



## **EE314 – Mini-Project**

**Task:** Design digital logic design to perform the functions of a calculator on Basys 3.

### **Group 4**

**GROUP: 4** \_\_\_\_\_

---

**DATE: 06/05/2025**

I declare that I have not plagiarised or excessively quoted the work of others, nor have I colluded with others to represent collaborative work as my own. I have appropriately cited all information derived from the published and unpublished work of others, and less than 5% of my submission quotes the source material directly.

I declare that parts of this submission have contributions from AI software and that it aligns with acceptable use as specified as part of the assignment brief/ guidance and is consistent with good academic practice. I understand that as long as my use falls within the scope of appropriate use as defined in the assessment brief/guidance then this declaration will not have any direct impact on the grades awarded.

We used the following AI tools: Chat GPT

I declare use of the above software to (tick as appropriate):

X	(i) Generate ideas or structure suggestions, for assistance with understanding core concepts, or other substantial foundational and preparatory activity.
X	(ii) Write, rewrite, rephrase and/or paraphrase part of this report.
	(iii) Generate some other aspect of the submitted assessment. Details below:

NAME: Joshua Inioluwa Salako

SIGNED:

NAME: Ayobami Samuel Alarape

SIGNED:

NAME: \_\_\_\_Fortune Egha Olose\_\_\_\_

SIGNED:

NAME: \_\_\_\_Destiny Nosa\_\_\_\_

SIGNED:

## 1. Project Completion Plan:

Tasks	Description	Name
Introduction to the mini project	We planned the structure of the calculator by identifying the inputs (+, -, x, /) and output.	All
Block diagram	We designed a labeled block diagram showing the main components of our calculator system and how they interact within the FPGA.	Fortune and Joshua
Module design	Internal structure of the calculator.	Joshua
Verilog's module	We implemented the calculator using Verilog, writing the top-level module and all submodules.	Joshua
Testbench coding	We developed a Verilog testbench to simulate and validate the behavior of our design, testing all operations and ensuring that the outputs matched expected values.	Destiny
Simulation and testing	We performed simulations in Vivado and we tested the synthesized design on the Basys 3 board using switches and buttons.	Ayobami and Fortune
Writing the report	Documentation of project approach, design, Verilog module, testbench results, Basys 3 calculator testing, and conclusions.	All
Formatting the report	Formatted the report to meet the EE314 submission requirements.	Destiny and Ayobami

