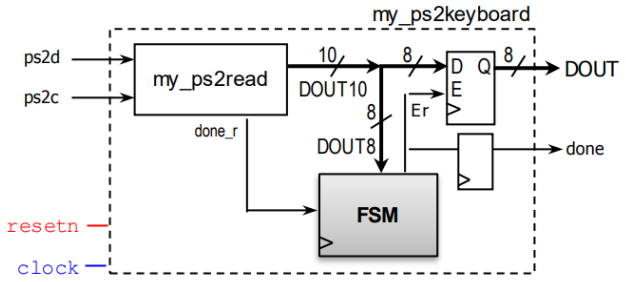

Figure 8: PS/2 Interface top file, as illustrated from the Unit 7 notes [1].

G. *Scan Code Encoders*

The scan code encoders were relatively simple to create, as converting from the scan code to hexadecimal and the scan code to operation only required a case-when statement for each component. These scan code encoders were based off of the scan codes given in the Nexys A7 Reference Manual [3]. Figure 9 showcases the corresponding scan code associated with each key on the keyboard. As can be seen in Figure 1, the inputs to each of the encoders come from the 40-bit register as a single bus. The 40 bit bus is then split into a set of five, 8-bit buses. This separates the data into their respective 8-bit scan codes, which are then sent through each encoder. The output of the hexadecimal encoders are 4-bits; the operation encoder only outputs 2-bits. The case-when statements to transform the scan code to hexadecimal or scan code to operation is shown in Figure 10 and Figure 11, respectively. Notice that if a scan code that is not listed is input, the component defaults the hexadecimal encoders to output "0000", and will also default the operation to multiplication.

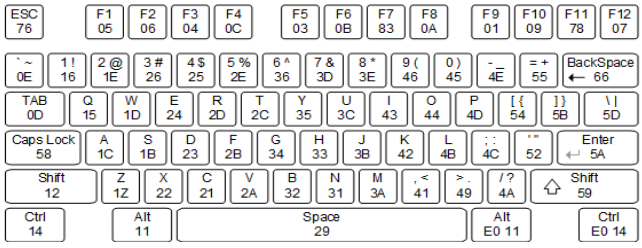

Figure 9: PS/2 keyboard scan codes [3]

```
entity ScanCodeToHex is
    Port ( Scan : in STD_LOGIC_VECTOR (7 downto 0);
           Hex : out STD_LOGIC_VECTOR (3 downto 0));
end ScanCodeToHex;

architecture Behavioral of ScanCodeToHex is
begin
    process (Scan)
    begin
        case Scan is
        when "0100"&"0101" => Hex <= "0000"; --0
        when "0001"&"0110" => Hex <= "0001"; --1
        when "0001"&"1110" => Hex <= "0010"; --2
        when "0010"&"0110" => Hex <= "0011"; --3
        when "0010"&"0101" => Hex <= "0100"; --4
        when "0010"&"1110" => Hex <= "0101"; --5
        when "0011"&"0110" => Hex <= "0110"; --6
        when "0011"&"1101" => Hex <= "0111"; --7
        when "0011"&"1110" => Hex <= "1000"; --8
        when "0100"&"0110" => Hex <= "1001"; --9
        when "0001"&"1100" => Hex <= "1010"; --10 'A'
        when "0011"&"0010" => Hex <= "1011"; --11 'B'
        when "0010"&"0001" => Hex <= "1100"; --12 'C'
        when "0010"&"0011" => Hex <= "1101"; --13 'D'
        when "0010"&"0100" => Hex <= "1110"; --14 'E'
        when "0010"&"1011" => Hex <= "1111"; --15 'F'
        when others => Hex <= "0000";        --anything else, '0'
    end case;
  end process;
end Behavioral;
```

