TÜRK-ALMAN ÜNİVERSİTESİ
TÜRKISCH-DEUTSCHE UNIVERSITÄT

Digital Design Course Fall Term Project Report

Metin Kerem Öztürk

December 11, 2023

Table of Contents

Introduction

The quest to design a functional and efficient electronic calculator is the cornerstone of this project. The task at hand involves the preparation of a simplified yet robust calculator that can execute the four basic arithmetic operations (addition, subtraction, multiplication and division) while accommodating operands limited to 4-digit decimal integers. Using the digital simulation tool as our platform, the aim is to reveal a meticulously crafted design through simulation alone, without requiring physical circuit implementation.

My goal throughout this project is to showcase a meticulously crafted design that relies solely on simulation to highlight the operational aspects and accuracy of the electronic calculator. The evaluation criteria are based on a personal demonstration where I will explain my unique approach to problem definition, the strategies used during implementation, and the intricate details underlying my design. This demonstration is not just about showcasing the functional aspects but also includes a detailed explanation of my problem-solving methodologies.

Throughout this report I will detail the design assumptions, approach, solution block diagram, summary of results, and invaluable insights gained along this journey.

Design Assumptions

1. **Duration of Key Press Event:** The assumption that each key press event lasts 100 ms will be taken into consideration for handling input operations.
2. **Debouncing Not Required:** Given the instruction that there's no need to debounce key presses, this assumption will affect how the input from keys is processed and managed.
3. **Interval between Key Presses:** The presence of a 100 ms inactivity interval between consecutive key presses will impact the input handling mechanism and sequence.
4. **Input Operand Types:** Only positive integers are allowed as operands, but the result can be negative. This constraint influences the input validation and processing logic.
5. **Integer Division:** Division operations are to be treated as integer divisions, for example, 11/5 = 2. This specific behavior will guide the design of the division module.
6. **Usage of Existing Blocks:** The option to use existing Adder, Subtractor, Multiplier, Divider boxes, or develop custom ones will influence the decision-making process for optimizing functionality and resources.
7. **Utilization of Available Digital Tool Components:** The freedom to use various components available in the Digital simulation tool will be leveraged to design different parts of the calculator, ensuring modularity and clarity.
8. **Hierarchical Design Approach:** Emphasizing a hierarchical design approach using subcircuits will enhance clarity, testability, and modularity. This will be considered to structure the calculator design for better understanding and manageability.
9. **Restrictions on Monolithic Verilog Coding:** Although Verilog language can be used to define building blocks, a monolithic approach for the entire calculator box is discouraged, aligning with course principles.

Design Approach

Throughout the development of this project, I embraced a modular approach to enhance the efficiency of testing and bug fixing. *Figure 1* is a hierarchical flowchart that represents the structure of a calculator project. The project consists of three main branches: “modular_circuits,” “verilog_codes,” and “Calculator.” The “modular_circuits” branch further breaks down into “arithmetic_circuits” and its sub-modules. The “verilog_codes” branch includes various modules that contain external building blocks designed in the icarus verilog language.

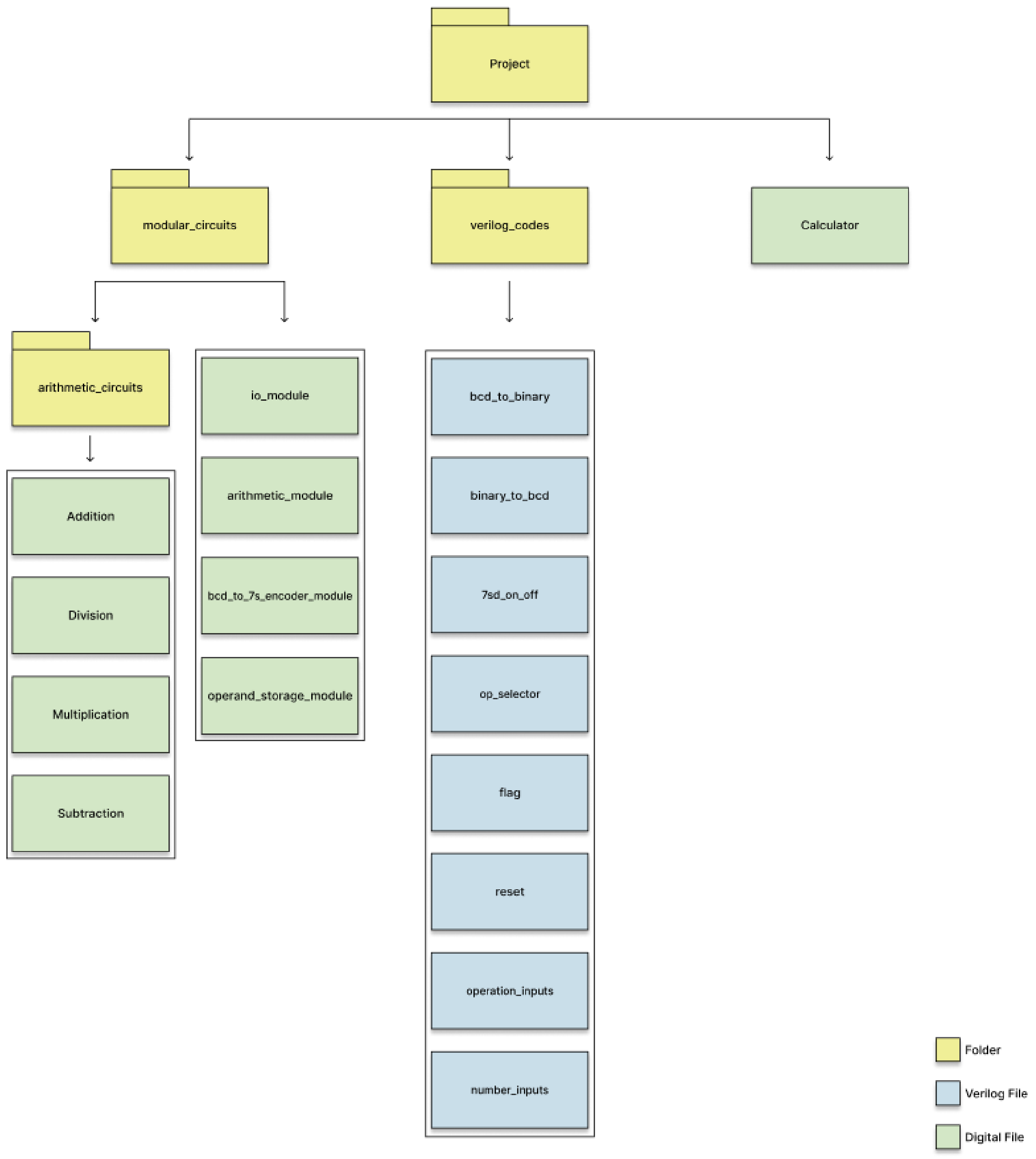


Figure 1: Hierarchical Flowchart of the Project

Verilog Modules

In the digital design of calculator, external Verilog modules serve as dedicated building blocks, each handling specific functions critical to the calculator's operation. These modules streamline complex tasks such as selecting and directing arithmetic operations, input processing, data conversion, display management, and flag control. By using these modules, the design achieves modularity and efficiency, enabling easier debugging, maintenance, and scalability while ensuring each component's dedicated functionality within the calculator's architecture. I have given more detailed explanations about these modules below.

1. 7sd_on_off.v:

Purpose: This module manages a 7-segment display, controlling the output display when turned on or off based on input values.

Inputs:

- `n`: 7-bit input used to display a value on the 7-segment display.
- `reset`: Input for resetting the display.

Outputs:

- `out`: 7-bit output representing the value displayed on the 7-segment display.

Functionality:

- Initializes the display to 0000000 upon reset (`reset == 1`).
- Passes the input `n` to the output `out` when the reset is not active (`reset == 0`).

2. bcd_to_binary.v:

Purpose: Converts a Binary Coded Decimal (BCD) input to binary representation.

Inputs:

- `bcd_input`: 16-bit BCD input.

Outputs:

- `binary_output`: 16-bit binary output.

Functionality:

- Extracts BCD digits from the input and converts them to binary.
- Combines the BCD digits to form a 16-bit binary output.

3. binary_to_bcd.v:

Purpose: Converts a binary input to Binary Coded Decimal (BCD) representation.

Inputs:

- `in`: 16-bit binary input.

Outputs:

- `out`: 16-bit BCD output.

Functionality:

- Converts the 16-bit binary input into its BCD representation using specific logic and algorithms.

4. flag.v:

Purpose: Manages flags and numbers, directing inputs to specific outputs based on a flag condition.

Inputs:

- \`flag\`, \`enable\`, \`number\`, \`reset\`, \`da\`, \`db\`: Inputs used for flag control, number selection, reset and control for maximum input for each operand.

Outputs:

- \`a\`, \`b\`: 4-bit outputs for number representation.
- \`en_a\`, \`en_b\`, \`en2\`: Control signals for enabling specific outputs.

Functionality:

- Sets initial states upon reset.
- Based on the \`flag\`, directs input \`number\` to output \`a\` or \`b\`.
- 0 was given special treatment to detect the change, if input is 1111, convert it back to 0.
- Limit input to 4 times for operands

5. number_inputs.v:

Purpose: Outputs a 4-bit representation based on a 10-bit input.

Inputs:

- \`inp\`: 10-bit input.
- \`reset\`: Reset signal.

Outputs:

- \`out\`: 4-bit output representing a number.
- \`enable\`: Control signal for output activation.

Functionality:

- Resets the output and disable signal.
- Maps specific input values to corresponding 4-bit outputs.
- Converting the output to 1111 instead of 0000, giving special treatment to 0 in order to detect the changes.

6. op_selector.v:

Purpose: Selects and performs arithmetic operations based on input selection.

Inputs:

- Various inputs (\`a_in\`, \`s_in\`, \`m_in\`, \`d_in\`) for arithmetic operations and associated error flags (\`a_er\`, \`m_er\`, \`d_er\`).
- Control inputs (\`a\`, \`s\`, \`m\`, \`d\`) for selecting operations.

Outputs:

- `enable`, `abs_out`, `err`, `neg`: Outputs for operation execution and error checking.

Functionality:

- Executes arithmetic operations based on the selected input.
- Handles error flags and determines the absolute value and sign of the output.

7. operation_inputs.v:

Purpose: Sets operation selection signals based on input.

Inputs:

- `inp`: 4-bit input for operation selection.
- `reset`: Reset signal.

Outputs:

- `a`, `s`, `m`, `d`: Signals for selecting addition, subtraction, multiplication, or division.

Functionality:

- Sets the operation selection signals based on the input value.

8. reset.v:

Purpose: Handles resetting a 4-bit output.

Inputs:

- `r`: Reset signal.
- `n`: 4-bit input for setting the output.

Outputs:

- `o`: 4-bit output.

Functionality:

- Resets the output to `0000` when the reset signal is active.
- Otherwise, sets the output to the value of the input `n`.

Digital Modules

In the design of a digital calculator project, the use of Digital (.dig) modules is indispensable to create and verify the gate-level implementation of the calculator's components. These .dig modules serve as virtual representations of logic gates, registers, multiplexers, and other digital elements within the calculator's architecture. They provide a simulated environment to test, verify and fine-tune the functionality of complex gate-level designs. The use of .dig modules streamlines the development process, allowing comprehensive testing and validation of gate-level designs on a digital calculator, resulting in a reliable and functional end product.

1. Calculator Module

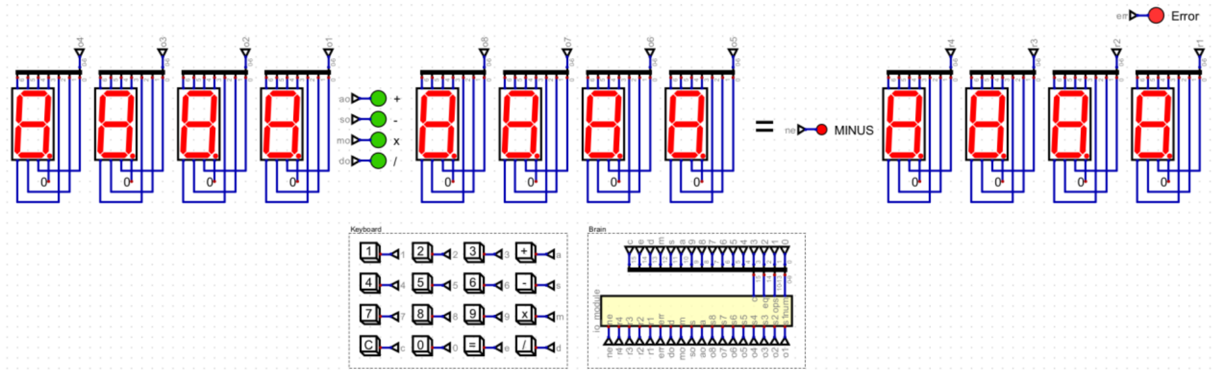


Figure 2: Calculator.dig Module

The Calculator module serves as the interface at the highest level. There is a keyboard component with input buttons to perform arithmetic operations via this module. There are seven segment displays that allow these inputs and the result to be displayed. In addition, some lamps indicate error, operation and error conditions. I named the most important part brain. This component is a gateway to io_module.dig. It takes the inputs there and returns the incoming outputs to the main module.

2. Addition Module

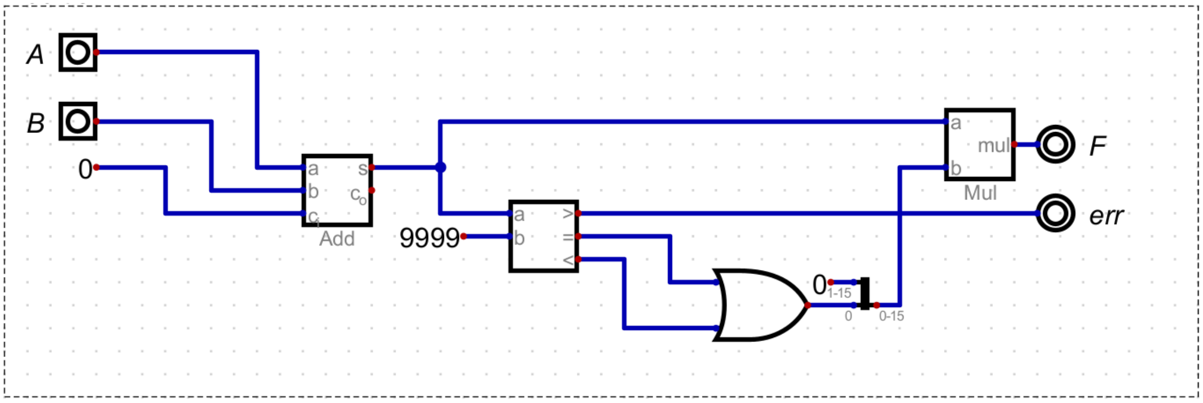


Figure 3: Addition.dig Module

Addition Module takes 16 bit inputs and passes them through a 16 bit adder. The initial carry is given as zero. The result is then compared to the maximum allowed number of 9999. If the result is greater than this, the error is positive. In other cases, the main result is the main output.

3. Division Module

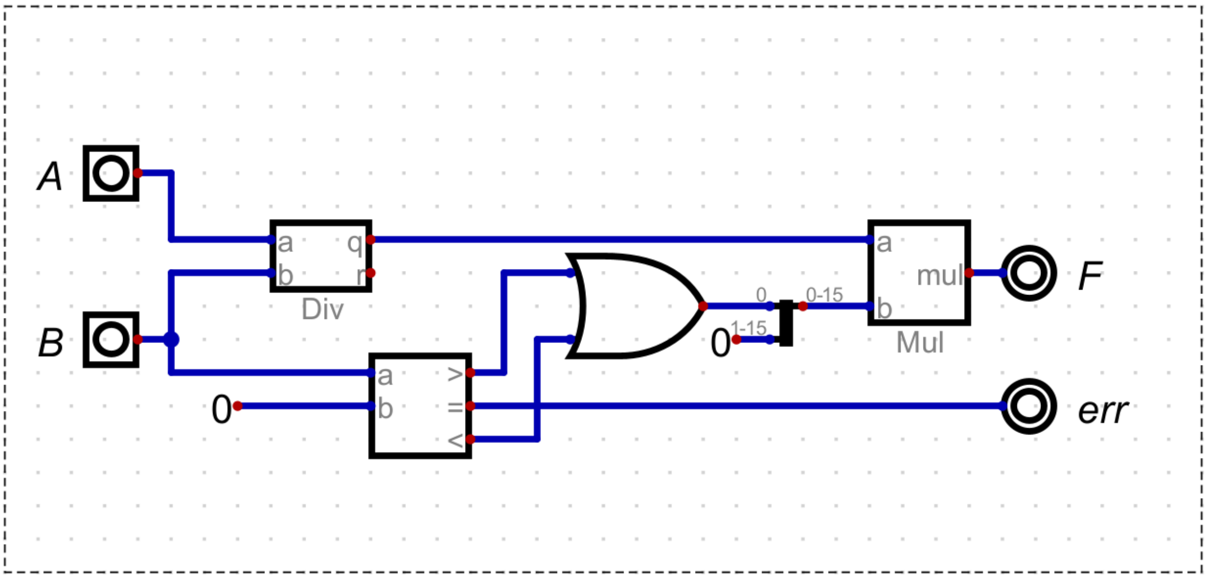


Figure 4: Division.dig Module

The multiplication module also takes 16-bit imputations and divides input A into input B, thanks to the division arithmetic building block. If input B is 0, error output is positive. Then, if there is no error, the result will be 32 bits. This will be reduced again to a smaller number of bits in the future with the op_selector.v module.

4. Multiplication Module

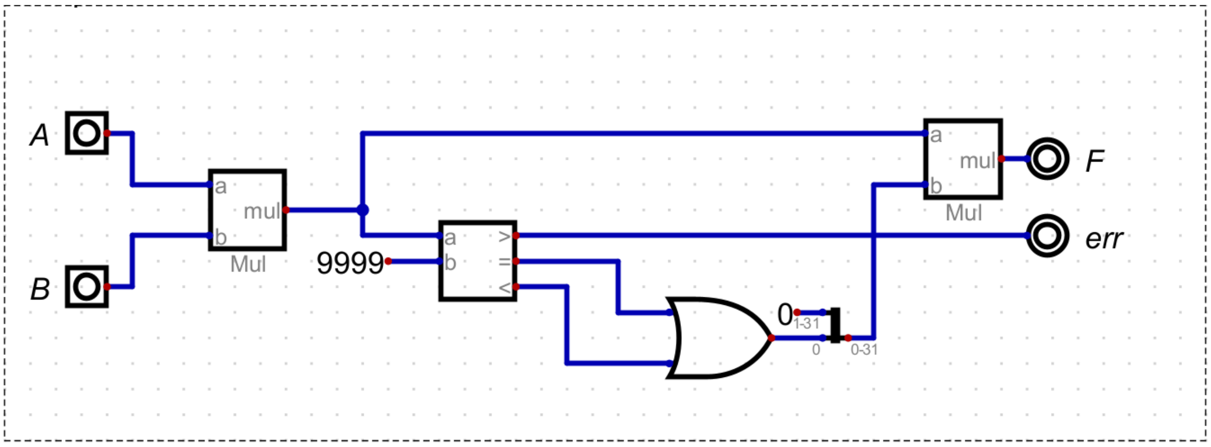


Figure 5: Multiplication.dig Module

The multiplication module works in a similar way to the others. It takes 16 bit imputs and multiplies them with each other thanks to the arithemetic building block of multiplying inputs A and B. The result is compared with 9999 via the comparator. If it is large, the error output will be positive. Then, if there is no error, the result will be 64 bit. This will be reduced again to a smaller number of bits in the future with the op_selector.v module.

5. Subtraction Module

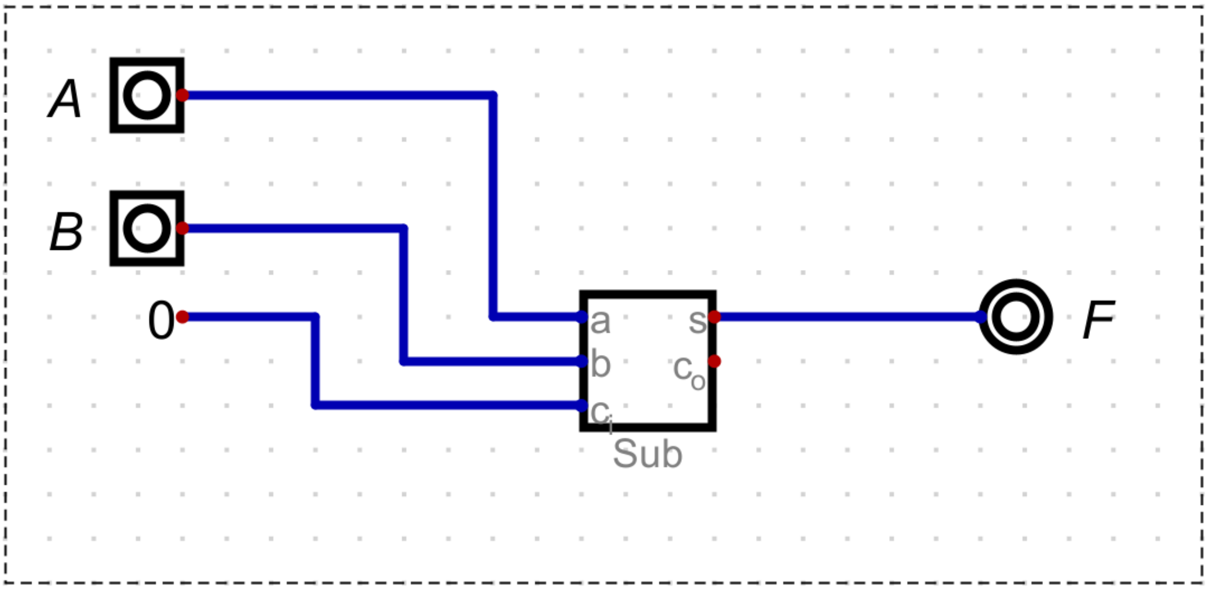


Figure 6: Subtraction.dig Module

Subtraction Module takes 16-bit inputs and passes them through a 16-bit subtraction circuit. Initial carry is 0 again. It then passes the result as a signed integer to the output.

6. Arithmetic module

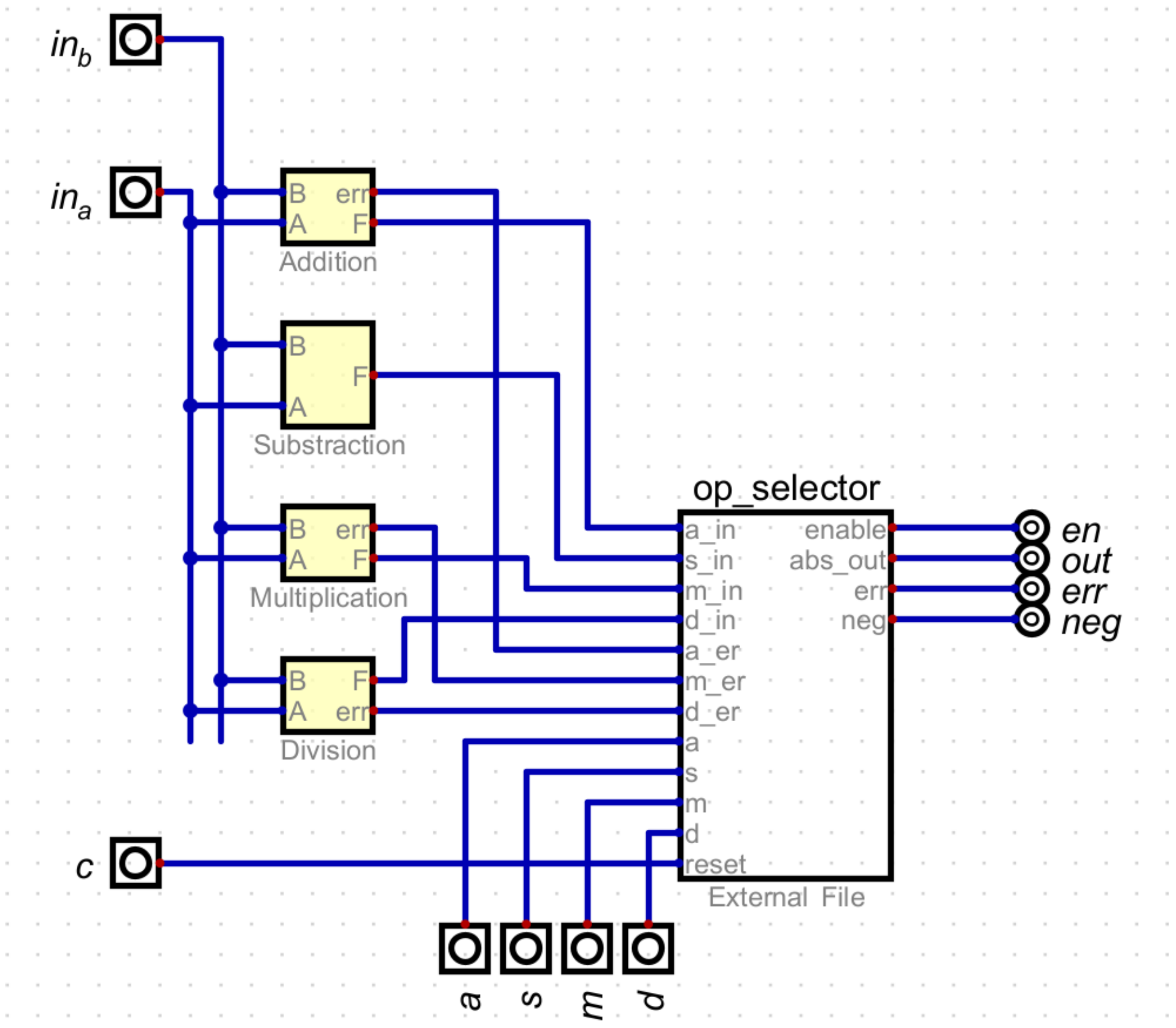


Figure 7: arithmetic_module.dig Module

By using many digital modules and Verilog modules described above, it enables the selection of the operation and then passing the A and B numbers through these operations to obtain the necessary results that will be used in io_module.

7. BCD to 7 Segment Encoder Module

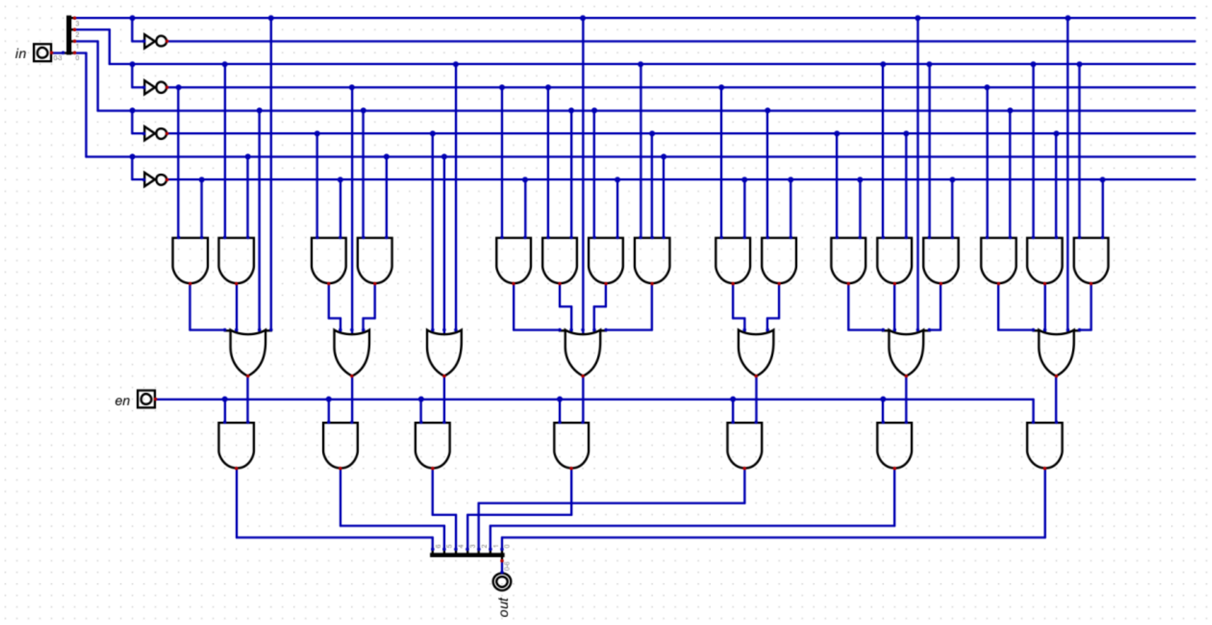
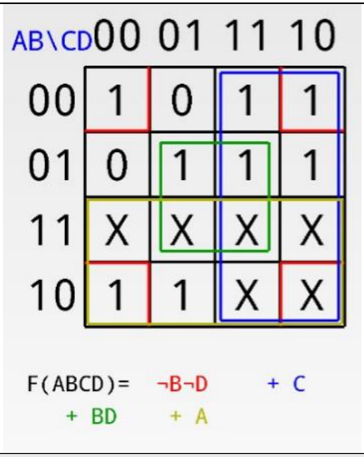
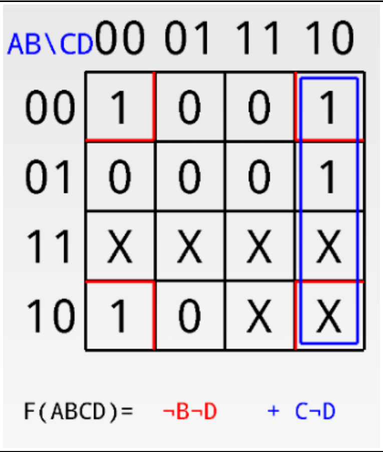
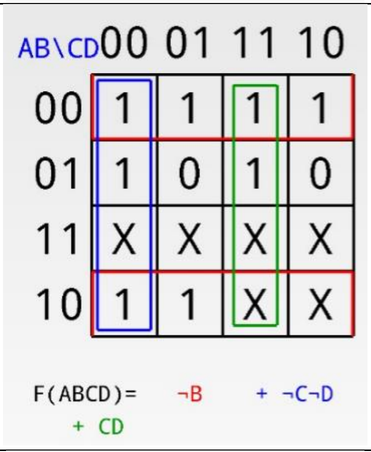
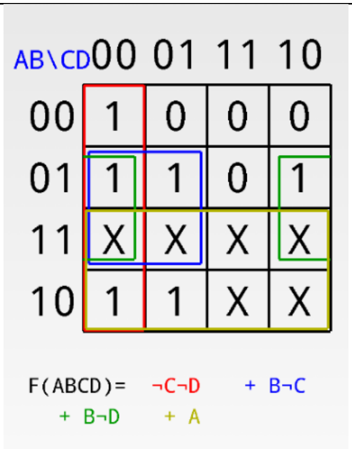
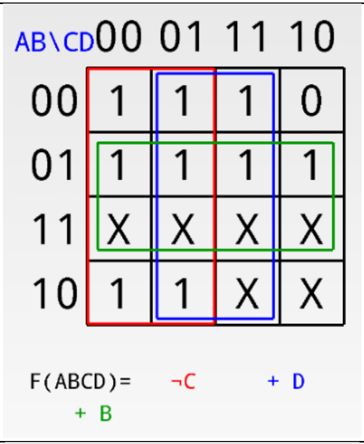
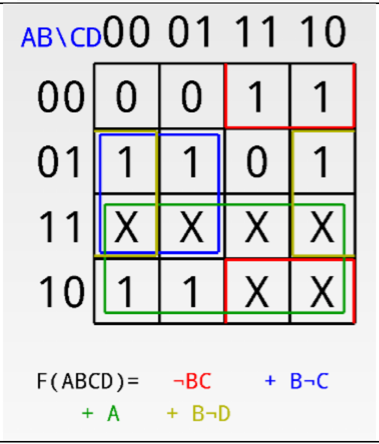
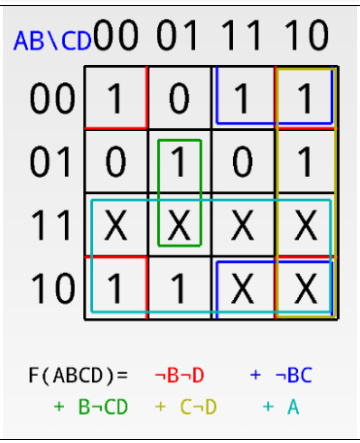


Figure 8: bcd_to_7s_encoder_module.dig Module

Thanks to this Module, encoding is provided from BCD to the seven segment so that the inputs entered from the keyboard can be seen on the seven segment display. Before this part, numbers are stored in D flip flops. Below I am adding the truth table and simplification karnaugh maps used to produce this circuit.

A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

Figure 9:Truth Table for BCD to Seven Segment Display Encoder

A	 AB\CD 00 01 11 10 00 1 0 1 1 01 0 1 1 1 11 X X X X 10 1 1 X X F(ABCD)= ¬B¬D + C + BD + A	E	 AB\CD 00 01 11 10 00 1 0 0 1 01 0 0 0 1 11 X X X X 10 1 0 X X F(ABCD)= ¬B¬D + C¬D
B	 AB\CD 00 01 11 10 00 1 1 1 1 01 1 0 1 0 11 X X X X 10 1 1 X X F(ABCD)= ¬B + ¬C¬D + CD	F	 AB\CD 00 01 11 10 00 1 0 0 0 01 1 1 0 1 11 X X X X 10 1 1 X X F(ABCD)= ¬C¬D + B¬C + B¬D + A
C	 AB\CD 00 01 11 10 00 1 1 1 0 01 1 1 1 1 11 X X X X 10 1 1 X X F(ABCD)= ¬C + D + B	G	 AB\CD 00 01 11 10 00 0 0 1 1 01 1 1 0 1 11 X X X X 10 1 1 X X F(ABCD)= ¬BC + B¬C + A + B¬D
D	 AB\CD 00 01 11 10 00 1 0 1 1 01 0 1 0 1 11 X X X X 10 1 1 X X F(ABCD)= ¬B¬D + ¬BC + B¬CD + C¬D + A		

8. Operand Storage Module

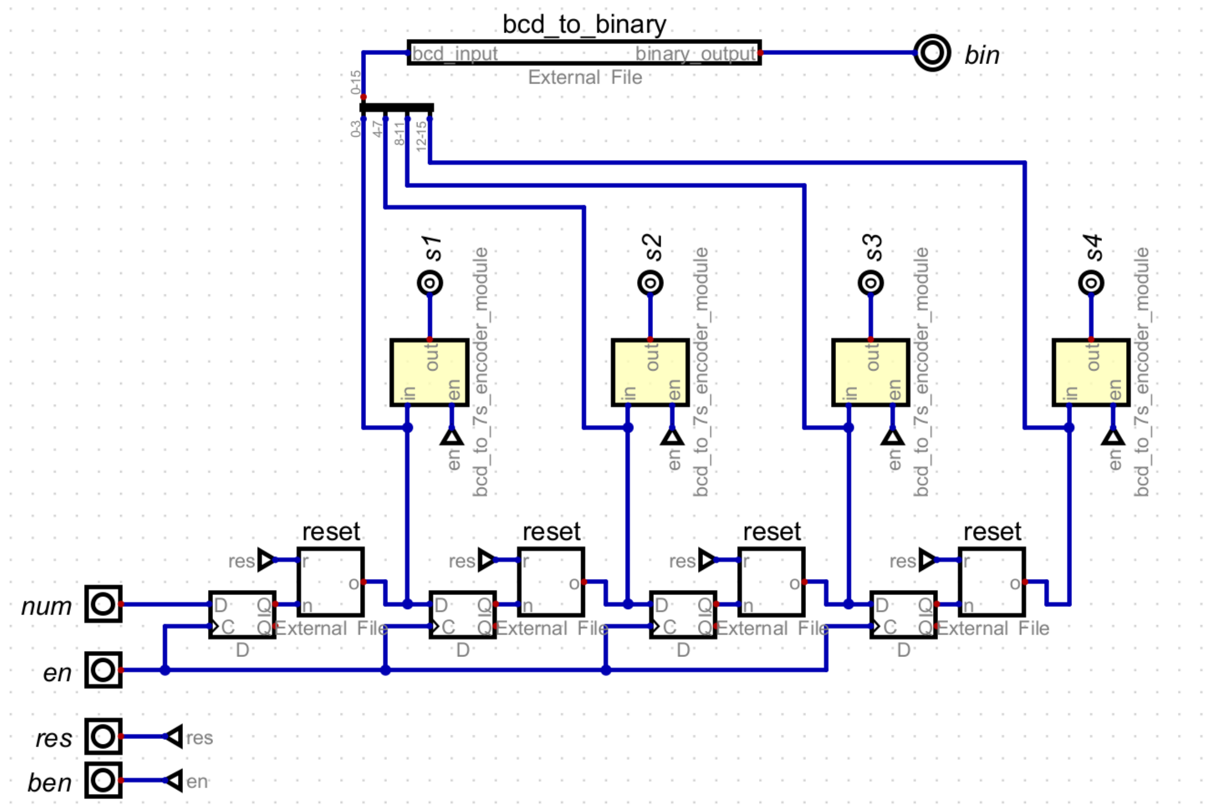


Figure 10: operand_storage_module.dig Module

Thanks to this module, we store the incoming number as a shift in D Flip Flops, and then these stored numbers are made suitable for displaying on the seven segment display with the encoder. These numbers are then brought together and expressed as a 4-digit number. And it is served outside as another output in binary form, ready for arithmetic operations.

9. IO Module

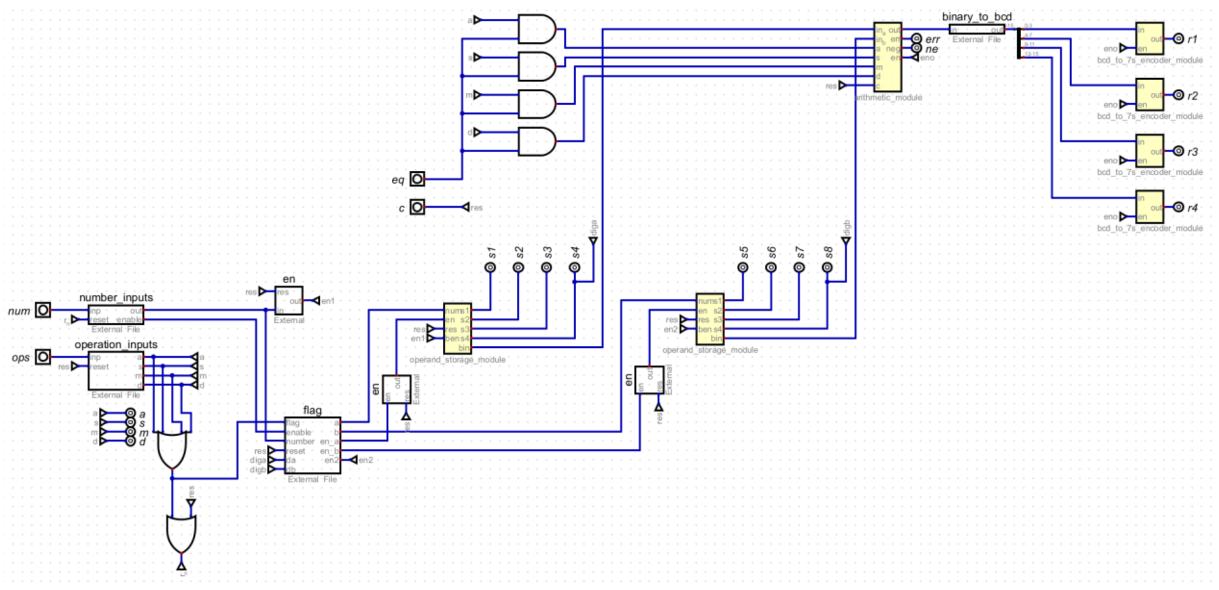


Figure 11: io_module.dig Module

The IO Module is the most important and complex module that I can call the brain of the calculator. It must receive the inputs from the calculator and forward them to the correct submodules. At the same time, the reset process, which starts with the clear button, occurs in many sections and important operations such as turning on and off seven segment displays must be done here. Numbers are first shifted and stored in D Flip Flops (Operand Storage Module). These inputs pass through the seven segment decoder and are sent out to be displayed on the calculator interface. When an arithmetic operation is selected, the second part comes into play and the number B is taken and the previous operations are repeated. Then, the numbers converted from BCD to binary are processed by the arithmetic module, and finally the result is encoded to be transmitted to the seven segment displays. The outputs s4 and s8 are the first digit of the 4-digit numbers. Therefore, I check whether they are different from the default values in the flag building block and allow a maximum of 4 inputs for each operand. A similar control mechanism occurs in the operation_inputs.v module, allowing only one input to be received until reset.

Solution Block Diagram

Figure 12 illustrates a fundamental component of our calculator digital design project, showcasing the interaction between the calculator's control unit and its various peripheral modules. At the heart of the diagram, we see the Control Signal pathways, which serve as the conduits through which operational commands are transmitted, ensuring coherent communication across the system.

Above the Control Signal pathways, the 'Operand Storage Module' forms a crucial part of the calculator's memory system, where operands are stored temporarily during calculations. This module is essential for the arithmetic and logic unit to accurately process and compute the mathematical operations.

The keyboard interface, represented below the Operand Storage Module, is where the user input is received. This is the point of interaction with the calculator, allowing users to enter numbers and operations into the system for processing. Lastly, we have the I/O Module, which stands for the

Input/Output Module. This component manages the flow of data into and out of the calculator, interfacing with the external environment, be it for displaying results, or for additional data input/output operations.

Together, these modules work in tandem, orchestrated by the control signals, to execute the complex series of actions that bring the digital calculator to life, making it capable of performing a wide array of mathematical calculations.

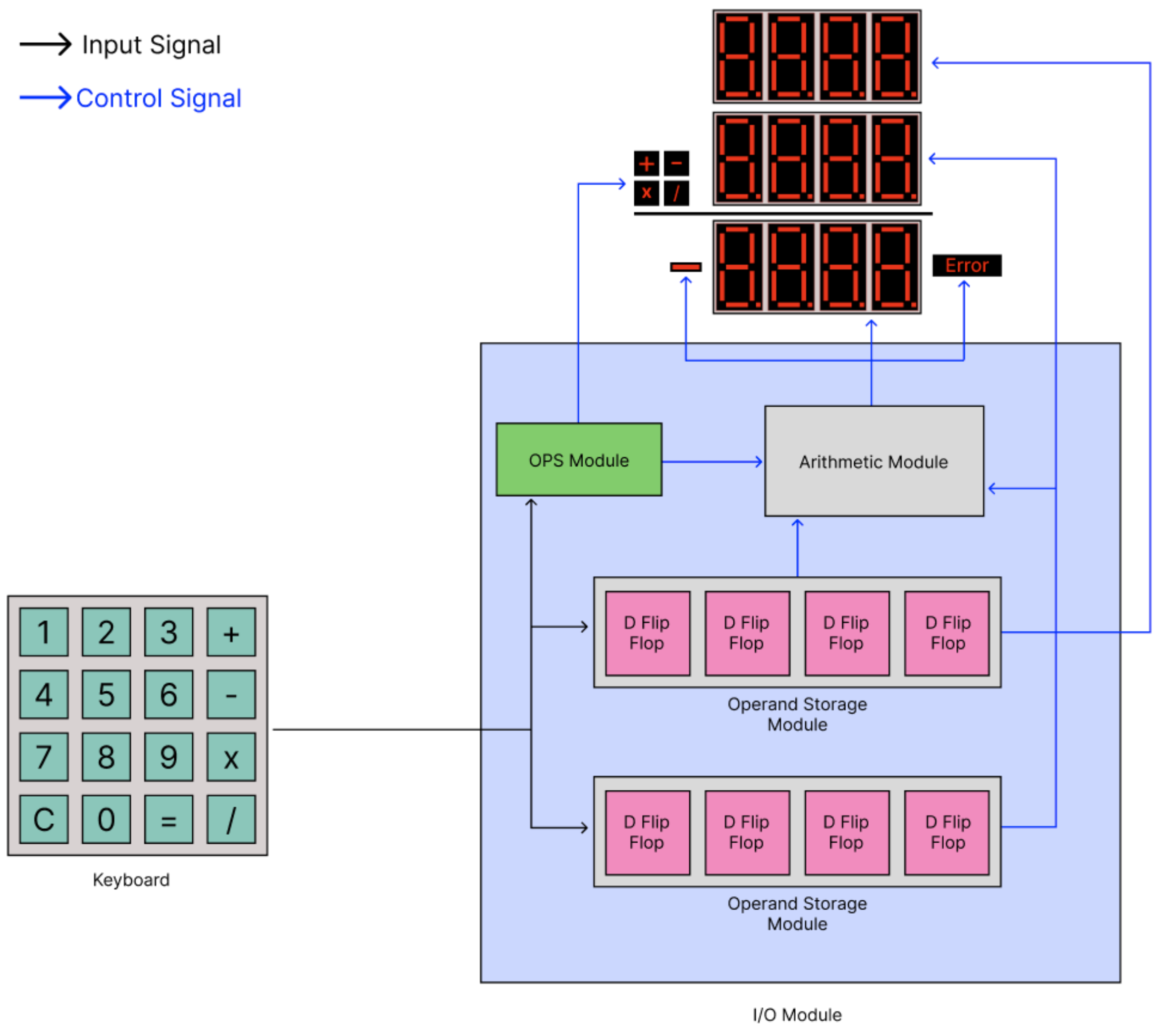


Figure 12: Solution Block Diagram

Summary of Results

This section presents the comprehensive evaluation outcomes of the calculator application. The testing process encompassed various example test cases aimed at gauging its performance and functionality. Notably, the calculator consistently demonstrated the expected behavior across all test scenarios, affirming its reliability and adherence to predefined functionalities.

						MINUS SIGN	RESULT	ERROR SIGN
(C)	->					[Minus OFF]	[OFF]	[Err OFF]
(C)	(1)(2)(3)(4)	(+)	(5)(6)(7)(8)	(=)	->	[Minus OFF]	[6912]	[Err OFF]
(C)	(1)(2)(3)(4)	(-)	(5)(6)(7)(8)	(=)	->	[Minus ON]	[4444]	[Err OFF]
(C)	(1)(2)(3)(4)(5)	(+)	(5)(6)(7)(8)(9)(=)		->	[Minus OFF]	[6912]	[Err OFF]
(C)	(1)	(+)	(5)	(=)	->	[Minus OFF]	[0006]	[Err OFF]
(C)	(1)	(/)	(0)	(=)	->	[Minus OFF]	[OFF]	[Err ON]
(C)	(9)(9)(9)(9)	(+)	(1)	(=)	->	[Minus OFF]	[OFF]	[Err ON]
(C)	(9)(9)(9)(9)	(x)	(9)	(=)	->	[Minus OFF]	[OFF]	[Err ON]
(C)	(0)	(-)	(1)	(=)	->	[Minus ON]	[0001]	[Err OFF]
(C)	(1)(=)(=)	(-)(+)	(9)(9)(/)(x)(8)(=)		->	[Minus ON]	[0997]	[Err OFF]

Table 1: Test Cases

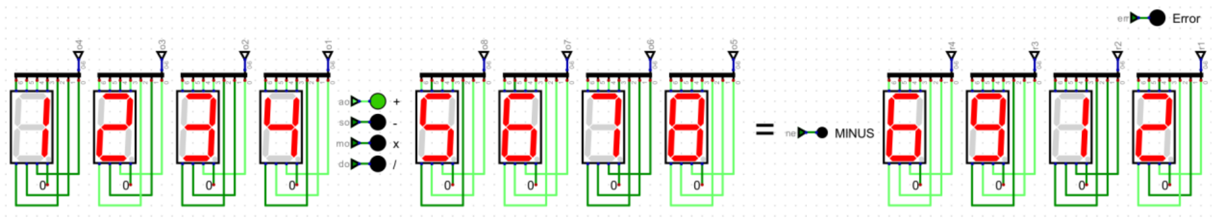


Figure 13: Test Case 1

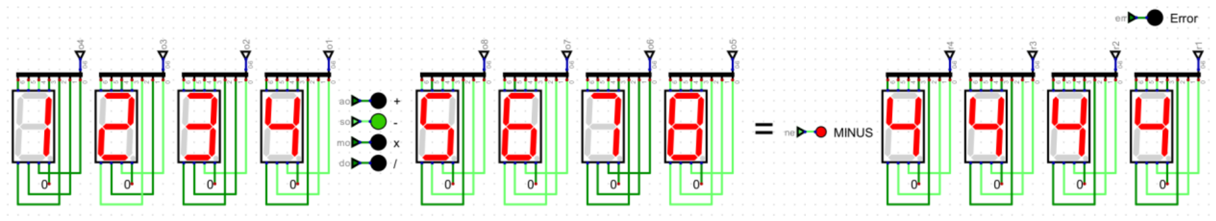


Figure 14: Test Case 2

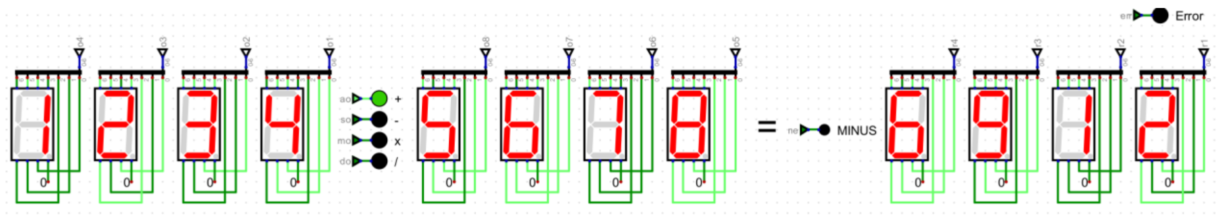


Figure 15: Test Case 3

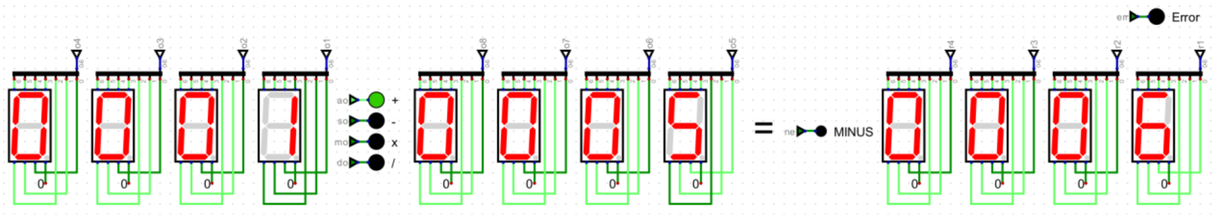


Figure 16: Test Case 4

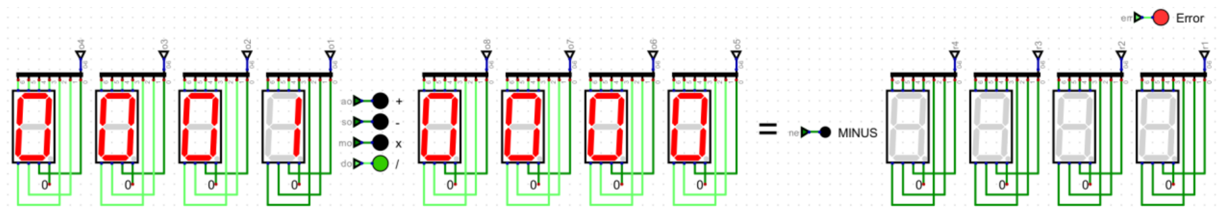


Figure 17: Test Case 5

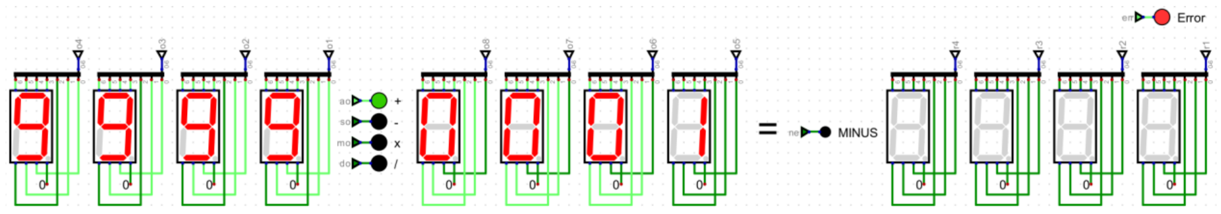


Figure 18: Test Case 6

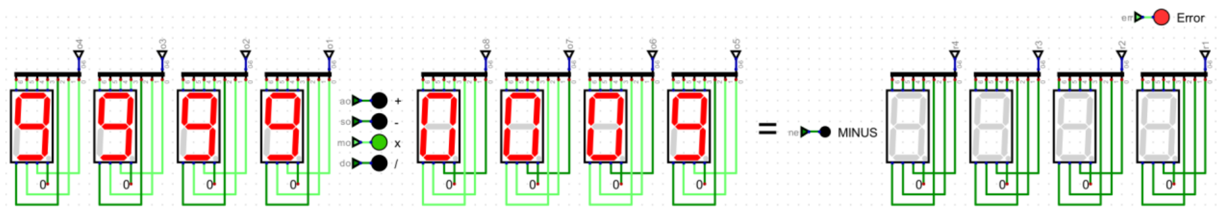


Figure 19: Test Case 7

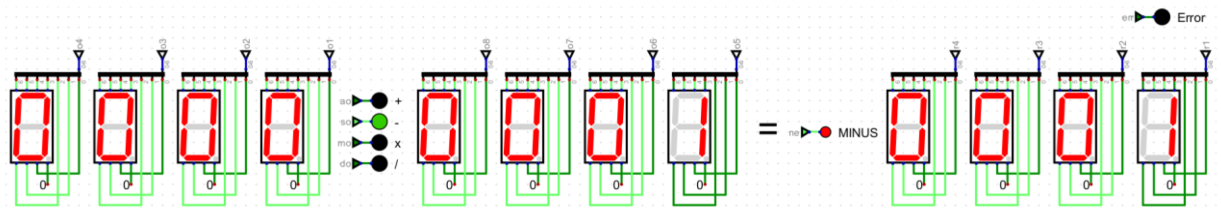


Figure 20: Test Case 8

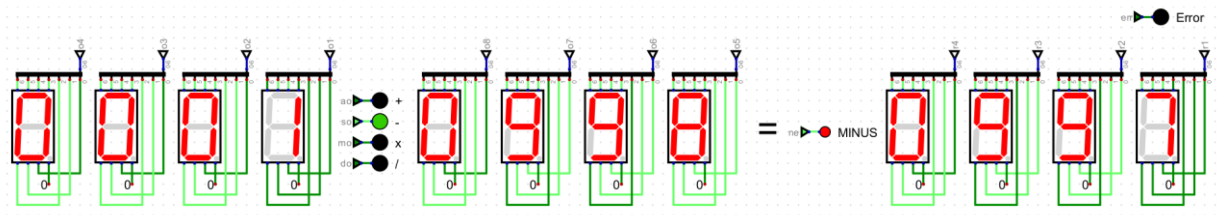


Figure 21: Test Case 9

Lessons Learned

Throughout this project, I've gained valuable insights into design and problem-solving, especially in modular design principles. Unlike my previous work, this venture emphasized breaking down the project into independent modules. This approach streamlined development, boosted maintainability, and allowed focused adjustments without disrupting the entire system.

Encountering and overcoming various errors and bugs became an essential part of my learning journey. Each challenge served as a unique problem to solve, enhancing my problem-solving skills and instilling a methodical approach to troubleshooting.

Simultaneously, my proficiency in Verilog significantly improved. Hands-on experience helped me navigate its complexities, empowering me to write more efficient and reliable code. What was once intimidating became a powerful tool in my skill set.

In the context of creating a calculator, learning how to handle storage elements was instrumental. Getting a grasp on manipulating storage mechanisms was essential for ensuring the smooth flow of data within the system and maintaining accurate timing in digital designs.