## 2. Introduction (~ 1 page):

### a) Block Diagram

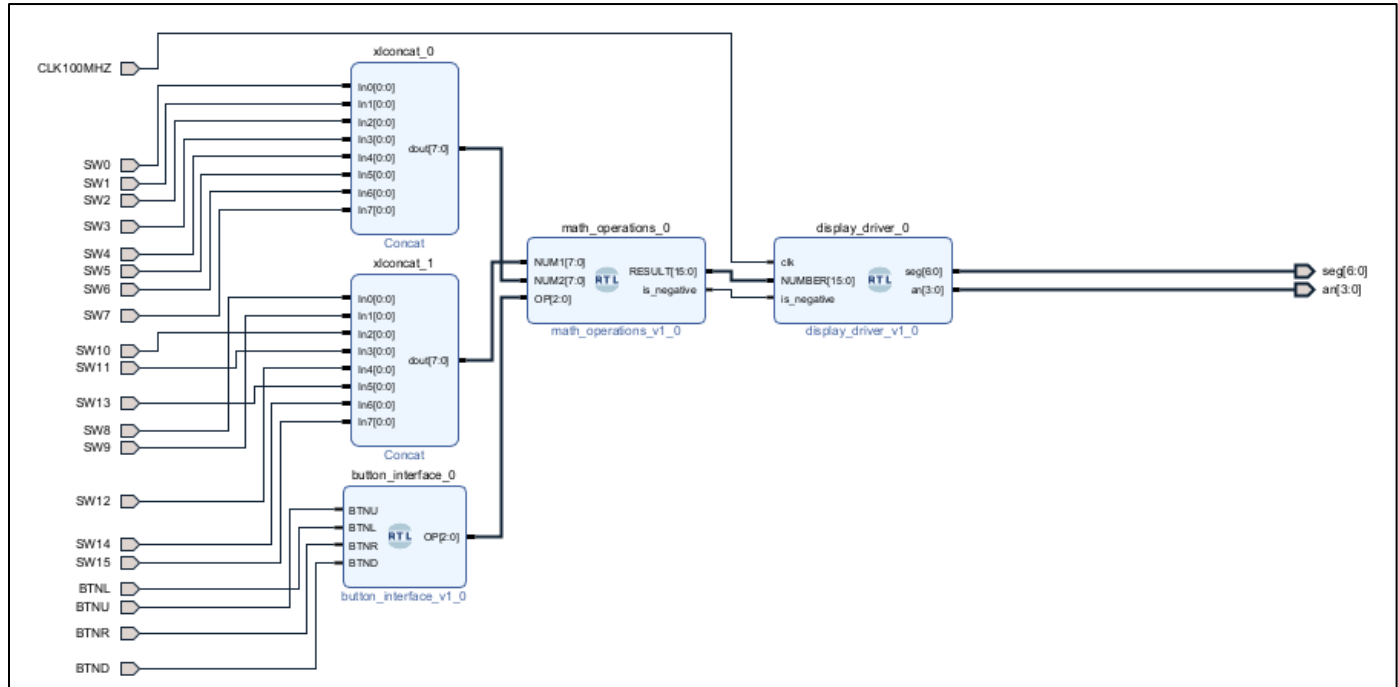


Figure 1 - Block Design

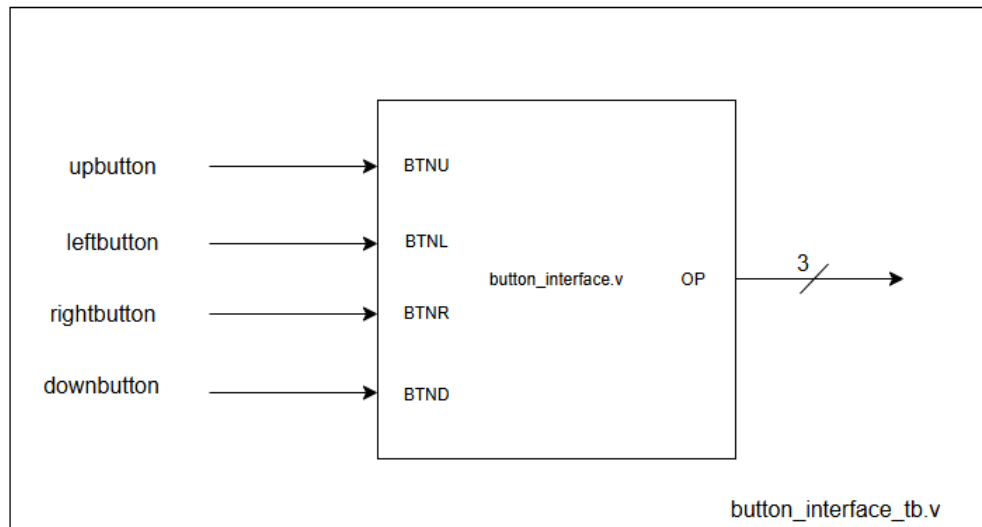


Figure 2 - button\_interface module

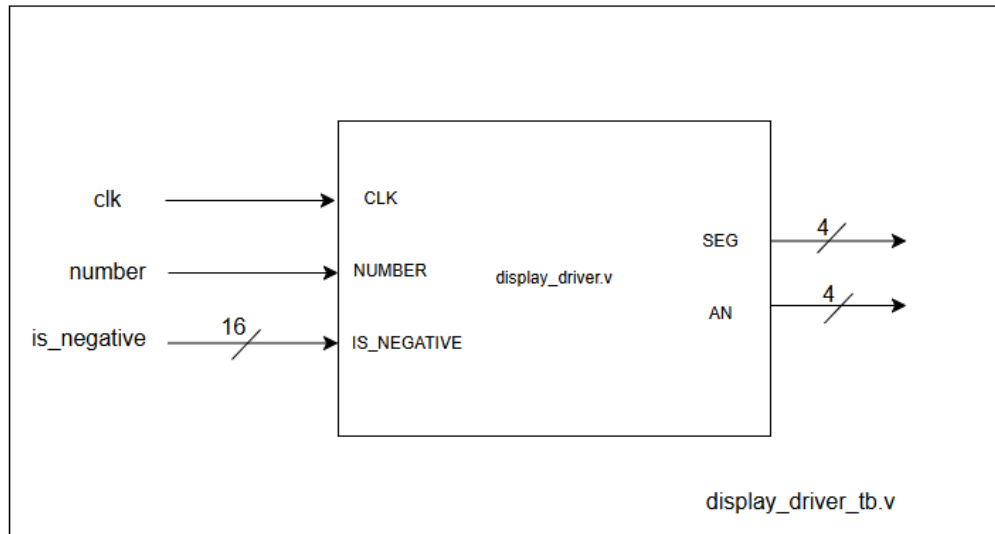


Figure 3 - display\_driver module

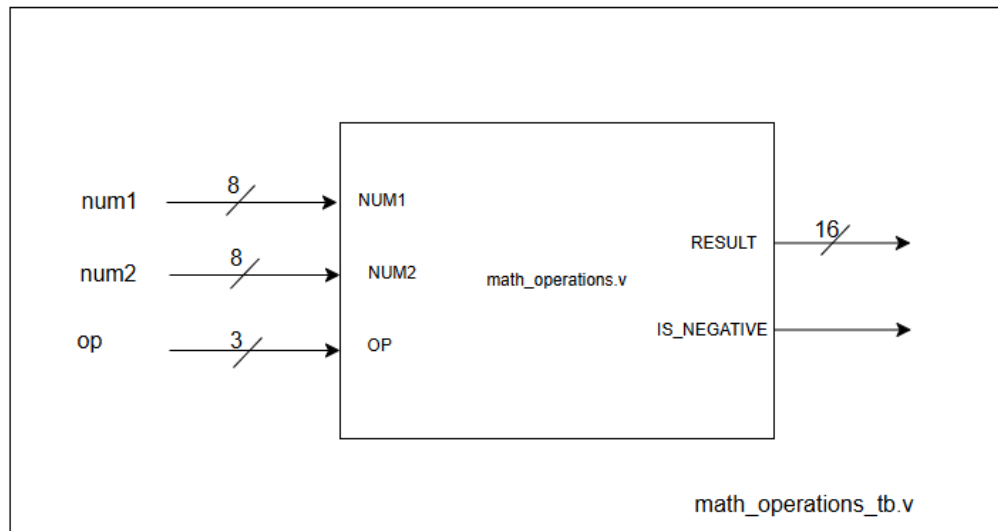


Figure 4 - math\_operations module

## b) Description of the use of the design

The calculator operates using input switches and directional buttons on the BASYS 3 FPGA.

The user sets two 8-bit input values (NUM1 and UM2) using the switches and the first number displays while the second one doesn't.