Figure 10: Scan Code to Hexadecimal Encoder. The hexadecimal value is defaulted to "0000".

```
entity ScanCodeToOP is
    Port ( Scan : in STD_LOGIC_VECTOR (7 downto 0);
           op : out STD_LOGIC_VECTOR (1 downto 0));
end ScanCodeToOP;

architecture Behavioral of ScanCodeToOP is
begin
    process (Scan)
    begin
        case Scan is
        when "0101"&"0101" => op <= "00"; --0 = addition
        when "0100"&"1110" => op <= "01"; --1 = subtraction
        when "0100"&"1001" => op <= "10"; --2 = multiplication
        when "0100"&"1010" => op <= "11"; --3 = division
        when others => op <= "10";        --anything else, default to multiplication
      end case;
    end process;

end Behavioral;
```

Figure 11: Scan Code to Operation Encoder. The operation value is defaulted to "10" (multiplication).

H. *7-Segment Serializer*

A serializer is required to display the 16-bit calculation at once on the Nexys A7 board. This is due in part by the architecture of the board, where each cathode on every individual 7-segment digit shares a single node [3]. The serializer consists of a FSM, a four input multiplexor, a 1 millisecond counter, a 2-to-4 decoder, and a hexadecimal to 7-segment decoder. The FSM is controlled by the 1 ms counter, which outputs a 2-bit value for every 1 ms that passes. The output of the FSM controls the select line of the multiplexor and passes through the 2-to-4 decoder. The 2-to-4 decoder essentially activates one of four 7-segment digits every millisecond, allowing the user to see the individual digits making up the 16-bit result. To display the calculation, the output of the calculator is connected to the input of the serializer, or the multiplexor. To match the bus size of the calculator to the input of the multiplexor, the 16-bit single bus is split into four separate 4-bit buses,

which are connected to the input of the multiplexor. The block diagram of the serializer is shown in Figure 12 below. The 7-segment serializer was supplied in the Unit 7 notes [1].

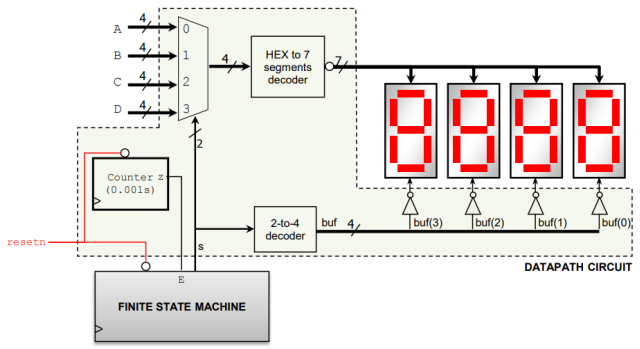

Figure 12: 7-Segment Serializer top file, from the Unit 7 notes [1].

I. Finite State Machine and Datapath Circuit

A few designs of the FSM and datapath circuit were considered before settling on the design pictured in Figure 1. The datapath circuit consists of all of the components mentioned previously, as well as two counters and two registers that are controlled by the FSM of the 8-bit signed calculator. In fact, the FSM functions as the control circuit of the entirety of the datapath circuit. The FSM allows the user to input a value from the keyboard while it is in state 1. Once the FSM receives a signal from the keyboard, the scan code output from the PS/2 Interface component becomes stored in the 8-bit register, and the FSM enters state 2.

During state 2, the 8-bit register shifts all of the scan code data (MSB first) into the 40-bit register. This is controlled by the modulo-8 counter; once the data is stored in the 40-bit register and the 40-bit register isn't yet filled up (the modulo-40 counter keeps track of that), the FSM goes back to state 1, awaiting another input. Once five values are input by the user (the modulo-40 has reached maximum count and the 40-bit register is filled), the FSM goes into state 3.

During state 3, the calculation is displayed automatically for the user to see; there is no need to press any buttons to see the result. In order to begin a new calculation, the user must press the enter key (any other key will not work) so that the FSM returns to state 1, clearing the registers and counters.

