ENGS 31 Final Project Report: The Calculator

Gavin Burns and Sarah Hutchinson

August 2021

Abstract

The goal of our project was to use digital electronic design to create a calculator that can add, subtract, multiply, and divide numbers with results up to 3 digits. Our resulting design uses a computer-connected universal asynchronous receiver-transmitter system (UART) to send signals to the FPGA, where the calculations take place. The calculation process and results are outputted to the 7-segment display on the FPGA, with additional features such as LEDs to indicate which step of the entry process is occurring, and buttons to view past operand entries.

Contents

1 Introduction

For our final project in ENGS31 we were tasked with designing a hardware calculator in VHDL. Specifically, we wanted a calculator that could represent integers between -999 to 999, carry the four main mathematical operations(+ - * /), and be able to recursively apply answers to the next equation. Using processes and ideas we've learned throughout the term we came up with a thorough design and began working. Over the course of the project we practiced top-down circuit design and many different debugging techniques. Furthermore, we added some ease of life capabilities and additive features to separate our design from a simple calculator.

2 Design Solution

2.1 Specifications

Our calculator takes inputs coming from a connected computer, manipulates entered values and operators within the FPGA (calculator functions), and outputs values and results onto the FPGA seven-segment display. Values come into the system through a universal asynchronous receiver-transmitter (UART) system. This is a form of serial input. Using the Waveforms application on the computer, we enter numbers and operators which are then sent to the FPGA, coded as an ASCII code in serial. Our calculator connected to the computer via Analog Discovery 2 (transmits signal). Data is sent from the computer at a baud rate of 115.2k. This rate refers to how many different signals occur per second. This is relevant because since the UART is asynchronous, a clock signal is not sent to the FPGA and therefore it is necessary to use the baud rate to determine how to read the incoming signal. Our system is clocked at 1MHz, which means that a new signal is sent from the computer approximately every 8 clock cycles. Our calculator outputs to the 7-segment display on the FPGA, as well as to the LED lights. Depending on what part of the calculation process is occurring, either the operands, a decimal point to indicate which operator is in use, or the answer is displayed on the board. One of four LED lights illuminates at any given time to indicate which part of the calculation is occurring and what the next entry should be from the user (waiting for operand, waiting for operator, showing answer, or overflow). Additionally, there are input buttons on the FPGA which can be pressed to view past operands on the 7-segment display.

2.2 Operating Instructions

2.2.1 Setup

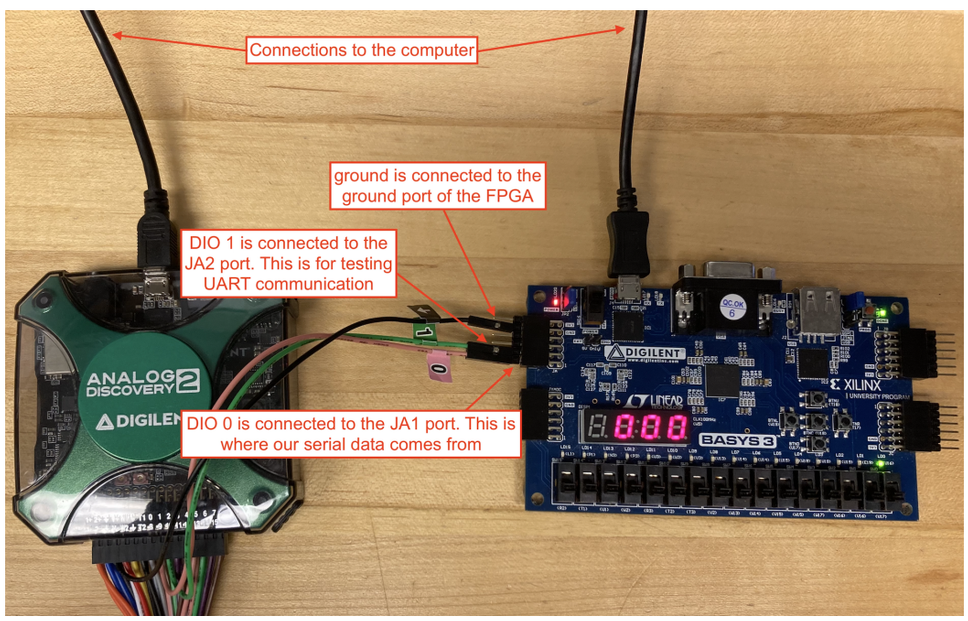

Figure 1: AD2 and FPGA connections. DIO 0(pink) and 1(green) are used to send and receive signals from the AD2 to FPGA respectively. Additional connections are made individually between the Ad2 and FPGA to the computer. Signals are typed in via a keyboard, sent though the AD2 and finally delivered to the FPGA vai serial data.

In order to operate our calculator it is important to understand how to set it up. Since we are using a FPGA and AD2 converter we'll need to connect the data ports together to communicate between them. According to out constraints file in the Appendix, we will be using the DIO 0 and DIO 1 wires on the AD2 to control the transmission and receiving of signals respectively. On our FPGA, the JA1 port is where our serial data from the computer comes from, so we'll connect the DIO 0(pink cable) wire to the JA1 port. Likewise, the JA2 port is the transmission port of our FPGA back to the AD2 and computer moniter. Although we used this port for checking UART capabilities and debugging you can connect the this port to the DIO 1 wire as shown in the figure above. Do not forget to also connect the ground wire of the AD2 to the any ground port on the FPGA. For ease of use we used the ground port closest to the AD2 (JA5).

Lastly, the micro USB ports for both the FPGA and AD2 are connected to the computer to provide power and allow us to enter digits into the calculator via the keyboard.

2.2.2 Communicating

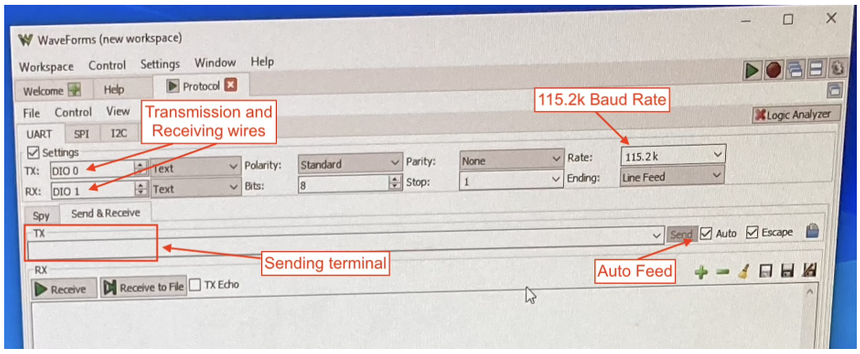

Figure 2: UART Protocol workspace for the Waveforms application. In it you can send data in the sending terminal, set your baud rate, identify transmission and receiving wires, as well as view any receiving signals in the Rx window.

In order to communicate with the calculator, we'll be using the Waveforms app provided by the AD2 to send serial signals to the FPGA. In the protocol workspace, we can send serial data to our system to be processed and displayed on the seven segments displays.

As shown in the figure above it is important to set the baud rate to 115.2k to make sure serial communication is synchronized between the computer and the FPGA.

In the transmission terminal window you can type in inputs to be sent to the AD2 and subsequently the FPGA. In our tests it was helpful to set the feed to auto send such that we wouldn't need to press enter after each input to successfully send it.

2.2.3 Display

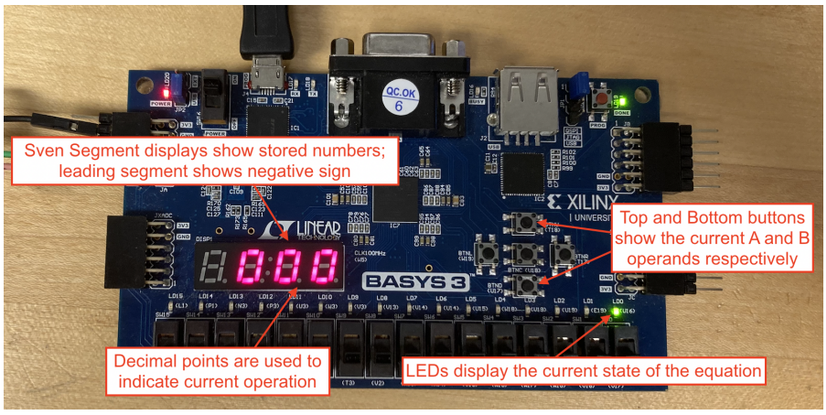

Figure 3: FPGA Display for Calculator. The four seven segment displays show the negative and 3 digits of the inputted and calculated numbers. Decimal points are used to indicate operators(+ - * /). LEDs in the bottom right corner indicate current state of the equation. Starting with operand A and moving to operator, operand B, and Answer.

When entering data from the keyboard it is important to receive some form of feedback to clarify your entering the correct digits. When using our calculator digits will appear in the seven segment display on the FPGA. Our calculator is reserved to only storing 3 digits so the fourth segment will be used to display the negative sign of the number.

Additionally, valid operators will be displayed as specific decimal points found in the bottom right corner of each segment. Only one decimal point will be lit up at any given time and moving left to right on the FPGA, they represent the division, multiplication, subtraction, and addition.

The LEDs on the bottom right of the board are used to communicate to the user the current state of the equation. In the figure above, the rightmost LED is on, indicating that the user will be inputting the A operand. Moving leftward, the second LED represent entering an operator, the third LED is for the B operand, and the fourth LED is on when an answer is being diplayed.

The last feature we have for the display is the control of the top and bottom buttons of the FPGA to display the currently stored A and B operands. When the top button is pressed, the current display will be overwritten to show the A operand used in the equation. The same goes for the B operand when you press the bottom button.

2.3 Theory of Operation

The following steps indicate the typical flow of use for a person using our calculator: (See Figure 11 in Appendix 1.)

Step 1: The user inputs an integer operand between -999 to 999 one digit at a time

Step 2: The user inputs their desired operation between addition, subtraction, multiplication, and division

Step 3: The user inputs their second integer operand between -999 to 999, which will enact their desired operation on the previous operand

Step 4: The user receives an answer to their equation and can either chose to start a new equation(return to step 1) or save this answer and chain it into another equation(move to step 2)

2.3.1 Top Level Design

Before diving into the details of each component, we first figured out what problems we'd come across and describe how our deign plans to overcome them.

Starting to our problem with transmitting to the FPGA, we decided to use UART to process the serial data inputted from our computer and turn it into a single code.

As shown in Figure 5 in Appendix 1, this code would then be fed through an ASCII Converter to interpret the code as one of many keyboard inputs and output the necessary signals. Operators will receive a 2-bit code to differentiate the 4 potential operators; digits will receive there associated 4-bit BCD code; and the negative, enter, and delete keys will all get a single standard logic signal.

Signals sent from the ASCII Converter would then be read by the Controller; which in turn communicates with the Datapath; to organize and store most of this data. The Datapath works to shuttle in digits and operator in from the ASCII Converter and the Controller moderates when, where, and how many signals get stored.

Once all necessary data is collected, the controller will signal the Math Block to preform the calculation. Specifically, the Math Block converts the BCD numbers into signed binary which can then be used to calculate the answer.

Lastly, we decided to use a Binary to BCD converter to take our binary answer (and any other variables) and transform them into a BCD signal readable by the seven segment display. Likewise, other signals outputted by the Controller are given to the seven segment displays to select what should be displayed at each point in the calculation.