Arithmetic operations: addition, subtraction, multiplication and division are selected using the directional buttons (BTNU,BTN,BTNR,BTND) and are help down to show the calculated result of the two numbers on the 4-digit 7 -segment display.

Additionally this design supports negative values (e.g. when the first number is less than the second number) and displays the value as well as a negative sign.

### 3. Verilog Testbench (with Comments):

Code + Comments

```
`timescale 1ns / 1ps

// 02/05/2025
// Module Name: button_interface_tb
// Project Name: EE314_Calculator
// Description: Testbench to test full functionality
// of button_interface module

module button_interface_tb();
// Declaring input/output signals
reg upbutton, leftbutton, rightbutton, downbutton;
wire [2:0] op;

// Instantiate Unit Under Test
button_interface uut (
    .BTNU(upbutton),
    .BTNL(leftbutton),
    .BTNR(rightbutton),
    .BTND(downbutton),
    .OP(op)
);

// Test Sequence
initial begin
    // Default state
    upbutton = 0; leftbutton = 0; rightbutton = 0; downbutton = 0;
    #10;

    // Test UP (Addition)
    upbutton = 1; leftbutton = 0; rightbutton = 0; downbutton = 0;
    #10;

    // Test LEFT (Subtraction)
    upbutton = 0; leftbutton = 1; rightbutton = 0; downbutton = 0;
    #10;

    // Test RIGHT (Multiplication)
    upbutton = 0; leftbutton = 0; rightbutton = 1; downbutton = 0;
    #10;

    // Test DOWN (Division)
    upbutton = 0; leftbutton = 0; rightbutton = 0; downbutton = 1;
    #10;

    // Test no buttons pressed (pass-through)
    upbutton = 0; leftbutton = 0; rightbutton = 0; downbutton = 0;
    #10;

    // Test multiple buttons (BTNU priority)
    upbutton = 1; leftbutton = 1; rightbutton = 1; downbutton = 1;
    #10;

    // Finish simulation
    $finish;
end

endmodule
```

```

`timescale 1ns / 1ps

module math_operations_tb;

// Declare testbench registers to drive the inputs of the DUT
reg [7:0] NUM1;    // First input number
reg [7:0] NUM2;    // Second input number
reg [2:0] OP;      // Operation code selector

// Declare wires to capture outputs from the DUT

wire [15:0] RESULT; // Result from math_operations
wire is_negative;   // Flag indicating if the result of subtraction was negative

// Instantiate the math_operations module (Device Under Test)
math_operations uut (
    .NUM1(NUM1),
    .NUM2(NUM2),
    .OP(OP),
    .RESULT(RESULT),
    .is_negative(is_negative)
);

// Initial block runs once at time 0
initial begin

    // Test Case 1: Addition
    // Expect: 10 + 15 = 25
    NUM1 = 8'd10;
    NUM2 = 8'd15;
    OP = 3'b000; // Select addition
    #10;         // Wait 10ns for result to propagate

    // Test Case 2: Subtraction (positive)
    // Expect: 30 - 20 = 10, is_negative = 0
    NUM1 = 8'd30;
    NUM2 = 8'd20;
    OP = 3'b001; // Select subtraction
    #10;

    // Test Case 3: Subtraction (negative)
    // Expect: 10 - 30 = -20 → Result = 20, is_negative = 1
    NUM1 = 8'd10;
    NUM2 = 8'd30;
    OP = 3'b001; // Select subtraction
    #10;
end

```

```

// Test Case 4: Multiplication (small)
// Expect: 12 * 12 = 144
NUM1 = 8'd12;
NUM2 = 8'd12;
OP = 3'b010; // Select multiplication
#10;

// Test Case 5: Multiplication (large)
// Expect: 255 * 255 = 65025
NUM1 = 8'd255;
NUM2 = 8'd255;
OP = 3'b010; // Select multiplication
#10;

// Test Case 6: Division (normal)
// Expect: 100 / 5 = 20
NUM1 = 8'd100;
NUM2 = 8'd5;
OP = 3'b011; // Select division
#10;

// Test Case 7: Division by zero
// Expect: 100 / 0 = 0 (safe handling)
NUM1 = 8'd100;
NUM2 = 8'd0;
OP = 3'b011; // Select division
#10;

// Test Case 8: No operation (pass-through)
// Expect: Output = NUM1 = 77
NUM1 = 8'd77;
NUM2 = 8'd123; // Irrelevant in this case
OP = 3'b100; // Display NUM1
#10;

// Test Case 9: Invalid operation
// Expect: Default to 0
OP = 3'b111; // Invalid code
#10;

// End the simulation
$stop; // Use $stop for waveform tools; replace with $finish if not needed
end

endmodule

```



```

(change # and 16d' to binary or 8d')
`timescale 1ns / 1ps

module display_driver_tb;

    reg clk;
    reg [15:0] number;
    reg is_negative;

    wire [6:0] seg;
    wire [3:0] an;

    // Instantiate the display driver module
    display_driver uut (
        .clk(clk),
        .NUMBER(number),
        .is_negative(is_negative),
        .seg(seg),
        .an(an)
    );

    // 100 MHz clock (10 ns period)
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    // Apply test cases one after another
    initial begin
        number = 16'd0;
        is_negative = 0;

        // Wait 100,000 ns for initial setup
        #100_000;

        // Test 1: Show 1234
        number = 16'd1234;
        is_negative = 0;
        #200;

        // Test 2: Show 45
        number = 16'd45;
        is_negative = 0;
        #200;

        // Test 3: Show 0
        number = 16'd0;
        is_negative = 0;
        #200;

        // Test 4: Show -20
        number = 16'd20;
        is_negative = 1;
        #200;
    end
endmodule

```

```

        // Test 5: Show 9999
        number = 16'd9999;
        is_negative = 0;
        #200;

        $stop; // Stops simulation in Vivado
    end
endmodule

```

## 4. Verilog Module(s) (with Comments):

Code + Comments

### Button\_interface module

```

`timescale 1ns / 1ps

// 02/05/2025
// Module Name: button_interface
// Project Name: EE314_Calculator
// Description: Module to handle arithmetic operations using
// buttons on FPGA

module button_interface(
    input BTNU, BTNL, BTNR, BTND, // UP, LEFT, RIGHT and DOWN Buttons
    output reg [2:0] OP           // operator register
);

//
always @(*) begin
    if (BTNU)
        OP = 3'b000; // Up Button is Addition
    else if (BTNL)
        OP = 3'b001; // Left Button is Subtraction
    else if (BTNR)
        OP = 3'b010; // Right Button is Multiplication
    else if (BTND)
        OP = 3'b011; // Down Button is Division
    else
        OP = 3'b100; // No operation - Just display first number
end
endmodule

```

## Math\_operations module

```
`timescale 1ns / 1ps

// 02/05/2025
// Module Name: math_operations
// Project Name: EE314_Calculator
// Description: Module performs basic math operations on
// two 8-bits numbers and outputs a 16-bit output to handle
// large numbers. Also outputs a flag to say if result is negative
// Declaring output and input ports
module math_operations(
    input [7:0] NUM1,          // two 8-bit unsigned numbers
    input [7:0] NUM2,          //
    input [2:0] OP,            // operation selector ( +,-,/,*)
    output reg [15:0] RESULT,  //16-bit result for large multiplication)
    output reg is_negative     // flag for subtraction
);

reg signed [8:0] temp_sub; // temporary register for signed subtraction

always @(*) begin // combinational logic block that executes whenever
signals change
    is_negative = 1'b0; // Default

    case (OP)
        3'b000: RESULT = NUM1 + NUM2; // Addition
        3'b001: begin // Subtraction
            temp_sub = $signed({1'b0, NUM1}) - $signed({1'b0, NUM2}); //
directive used to
            is_negative = (temp_sub < 0); //
convert to signed 9-bit
            RESULT = is_negative ? -temp_sub : temp_sub;
        end
        3'b010: RESULT = NUM1 * NUM2; // Multiplication
        3'b011: RESULT = (NUM2 != 0) ? NUM1 / NUM2 : 16'd0; // Divison
        (avoiding divison by zero)
        3'b100: RESULT = NUM1; // No operation - Simply displays first
number
        default: RESULT = 16'd0;
    endcase
end

endmodule
```

## Display\_driver module

```

`timescale 1ns / 1ps

// 02/05/2025
// Module Name: display_driver
// Project Name: EE314_Calculator
// Description: Drives a 4-digit 7-segment display by converting a 16-bit
// unsigned number to 4 BCD digits (thousands, hundreds, tens, units).
// Also multiplexes the digits at high speed to create the illusion of a static display.
// And handles negative numbers (shows '-' on the leftmost digit if `is_negative` is high).

// Declaring output and input ports
module display_driver(
    input clk,
    input [15:0] NUMBER,          // Unsigned result to display
    input is_negative,           // Signal from math operations module for subtraction
    output reg [6:0] seg,        // Output register for segment display
    output reg [3:0] an          // Output register for anodes
);

reg [3:0] digit3, digit2, digit1, digit0; // Registers to hold BCD digits
reg [19:0] refresh_counter = 0;
reg [1:0] digit_select = 0; // Digit selector

// Binary to BCD procedural block to
// split 16-bit NUMBER into BCD digits
always @(*) begin
    digit3 = (NUMBER / 1000) % 10; // thousands
    digit2 = (NUMBER / 100) % 10;  // hundreds
    digit1 = (NUMBER / 10) % 10;   // tens
    digit0 = NUMBER % 10;          // ones
end

// Refresh counter for multiplexing
always @(posedge clk) begin
    refresh_counter <= refresh_counter + 1;
    if (refresh_counter == 100000) begin
        refresh_counter <= 0;
        digit_select <= digit_select + 1;
    end
end

// Active-low 7-segment decoder
// Converts a 4-bit BCD digit to its 7-segment pattern
function [6:0] bcd_to_7seg;
    input [3:0] digit;
    case (digit)
        4'd0: bcd_to_7seg = 7'b1000000;
        4'd1: bcd_to_7seg = 7'b1111001;
        4'd2: bcd_to_7seg = 7'b0100100;
        4'd3: bcd_to_7seg = 7'b0110000;
        4'd4: bcd_to_7seg = 7'b0011001;
        4'd5: bcd_to_7seg = 7'b0010010;
        4'd6: bcd_to_7seg = 7'b0000010;
        4'd7: bcd_to_7seg = 7'b1111000;
        4'd8: bcd_to_7seg = 7'b0000000;
        4'd9: bcd_to_7seg = 7'b0010000;
        default: bcd_to_7seg = 7'b1111111;
    endcase
endfunction

```

```

// Multiplexing logic
// Selects which digit to display based on `digit_select` and anode.
always @(*) begin
    case (digit_select)
        2'd0: begin
            seg = bcd_to_7seg(digit0);
            an = 4'b1110;
        end
        2'd1: begin
            seg = bcd_to_7seg(digit1);
            an = 4'b1101;
        end
        2'd2: begin
            seg = bcd_to_7seg(digit2);
            an = 4'b1011;
        end
        2'd3: begin
            seg = is_negative ? 7'b0111111 : bcd_to_7seg(digit3); // '-' or digit3
            an = 4'b0111;
        end
    endcase
end

endmodule

```

## 5. Description of the Design

Explain the functionality of the design in plain English.

**Math\_operations.v** takes in two 8-bit numbers (NUM1, NUM2) and a 3-bit operation selector (OP). It calculates a result and determines if the result of a subtraction is negative.

This is the code that performs the operations and it works like this:

If OP is 000, it performs addition, calculating NUM1 + NUM2. If OP is 001, it performs subtraction by computing the absolute difference between NUM1 and NUM2 and sets the `is_negative` flag if the result is negative.

If OP is 010, it carries out multiplication using NUM1 \* NUM2. If OP is 011, it executes division, safely computing NUM1 / NUM2 and setting the result to zero if NUM2 is zero to avoid division by zero. If OP is 100, it performs a pass-through, with the value of NUM1 as the output. If OP is anything else, it defaults the result to zero.

The outputs are RESULT which is 16 bits and it's the outcome of the operation, and the other output is `is_negative` that is only used in subtraction to show a minus sign.

**Display\_driver.v** shows the operations in the 4 digit 7 segment display (the digital clock display with 4 slots) on Basys 3 board.

In this module we have the BCD Conversion used to convert the 16-bit result into 4 decimal digits. Then the multiplexing that switches between digits very fast, so they all appear lit up simultaneously. And the negative display for the minus sign, if `is_negative` is true, it shows a minus (-) sign in the leftmost digit position.

**Button\_interface.v** module maps Basys 3 buttons to specific operations for the calculator

BTNU (Up button) → Performs addition.

BTNL (Left button) → Performs subtraction.

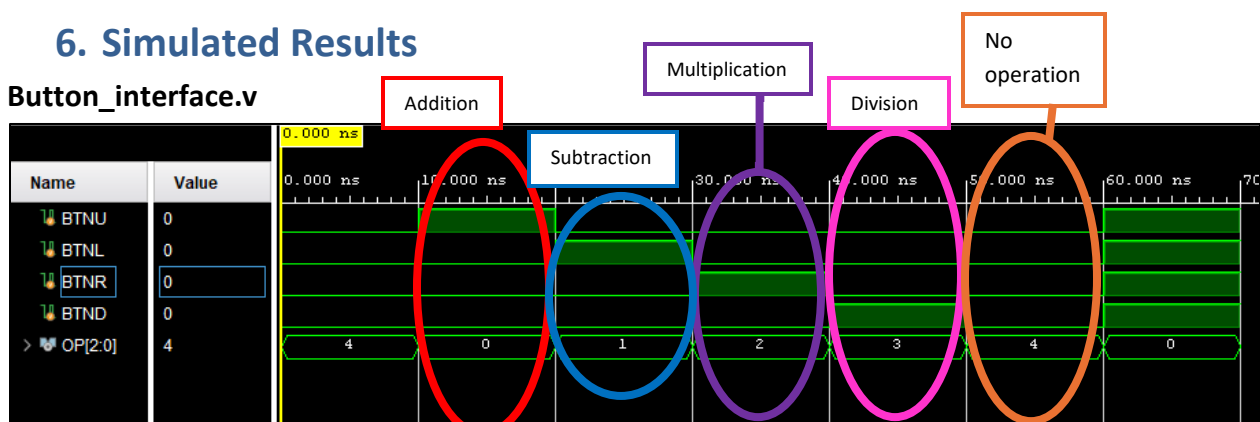
BTNR (Right button) → Performs multiplication.

BTND (Down button) → Performs division.

If no button is pressed, the OP code is set to 100, meaning just show NUM1 on the display (pass-through). This module outputs a 3-bit signal based on the button being pressed, which controls what operation the calculator performs, so the up button is 000, the left button 001, the right button 010, the down button is 011.

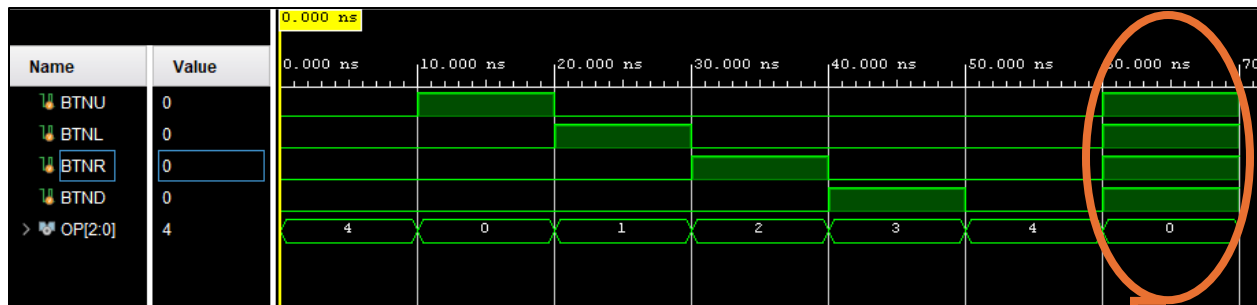
## 6. Simulated Results

### Button\_interface.v



The button interface module is responsible for handling user input from buttons on the FPGA. It outputs a 3-bit signal and this shows which mathematical operation is selected.

- 0 - when the UP BUTTON is pressed the output signal is 0 which means that the operation is addition
- 1 - when the LEFT BUTTON is pressed the output signal is 1 which means that the operation is subtraction
- 2 - when the RIGHT BUTTON is pressed the output signal is 2 which means that the operation is multiplication
- 3 - when the DOWN BUTTON is pressed the output signal is 3 which means that the operation is division
- 4 - highlights when no button is pressed OP is 4 meaning there is no operation done.



Additionally if all buttons are pressed simultaneously it defaults to order of the operations, hence when all the button signals are high the output is 0 which is addition.

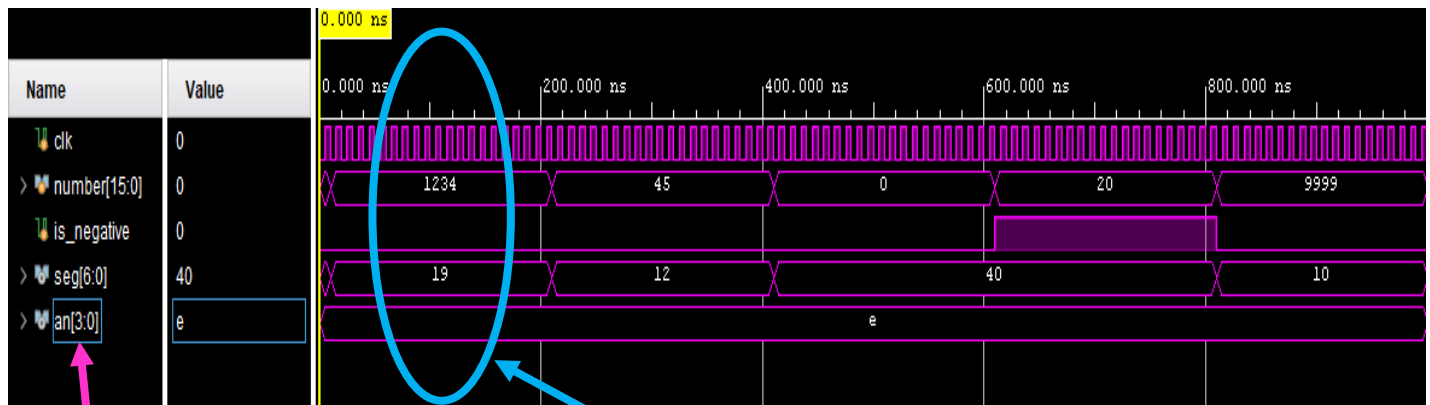
The operations correspond to the buttons in the following manner:

- **Up button:** Addition (3'b000)
- **Left button:** Subtraction (3'b001)
- **Right button:** Multiplication (3'b010)
- **Down button:** Division (3'b011)

If no buttons are pressed, the output is set to 3'b100, indicating no operation.

### Display driver

The module is cycling through the 4 digits using an, with each cycle showing a new digit and its corresponding seg output. The digits are displayed one at a time but the fast refresh rate gives illusion of all digits being displayed simultaneously using multiplexing.



an represents the anode control for the 4-digit 7 segment display which tells which digit is currently active. The anode lines are active low. In the display driver module the bcd is converted into 7 segment pattern

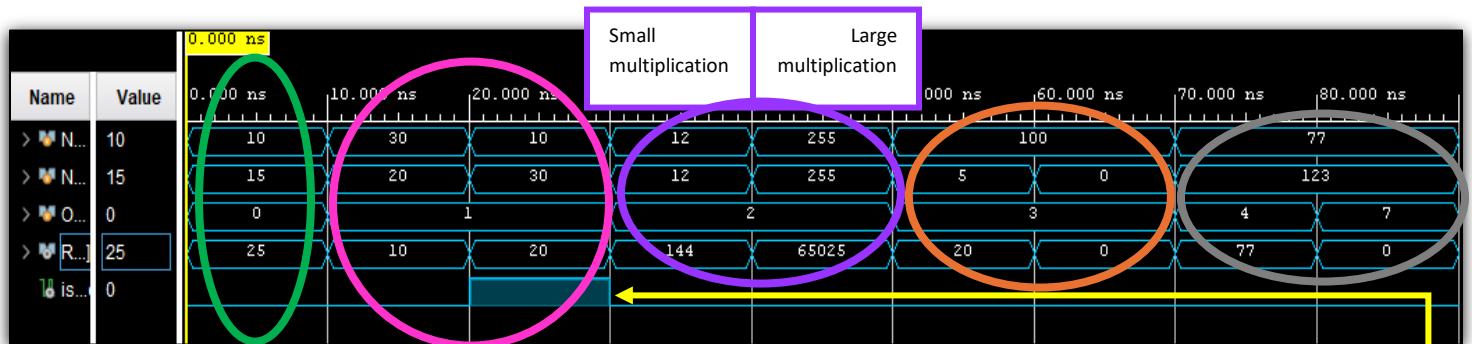
Value is equal to 'e' in hex which is 1110 in bit binary. This means only rightmost bit (units) is active this and occurs in the simulation as the clock signal isn't toggling fast enough.

Example:

19 in decimal represents 0011001 in binary and from bcd\_to\_7seg this corresponds to **digit 4** which correlates to the last digit of the number **1234**

## Math\_operations module

This module is responsible for performing mathematical operations on two unsigned 8-bit numbers (**NUM1** and **NUM2**) and provide a 16-bit '**RESULT**'. It also has a flag (**is\_negative**) that indicates if the results of a subtraction operation was negative. (**OP** also from button interface)



0 – Tests the addition functionality of code, **OP** is set to 0 therefore it is addition and result is sum of the two numbers, the **is\_negative** flag is also low as **NUM1 > NUM2**.

1 – Tests subtraction functionality for when **NUM1 > NUM2** and for when **NUM1 < NUM2**.

3 – Tests multiplication as **OP** = 3 with small numbers as well as large numbers

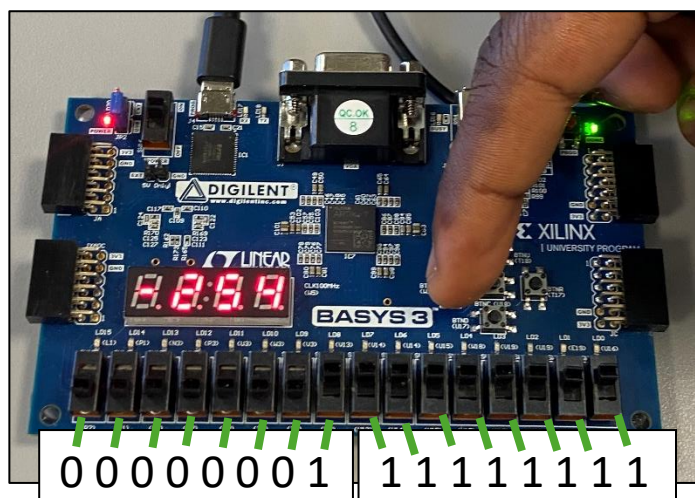
2 – Tests normal division functionality as well as division by 0.

4 – Test of no operation functionality e.g. when **OP** = 4, (Pass-through of **NUM1**)

Here the **is\_negative** flag is asserted as the first number is less than the second indicated that it is a negative value to be displayed

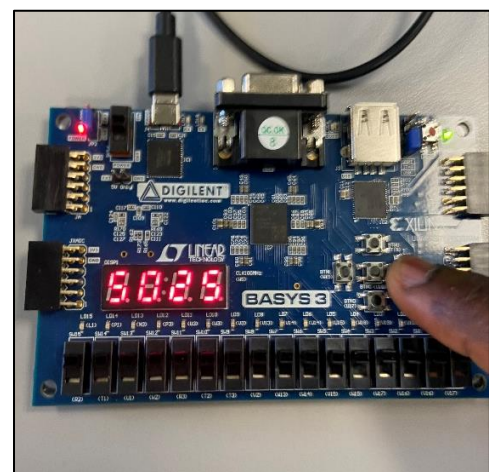
## Visual Examples,

### Case when NUM1 < NUM2 -



In this FPGA board picture Switches 15 – 8 represent **NUM1** while Switches 7 – 0 represent **NUM2**  
Therefore **NUM1** = 1 and **NUM2** = 255

### Case when RESULT is 5 digit number



In this case all Switches are asserted (set high) meaning both **NUM1** and **NUM2** are 255. The value on the display highlights that only the last 4 digits of the product are shown. As  $255 \times 255 = 65,025$  but only 5025 is shown on the display.



## 7. Measured Results – From Experiments

The calculator works as expected in simulation. It correctly performs addition, subtraction, multiplication, division, and pass-through operations based on button inputs. The result is displayed properly on the 4-digit 7-segment display, and if a subtraction result is negative, a minus sign is shown. All operations were tested using a testbench, and the outputs matched the expected results.

### What are its limitations?

The Basys 3 FPGA board has only a few ways to get input from the user, just 16 switches and 5 buttons. It doesn't have things like a keypad or touchscreen. It also has a small 4-digit display that's good for showing short numbers, but it can't show words or long results, since it can only show 4 digits at a time.

About our calculator it has other limitations. We don't have an equal button so to make the calculation we need to hold the button of the operation sign we use (e.g.,  $17 + 10$  ---- number 17, then number 10 and then hold sum button) if the button is not pressed, the operation is not shown. Also it only shows the left hand side numbers (NUM1) before doing the operation, so we can't see NUM2 displaying.

## 8. Conclusions (~ 1 page)

### a) Explain in your own words how you proved that the design works correctly, and what evidence there is to show that it works.

A testbench in Verilog was created that executed all supported operations of the math\_operations module, including addition, subtraction, multiplication, division, and pass-through (displaying a single number), in order to confirm the accuracy of the calculator design. Every test was selected with care to cover both common use cases and possible edge cases. With common Use Cases we mean the typical or expected ways that the calculator will be used. They represent standard, everyday scenarios where the calculator is used as intended. For example:

Adding two small numbers, like  $12 + 34$ .

Multiplying  $7 \times 8$ .

Dividing  $20 \div 4$ .

Subtracting  $15 - 5$ .

Then Edge Cases are the unusual, extreme, or unexpected scenarios that can reveal weaknesses or bugs in the design. They test the boundaries or limits of the system. In our case these are attempting to divide by zero ( $15 \div 0$ ), multiplying large numbers ( $255 \times 255$ ) that cause overflow and pressing multiple operation buttons at once.

## **b) What additional features if any did you incorporate into the design to improve it?**

### Negative Result Handling

When doing subtraction, the design checks if the result is negative. If it is, it sets a signal and shows a minus sign (-) on the display. This helps the user understand the result better.

## **c) What are the limitations of your design?**

### Priority Logic Bias

In the button\_ interface module the logic structure gives higher priority to BTNU over the others due to its order in the if-else chain. If multiple buttons are accidentally pressed or bouncing occurs, only the first true condition is honored. This could confuse users during rapid inputs.

### Only One Operation at a Time

The design only supports one button press at a time. If multiple buttons are pressed together, only the first one in the priority order (Up > Left > Right > Down) is used, which may cause confusion.

### Division Simplification

Division is implemented with a basic check for division by zero. While it prevents hardware errors, it silently returns 0, which can be misleading to the user. More informative feedback (e.g., a display message or error flag) would improve robustness. There is also no decimal place or floating point value support. The calculator only handles whole numbers (integers). It cannot display decimal values or perform operations with fractions.

### No Decimal Point or Error Display

The system cannot display non-integer results (e.g.,  $7 \div 2 = 3.5$  will show just 3) or indicate errors (e.g., division by zero). It lacks flexibility for real-world calculator functions like floating-point display or status flags

### Limited Display Range

The 4-digit 7-segment display can only show numbers from 0000 to 9999. If the result is larger than 9999, it will only show the last 4 digits or it won't show the minus sign, which could be misleading.

### No Overflow Detection

The design doesn't check if an operation overflows the 16-bit result range, which could lead to incorrect values being displayed without warning.

### No Debouncing for Buttons

The button interface doesn't include debouncing logic, so on the Basys 3, rapid or noisy button presses might cause unexpected behavior.

## Appendix – All AI tool related interactions

**ALL** Questions asked of AI tools and Answers received from AI tools:

**Q: By increasing the result register to 16 bits wide would that allow for large multiplication**

A: Increasing the result register to **16 bits** *partially* allows for larger multiplication results—but only up to a certain limit. Here's a breakdown:

A **16-bit register** can store values up to:

$$2^{16} - 1 = 65,535$$

So, if your operands are **8-bit numbers** (maximum value 255), the **largest possible result** is:

$$255 \times 255 = 65,025$$

This result **just fits** within 16 bits, which means:

Yes, **16 bits are enough** for the maximum possible result of two 8-bit numbers.

**Q: I'm using block design for my project how would I group all of my switches into one array**

A: To group all your switches into a single array when using **Block Design (BD)** in Vivado, you typically use the **xlconcat** IP block to concatenate multiple single-bit switch inputs into a single vector (array)

### 1. Open Your Block Design

- Go to your Vivado project and open the **Diagram** view.

### 2. Add an xlconcat IP Block

- Click on the “+ Add IP” icon.
- Search for **xlconcat** and add it to your block design.

### 3. Configure xlconcat Width

- Double-click the xlconcat block.
- Set the **Number of Inputs** equal to the number of switches you want to group (e.g., 8 or 16).
- Each input will be **1-bit wide** by default—this is correct for switches.

### 4. Connect Switches to xlconcat Inputs

- From your board interface (likely via the **Board tab** or from I/O ports), drag each switch (e.g., SW0, SW1, etc.) to the respective In0, In1, ... InN of the xlconcat.

### 5. Output Vector

- The xlconcat block will now output a single **bus** signal (dout[N:0]) that groups all your switches.
- Connect this dout to your custom IP or logic module that expects a switch array input.

**Q: What is the Double Dabble method, explain carefully.**

A: **Concept Behind Double Dabble:**

The Double Dabble algorithm works by:

1. **Initializing** a binary input and an empty BCD register.
2. **Shifting** the binary bits one at a time into the BCD.
3. **Checking each BCD digit**—if it is  $\geq 5$ , **add 3** to correct it.
4. **Repeat** until all bits have been processed.

### Steps of the Algorithm:

Let's say you are converting an 8-bit binary number b[7:0] to BCD.

1. Initialize a BCD register with enough digits (e.g., 4 BCD digits = 16 bits).
2. For each bit from the binary number (starting with MSB):
  - Shift BCD and the next binary bit into the register.

- For each digit in BCD:
  - If the digit is  $\geq 5$ , add 3.
- 3. Repeat for all binary bits.

**Q: The bit stream is not generating due to constraints file error, fix this error.**

A: To fix a bitstream generation error caused by the constraints (.xdc) file in Vivado, follow these step-by-step troubleshooting guidelines. These issues often arise in projects using Block Design like yours (Basys 3):

### 1. Understand What the Constraints File Does

The .xdc file maps logical ports in your HDL or Block Design to physical FPGA pins. If there is a mismatch, bitstream generation will fail.

### 2. Verify Your Block Design Port Names

**In Vivado:**

- Go to Sources > Design Sources > your block design wrapper (e.g., design\_1\_wrapper.v)
- Open this file and note the top-level port names (e.g., SW[15:0], BTNU, led, etc.)
- These names must match exactly in your .xdc.

**For example, if your block design uses input [15:0] SW, your .xdc should have:**

```
set_property PACKAGE_PIN V17 [get_ports {SW[0]}]
...
set_property PACKAGE_PIN U12 [get_ports {SW[15]}]
```

### 3. Clean Your .xdc File

**Open the .xdc and do the following:**

- Comment out all unused lines by prefixing them with #
- Uncomment only the lines corresponding to actual ports in your design
- Check for typos or case mismatch

### 5. Set the Correct Top Module

- Right-click your top module (like design\_1\_wrapper.v)
- Click “Set as Top”
- This ensures Vivado knows what design to implement

**Q: I have how the way I want to display negative values using a flag, how do I work the negative sign on the display**

A: Yes, you **can display a minus sign on a 7-segment display**, and here's how to do it in your Basys 3 project.

### How 7-Segment Displays Work

A 7-segment display consists of 7 LEDs (labelled a–g) that light up in different patterns to represent numbers and symbols. To display a **minus sign**, you need to light up only the **middle horizontal segment** — this is **segment 'g'**. In binary form for **active-low logic** (0 = ON, 1 = OFF), the minus sign looks like this:  
7'b0111111 // Only segment 'g' is ON

Segment	a	b	c	d	e	f	g
Value	1	1	1	1	1	1	0