The ASM is shown in Figure 13, while the block diagram of the FSM is shown in Figure 14. To aid in debugging the design of the FSM, a total of three LEDs were assigned, one to each of the FSM's states. This way, if the calculator required more or less than five inputs before it would reach state 3, the debugger would be able to tell based on which LED is illuminated, and could adjust the counters or outputs of the ASM accordingly. The LEDs also help the user recognize what state they are in. State 2 lasts for around 100 nanoseconds before switching states, so the LED associated with state 2 never appears to be on for the user.

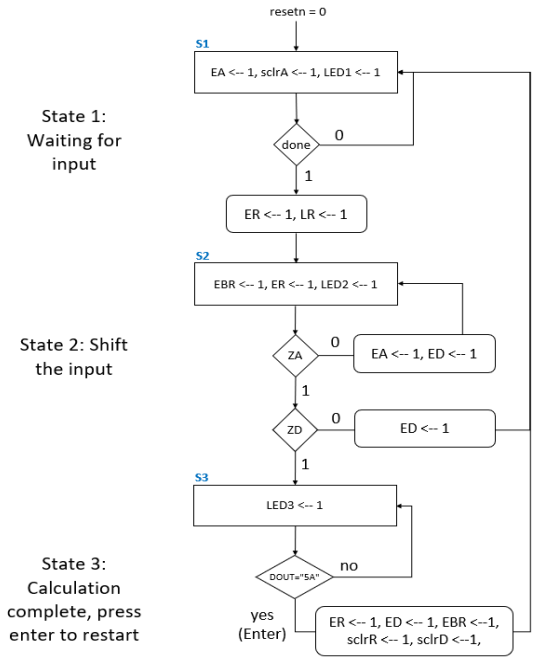

Figure 13: ASM of the 8-bit signed calculator.

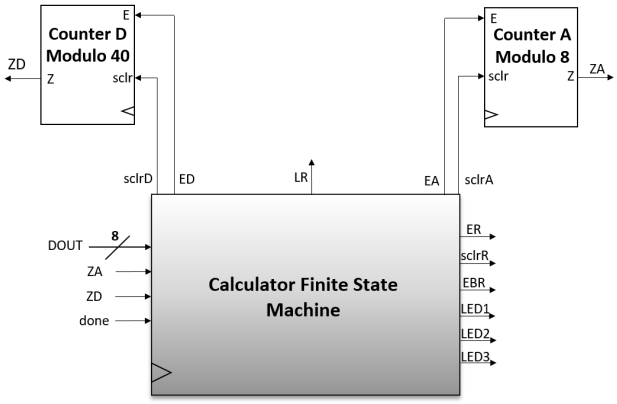

Figure 14: FSM of the 8-bit signed calculator, with the counters attached.

III. Experimental Setup

After each component was completed a testbench was then written for each arithmetic component to search for any possible errors. These errors included things such as not receiving an output at a specific operation on the timing diagram or not having the simulation run at all. However this was easily fixed after reviewing the code and solving the errors. Screenshots of the Vivado simulation results can be seen for each of the operations below.

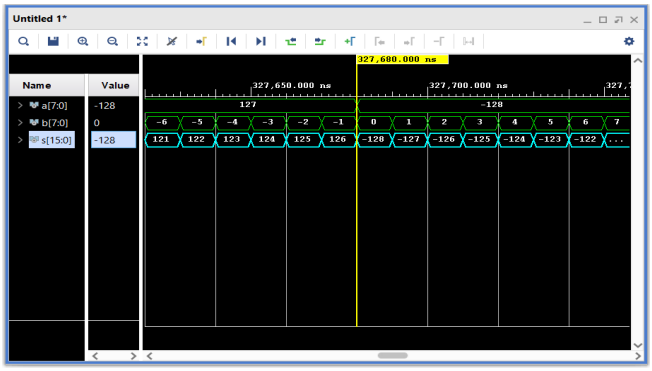

Figure 15: Simulation results for the adder.