2.3.2 UART

The UART input component takes the serial input from the computer, which arrives at a rate of 115,200 signals per second, and compiles it into an 8-bit parallel signal which can be read by the ASCII converter to determine which value it represents (ASCII codes are 8 bits long). In order to understand how the UART reciever works, it is necessary to understand the components of a UART signal. A UART signal is 10 bits long. The signal is at 1 when it is idle, and the first bit is always 0. This is called the start bit and indicates that the next 8 bits will represent a value. The last bit is always 1, and is called the stop bit. The synchronizer uses two flip-flops to ensure that the incoming signal is being fed to the circuit at the 1MHz system clock speed. This is particularly important because the incoming UART signal is asynchronous. See Figure 6 in Appendix 1 for UART Block Diagram reference. The synchronized signal is passed into the deserializer shift register, which, in conjunction with the UART state machine, converts the serial signal to a parallel signal. The state machine starts in the idle state. See Figure 12 in Appendix 2 for UART State Diagram. When the first bit is detected to be a 0, the state machine moves into the count_baud_1 state. This sends a signal to the counter that to start counting. Since the signal comes in at a rate of 115,200 signals per second and the system is at 1MHz, in general, the counter in general counts to 8 before it enables timeout, which, causes the state machine to move to the shift_1 state, make shift_en high, and have the contents of the shift register to be shifted. It is ideal to read the incoming signal in the middle of the signal to get a steady value, so the count_baud_1 state enables timeout after 1 clock cycle, thus there is a different state for the subsequent counting and shifting states. After each shift, the shift counter increments. While the counter is counting, the shift counter holds its current value. Once there have been 10 shifts, the load state is entered, during which load_en is high and the contents of the shift registers are loaded into the parallel output.

See Figure 15 in Appendix 3 for the simulation image.

2.3.3 ASCII Converter

The goal of the ASCII converter is to take the 8-bit signal provided by the UART and decode it into specific signals. Knowing that the 8-bit signal (labeled as par_output in block diagram) represents a single key stroke from a keyboard, we deduced that it would be best to utilize a look-up-table to assign the input number to a specific output signal. (See Figure 7 in Appendix 1 for ASCII Converter Block Diagram).

To do this we first had to decide which keys would be registered to which signals. This was fairly straightforward and by using an ASCII look up table we were able to figure what the par_output code would be for keys associated with the numbers 0-9, all necessary operators(+ - * /), and the enter key. Consequently, both the backspace and delete key were not eligible inputs for the Waveform program that we were using to communicate with the system. Instead, we chose to register the spacebar as our primary delete key for the continuation of our project.

The actual hardware implementation of our ASCII Converter was just a look-up-table to check the input signal and set the corresponding output signals. For example; when the par_output signal is recognized to be some number, valid_int will go high and dig_code will parse the input into its BCD counterpart. The same goes for valid_op and op_code for when an operator code is detected, valid_neg for when the '-' key is detected, and entr/dlt for when the enter key or spacebar is detected.

For a more detailed analysis of the ASCII converter's operation please review ASCII Converter testbenches 1 (Figure 16) and 2 (Figure 17 in Appendix 3.

2.3.4 Main Controller

The crux of our calculator is the main controller that signals to many of the other components when to do each of their processes. (See 13 in Appendix 2 for Calculator State Diagram). Once data is decoded by the ASCII Converter it is up to the Controller to read these signals and move though its states, commanding the other components as it goes. The structure of the Controller's state machine is such that we can break down the flow of state into four individual sections(all of which are cycled through using the enter key).

The initial state of the machine begins in the first section that we'll call the "operands" section (colored in blue in the above figure). In this section, user inputs are read to store and delete numbers to form the A and B operands of the equation. Valid integers have the controller signal the datapath to store this number while the spacebar has the controller signal to delete the least significant digit currently stored. Furthermore, the minus symbol in this section is used to toggle the negativity of a number and other operator symbols have no effect on the system. Another feature that is present while in this section is a counter that used to make sure the user does not input more than 3 digits. Our calculator has a capacity to work with numbers between -999 to 999, so it was important to us to keep the user within these bounds during number inputs. Upon leaving this section the controller will signal to take whatever 3-digit number that has been created and store it as the A or B operand as determined by the FLAG signal.

The FLAG signal is used to differentiate A operand from B operand storage. Upon storing a number in operand A, the signal toggles high to signal that the next number stored will be operand B. It is then toggled down when to calculation is complete and the Restart state is reached.

The next section of states is the "operator" section(yellow). Similar to the operands section except user inputs are read to store and delete operators. In this section numbers have no effect and the minus key is registered as a minus symbol. Following this section, the Controller briefly enters the Clear state to make sure all previous digits have been cleared before returning to the Operand section.

After operand A, B, and an operator have been stored the next section the Controller goes to is "computation". This section signals the Math Block to compute an answer and waits to see if the user wishes to chain the answer with another equation or start again from scratch. If the answer generated, overflows the calculator's capacity then the Controller will move to the Overload state which requires the user to begin a new equation. Alternatively, any other numerical answer can be chained into a new equation by entering an operator. This will move the Controller into a brief buffer state to store the answer as the A operand before going to the final section.

The "chain operator" state(Red) is nearly identical to the "operator" section except for the signal_output signal. This signal(along with LED_output) communicate with the seven segment display to tell them which numbers to show. In order to show the answer and new operator at the same time, we needed to create this additional section with a separate signal_output code.

For a waveform clarification on the controller's operation, please see the Calculator Controller Testbench (Figure 18) in the Appendix 3.

2.3.5 Main Datapath

According to the operating instructions of our system, the user will enter in a single digit at a time to ultimately form a 3-digit number between -999 and 999. In order to take single digits and assemble them into full numbers, we needed a way to preserve their order as multiple numbers are entered; this is done in the calculator's Datapath. (See Figure 8 in Appendix 1 for Datapath Block Diagram).

When the dig_en signal is sent from the Controller, each number register receives the previous registers digit and num0 receives the newest dig_code from the ASCII converter. Likewise, when dig_dlt is sent, each number register is enabled to tak in a new digit, but the digit they receive is taken from the register ahead of them.

As you can see in the diagram above the multiplexer determine where the data being stored comes from. Its select bit is the dig_dlt signal from the Controller; when low it sends the previous register's digit, and when its high it sends the following register. It should also be noted that the num2 register will always be reset when the dig_dlt or dig_clr signals are passed to ensure that a zero is passed back to the previous register when the delete signal is sent.

This Datapath also controls the current state of the the numbers negativity. Whenever the neg_en signal is passed to the Datapath, the neg output will toggle between high and low with every press. We can also see in the figure above that the dig_clr signal also resets the current state of the neg register because negativity is very much tied to the actual number itself.

Lastly, the Datapath stores and outputs the current operator inputted into the system. When op_en is sent, the op_code from the ASCII converter is stored in the register and an op_full signal is sent back to the controller. This signal is to indicate that a operator has been stored and user can officially move on to the next state.

A full waveform of the Datapath's operations can viewed in Appendix 3 under Calculator Datapath Testbench. (Figure 19.)

2.3.6 Math Block

Following up on the calculator's Datapath, the Math Block takes in its digits(num0-2), negative, and operator to form an equation for computation. (See Figure 9 in Appendix 1 for Math Block Diagram).

As a reminder, the num signals from the Datapath represent the 3 digits of a base 10 number. So before computation num0, num1, num2 are turned into a single binary number by multiplying the hundreds digit(num2) by 100, the tens digit(num0) by 10, and adding them all together. Additionally, if a negative is present for this number then the system will preform a two's compliment conversion to make the number a

negative. After this conversion, the new signed binary number is stored in operand A or B (as decided by the controller) until it is needed for computation.

Speaking of which, the computation preformed is determined by the stored operator value also from the datapath. This can be seen as the multiplexer that takes its select bit as the op signal. This value is computed but not stored officially until the math_en signal is outputted high by the controller.

After an answer is calculated it is stored in the Answer_temp register and answer_full signal is fired directly back to the Controller. The new answer is then immediately checked to see if it exceeds the calculators maximum value; if so the overflow signal goes high.

Likewise, after the an answer is generated, it runs through binary conditioning to separate the negative sign from the rest of the number. This is to make the outputting answer digestible for the binary to BCD converter while preserving the negativity of the number. The specific process of the binary conditioning block is as follows: the number is check to see if it less than 0; if so, two's complement is preformed and y_neg/A_neg is set high. The same process is applied to the A operand which is also run through its own binary to BCD converter so it can also be shown on the seven segment display.

One last feature of the Math block is its ability to store the computed answer back into the A operand. When the controller sends the Answer_store signal it'll loop the Answer_temp signal back to the operand A register for storage. This allows the user to chain answers into new equations following a calculation.

Please refer to Computation Testbench (Figure 20) in Appendix 3 for an in-depth example of operation.

2.3.7 Binary to BCD

The binary to BCD block is necessary to convert the signed binary number, the calculation form, into sets Binary Coded Decimal bytes, which are sent to the 7-segment display for output. (See Figure 10 in Appendix 1 for Binary to BCD Converter). This is done according to the Double Dabble algorithm. (https://en.wikipedia.org/wiki/Double_dabble) The binary to BCD converter takes in a 10-bit unsigned value. The output will be three 4-bit numbers that correspond to the three digits of the associated decimal number. (0000 0000 0000, with the first group of four representing the 100s digit and the third group of four representing the 1s digit). The most significant bit of the binary number gets shifted into the least significant bit of the 12 BCD bits. Then the binary number is shifted over so there is a new msb. This occurs 12 times. If at any point a group of 4 has a value greater than or equal to 5, 3 is added to it. This is implemented with multiplexers and a state machine. (Intially loads all the digits in, then shifts and checks if anything is greater than or equal to 5 each time). (See Figure 14 in Appendix 2 for the Binary to BCD State Diagram.) Each time a shift occurs, a shift count is incremented. Once 10 shifts have occurred, the 3 BCD numbers are outputted.

(See Figure 21 in Appendix 3 for the simulation of binary to BCD).

2.3.8 Display

(See Figure 11 in Appendix 1 for Display Block Diagram).

Although we now have all of our desired signals in the form that is transferable to the seven segment display; we'll need some sort of way to select which signal is displayed at any given time. And by focusing on the keyword "select" we immediately understood that we should use a multiplexer to do this job. There are 4 seven segment displays that we have programmed to take a 5-bit input determining there output. Including the decimal point to signify the current operator and we have a total of 5 inputs to shuttle to the seven segments. Furthermore, we have four potential outputs to display at any given time: the digits and operators being inputted by the user, the answer, the answer and the operator the user is inputting(for chaining), and an overload sequence for when the answer is too big.

An RTL design of this block can be seen in the figure above, as you see, the first round of signal selection is conducted by the signal_output signal from the controller. This clarifies what section of the equation the system is currently in and selects the appropriate output.

Two multiplexers follow this selection, each of which are for the additional button features that allow the user to view each operand of the equation. Signals button_A and button_B represent the top and bottom buttons on the FPGA board and when pressed will select the corresponding operand to be displayed on the seven segments. It should also be noted that selection of operand A has priority over operand A as seen in the figure above.

This same sort of selection process is used to determine which LED should light up in the calculator sequence. As you progress through the calculator, the signal LED_output will count from 1 to 4 indicating if the user needs to input operand A, an operator, operand B, or what to do with the answer.

3 Evaluation

3.1 Functionality

The goal of this project was to design and build a functioning calculator using what we've learned form ENGS31. Specifically, we were tasked with creating calculator that could receive external keyboard inputs, preform the basic mathematics operations, and display our answer on seven segment displays. With those criteria (and some we created) we were very much successful in our endeavour.

Our calculator works almost exactly as we planned it to; it takes in numbers from a keyboard and actively displays them on the seven segments. These numbers then form our operands and can be applied to the main mathematical operations with the answer successfully being displayed.

Within the provided time frame for design we were also able to add additional feature to help users and increase functionality. We made it so both positive and negative numbers can be processed. Added LED

sequencing to notify the user their current state within the equation. Allowed the user to check their A and B operands after calculating their answer. Handle potential answer overflow scenarios. And gave the users the option to chain answers into another equation for more options.

3.2 Review

Looking back on the construction of the calculator there were a lot of aspects that were really cool to work on. Figuring out the seven segment displays and being able to control their output using the FPGA buttons was really nice. We originally included this feature as a way to debug our code; such that we could check each intermediate signal to see how out numbers changed as they flowed through the system. Button upon completion we adapted it slightly to offer some useful information for our user. It was also interesting (and a little bit challenging) to get the system to work with both positive and negative numbers. Casting the variables into unsigned, doing the computation, and translating/resizing the back proved to be more of a challenge than we expected but it was ultimately very cool to see working.

One functionality that we failed to implement was the ability to receive serial signals from the FPGA and display them back on the computer monitor. We ran into many issues with converting the signals back into ASCII codes and getting the timing working for the components to work together. In the end we were unfortunately unsuccessful with this plan and were forced to redesign the display location to the seven segments on the FPGA. But even with this setback, it was really amazing to design a project from the ground up and get to use everything I've learned thus far into making it work.

4 Conclusions

The initial goal of our project was to create a calculator with computer numeric input and computer numeric output that could handle addition, subtraction, multiplication, and division of positive and negative numbers for results up to 3 digits. Our final product is a calculator that uses computer input to add, subtract, multiply, and divide positive or negative numbers. The output is displayed instead on the 7-segment display of the FPGA. We added usability features such as LED status lights and buttons to review inputs. Therefore, the functionality of our design was exactly the same as in our proposal. We chose to output our results differently because we found that sending data back to the computer was not working well. Using the FPGA outputs allowed our design to look more like a calculator and to have the added LED and button features. Future groups considering this project should work on testing the hardware/computer connection early, as this is actually somewhat difficult. It also allows them to see what the system is outputting, which is helpful for debugging. Additionally, connecting the 7-segment display for debugging purposes is very useful for monitoring what is happening at every step of the calculation process. Overall, implementing ways to assess what is happening within the design is essential. Wheres with computer coding it is possible to have the computer print values, with digital design, it is necessary and important to implement a hardware equivalent.

5 Acknowledgements

In this project Sarah worked on the design and construction of the UART and serial communication of the computer with the calculator. Likewise, she also designed the data conversion between Binary to BCD using the double dabble algorithm. Gavin was tasked with the main controller and datapath of calculator and how it would sort and process inputs from the user. He also designed the Math block and how to chain answers into new equations. Both team members worked together to debug, testbench, and implement every aspect of the program and worked together to design an intuitive display and additive features for the users.

We would also like to thank Ben Dobbins, Professor Eric Hansen, and our TA Yefri for their help and guidance during the constructs of this calculator.

6 References

For the binary to BCD conversion process: https://en.wikipedia.org/wiki/Double_dabble

Appendix

The Calculator
Group 9 - Gavin Burns and Sarah Hutchinson

Appendix 1: Block Diagrams

System Flow

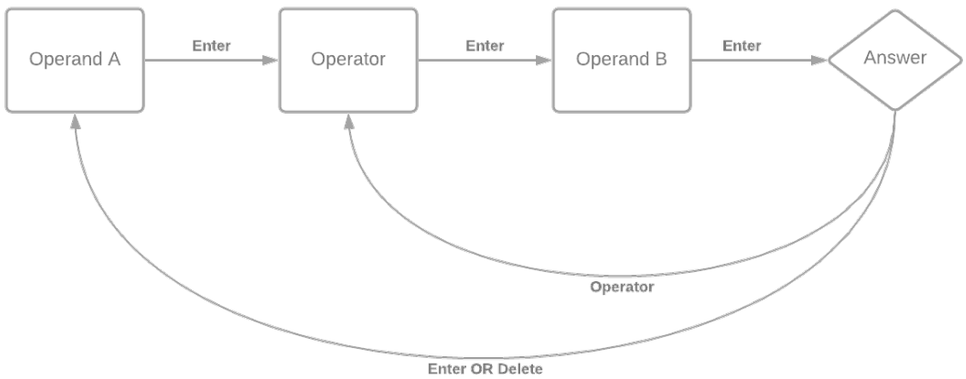

Figure 4: Example Flow of Use

System Block Diagram

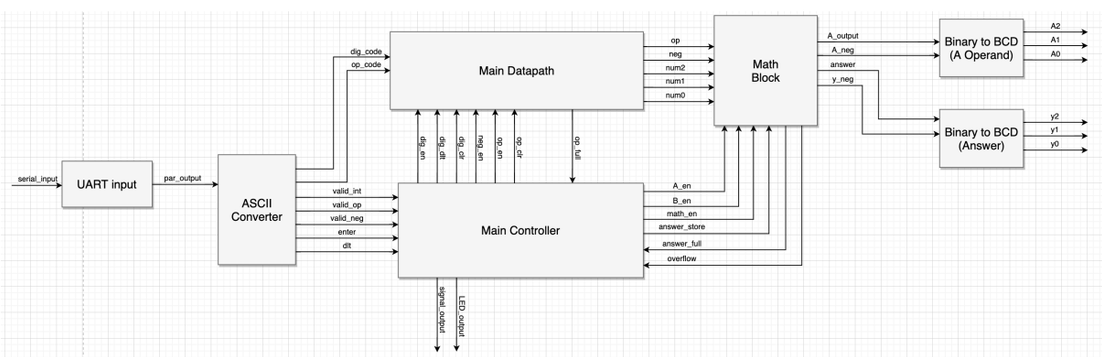

Figure 5: The top level diagram for our project. Serial data is processed in the UART where it is turned into an 8-bit signal. The ASCII Converter decodes this signal into codes denoting integers, operators, and other valid inputs. These signals are read by the Controller which tells the Datapath to store or delete these values. After two operands and an operator are entered, the Math Block computes the answer. The generated answer is then converted into BCD which will be displayed on the seven segments

UART Block Diagram

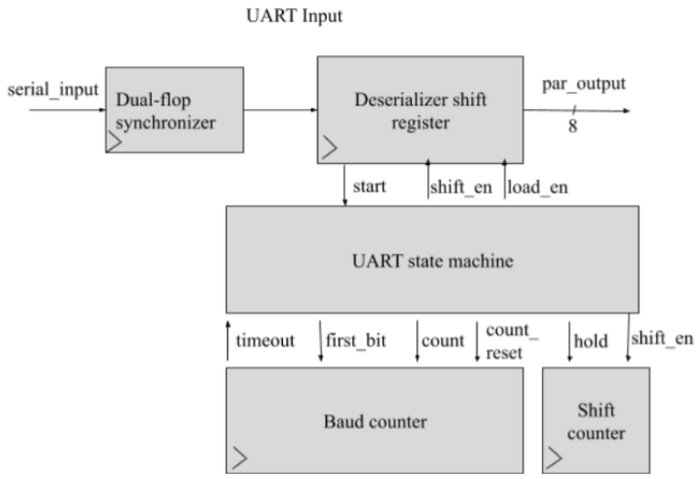

Figure 6: UART converter Diagram. Serial input signal goes through a synchronizer and a deserializer and is outputted as an 8-bit par_output signal, which is sent to the ASCII Converter

ASCII Converter Block Diagram

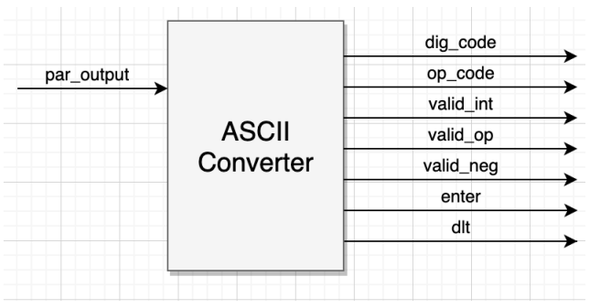

Figure 7: ASCII converter Diagram. 8-bit par_output signal is fed in and decoded into a variety of signals to be interpreted by the Datapath and Controller

Datapath Block Diagram

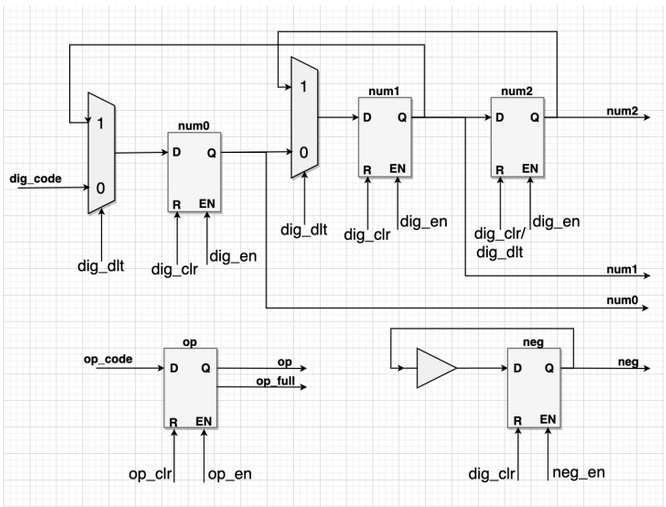

Figure 8: Block diagram of the calculator's datapath. dig_code and op_code are taken from the ASCII converter and stored in their respective registers. As multiple digits are input, currently stored digits are shifted through the registers to preserve their order. The reverse process occurs when digits our deleted. A T flip-flop is used to toggle between the numbers negativity as neg_en is triggered.

Math Block Diagram

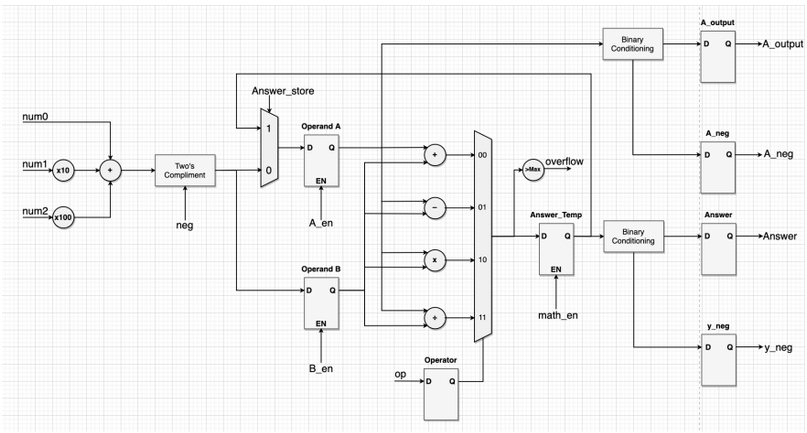

Figure 9: RTL design of the Math function block. num0, num1, num2, and neg are signals from the Datapath that are converted into a signed binary number and stored in either operand A or B. At the command of the Controller, the math block will compute an answer using the inputted operands and operator; and before outputting the signal, will separate the answer into an unsigned binary number and its negativity. Additionally, operand A can be converted in the same fashion for display on the seven segments and the answer can be stored back into operand A for chaining equations.

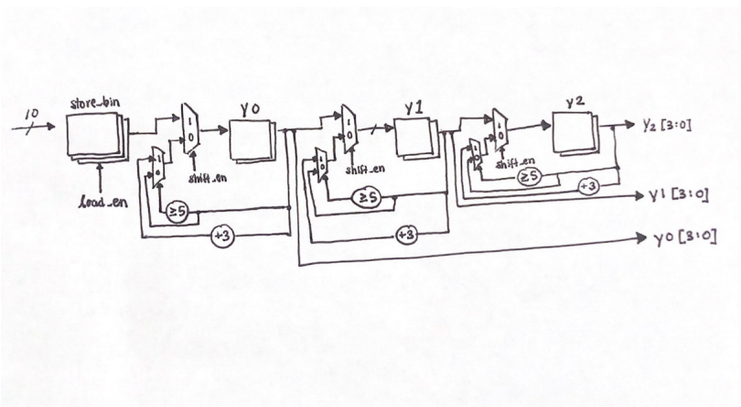

Figure 10: Block diagram for Binary to BCD Conversion. 10-bit binary converted to BCD via the Double Dabble Algorithm, which involves shifting the 10-bit binary input into 3 sets of 4 bit numbers (BCD) and adding 3 if the value of a 4 bit group is ever greater than or equal to 5.

Display Block Diagram

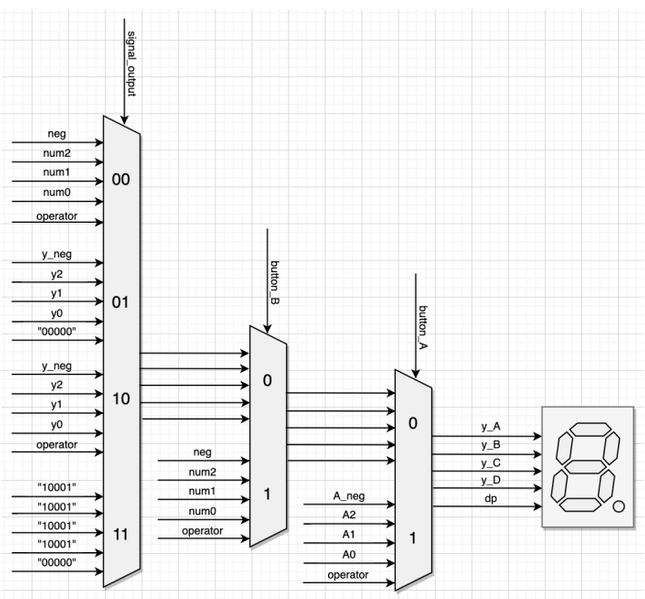

Figure 11: Diagram for the seven segment display output. During the typical process of the calculator, the controller will send signals to the first multiplex to determine what data to display. The user also has the option to press the top and bottom button on the FPGA to display the current A and B operands respectively.

Appendix 2. State Diagrams

UART State Diagram

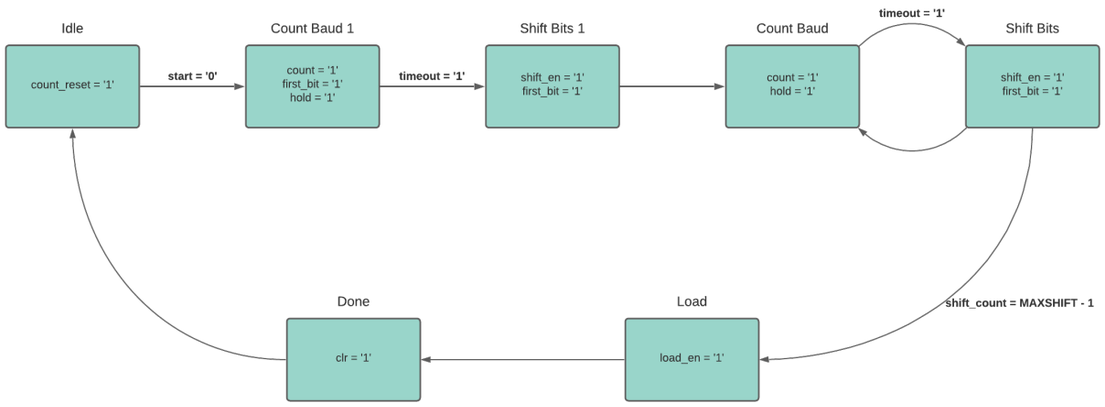

Figure 12: UART State Diagram

Calculator State Diagram

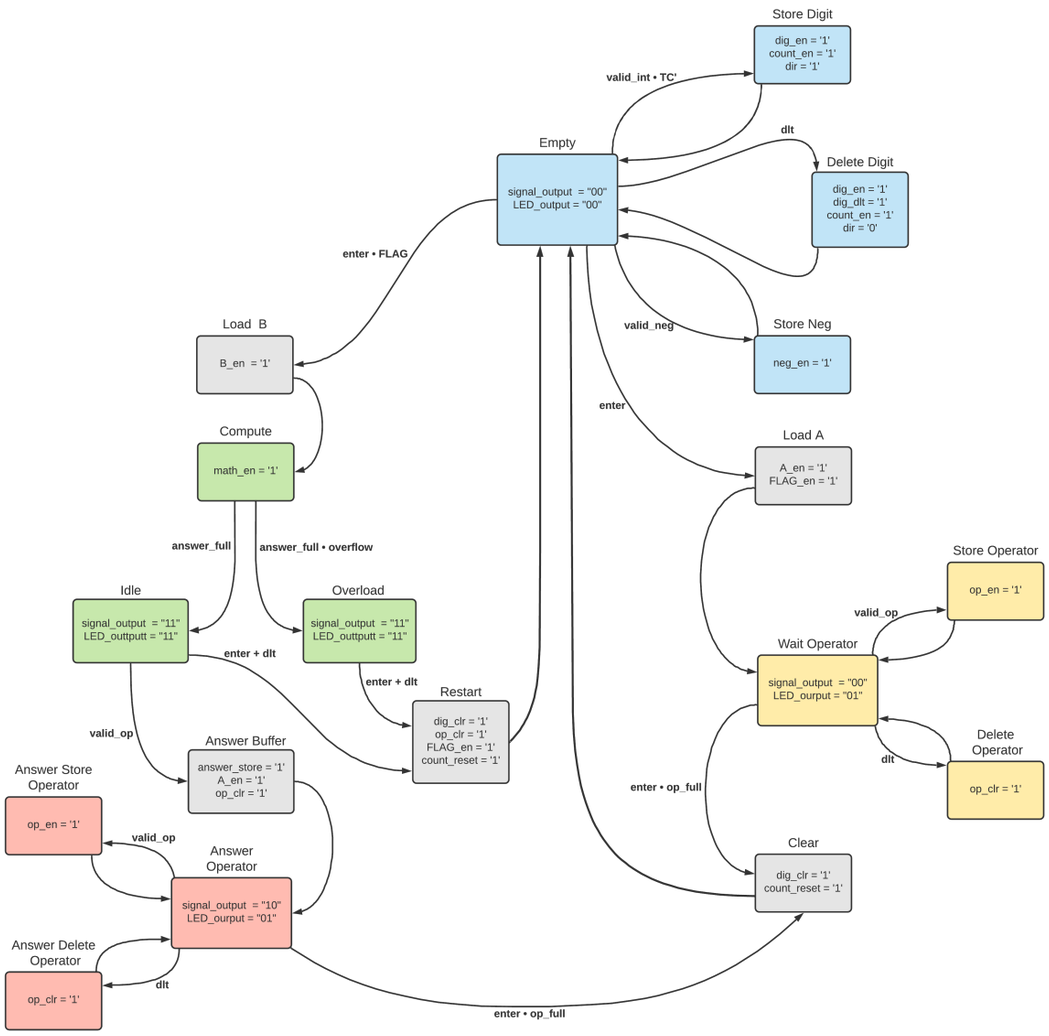

Figure 13: State diagram of the main controller. The flow of the controller is split up into 4 main sections with conditioning states in between. Each of the sections in state machine represent holding areas that are waiting on user input to process data and move to the next sections. The blue represents inputting operands, the yellow is inputting operators, the green is from answer computation, and the red is designated for chaining equations together

Binary to BCD State Diagram

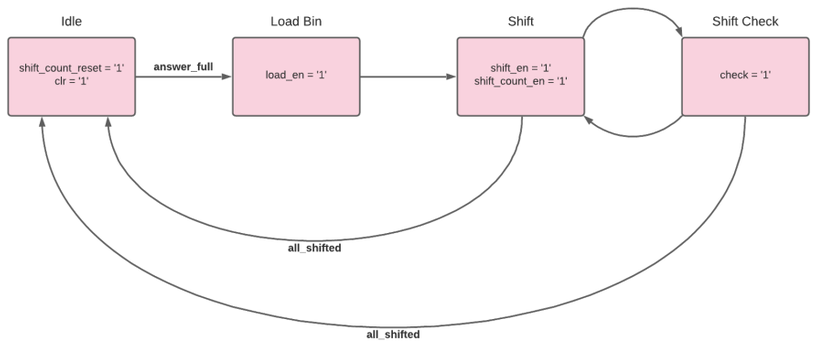

Figure 14: Binary to BCD State Diagram

Appendix 3: Testbench Waveforms

UART Testbench

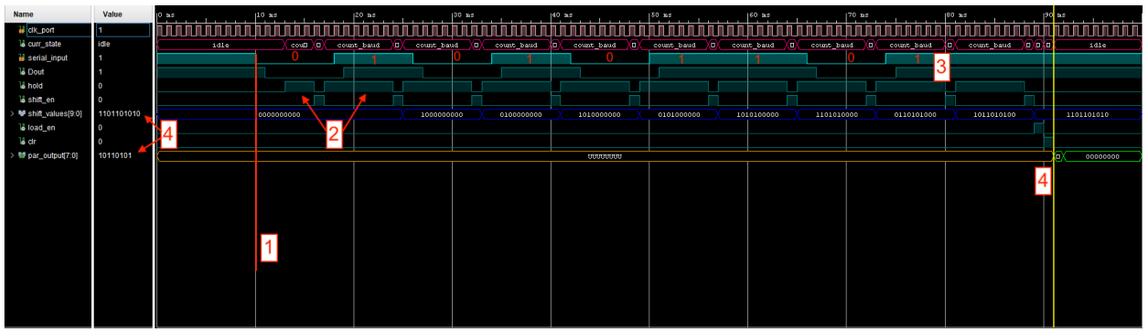

Figure 15: Simulation waveform of our UART converter. Serial signal is delivered in reverse to the UART converter. 1)When the serial input goes low the block is trigger to begin shifting in numbers. 2)In between shifts the values are held constant for the duration of the baud rate. Counter is active for this period and the system is held in the Count_Baud state. 3)Signal is being fed in reverse order into the UART. 4) When the maximum number of shifts is reached, the shifted values are sent out as a parallel signal. The parallel out signal is taken from the shift_signal[8 downto 1].

ASCII Converter Testbench 1

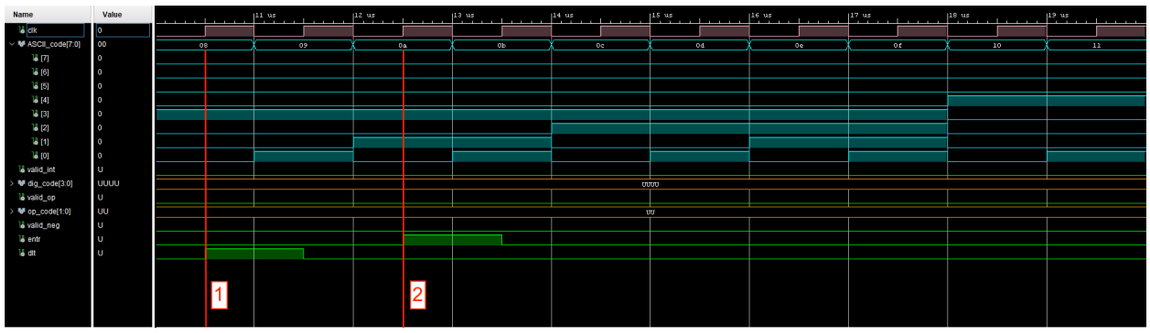

Figure 16: Simulation waveform of the ASCII converter. 1)entr signal goes high when enter key is pressed. 2)dlt(spacebar) signal goes high when delete key is pressed

ASCII Converter Testbench 2

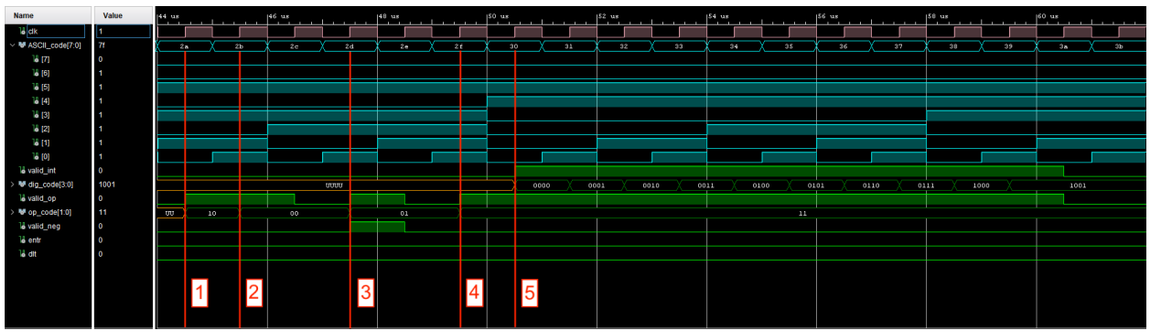

Figure 17: Simulation waveform for the ASCII converter. 1)Asterisks key ASCII code is recognized; valid_op goes high and op code gains the signal for multiplication. 2)Plus key ASCII code; addition stored. 3)Subtraction key ASCII code; subtraction stored and valid_neg goes high. 4)Slash key ASCII code; division is stored. 5)As digits are recognized, their binary form is stored in dig_code and valid_int goes high

Calculator Controller Testbench

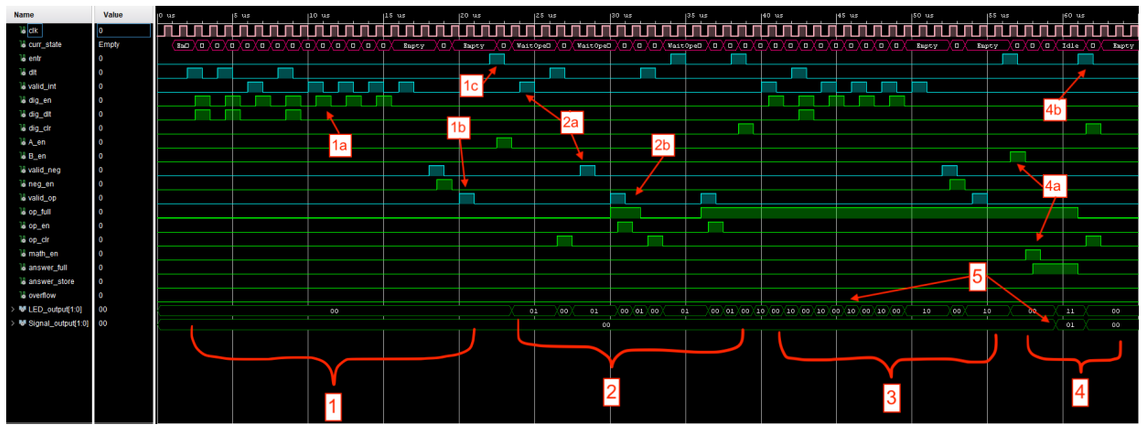

Figure 18: Simulation waveform for the main calculator controller. 1)The "operand A" state; where the user can input any 3 digit integer. 1a)When valid_dlt and valid_int signals are detected controller will output dig_dlt and dig_en to add/remove stored digits. A max of 3 digits can ever be stored. 1b)Detection of a valid_op signal has no effect in this state. 1c)Progression to the next state is enacted with the entr signal. 2)The "operator" state. 2a)detecting a valid_int and valid_neg signals have no effect in this state. 2b)Detecting a valid_op will enable op storage and signal that an operator is full. 3)The "operand B" state, same as "operand A" state. 4)Answer state, inputs of previous states our processed and an answer is calculated. 4a)When state is entered math_en goes high and an answer is generated. 4b)Upon leaving state, all currently stored variables are cleared. 5)As the system progresses through different states, it signals to the seven segment display what should be shown.

Calculator Datapath Testbench

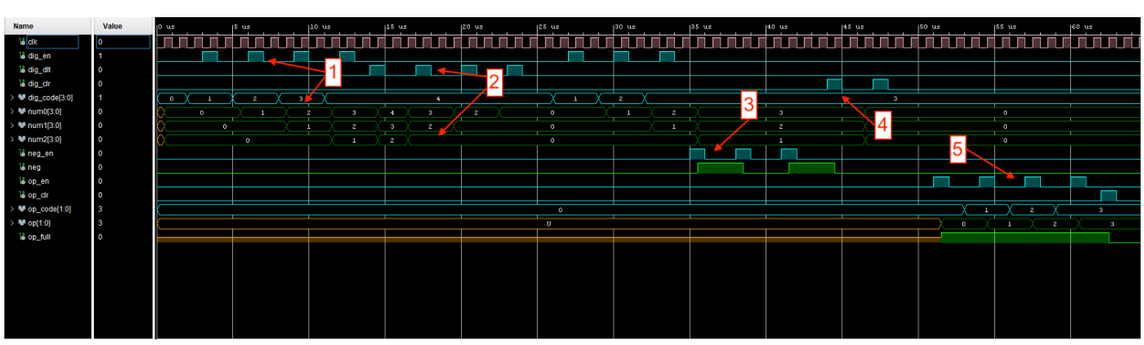

Figure 19: Simulation waveform of the main calculator datapath. 1)When a dig_en signal is detected, current dig_code is stored in the num0 register and currently stored dig_codes are shifted over in place. 2)dig_dlt signals will delete the dig_code stored in num0 and shift other codes back. 3)neg_en will toggle the negativity of a number. 4)dig_clr will wipe all number and negativity registers. 5)op_en and op_clr store and delete operators respectively. op_full is only high if an operator is present and op_clr has not been pressed

Computation Testbench

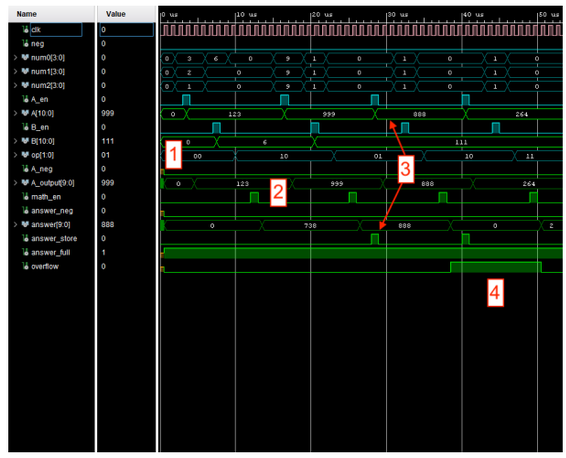

Figure 20: Simulation waveform of computation block; where previous inputs are processed and an answer is output. 1)A_en and B_en take the current BCD numbers, compute them into binary and store them in the A and B operands respectively. 2)When math_en triggers, the system will take the current values of A and B, along with the current operator and calculate and answer. 3)When answer_store is detected, previous answer is stored in the A operand and can be used for recursion. 4)When an answer exceeds the bounds (-999 to 999) the answer will be set to zero and the overflow signal will go high.

Binary to BCD Testbench

Figure 21: Simulation waveform of the BCD to binary converter. 1)When answer_full goes high converter begins shifting in binary_signal. 2)Between each shift the current BCD variables are checked to see if they are greater than 5. If so, the converter adds 3 to them an continues with the shifts. 3)Upon all shifts being conducted each BCD bin is filled with their respective quantity. In this waveform, the number coming in was 999 and the output was a 9, 9, and 9.

Appendix 4: VHDL Programs

UART_Input Code

```
----------------------------------------------------------------------------------
-- Company: Thayer School of Engineering
-- Engineer: Gavin Burns & Sarah Hutchinson
--
-- Create Date: 08/14/2021 12:37:03 AM
-- Design Name:
-- Module Name: ASCII_converter - behavior
-- Project Name: ENGS31 - [REDACTED]
-- Target Devices: Basys3 FPGA
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

entity UART_input is
port(
  clk_port:     in std_logic; -- 1 MHz serial clock
```

```
  serial_input: in std_logic; --inupt from the keyboard
  par_output:   out std_logic_vector(7 downto 0); --output signal

  par_output_check: out std_logic_vector(7 downto 0)); --for debugging

end UART_input
architecture behavior of UART_input is
  constant MAXTIME:     integer := 8; -- ~1 MHz /115,200 baud
    constant MAXSHIFT:    integer := 10; --shift 10 times
  signal time_count:    unsigned(13 downto 0) := (others => '0'); --keeps track of time
    signal shift_count:   unsigned(4 downto 0) := (others => '0'); --keeps track of number
    of shifts
    signal shift_values:        std_logic_vector(9 downto 0) := (others => '0'); --currently
     shifting values

    signal hold:            std_logic := '0'; --passes the shifting process
    signal shift_en:    std_logic := '0'; --enables shift_values to be shifted
    signal load_en:     std_logic := '0'; --loads current values into par_out_temp
    signal clr:             std_logic := '0'; --clears all currently held values

    signal count:     std_logic := '0'; --keeps track of time between shifts
    signal timeout:     std_logic := '0'; --signals when MAXTIME has been reached
    signal count_reset:     std_logic := '0'; --resets the count

    signal Dsync:       std_logic := '1'; --Synchronizes incoming signal
    signal Dout:      std_logic := '1'; --^^^^^^^^^^^^^^^^^^^^^^^^^^
    signal Start:           std_logic := '1'; --signals the start of the reading

    signal first_bit:       std_logic := '0';
    signal par_out_temp:  std_logic_vector(7 downto 0); --temporarily holds the parallel out
     signal

    type state_type is (idle, count_baud_1, shift_bits_1, count_baud, shift_bits, load, done
    ); --7 state machine
    signal curr_state:    state_type := idle; --begin in Idle
    signal next_state:    state_type;


begin

    synchronizer: process(clk_port) --syncronizes the incoming signal
    begin
      if rising_edge(clk_port) then
          Dsync <= serial_input;
            Dout <= Dsync;
        end if;
    end process synchronizer;

    FSM_inc: process(clk_port) --moves through the state machine at the rising edge of the
    clock
    begin
        if rising_edge(clk_port) then
           curr_state <= next_state;
        end if;
    end process FSM_inc;


    FSM_comb: process(curr_state, timeout, Start, shift_count)
    begin
      next_state <= curr_state; --default values
        shift_en <= '0';
        load_en <= '0';
        hold <= '0';
        count <= '0';
        count_reset <= '0';
        first_bit <= '0';
        clr <= '0';

        case curr_state is
            when idle =>
                count_reset <= '1'; --reset count

                if Start = '0' then --if incoming signal goes low
                    next_state <= count_baud_1; --move to Count_Baud_1(start shifting in
    bits)
                end if;

            when count_baud_1 =>
              count <= '1'; --begin counting(for baud rate)
              first_bit <= '1';
              hold <= '1'; --hold onto current value

                if timeout = '1' then --when MAXCOUNT is reached
                    next_state <= shift_bits_1; --move to Shift_Bits_1
                end if;

            when shift_bits_1 =>
                shift_en <= '1'; --shift in new bit
                first_bit <= '1';

                next_state <= count_baud; --else move to count Baud
```

```
113
114             when count_baud =>
115               count <= '1'; --begin counting(for baud rate)
116               hold <= '1'; --hold current values
117
118                 if timeout = '1' then --when MAXCOUNT is reached
119                     next_state <= shift_bits; --move to Shift_Bits
120                 end if;
121
122             when shift_bits =>
123                 shift_en <= '1'; --shift in new bit
124
125                 if shift_count >= (MAXSHIFT-1) then --if shift_count is greater than
    MAXSHIFT - 1
126                   next_state <= load; --move to Load state
127                 else
128                   next_state <= count_baud; --else move to count Baud
129                 end if;
130
131             when load =>
132               load_en <= '1'; --load in currently shifted values
133
134                 next_state <= done; --move to Done
135
136             when done =>
137               clr <= '1'; --clear all currently held values(get ready for next set of values
    )
138
139                 next_state <= idle; --move to Idle
140
141         end case;
142   end process FSM_comb;
143
144     shift_counter: process(clk_port, shift_en, hold) --counts the number of shifts
145     begin
146         if rising_edge(clk_port) then --on rising edge
147             if hold = '1' then --if hold is high
148                 shift_count <= shift_count; --hold values
149             elsif shift_en = '1' then --if shift_en is high
150                 shift_count <= shift_count + 1; --increment shift count
151             else
152                 shift_count <= (others => '0'); --else, reset count
153             end if;
154         end if;
155     end process;
156
157   timer: process(clk_port, count_reset, first_bit, time_count, count) --counts for baud rate
158   begin
159     if rising_edge(clk_port) then --on rising edge
160       if count = '1' then --if count is high
161               time_count <= time_count + 1; --increment time_count
162         if first_bit = '1' then --if this is the first bit
163                   if time_count >= (MAXTIME/2 - 3) then --set timeout a little sooner
164                     timeout <= '1';
165                     time_count <= (others => '0'); --reset count
166                     else
167                       timeout <= '0';
168                     end if;
169               else --if not the first bit
170                   if time_count = (MAXTIME - 2) then --timeout triggers at normal rate
171                       timeout <= '1';
172                       time_count <= (others => '0');
173                     else
174                       timeout <= '0';
175                     end if;
176               end if;
177       else
178               time_count <= time_count; --defualt
179             end if;
180
181             if count_reset = '1' then --if count_reset is high
182                time_count <= (others => '0'); --reset time_count
183             end if;
184     end if;
185   end process timer;
186
187     RTL: process(clk_port) --controls the shifting and loading of the bits
188     begin
189       if rising_edge(clk_port) then --on rising edge
190           if Dout = '0' then --if Dout gows low
191               Start <= '0'; --Start mimics
192             else
193               Start <= '1'; --else, Start also mimics
194             end if;
195
196           if shift_en ='1' then --if shift_en is high
197             shift_values <= Dout & shift_values(9 downto 1); --take in new value and right
    shift
198             else
199                 shift_values <= shift_values; --else, hold current value
200             end if;
```

```
201
202             if(clr = '1') then --if clear is high (priority)
203                 par_out_temp <= (others => '0'); --clear currently stored values
204             elsif load_en = '1' then --if load is high
205               par_out_temp <= shift_values(8 downto 1); --load bits into par_out_temp
206               par_output_check <= shift_values(8 downto 1); --just for debugging(lets us see
      the values being input)
207             else
208               par_out_temp <= par_out_temp; --else, hold current values
209             end if;
210
211             par_output <= par_out_temp; --at the end of the cycle output the newly loaded
     values
212     end if;
213    end process RTL;
214
215 end behavior;
```

ASCII_Converter Code

```
----------------------------------------------------------------------------------
-- Company: Thayer School of Engineering
-- Engineer: Gavin Burns & Sarah Hutchinson
--
-- Create Date: 08/16/2021 09:33:50 AM
-- Design Name:
-- Module Name: ASCII_converter - behavior
-- Project Name: ENGS31 - [REDACTED]
-- Target Devices: Basys3 FPGA
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ASCII_converter is
    Port ( clk: in STD_LOGIC; -- 1 MHz
           ASCII_code : in STD_LOGIC_VECTOR(7 downto 0); --8bit ASCII code from serial

           op_code : out STD_LOGIC_VECTOR(1 downto 0); --operator code for (+ - * /)
           dig_code : out STD_LOGIC_VECTOR(3 downto 0); --digit code for (0-9)
           valid_int : out STD_LOGIC;--signal valid integer
           valid_op : out STD_LOGIC; --signals valid operator
           valid_neg : out STD_LOGIC; --signals negative symbol
           entr : out STD_LOGIC; --signals enter key
           dlt : out STD_LOGIC); --signals delete key
end ASCII_converter;

architecture behavior of ASCII_converter is

begin

    ASCII_converter: process(clk)
    begin

        if rising_edge(clk) then
            case ASCII_code is

                when "00100000" => --delete key
                    dlt <= '1'; --delete code

                when "00001000" => --backspace key
                    dlt <= '1'; --delete code

                when "00001010" => --enter key
                    entr <= '1'; --enter code

                when "00101011" => --plus key

                    valid_op <= '1';
                    op_code <= "00"; --addition code

                when "00101101" => --minus key
                    valid_neg <= '1';
                    valid_op <= '1';
                    op_code <= "01"; --subtraction code

                when "00101010" => --asterisks key
                    valid_op <= '1';
                    op_code <= "10"; --multiplication code

                when "00101111" => --slash key
                    valid_op <= '1';
                    op_code <= "11"; --division code

                when "00110000" => --0 key
                    valid_int <= '1';
                    dig_code <= ASCII_code(3 downto 0); --take the last 4 digits

                when "00110001" => --1 key
                    valid_int <= '1';
                    dig_code <= ASCII_code(3 downto 0); --take the last 4 digits

                when "00110010" => --2 key
```

```
                    valid_int <= '1';
                    dig_code <= ASCII_code(3 downto 0); --take the last 4 digits

                when "00110011" => --3 key
                    valid_int <= '1';
                    dig_code <= ASCII_code(3 downto 0); --take the last 4 digits

                when "00110100" => --4 key
                    valid_int <= '1';
                    dig_code <= ASCII_code(3 downto 0); --take the last 4 digits

                when "00110101" => --5 key
                    valid_int <= '1';
                    dig_code <= ASCII_code(3 downto 0); --take the last 4 digits

                when "00110110" => --6 key
                    valid_int <= '1';
                    dig_code <= ASCII_code(3 downto 0); --take the last 4 digits

                when "00110111" => --7 key
                    valid_int <= '1';
                    dig_code <= ASCII_code(3 downto 0); --take the last 4 digits

                when "00111000" => --8 key
                    valid_int <= '1';
                    dig_code <= ASCII_code(3 downto 0); --take the last 4 digits

                when "00111001" => --9 key
                    valid_int <= '1';
                    dig_code <= ASCII_code(3 downto 0); --take the last 4 digits

                when others =>
                    --op_code <= (others => '0');
                    --dig_code <= (others => '0');
                    valid_int <= '0'; --set them all off for all other cases
                    valid_op <= '0';
                    valid_neg <= '0';
                    entr <= '0';
                    dlt <= '0';
            end case;
        end if;
    end process;
end behavior;
```

Num_Controller Code

```
----------------------------------------------------------------------------------
-- Company: Thayer School of Engineering
-- Engineer: Gavin Burns & Sarah Hutchinson
--
-- Create Date: 08/11/2021 11:39:17 AM
-- Design Name:
-- Module Name: num_controller - behavior
-- Project Name: ENGS31 - [REDACTED]
-- Target Devices: Basys3 FPGA
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity num_controller is
    generic(
      NUM_DIGITS: integer:= 3); --control the largest number of digits
    Port ( clk: in STD_LOGIC; --clock signal
           valid_int : in STD_LOGIC; --signals valid integer
           valid_op : in STD_LOGIC; --signals valid operator
           valid_neg : in STD_LOGIC; --signals valid negative symbol
           entr : in STD_LOGIC; --signals enter key
           dlt : in STD_LOGIC; --signals delete key
           op_full : in STD_LOGIC; --signals that an operator has been input
           answer_full : in STD_LOGIC; --signals that an answer has been computed
           overflow : in STD_LOGIC; -- signals that the answer has overflown

           dig_en : out STD_LOGIC; --enables digit storage
           dig_dlt : out STD_LOGIC; --deletes least significant digit
           dig_clr : out STD_LOGIC; --clears all digit registers
           op_en : out STD_LOGIC; --enables operator storage
           op_clr : out STD_LOGIC; --clears operator register
           neg_en : out STD_LOGIC; --toggles number negativity
           A_en : out STD_LOGIC; --enable storage into the A operand
           B_en : out STD_LOGIC; --enables storage into the B operand
           math_en : out STD_LOGIC; --enables computation
           LED_output: out STD_LOGIC_VECTOR(1 downto 0); --determines what is output to the
    computer
           signal_output: out STD_LOGIC_VECTOR(1 downto 0); --determines what is output to
    the computer
           answer_store: out STD_LOGIC); --stores the operand in operand A

end entity;

architecture behavior of num_controller is

    type state_type is (Empty, StoreDigit, StoreNeg, DeleteDigit, LoadA, LoadB, WaitOperator
    , StoreOperator, DeleteOperator, Clear, Restart, Compute, oops, Idle, AnswerBuffer,
    AnswerWaitOperator, AnswerStoreOperator, AnswerDeleteOperator); --13 states
    signal curr_state: state_type:= Empty; -- start in EMPTY state
    signal next_state: state_type;

    signal count: unsigned(2 downto 0):= (others => '0'); --counts the number of times a
    digit has been entered (MAX 3)
    signal output_temp: std_logic_vector(1 downto 0):= (others => '0');
    signal count_en: std_logic:= '0'; --enables count increment
    signal count_reset: std_logic:= '0'; --signal to reset counter
    signal dir: std_logic:= '0'; --determine if count goes up or down
    signal TC: std_logic:= '0'; --output signal when max is reached

    signal FLAG_en: std_logic:= '0'; --enable FLAG toggle
    signal FLAG: std_logic:= '0'; --determines which operand

begin

    Controller_Update: process(clk)
  begin
      if rising_edge(clk) then
          curr_state <= next_state; --move to next state at the start of the next clock
    cycle
        end if;
  end process;

```

```
    Controller_Logic: process(curr_state, valid_int, valid_op, valid_neg, entr, dlt, op_full
    , answer_full, TC, FLAG, overflow, output_temp)
    begin
        next_state <= curr_state; --default values
        dig_en <= '0';
        dig_dlt <= '0';
        dig_clr <= '0';
        op_en <= '0';
        op_clr <= '0';
        neg_en <= '0';
        A_en <= '0';
        B_en <= '0';
        math_en <= '0';
        answer_store <= '0';
        count_en <= '0';
        count_reset <= '0';
        FLAG_en <= '0';
        dir <= '0';
        signal_output <= "00";
        LED_output <= "00";

        case curr_state is
            when Empty => --begin in empty
                signal_output <= "00";

                if(FLAG = '1') then
                    LED_output <= "10";
                else
                    LED_output <= "00";
                end if;


                if ((valid_int AND NOT(TC)) ='1') then --if a number is recognized and the
    registers aren't filled
                    next_state <= StoreDigit; --go to STORE_DIGIT
                end if;

                if (valid_neg = '1') then --if a negative symbol is recognized
                    next_state <= StoreNeg; --go to STORE_NEG
                end if;

                if (dlt = '1') then --if delete key is recognized
                    next_state <= DeleteDigit; --go to DELETE_DIGIT
                end if;

                if (entr = '1') then --if enter key is recognized
                    if (FLAG = '1') then --and the flag is raised
                        next_state <= LoadB; --go to LOAD_B
                    else --if flag is down
                        next_state <= LoadA; --go to LOAD_A
                    end if;
                end if;


            when StoreDigit => --STORE_DIGIT state
                dig_en <= '1'; --enable digit storage

                count_en <= '1'; --enable counter
                dir <= '1'; --have it increment +1

                next_state <= Empty; --return to EMPTY


            when DeleteDigit => --DELETE_DIGIT state
                dig_en <= '1'; --enble digit storage
                dig_dlt <= '1'; --delete lsd and have the numbers shift back

                count_en <= '1'; --enable counter
                dir <= '0'; --have it increment -1

                next_state <= Empty; --return to EMPTY


            when StoreNeg => --STORE_NEG state
                neg_en <= '1'; --toggle negative sign

                next_state <= Empty; --return to EMPTY


            when LoadA => --LOAD_A state
                A_en <= '1'; --load all current digits into operand A
                FLAG_en <= '1'; --raise flag; indicates that next operand being stored is B

                next_state <= WaitOperator; --move to WAIT_OPERATOR


            when LoadB => --LOAD_B state
                B_en <= '1'; --load all current digits into operand A
                next_state <= Compute; --move to COMPUTE; carries out the math operations


```

```
            when WaitOperator => --WAIT_OPERATOR state
                signal_output <= "00";
                LED_output <= "01";

                if (valid_op = '1') then --if an operator is recognized
                    next_state <= StoreOperator; --go to STORE_OPERATOR
                end if;

                if (dlt = '1') then --if delete key is recognized
                    next_state <= DeleteOperator; --go to DELETE_OPERATOR
                end if;

                if ((entr AND op_full) = '1') then --if enter key is recognized and there is an operator present
                    next_state <= Clear; --move to CLEAR; clears previous digits and gets ready to store operand B
                end if;


            when StoreOperator => --STORE_OPERATOR state
                op_en <= '1'; --store the inputed operator

                next_state <= WaitOperator; --retrn to WAIT_OPERATOR; gives user option to change operator later


            when DeleteOperator => --DELETE_OPERATOR state
                op_clr <= '1'; --clear currently stored operator

                next_state <= WaitOperator; --return to WAIT_OPERATOR


            when Clear => --CLEAR state
                dig_clr <= '1'; --clear all current g=digits and reset negativity

                count_reset <= '1'; --reset count

                next_state <= Empty; --move to EMPTY


            when Compute => --COMPUTE state
                math_en <= '1'; --compute the ansewer

                if (answer_full = '1') then --when an answer is ready
                     if(overflow = '1') then
                      next_state <= oops;
                      else
                      next_state <= Idle;
                      end if;
                end if;

            when oops =>
                signal_output <= "11";
                LED_output <= "11";

                if ((entr OR dlt) = '1') then --if enter or delete is recognized
                    next_state <= Restart; --restart the entire process; begining with operand A
                end if;

            when Idle =>
                signal_output <= "01";
                LED_output <= "11";

                if (valid_op = '1') then --if operator recgnized
                    next_state <= AnswerBuffer; --move to STORE_OPERATOR; skips A operand since the answer is stored in there
                end if;

                if ((entr OR dlt) = '1') then --if enter or delete is recognized
                    next_state <= Restart; --restart the entire process; begining with operand A
                end if;

            when AnswerBuffer =>
                answer_store <= '1'; --enable storage of the answer
                A_en <= '1'; --store the answer in operand A; this lets the user connect equations together
                op_clr <= '1'; --clear currently stored operator

                next_state <= AnswerStoreOperator;

            when AnswerWaitOperator => --WAIT_OPERATOR state
                signal_output <= "10";
                LED_output <= "01";

                if (valid_op = '1') then --if an operator is recognized
                    next_state <= AnswerStoreOperator; --go to STORE_OPERATOR
                end if;

                if (dlt = '1') then --if delete key is recognized
```

```
257                     next_state <= AnswerDeleteOperator; --go to DELETE_OPERATOR
258                 end if;
259 
260                 if ((entr AND op_full) = '1') then --if enter key is recognized and there is
     an operator present
261                     next_state <= Clear; --move to CLEAR; clears previous digits and gets
    ready to store operand B
262                 end if;
263 
264 
265             when AnswerStoreOperator => --STORE_OPERATOR state
266                 op_en <= '1'; --store the inputed operator
267 
268                 next_state <= AnswerWaitOperator; --retrn to WAIT_OPERATOR; gives user
    option to change operator later
269 
270 
271             when AnswerDeleteOperator => --DELETE_OPERATOR state
272                 op_clr <= '1'; --clear currently stored operator
273 
274                 next_state <= AnswerWaitOperator; --return to WAIT_OPERATOR
275 
276             when Restart => --RESTART state
277                 dig_clr <= '1'; --clear currently stored digits
278                 op_clr <= '1'; --clear operator
279                 FLAG_en <= '1'; --lower flag; returning to A operand storage
280 
281                 count_reset <= '1'; --reset count
282 
283                 next_state <= Empty; --move to EMPTY
284         end case;
285     end process;
286 
287     Counter: process(clk, count) --controls the user from pushing out the msd
288     begin
289         if rising_edge(clk) then
290             if(count_reset = '1') then --if reset count is enabled; priority over count
    enable
291                 count <= (others => '0'); --reset count
292             elsif(count_en = '1') then --else if count is enabled
293                 if(dir = '0' AND NOT(count = 0)) then --and direction is low and count isnt
    already at 0
294                     count <= count - 1; --increment -1
295                 elsif (dir = '1' AND count <= NUM_DIGITS) then -- or if direction is high
    and count isnt already larger than NUM_DIGITS
296                     count <= count + 1; --increment +1
297                 end if;
298             end if;
299         end if;
300 
301         TC <= '0'; --default value
302         if(count >= NUM_DIGITS) then --if count is higher or equal to than the number of
    permitted digits (3)
303             TC <= '1'; --TC goes high
304         end if;
305     end process;
306 
307     FlagPole: process(clk) --toggles the current state of the FLAG indicator
308     begin
309         if rising_edge(clk) then
310             if (FLAG_en = '1') then
311                 FLAG <= NOT(FLAG); --toggle FLAG
312             end if;
313         end if;
314     end process;
315 end behavior;
```

Num_Datapath Code

```
----------------------------------------------------------------------------------
-- Company: Thayer School of Engineering
-- Engineer: Gavin Burns & Sarah Hutchinson
--
-- Create Date: 08/11/2021 08:13:33 PM
-- Design Name:
-- Module Name: ASCII_converter_tb - testbench
-- Project Name: ENGS31 - [REDACTED]
-- Target Devices: Basys3 FPGA
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity num_datapath is
  --generic(
      --NUM_DIGITS: integer); --control the largest number of digits
  port(
      clk:in std_logic; --clock signal
      dig_code: in std_logic_vector(3 downto 0); --BCD number between 0-9
        op_code: in std_logic_vector(1 downto 0); --operator code: 00=addition, 01=sub, 10=
    mult, 11=div
        dig_en: in std_logic; --enables digit storage
        dig_dlt: in std_logic; --deletes least significant digit
        dig_clr: in std_logic; --clears all digit registers
        op_en: in std_logic; --enable operator storage
        op_clr: in std_logic; --clears operator register
        neg_en: in std_logic; --toggles number negativity

        op:  out std_logic_vector(1 downto 0); --entered operator
        num0: out std_logic_vector(3 downto 0); --one's place
        num1: out std_logic_vector(3 downto 0); --ten's place
        num2: out std_logic_vector(3 downto 0); --hundred's place
        neg: out std_logic; --current negativity
        op_full: out std_logic); --whether an operator is present

end num_datapath;

architecture behavior of num_datapath is

  signal temp_neg: std_logic:= '0'; --for T Flip-Flop
    signal dig0, dig1, dig2: std_logic_vector(3 downto 0):= (others => '0'); --tmeporary
    digits
--    signal num0_temp, num1_temp, num2_temp: std_logic_vector(3 downto 0):= (others => '0')
    ;

begin
    operator: process(clk)
    begin
      if rising_edge(clk) then --on the rising edge
          if(op_en = '1') then
              op <= op_code; --store operator
                op_full <= '1'; --set operator to full (lets you move to next state
            end if;

            if(op_clr = '1') then
              --op <= others => '0'; --clear operator (default to addition)
                op_full <= '0'; --set operator to empty (cannot move on until operator is
    input)
            end if;
        end if;
  end process;

    negative: process(clk, temp_neg)
    begin
      if rising_edge(clk) then --on rising edge
          if(neg_en = '1') then
              temp_neg <= NOT(temp_neg); --toggle negativity
            end if;

            if(dig_clr = '1') then
              temp_neg <= '0'; --reset negativity
            end if;
        end if;

        neg <= temp_neg; --output current negativity
    end process;

    number: process (clk, dig0, dig1, dig2)
    begin
```

```
      if rising_edge(clk) then -- on rissing edge
          if(dig_en = '1') then
              dig0 <= dig_code; --shift msd and store entered digit
                dig1 <= dig0;
                dig2 <= dig1;
            end if;

            if(dig_dlt = '1') then
              dig0 <= dig1; --delete lsd and shift
                dig1 <= dig2;
                dig2 <= (others => '0');
            end if;

            if(dig_clr = '1') then
              dig0 <= (others => '0'); --clear all digits
                dig1 <= (others => '0');
                dig2 <= (others => '0');
            end if;


--            num0_temp <= num0_temp;
--            num1_temp <= num1_temp;
--            num2_temp <= num2_temp;

            --if(busy_convert = '0') then
                num0 <= dig0; --output current digits
                num1 <= dig1;
                num2 <= dig2;
            --end if;

--            num0 <= num0_temp;
--            num1 <= num1_temp;
--            num2 <= num2_temp;
        end if;

    end process;

end behavior;
```

Math Code

```
----------------------------------------------------------------------------------
-- Company: Thayer School of Engineering
-- Engineer: Gavin Burns & Sarah Hutchinson
--
-- Create Date: 08/16/2021 11:43:22 AM
-- Design Name:
-- Module Name: Math - Behavioral
-- Project Name: ENGS31 - Operation_[REDACTED]
-- Target Devices: Basys3 FPGA
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Math is
    Port ( clk: in STD_LOGIC; --clock signal
           num0: in std_logic_vector(3 downto 0); --one's place
           num1: in std_logic_vector(3 downto 0); --ten's place
           num2: in std_logic_vector(3 downto 0); --hundred's place
           neg: in std_logic; --current negativity
           op: in STD_LOGIC_VECTOR(1 downto 0); --operator
           A_en : in STD_LOGIC; --enables storage of number into A operand
           B_en : in STD_LOGIC; --enables storage of number into B operand
           math_en : in STD_LOGIC; --enables computation of A and B by op
           answer_store : in STD_LOGIC; --stores the answer in the A operand

           answer : out STD_LOGIC_VECTOR(9 downto 0); --outputted answer
           A_output : out STD_LOGIC_VECTOR(9 downto 0);
           A_neg : out STD_LOGIC;
           overflow: out STD_LOGIC; --signals that the value is beyond -999 or 999
           y_neg : out STD_LOGIC; --whether answer is negative our not
           answer_full : out STD_LOGIC); --signals that an answer is ready to be presented
end Math;

architecture behavior of Math is

    signal A: SIGNED(10 downto 0):= (others => '0'); --Operand A; 11 bits to store numbers
    between -999 to 999
    signal B: SIGNED(10 downto 0):= (others => '0'); --Operand A; 11 bits to store numbers
    between -999 to 999
    signal answer_temp: SIGNED(20 downto 0):= (others => '0'); --answer; 21 bits to store
    numbers between -998001 to 998001
    --signal num0_temp: UNSIGNED(10 downto 0):= (others => '0');
    --signal num1_temp: UNSIGNED(10 downto 0):= (others => '0');
    --signal num2_temp: UNSIGNED(10 downto 0):= (others => '0');
    --signal num3_temp: UNSIGNED(10 downto 0):= (others => '0');
    constant I: unsigned(6 downto 0):= "1100100"; --needed for BCD conversion

begin

    Store: process(clk)
    begin
        if rising_edge(clk) then

            --num0_temp(3 downto 0) <= unsigned(num0);
            --num1_temp <= resize(10*unsigned(num1), num1_temp'length);
            --num2_temp <= resize(I*unsigned(num2), num2_temp'length);
            --num3_temp <= resize(10*unsigned(num2_temp), num3_temp'length);

            if (A_en = '1') then
                if (answer_store = '1') then
                    A <= answer_temp(10 downto 0); --store the answer into operand A(for
    recursion)
                else
                    if(neg = '1') then --if value is negative
                        A <= NOT(signed(resize(I*unsigned(num2) + 10*unsigned(num1) +
    unsigned(num0), A'length))) + 1; --two's compliment of the digits
                    else
                        A <= signed(resize(I*unsigned(num2) + 10*unsigned(num1) + unsigned(
    num0), A'length)); --store digits in operand A
                    end if;
                end if;
            elsif (B_en = '1') then
```

```
                if(neg = '1') then --if value is negative
                        B <= NOT(signed(resize(I*unsigned(num2) + 10*unsigned(num1) + unsigned(num0), B'length))) + 1; --two's compliment of the digits
                else
                        B <= signed(resize(I*unsigned(num2) + 10*unsigned(num1) + unsigned(num0), B'length)); --store digits into operand B
                end if;
            end if;

            if(A < 0) then
                    A_neg <= '1'; --separate the negative
                    A_output <= std_logic_vector((NOT(A(9 downto 0)) + 1)); --store the value in answer
                else
                    A_neg <= '0';
                    A_output <= std_logic_vector(A(9 downto 0)); --store the value of answer
                end if;
        end if;
    end process;

    Computation: process(clk, answer_temp)
    begin

        if rising_edge(clk) then

            answer_full <= '0';

            if (math_en = '1') then

                --answer_temp <= answer_temp;

                case op is
                    when "00" => --addition operator
                        answer_temp <= resize(A + B, answer_temp'length); --add A and B

                    when "01" => --subtraction operator
                        answer_temp <= resize(A - B, answer_temp'length); --subtract B from A

                    when "10" => --multiplication operator
                        answer_temp <= resize(A * B, answer_temp'length); --multiply A and B

                    when "11" => --division operator
                        answer_temp <= resize(A / B, answer_temp'length); --divide A by B

                    when others =>
                        answer_temp <= (others => '0'); --set to zero so we can easily see there's an error
                end case;

            end if;

            if(answer_temp > 999 OR answer_temp < -999) then --if the answer is out of bounds
                    answer <= (others => '0'); --store the value "10000000000" (indicates an overvalued answer)
                    overflow <= '1'; --signal overflow(this value cannot be stored in A)
                    answer_full <= '1'; --signal that an answer has been generated
                else
                    if(answer_temp < 0) then
                        y_neg <= '1'; --separate the negative
                        answer <= std_logic_vector((NOT(answer_temp(9 downto 0)) + 1)); --store the value in answer
                    else
                        y_neg <= '0';
                        answer <= std_logic_vector(answer_temp(9 downto 0)); --store the value of answer
                    end if;
                    overflow <= '0';
                    answer_full <= '1'; --signal that an answer has been generated
                end if;

        end if;
    end process;

end behavior;
```

Binary to BCD Code

```
----------------------------------------------------------------------------------
-- Company: Thayer School of Engineering
-- Engineer: Gavin Burns & Sarah Hutchinson
--
-- Create Date: 08/17/2021 10:14:52 PM
-- Design Name:
-- Module Name: ASCII_converter - behavior
-- Project Name: ENGS31 - [REDACTED]
-- Target Devices: Basys3 FPGA
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------


library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

entity binary_BCD is
  port( clk:      in  std_logic; --clock signal
        answer_full:        in  std_logic; --signals that an answer has been generated
              binary_in:    in  std_logic_vector(9 downto 0); --binary answer

      y0:       out std_logic_vector(3 downto 0); --BCD number (0-9) lsd
      y1:       out std_logic_vector(3 downto 0); --BCD number (0-9) mid digit
            y2:       out std_logic_vector(3 downto 0)); --BCD number (0-9) msd

end binary_BCD;

architecture behavior of binary_BCD is
  constant NSHIFT:              integer := 10; --number of total shifts
    signal store_bin:             unsigned(9 downto 0) := (others => '0');
    signal temp_bcd:              unsigned(11 downto 0) := (others => '0');
    signal shift_count_en:      std_logic := '0'; --enables shift_cout to increment
    signal shift_count_reset:       std_logic := '0'; --resets shift count
  signal shift_count:             unsigned(5 downto 0) := "000000"; --counts the number of
    shifts
    signal load_en:               std_logic := '0'; --loads the answer into the store_bin
    signal clr:                     std_logic := '0'; --clears store bin
    signal shift_en:              std_logic := '0'; --right shifts the store_bin into the
    BCD generator
    signal all_shifted:             std_logic := '0'; --signals that all of the bits have
    been shifted
    signal check:                   std_logic := '0'; --signals to check if each BCD bin is
    greater than 5

    type state_type is (idle, load_bin, shift, shift_check);
    signal curr_state:              state_type := idle;
    signal next_state:              state_type;

begin

  RTL: process(clk)
  begin
      if rising_edge(clk) then

          if(clr = '1') then
              store_bin <= (others => '0');
              temp_bcd <= (others => '0');
          elsif load_en = '1' then
              store_bin <= unsigned(binary_in); --load the answer into store_bin
          else
              store_bin <= store_bin;
          end if;


          if shift_en = '1' then
            temp_bcd <= temp_bcd(10 downto 0) & store_bin(9); --left shift bits into BCD
    generator
            store_bin <= store_bin(8 downto 0) & '0'; --left shift store_bin
          end if;

          if(check = '1') then
            if temp_bcd(3 downto 0) >= 5 then --check if the first BCD bin is >5
              temp_bcd(3 downto 0) <= temp_bcd(3 downto 0) + 3; --if so; add 3
            else
              temp_bcd(3 downto 0) <= temp_bcd(3 downto 0); --else; no change
            end if;

            if temp_bcd(7 downto 4) >= 5 then --check if second BCD bin is >5
              temp_bcd(7 downto 4) <= temp_bcd(7 downto 4) + 3; --if so; add 3
            else
```

```
            temp_bcd(7 downto 4) <= temp_bcd(7 downto 4); --else; no change
          end if;

          if temp_bcd(11 downto 8) >= 5 then --check if third BCD binis >5
            temp_bcd(11 downto 8) <= temp_bcd(11 downto 8) + 3; --if so; add 3
          else
            temp_bcd(11 downto 8) <= temp_bcd(11 downto 8); --else; no change
          end if;
        end if;

        if all_shifted = '1' then --once all bits have been shifted
          y0 <= std_logic_vector(temp_bcd(3 downto 0)); --store first BCD bin in num0 (lsd
  )
          y1 <= std_logic_vector(temp_bcd(7 downto 4)); --store second BCD bin in num0 (
  mid digit)
          y2 <= std_logic_vector(temp_bcd(11 downto 8)); --store third BCD bin in num0 (
  msd)
    --else
          --num0 <= "0000";
          --num1 <= "0000";
          --num2 <= "0000";
        end if;

    end if;

end process RTL;


  FSM_comb: process(curr_state, answer_full, all_shifted)
  begin
    next_state <= curr_state;
      shift_en <= '0';
      load_en <= '0';
      clr <= '0';
      check <= '0';

      shift_count_en <= '0';
      shift_count_reset <= '0';

      case curr_state is
          when idle =>
              shift_count_reset <= '1';
              clr <= '1';

              if answer_full = '1' then
                  next_state <= load_bin;
              end if;

          when load_bin =>
              load_en <= '1';

              next_state <= shift;

          when shift =>
              shift_en <= '1';
              shift_count_en <= '1';

              if all_shifted = '1' then
                next_state <= idle;
              else
                  next_state <= shift_check;
              end if;

          when shift_check =>
              check <= '1';

              if all_shifted = '1' then
                next_state <= idle;
              else
                  next_state <= shift;
              end if;

      end case;
end process FSM_comb;

counter: process(clk, shift_count, shift_count_reset)
begin
   if rising_edge(clk) then
          if(shift_count_en = '1') then --else if shift_count is enabled
                  shift_count <= shift_count + 1; --increment shift_count by 1;
          end if;
      end if;

      if(shift_count_reset = '1') then --if reset shift_count is enabled; priority over
  count enable
              shift_count <= (others => '0'); --reset shift_count
      end if;

      all_shifted <= '0'; --default value
      if(shift_count >= NSHIFT) then --if shiftcount is greater than number of required
  shifts
```

```
171             all_shifted <= '1'; --all_shifted goes high
172             --shift_count <= (others => '0');
173         end if;
174   end process;
175 
176   FSM_update: process(clk)
177   begin
178       if rising_edge(clk) then
179           curr_state <= next_state;
180         end if;
181   end process FSM_update;
182 
183 
184 end behavior;
```

Appendix 5: Residual Warnings

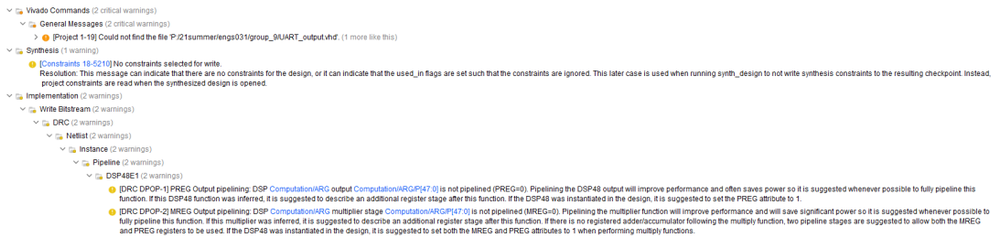

Figure 22: Residual error messages

Figure 23: Zoomed in Residual error messages

We had four residual error messages. The first error message is due to the fact that an earlier version of our project also used a UART_out block to reformat the signals to be outputted back to the computer. However, we ultimately decided to use the 7-segment display for the output instead. Since the UART_is no longer part of our project but various testbenches and debugging shells reference the block, an error was produced. There is also a constraints error, however, this is not relevant because we did have constraints selected.
Our final two errors are about pipelining. Vivado is suggesting that due to some of the time delays associated with computation, additional registers should be added in the middle of the computation so that the computation delays do not slow down the rest of the system. However, we were not experiencing any errors resulting from this, so we left our design as it was.

Appendix 7: Resource Utilization

Summary

Resource	Utilization	Available	Utilization %
LUT	461	20800	2.22
FF	288	41600	0.69
DSP	1	90	1.11
IO	20	106	18.87

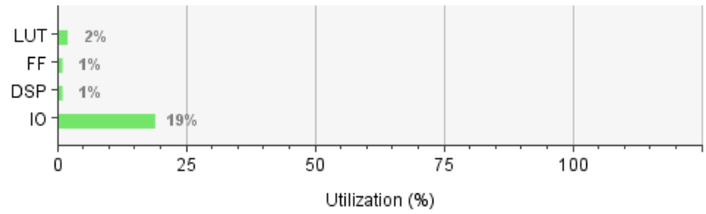

Figure 24: Resource Utilization

Our design used 2.22% of the look-up tables, 0.69% of the flip-flops, 1.11% of the digital signal processors, and 18.87% of the input/output buffers.