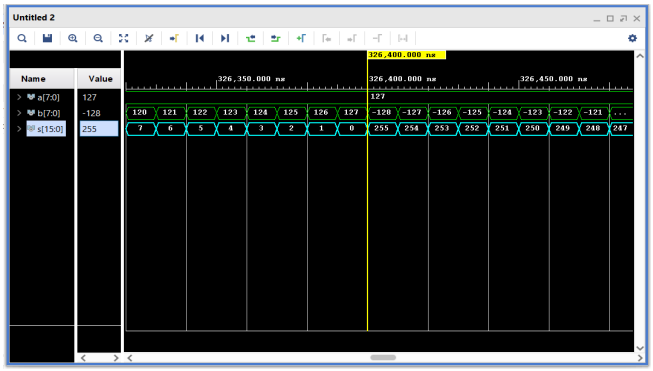

Figure 16: Simulation results for the subtractor.

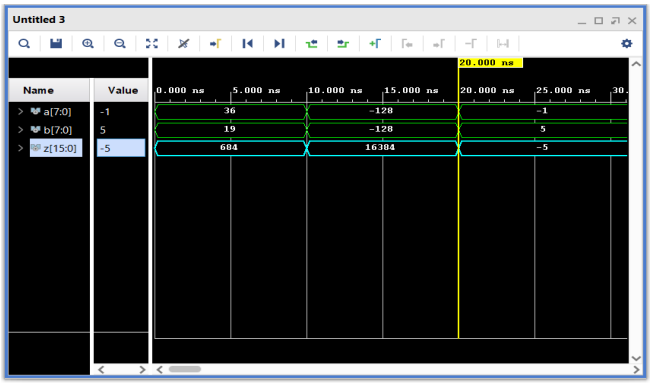

Figure 17: Simulation results for the multiplier.

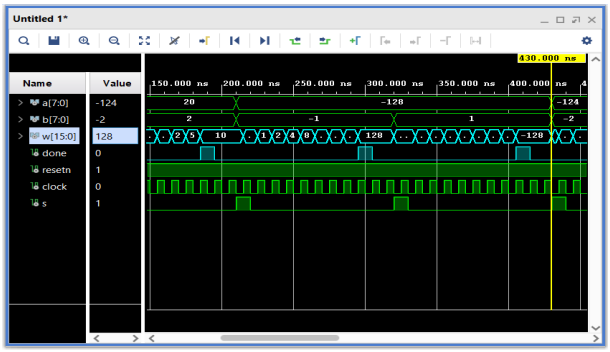

Figure 18: Simulation results for the divider.

IV. Results

The calculator designed in this project works similarly to that of a traditional calculator as it uses multiple operations such as addition, subtraction, multiplication, and division. Instead of having a keypad to make inputs, this calculator instead uses a traditional USB keyboard where the operations are "+ , - , . , /". This calculator can have an input between -128 to 127. The resulting output number will always be in hexadecimal so that if the result is negative then that will be accounted for. These results will be displayed using the seven segment displays on a Nexys A7 board. To view the demonstration of the calculator, visit the link here: https://www.youtube.com/watch?v=1yo0TZqZtCY

Conclusions

Once this calculator was finished, some conclusions were made. The use of block diagrams were super important as without them it would have been nearly impossible to visualize the circuit that was being made. This project was a great way to learn and reinforce past topics such as, designing multiple operations and connecting them in a single top file. To complete this project multiple design files were made along with a testbench and a constraint file. With all of these files properly working the calculator is finished. Once the project was completed it was simple to see why Vivado and VHDL coding is used and how it can be applied in a professional environment.

References

[1] D. Llamocca. (2021). ECE2700 - Unit 7: Introduction to Digital System Design [PDF].

[2] D. Llamocca. "Courses." Reconfigurable Computing Research Laboratory. Accessed: Apr. 20, 2023. [Online.] Available: https://www.secs.oakland.edu/~llamocca/Courses.html

[3] Digilent, Pullman WA, USA. Nexys A7 FPGA Board Reference Manual (2019). Accessed: Apr. 20, 2023. [Online]. Available: https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual