UNIVERSITI TUNKU ABDUL RAHMAN
FACULTY OF INFORMATION & COMMUNICATION TECHNOLOGY
DEPARTMENT OF COMPUTER AND COMMUNICATION TECHNOLOGY

BACHELOR OF INFORMATION TECHNOLOGY (HONOURS)
COMPUTER ENGINEERING

YEAR TWO SEMESTER ONE
Session: 202502

UCCE2023 DIGITAL SYSTEMS DESIGN

PRACTICAL ASSIGNMENT:
Specifications

Instructor: Mok Kai Ming
Date of Submission: 16 May 2025
You are not allow to change the following grouping or sequence

	Group Members	Tutorial Group	Task (component)	Mobile Contact
	Team work		Full Chip (Chip spec and verification)	
1	Cheah Lik Ding	2	Hex keypad code generator, synchronizer	011-59540693
2	Shu Mee Ang	2	ALU	0175418434
3	Jayson Chen Min Xiang	2	Multiplier, Barrel shifter	011-1141 9862
4	Chong Bing Hong	2	Storage + frequency generator Input / output converter	010-9105719

Table of Content

1.Architecture Specification: Written Spec
1.1 Functionality / Feature

Input Management:

- Uses hex keypad code generator and synchronizer to capture numeric digits and operation commands.
- Converts multi-digit hexadecimal inputs into an 8-bit binary number for processing.
- Support up to 255 digits.

8-Bit Integer Operations:

- Performs arithmetic operations (addition, subtraction, multiplication, division) on 8-bit numbers.
- Supports bitwise logic operations (AND, OR and XOR) on 8-bit operands.

Barrel Shifting:

- Implements left/right shift operations using a barrel shifter based on combinational logic

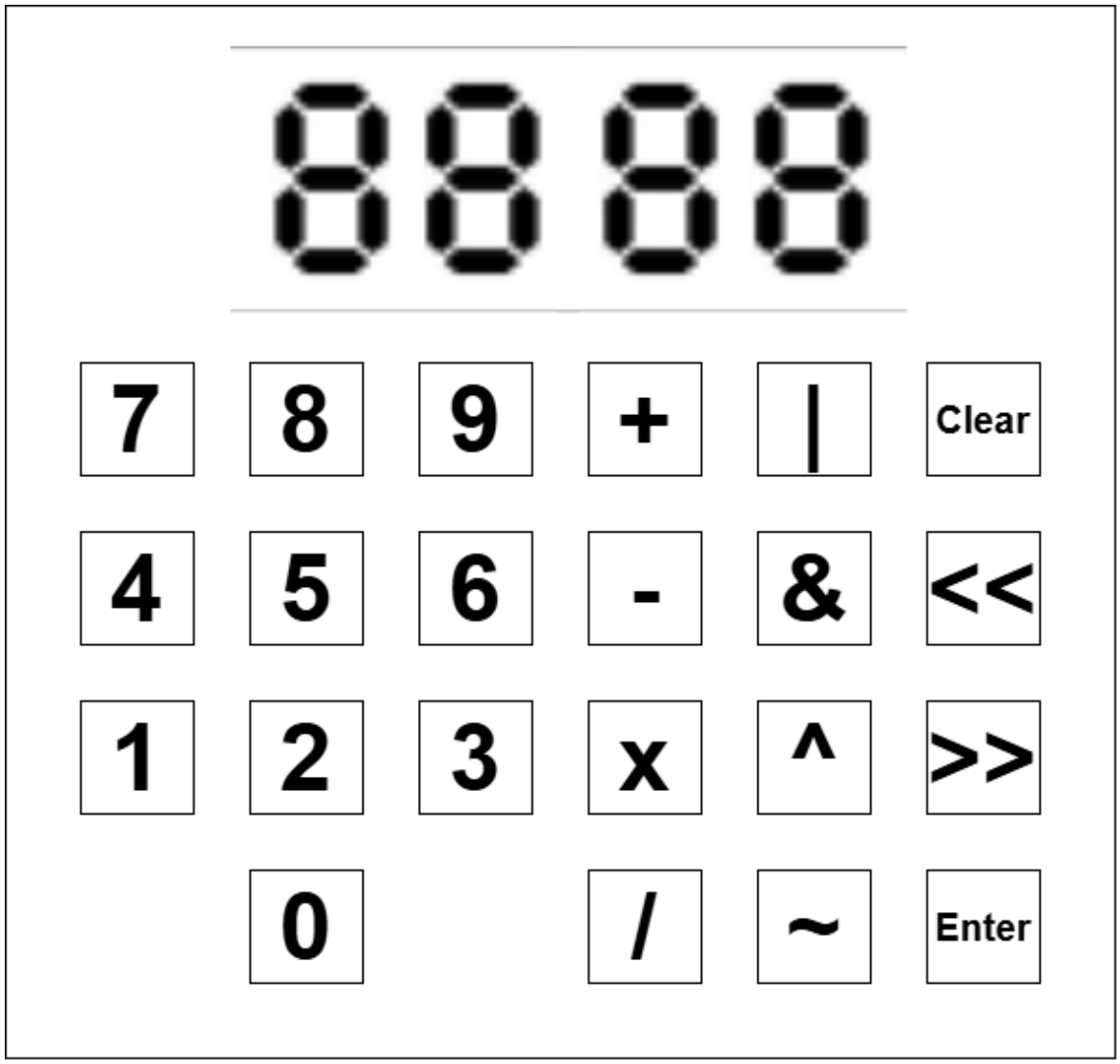
Output Conversion & Display:

- Converts computed binary results back into decimal digits for user display.
- Drives 7-segment displays to show both the input numbers and the results.

System Integration & Control:

- Coordinates the operation sequence among input, computation, and display modules.
- Provides clear and reset functions to start new calculations as needed.

1.2 Operating Procedures and Application
1.2.1 Examples of Application



1.2.2 Operating Procedures (External Operations)

1. A system clock, a system reset, a power line, and a ground line are supplied to the 8bit Integer Calculator.
2. The input devices including 2 hex keypad which are number buttons (for entering numbers) and system buttons (for Clear, Enter, and operation commands) as well as the output devices which is 7 segment displays.
3. To start the calculator, first power up the system by turning on the power switch.
4. If any previous input exists, press the Clear button to reset the calculator.
5. Key in the desired numeric values using the hex keypad and press the Enter button. The system captures these keycodes and converts the multi-digit number into an 8bit binary value.
6. Select the desired computation (addition, subtraction, multiplication, division, or bitwise/shift operations) by pressing the corresponding operation button and press the Enter button.

7. Key in the second desired numeric values using the hex keypad. Press the Enter button to initiate the computation. The operation will process the entered operands.
8. The computed binary result is then converted back into decimal digits and displayed on the 7segment display.
9. If a new calculation is desired, reset the system (using the Clear button) and repeat the input and operation selection process.
10. To continue with the result value, Press the enter Button and select the desired computation again.
11. To reinitialize the system at any point, press the Clear button; when finished, power off the system by turning off the power switch.

1.3 Naming Convention

<direction>_<level>_<module name>_<function><port size>

direction

Name	**description**
i	input
o	output

l**evel**

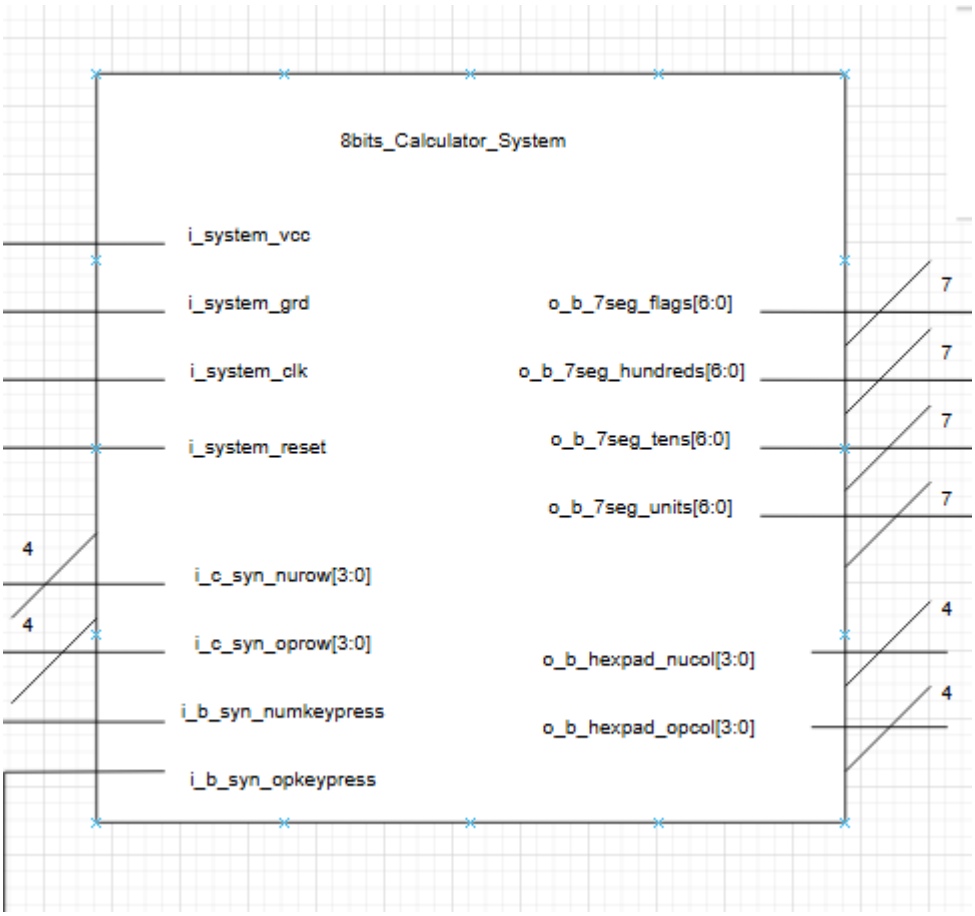
Name	description
b	block
c	chip
system	system

Module name

Name	description
SYN	synchronizer
hexpad	Hexadecimal keypad
CON	Data Flow Control and input Converter
ALU	arithmetic logic unit
BS	Barrier shifter
IOD	Input/output display

1.4 Calculator Chip Interface and I/O Pin Description

1.4.1 Calculator Interface



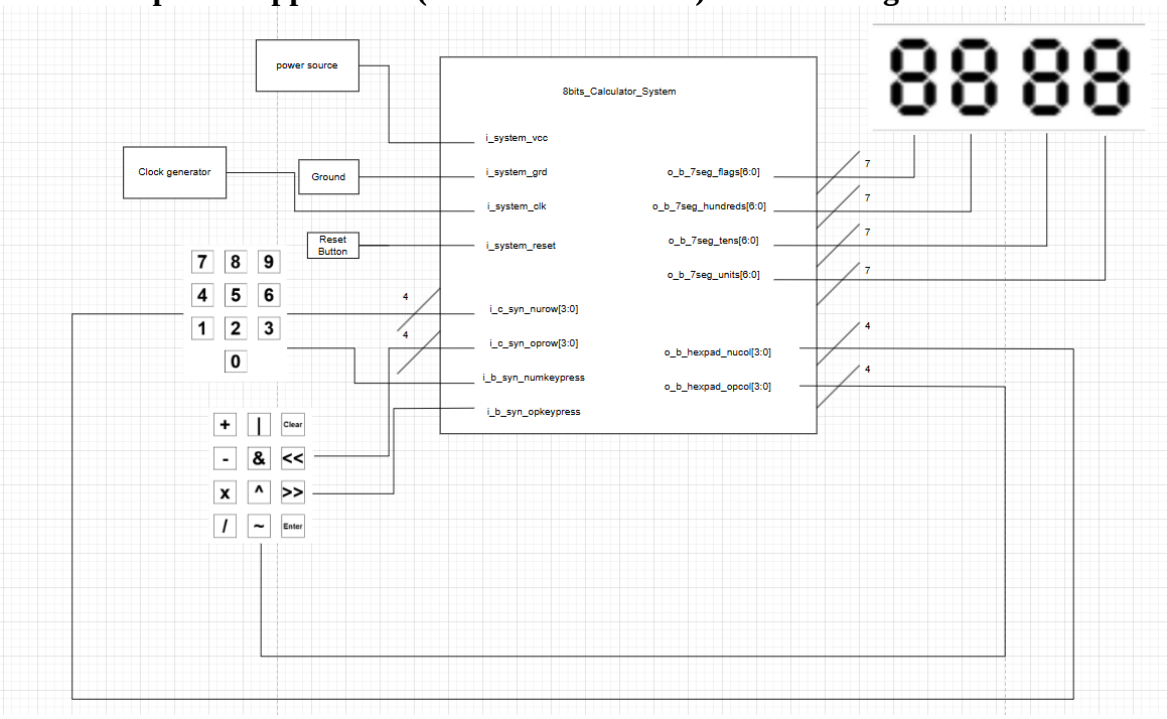
1.4.2 I/O Pin Description

Input Pins

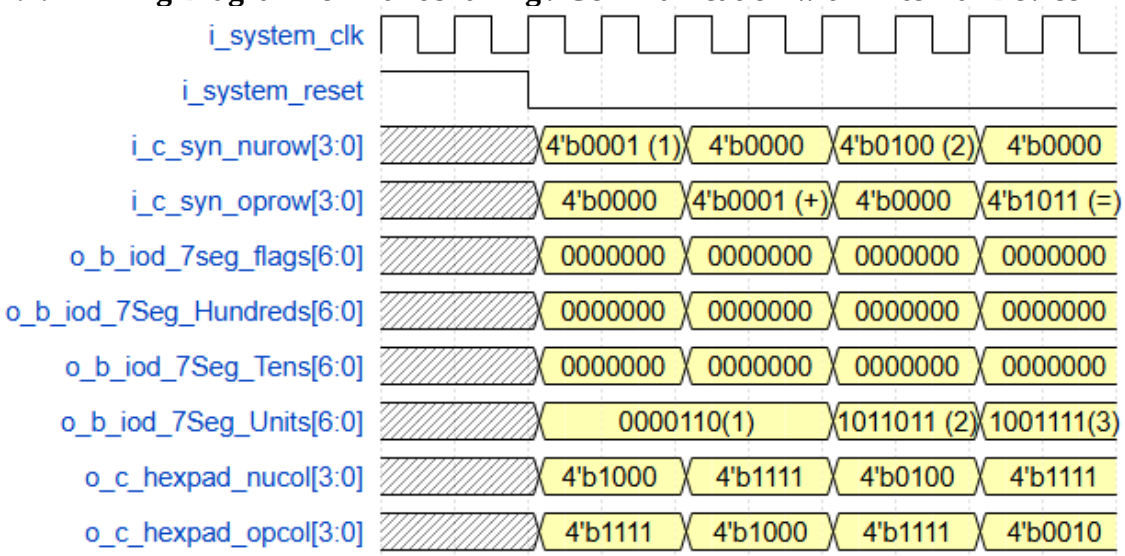
Pin Name	Pin Class	Pin direction	Source to direction	Pin Function
i_system_reset	Global	input	Reset button to calculator	System reset signal to initialize the keypad interface.
i_system_clk	Global	input	System clock to calculator	Synchronize system with clock system
i_c_syn_nurow[3:0]	data	input	Keypad to calculator	Determine number's row number
i_c_syn_oprow[3:0]	data	input	Keypad to Calculator	Determine opcode's row number
i_b_syn_numkeypress	data	input	Keypad to Calculator	Record clock cycle of

				opcode’s row number
i_b_syn_opkeypress	data	input	Keypad to Calculator	Record clock cycle of opcode’s row number
o_b_iod_7seg_flags [6:0]	data	output	Calculator to 7-segment display	Display the flags digit of 7-segment
o_b_iod_7Seg_hundreds [6:0]	data	output	Calculator to 7-segment display	Display the hundreds digit of 7-segment
o_b_iod_7Seg_tens [6:0]	data	output	Calculator to 7-segment display	Display the tens digit of 7-segment
o_b_iod_7Seg_units [6:0]	data	output	Calculator to 7-segment display	Display the units digit of 7-segment
o_b_hexpad_nucol [3:0]	data	output	Calculator to keypad	Scan number’s row number
o_b_hexpad_nucol [3:0]	data	output	Calculator to keypad	Scan opcode’s row number

1.4.3 Examples of Application (Include I/O Devices) in Block Diagram

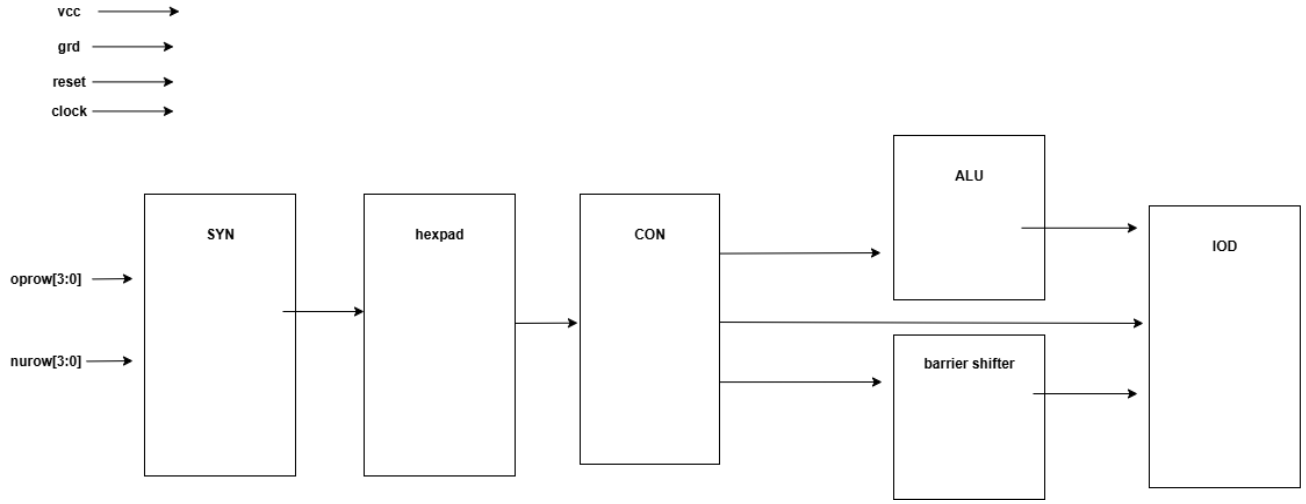


1.4.4 Timing Diagram for Handshaking / Communication with External Device

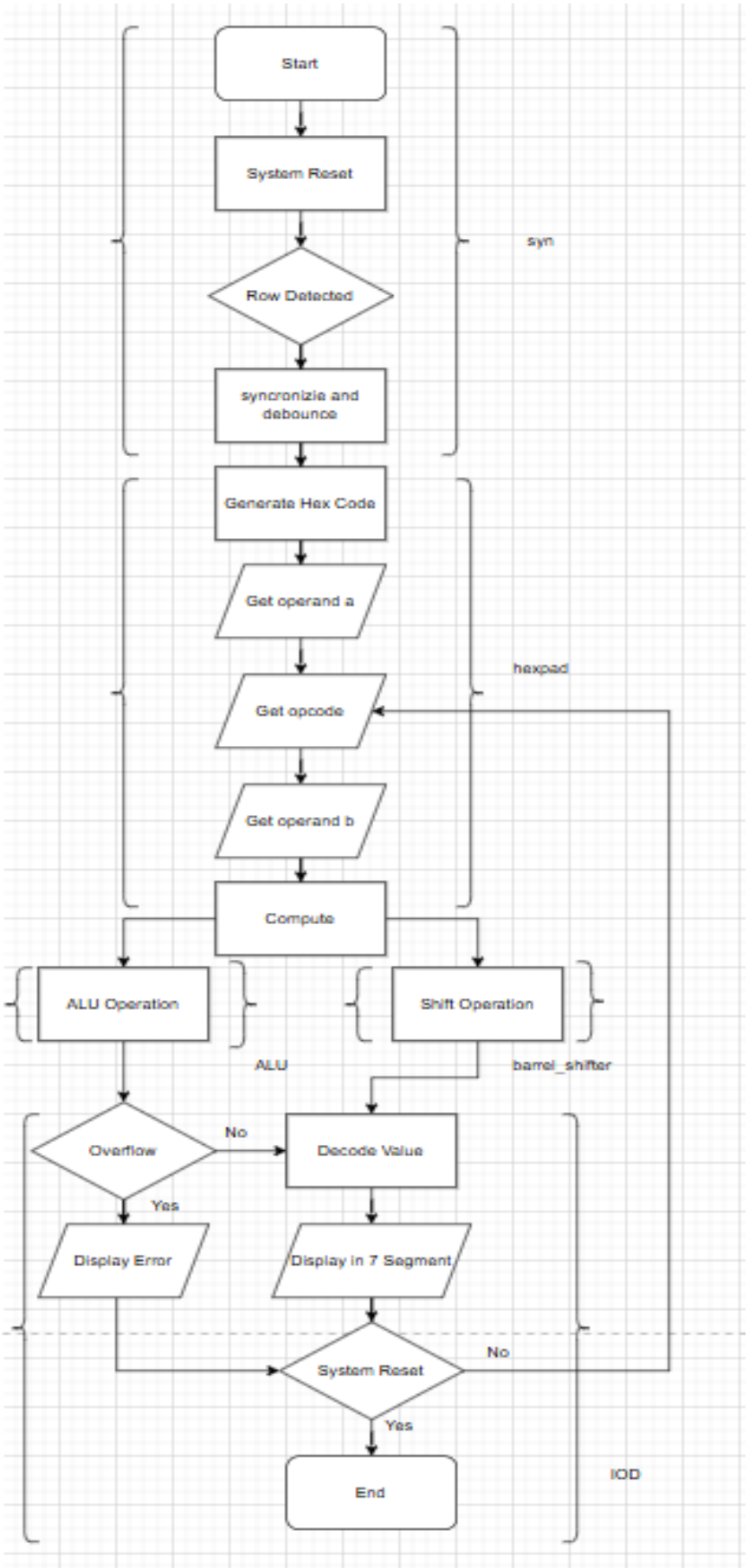


1.5 Internal Operation

1.5.1 Block Diagram with Information Flow



1.5.2 Flowchart



1.6 System Register

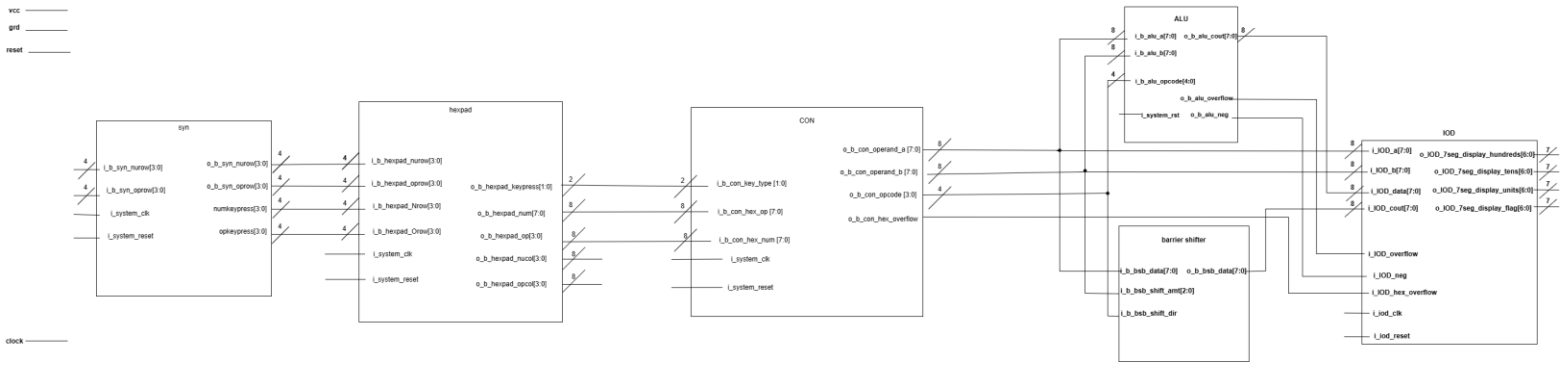
Register	Width	Reset Value	Function
bcd_input_reg	12-bit	12'b000000000000	Temporarily stores BCD digits entered via hexpad. bcd_input_reg[11:8]: 4’b0000 Hundreds place (Digit 2) bcd_input_reg[7:4]: 4’b0000 Tens place (Digit 1) bcd_input_reg[3:0]: 4’b0000 Units place (Digit 0)
operand_reg_a	8-bit	8'b00000000	Stores the first 8-bit binary operand converted from bcd_input_reg.
operand_reg_b	8-bit	8'b00000000	Stores the second 8-bit binary operand converted from bcd_input_reg.
opcode_reg	4-bit	4'b0000	Stores the operation code
bcd_display_reg	12-bit	12'b000000000000	Holds the BCD representation of result_reg for seven-segment display bcd_display_reg[11:8]: 4’b0000 Hundreds place (Digit 2) bcd_ display _reg[7:4]: 4’b0000 Tens place (Digit 1) bcd_ display _reg[3:0]: 4’b0000 Units place (Digit 0)
overflow_flag	1-bit	1'b0	Indicates overflow input when operand >255

2. Micro-Architecture Specification

2.1 Design Hierarchy

Chip Partitioning (Top Level) at System Level	Block and Sub-block Partitioning at MicroArchitecture Level
	Block Partitioning
Calculator	**b_syn (for synchronization)**
	b_hexpad (for get value and sent to next block)
	b_con (for convert the value form bcd to binary and sent to next block)
	b_ALU (for athematic computation)
	b_BS (for barrel shift computation)
	b_IOD (for display)

2.2 Block-level Functioning Partitioning



2.3 Full chip verilog model

```
module calculator;
// Clock and reset
logic clk;
logic reset;

// Inputs to syn module
logic [3:0] i_b_syn_nurow;
logic [3:0] i_b_syn_oprow;
logic i_b_syn_numkeypress;
logic i_b_syn_opkeypress;

// Outputs from syn module
logic [3:0] o_b_syn_nurow;
logic [3:0] o_b_syn_oprow;
logic o_b_syn_numkeypress;
logic o_b_syn_opkeypress;

// Outputs from hexpad module
logic [7:0] o_b_hexpad_num;
logic [3:0] o_b_hexpad_op;
logic [3:0] o_b_hexpad_nucol;
logic [3:0] o_b_hexpad_opCol;
logic [1:0] o_b_hexpad_keypress;

// Outputs from cub module
logic [3:0] o_b_con_opcode;
logic [7:0] o_b_con_a;
logic [7:0] o_b_con_b;
logic  o_b_con_hex_overflow;

//Outputs from ALU
   logic [7:0] o_alu_out;
   logic o_neg;
   logic o_overflow;

//Outputs from BS
   logic[7:0]   o_b_bs_data;

//Outputs from IOD

logic[6:0] o_b_iod_7seg_hundreds; logic[6:0] o_b_iod_7seg_tens; logic[6:0]
o_b_iod_7seg_units; logic[6:0] o_b_iod_7seg_flags;
// Instantiate syn module
syn u_syn (
   .i_system_clk(clk),
   .i_system_reset(reset),
   .i_b_syn_nurow(i_b_syn_nurow),
   .i_b_syn_oprow(i_b_syn_oprow),
   .i_b_syn_numkeypress(i_b_syn_numkeypress),
   .i_b_syn_opkeypress(i_b_syn_opkeypress),
```

```
    .o_b_syn_nurow(o_b_syn_nurow),
    .o_b_syn_oprow(o_b_syn_oprow),
    .o_b_syn_numkeypress(o_b_syn_numkeypress),
    .o_b_syn_opkeypress(o_b_syn_opkeypress)
);

// Instantiate hexpad module
hexpad u_hexpad (
    .o_b_hexpad_num(o_b_hexpad_num),
    .o_b_hexpad_op(o_b_hexpad_op),
    .o_b_hexpad_nucol(o_b_hexpad_nucol),
    .o_b_hexpad_opCol(o_b_hexpad_opCol),
    .o_b_hexpad_keypress(o_b_hexpad_keypress),
    .i_b_hexpad_nurow(o_b_syn_nurow),
    .i_b_hexpad_oprow(o_b_syn_oprow),
    .i_b_hexpad_Nrow(o_b_syn_numkeypress),
    .i_b_hexpad_Orow(o_b_syn_opkeypress),
    .i_system_clk(clk),
    .i_system_reset(reset)
);

// Instantiate con module
con u_con (
    .i_system_clock(clk),
    .i_system_reset(reset),
    .i_b_con_key_type(o_b_hexpad_keypress),
    .i_b_con_hex_num(o_b_hexpad_num),
    .i_b_con_hex_op(o_b_hexpad_op),
    .o_b_con_opcode(o_b_con_opcode),
    .o_b_con_a(o_b_con_a),
    .o_b_con_b(o_b_con_b),
    .o_b_con_hex_overflow(o_b_con_hex_overflow)
);
// Instantiate ALU module
ALU u_alu(
.i_operand_a(o_b_con_a),
.i_operand_b(o_b_con_b),
.i_opcode(o_b_con_opcode),
.i_system_rst(reset),
.o_alu_out(o_alu_out),
.o_neg(o_neg),
.o_overflow(o_overflow)
);

// Instantiate BS module
BS u_bs(
.i_b_bs_data(o_b_con_a),
.i_b_bs_shift_amt(o_b_con_b),
 .i_b_bs_shift_dir(o_b_con_opcode),
 .o_b_bs_data(o_b_bs_data)
);
// Instantiate IOD module
```

```
 iod u_iod(
.i_system_clk(clk),
.i_system_reset(reset),
.i_b_iod_overflow(o_overflow),
 .i_b_iod_neg(o_neg),
.i_b_iod_hex_overflow(o_b_con_hex_overflow),
 .i_b_iod_a(o_b_con_a),
.i_b_iod_b(o_b_con_b),
.i_b_iod_data(o_b_bs_data),
 .i_b_iod_cout(o_alu_out),
.i_b_iod_opcode(o_b_con_opcode),
 .o_b_iod_7seg_hundreds(o_b_iod_7seg_hundreds), .o_b_iod_7seg_tens(o_b_iod_7s
eg_tens),
.o_b_iod_7seg_units(o_b_iod_7seg_units),
.o_b_iod_7seg_flags(o_b_iod_7seg_flags)
);

endmodule
```

3. Architecture Specification: Verification Specification

3.1 Test Plan

No	Test Feature / Functionality	Test Vector Generator	Expected Outputs	Pass/ Fail

1	**Reset**	clk = 0;
		reset = 1;
		i_b_syn_nurow = 4'b0000;
		i_b_syn_oprow = 4'b0000;
		i_b_syn_numkeypress = 0;
		i_b_syn_opkeypress = 0;
		#20;
		reset = 0;
		#20;
			//Display '000'
			
			o_b_iod_7seg_hundreds[6:0]=0111111
			o_b_iod_7seg_tens[6:0]=0111111
			o_b_iod_7seg_units[6:0]= 0111111
			o_b_iod_7seg_flags[6:0]=0000000
				Pass

2	**Addition (63 + 36)**	// Press '6'
		i_b_syn_numkeypress = 1;
		#20;
		i_b_syn_nurow = 4'b0010;
		#20;
		i_b_syn_nurow = 4'b0000;
		i_b_syn_numkeypress = 0;
		#20;
		
		// Press '3'
		i_b_syn_numkeypress = 1;
		#20;
		i_b_syn_nurow = 4'b0100;
		#20;
		i_b_syn_nurow = 4'b0000;
		i_b_syn_numkeypress = 0;
		#20;
		
		// Press '+'
		i_b_syn_opkeypress = 1;
		#40;
		i_b_syn_oprow = 4'b0001;
		#20;
		i_b_syn_oprow = 4'b0000;
		i_b_syn_opkeypress = 0;
		#20;
		
		// Press '3'
		i_b_syn_numkeypress = 1;
		#20;
		i_b_syn_nurow = 4'b0100;
			//Display '099'
			
			o_b_iod_7seg_hundreds[6:0]=0111111
			o_b_iod_7seg_tens[6:0]=1101111
			o_b_iod_7seg_units[6:0]= 1101111
			o_b_iod_7seg_flags[6:0]=0000000
				Pass

		#20;
		i_b_syn_nurow = 4'b0000;
		i_b_syn_numkeypress = 0;
		#20;
		
		// Press '6'
		i_b_syn_numkeypress = 1;
		#20;
		i_b_syn_nurow = 4'b0010;
		#20;
		i_b_syn_nurow = 4'b0000;
		i_b_syn_numkeypress = 0;
		#20;
		
		//press ‘=’
		i_b_syn_opkeypress = 1;
		#20;
		i_b_syn_oprow = 4'b1000;
		#20;
		i_b_syn_oprow = 4'b0000;
		i_b_syn_opkeypress = 0;
		#20;		

3	**Barrel Shift (6<<3)**	// Press '6'
		i_b_syn_numkeypress = 1;
		#20;
		i_b_syn_nurow = 4'b0010;
		#20;
		i_b_syn_nurow = 4'b0000;
		i_b_syn_numkeypress = 0;
		#20;
		
		// Press '<<'
		i_b_syn_opkeypress = 1;
		#20;
		i_b_syn_oprow = 4'b0010;
		#20;
		i_b_syn_oprow = 4'b0000;
		i_b_syn_opkeypress = 0;
		#20;
		
		// Press '3'
		i_b_syn_numkeypress = 1;
		#20;
		i_b_syn_nurow = 4'b0100;
		#20;
		i_b_syn_nurow = 4'b0000;
		i_b_syn_numkeypress = 0;
		#20;
			//Display ‘048’
			
			o_b_iod_7seg_hundreds[6:0]=0111111
			o_b_iod_7seg_tens[6:0]=1100110
			o_b_iod_7seg_units[6:0]= 1111111
			o_b_iod_7seg_flags[6:0]=0000000
				pass

		//press ‘=’
		i_b_syn_opkeypress = 1;
		#20;
		i_b_syn_oprow = 4'b1000;
		#20;
		i_b_syn_oprow = 4'b0000;
		i_b_syn_opkeypress = 0;
		#20;		

4	**Multiplication (105 * 3) Overflow**	// Press '1'
		i_b_syn_numkeypress = 1;
		#40;
		i_b_syn_nurow = 4'b0100;
		#20;
		i_b_syn_nurow = 4'b0000;
		i_b_syn_numkeypress = 0;
		#20;
		
		//Press ‘0’
		i_b_syn_numkeypress = 1;
		#30;
		i_b_syn_nurow = 4'b1000;
		#20;
		i_b_syn_nurow = 4'b0000;
		i_b_syn_numkeypress = 0;
		#20;
		// Press '5'
		i_b_syn_numkeypress = 1;
		#30;
		i_b_syn_nurow = 4'b0010;
		#20;
		i_b_syn_nurow = 4'b0000;
		i_b_syn_numkeypress = 0;
		#20;
		
		// Press '*'
		i_b_syn_opkeypress = 1;
		#40;
		i_b_syn_oprow = 4'b0100;
		#20;
		i_b_syn_oprow = 4'b0000;
		i_b_syn_opkeypress = 0;
		#20;
		
		// Press '3'
		i_b_syn_numkeypress = 1;
		#20;
		i_b_syn_nurow = 4'b0100;
			Display ’E’ at the Last Digit of 7 Segment
			
			o_b_iod_7seg_flags[6:0]=1111001
				pass

		#20;
		i_b_syn_nurow = 4'b0000;
		i_b_syn_numkeypress = 0;
		#20;
		// Press '='
		i_b_syn_opkeypress = 1;
		#20;
		i_b_syn_oprow = 4'b1000;
		#20;
		i_b_syn_oprow = 4'b0000;
		i_b_syn_opkeypress = 0;
		#20;		

5	**Substraction (15-30) Negative Flag**	// Press '1'
		i_b_syn_numkeypress = 1;
		#40;
		i_b_syn_nurow = 4'b0100;
		#20;
		i_b_syn_nurow = 4'b0000;
		i_b_syn_numkeypress = 0;
		#20;
		
		// Press '5'
		i_b_syn_numkeypress = 1;
		#30;
		i_b_syn_nurow = 4'b0010;
		#20;
		i_b_syn_nurow = 4'b0000;
		i_b_syn_numkeypress = 0;
		#20;
		
		// Press '-'
		i_b_syn_opkeypress = 1;
		#40;
		i_b_syn_oprow = 4'b0010;
		#20;
		i_b_syn_oprow = 4'b0000;
		i_b_syn_opkeypress = 0;
		#20;
		
		// Press '3'
		i_b_syn_numkeypress = 1;
		#20;
		i_b_syn_nurow = 4'b0100;
		#20;
		i_b_syn_nurow = 4'b0000;
		i_b_syn_numkeypress = 0;
		#20;
		
		//Press ‘0’
			Display ’-’ at the Last Digit of 7 Segment
			
			o_b_iod_7seg_flags[6:0]=1000000
				pass

		i_b_syn_numkeypress = 1;
		#30;
		i_b_syn_nurow = 4'b1000;
		#20;
		i_b_syn_nurow = 4'b0000;
		i_b_syn_numkeypress = 0;
		#20;
		
		// Press '='
		i_b_syn_opkeypress = 1;
		#20;
		i_b_syn_oprow = 4'b1000;
		#20;
		i_b_syn_oprow = 4'b0000;
		i_b_syn_opkeypress = 0;
		#20;		

3.2 Testbench

```
`timescale 1ns / 1ps

module calculator;

// Clock and reset
logic clk;
logic reset;

// Inputs to syn module
logic [3:0] i_b_syn_nurow;
logic [3:0] i_b_syn_oprow;
logic i_b_syn_numkeypress;
logic i_b_syn_opkeypress;

// Outputs from syn module
logic [3:0] o_b_syn_nurow;
logic [3:0] o_b_syn_oprow;
logic o_b_syn_numkeypress;
logic o_b_syn_opkeypress;

// Outputs from hexpad module
logic [7:0] o_b_hexpad_num;
logic [3:0] o_b_hexpad_op;
logic [3:0] o_b_hexpad_nucol;
logic [3:0] o_b_hexpad_opCol;
logic [1:0] o_b_hexpad_keypress;

// Outputs from con module
logic [3:0] o_b_con_opcode;
logic [7:0] o_b_con_a;
logic [7:0] o_b_con_b;
logic  o_b_con_hex_overflow;

//Outputs from ALU
    logic [7:0] o_alu_out;
    logic o_neg;
    logic o_overflow;

//Outputs from BS
    logic[7:0]   o_b_bs_data;

//Outputs from IOD


logic[6:0] o_b_iod_7seg_hundreds;

 logic[6:0] o_b_iod_7seg_tens;

 logic[6:0] o_b_iod_7seg_units;

logic[6:0] o_b_iod_7seg_flags;
```

```
// Instantiate syn module
syn u_syn (
   .i_system_clk(clk),
   .i_system_reset(reset),
   .i_b_syn_nurow(i_b_syn_nurow),
   .i_b_syn_oprow(i_b_syn_oprow),
   .i_b_syn_numkeypress(i_b_syn_numkeypress),
   .i_b_syn_opkeypress(i_b_syn_opkeypress),
   .o_b_syn_nurow(o_b_syn_nurow),
   .o_b_syn_oprow(o_b_syn_oprow),
   .o_b_syn_numkeypress(o_b_syn_numkeypress),
   .o_b_syn_opkeypress(o_b_syn_opkeypress)
);

// Instantiate hexpad module
hexpad u_hexpad (
   .o_b_hexpad_num(o_b_hexpad_num),
   .o_b_hexpad_op(o_b_hexpad_op),
   .o_b_hexpad_nucol(o_b_hexpad_nucol),
   .o_b_hexpad_opCol(o_b_hexpad_opCol),
   .o_b_hexpad_keypress(o_b_hexpad_keypress),
   .i_b_hexpad_nurow(o_b_syn_nurow),
   .i_b_hexpad_oprow(o_b_syn_oprow),
   .i_b_hexpad_Nrow(o_b_syn_numkeypress),
   .i_b_hexpad_Orow(o_b_syn_opkeypress),
   .i_system_clk(clk),
   .i_system_reset(reset)
);

// Instantiate con module
con u_con (
   .i_system_clock(clk),
   .i_system_reset(reset),
   .i_b_con_key_type(o_b_hexpad_keypress),
   .i_b_con_hex_num(o_b_hexpad_num),
   .i_b_con_hex_op(o_b_hexpad_op),
   .o_b_con_opcode(o_b_con_opcode),
   .o_b_con_a(o_b_con_a),
   .o_b_con_b(o_b_con_b),
   .o_b_con_hex_overflow(o_b_con_hex_overflow)
);
// Instantiate ALU module
ALU u_alu(
.i_operand_a(o_b_con_a),
.i_operand_b(o_b_con_b),
.i_opcode(o_b_con_opcode),
.i_system_rst(reset),
.o_alu_out(o_alu_out),
.o_neg(o_neg),
.o_overflow(o_overflow)
);


// Instantiate BS module

BS u_bs(
```

```
 .i_b_bs_data(o_b_con_a),

 .i_b_bs_shift_amt(o_b_con_b),

 .i_b_bs_shift_dir(o_b_con_opcode),

 .o_b_bs_data(o_b_bs_data)

);

// Instantiate IOD module

iod u_iod(

.i_system_clk(clk),

.i_system_reset(reset),

 .i_b_iod_overflow(o_overflow),

 .i_b_iod_neg(o_neg),

.i_b_iod_hex_overflow(o_b_con_hex_overflow),

.i_b_iod_a(o_b_con_a),

 .i_b_iod_b(o_b_con_b),

 .i_b_iod_data(o_b_bs_data),

.i_b_iod_cout(o_alu_out),

.i_b_iod_opcode(o_b_con_opcode),

 .o_b_iod_7seg_hundreds(o_b_iod_7seg_hundreds),

 .o_b_iod_7seg_tens(o_b_iod_7seg_tens), .o_b_iod_7seg_units(o_b_iod_7seg_units)
, .o_b_iod_7seg_flags(o_b_iod_7seg_flags)

);

// Clock generation

 always begin #5 clk = ~clk;

end

// Test 1 sequence
initial begin
    clk = 0;
    reset = 1;
    i_b_syn_nurow = 4'b0000;
    i_b_syn_oprow = 4'b0000;
    i_b_syn_numkeypress = 0;
    i_b_syn_opkeypress = 0;
    #20;
    reset = 0;
```

```
    #20;

//Test 2 (63+36)=99
 // Simulate number key press
 i_b_syn_numkeypress = 1;
 // activate number scanning
#20;
 i_b_syn_nurow = 4'b0010;
 // simulate key press (e.g., '6')
#20; i_b_syn_nurow = 4'b0000;
i_b_syn_numkeypress = 0;
 // release key
 #20;
 // Simulate number key press
 i_b_syn_numkeypress = 1;
 // activate number scanning
#20; i_b_syn_nurow = 4'b0100;
// simulate key press (e.g., '3')
#20;
 i_b_syn_nurow = 4'b0000;
 i_b_syn_numkeypress = 0;
 // release key
#20;

   // Simulate operator '+' press
    i_b_syn_opkeypress = 1;
    #40;
    i_b_syn_oprow = 4'b0001;  // simulate '+' press
    #20;
    i_b_syn_oprow = 4'b0000;
    i_b_syn_opkeypress = 0;
    #20;

    // Simulate number key press
    i_b_syn_numkeypress = 1;  // activate number scanning
    #20;
```

```
    i_b_syn_nurow = 4'b0100;  // simulate key press (e.g., '3')
    #20;
    i_b_syn_nurow = 4'b0000;
    i_b_syn_numkeypress = 0;    // release key
    #20;
    // Simulate number key press
    i_b_syn_numkeypress = 1;  // activate number scanning
    #20;
    i_b_syn_nurow = 4'b0010;  // simulate key press (e.g., '6')
    #20;
    i_b_syn_nurow = 4'b0000;
    i_b_syn_numkeypress = 0;    // release key
    #20;


    // Simulate another operator press (e.g., '=')
    i_b_syn_opkeypress = 1;
    #20;
    i_b_syn_oprow = 4'b1000;  // simulate '=' press
    #20;
    i_b_syn_oprow = 4'b0000;
    i_b_syn_opkeypress = 0;
    #20;


//test 3 (6<<3=48)

// Simulate number key press

i_b_syn_numkeypress = 1;

 // activate number scanning

 #20;

i_b_syn_nurow = 4'b0010;

// simulate key press (e.g., '6')

#20;

i_b_syn_nurow = 4'b0000;

 i_b_syn_numkeypress = 0;

// release key

#20;

   // Simulate  operator press (e.g., '<<')
    i_b_syn_opkeypress = 1;
    #20;
    i_b_syn_oprow = 4'b0010;  // simulate '<<' press
    #20;
    i_b_syn_oprow = 4'b0000;
    i_b_syn_opkeypress = 0;
    #20;
```

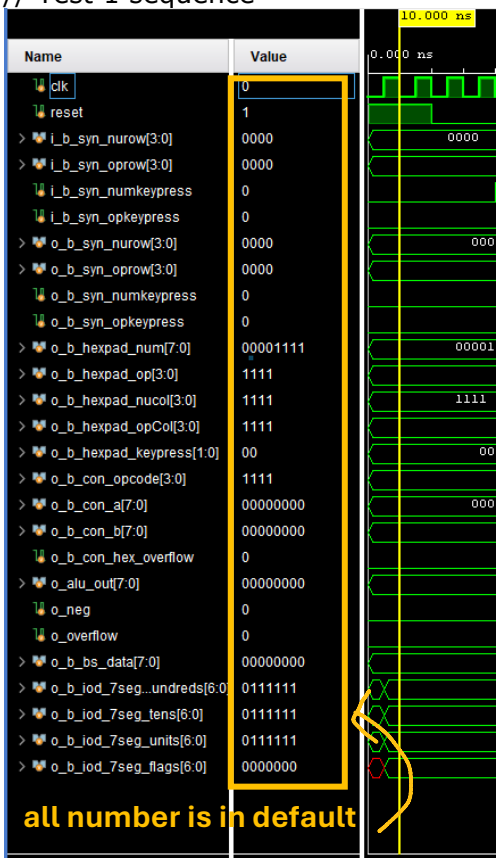
```
    // Simulate number key press
    i_b_syn_numkeypress = 1;  // activate number scanning
    #20;
    i_b_syn_nurow = 4'b0100;  // simulate key press (e.g., '3')
    #20;
    i_b_syn_nurow = 4'b0000;
    i_b_syn_numkeypress = 0;    // release key
    #20;

   // Simulate another operator press (e.g., '=')
    i_b_syn_opkeypress = 1;
    #20;
    i_b_syn_oprow = 4'b1000;  // simulate '=' press
    #20;
    i_b_syn_oprow = 4'b0000;
    i_b_syn_opkeypress = 0;
    #20;



//test 4 (105*3=315,overflow)

// Simulate number key press

i_b_syn_numkeypress = 1; // activate number scanning

#40;

 i_b_syn_nurow = 4'b0100;

// simulate key press (e.g., '1')

#20;

i_b_syn_nurow = 4'b0000;

 i_b_syn_numkeypress = 0; // release key #20;

   // Simulate number key press
    i_b_syn_numkeypress = 1;  // activate number scanning
    #30;
    i_b_syn_nurow = 4'b1000;  // simulate key press (e.g., '0')
    #20;
    i_b_syn_nurow = 4'b0000;
    i_b_syn_numkeypress = 0;    // release key
    #20;
    // Simulate number key press
    i_b_syn_numkeypress = 1;  // activate number scanning
    #30;
    i_b_syn_nurow = 4'b0010;  // simulate key press (e.g., '5')
    #20;
    i_b_syn_nurow = 4'b0000;
    i_b_syn_numkeypress = 0;    // release key
    #20;
```

```
    // Simulate operator press (e.g., '*')
    i_b_syn_opkeypress = 1;
    #40;
    i_b_syn_oprow = 4'b0100;  // simulate '*' press
    #20;
    i_b_syn_oprow = 4'b0000;
    i_b_syn_opkeypress = 0;
    #20;

    // Simulate number key press
    i_b_syn_numkeypress = 1;  // activate number scanning
    #20;
    i_b_syn_nurow = 4'b0100;  // simulate key press (e.g., '3')
    #20;
    i_b_syn_nurow = 4'b0000;
    i_b_syn_numkeypress = 0;    // release key
    #20;

   // Simulate another operator press (e.g., '=')
    i_b_syn_opkeypress = 1;
    #20;
    i_b_syn_oprow = 4'b1000;  // simulate '=' press
    #20;
    i_b_syn_oprow = 4'b0000;
    i_b_syn_opkeypress = 0;
    #20;
    // End simulation
    #100;



//test 5 (15-30=-15,negative flag) // Simulate number key press
i_b_syn_numkeypress = 1; // activate number scanning

#40;

 i_b_syn_nurow = 4'b0100; // simulate key press (e.g., '1')

#20;

 i_b_syn_nurow = 4'b0000;

i_b_syn_numkeypress = 0; // release key

#20;

   // Simulate number key press
    i_b_syn_numkeypress = 1;  // activate number scanning
    #30;
    i_b_syn_nurow = 4'b0010;  // simulate key press (e.g., '5')
    #20;
    i_b_syn_nurow = 4'b0000;
    i_b_syn_numkeypress = 0;    // release key
    #20;

    // Simulate another operator press (e.g., '-')
```

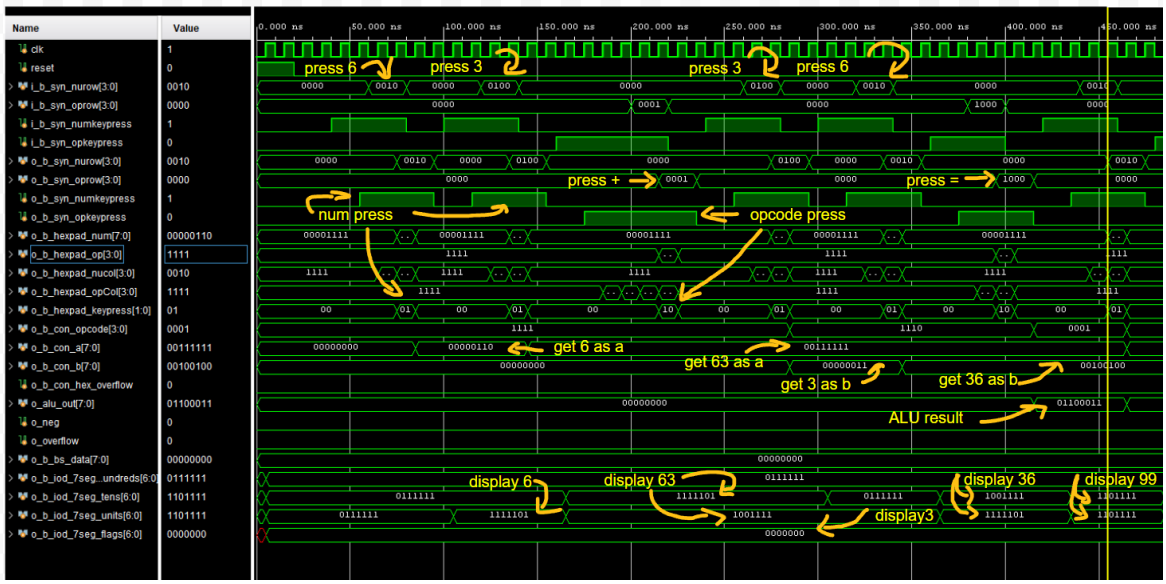
```
    i_b_syn_opkeypress = 1;
    #40;
    i_b_syn_oprow = 4'b0010;  // simulate '*' press
    #20;
    i_b_syn_oprow = 4'b0000;
    i_b_syn_opkeypress = 0;
    #20;

    // Simulate number key press
    i_b_syn_numkeypress = 1;  // activate number scanning
    #20;
    i_b_syn_nurow = 4'b0100;  // simulate key press (e.g., '3')
    #20;
    i_b_syn_nurow = 4'b0000;
    i_b_syn_numkeypress = 0;    // release key
    #20;

                // Simulate number key press
    i_b_syn_numkeypress = 1;  // activate number scanning
    #30;
    i_b_syn_nurow = 4'b1000;  // simulate key press (e.g., '0')
    #20;
    i_b_syn_nurow = 4'b0000;
    i_b_syn_numkeypress = 0;    // release key
    #20;

   // Simulate another operator press (e.g., '=')
    i_b_syn_opkeypress = 1;
    #20;
    i_b_syn_oprow = 4'b1000;  // simulate '=' press
    #20;
    i_b_syn_oprow = 4'b0000;
    i_b_syn_opkeypress = 0;
    #20;
    // End simulation
    #100;
    $finish;
end


endmodule
```

3.3 Test Simulation

// Test 1 sequence

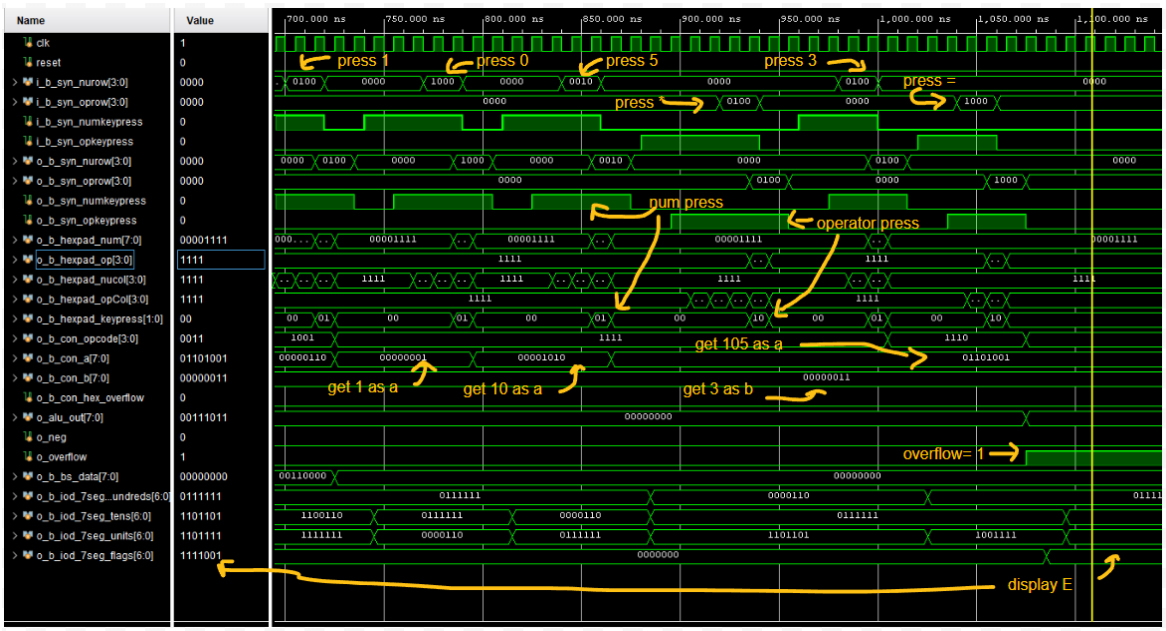


//Test 2 (63+36)=99

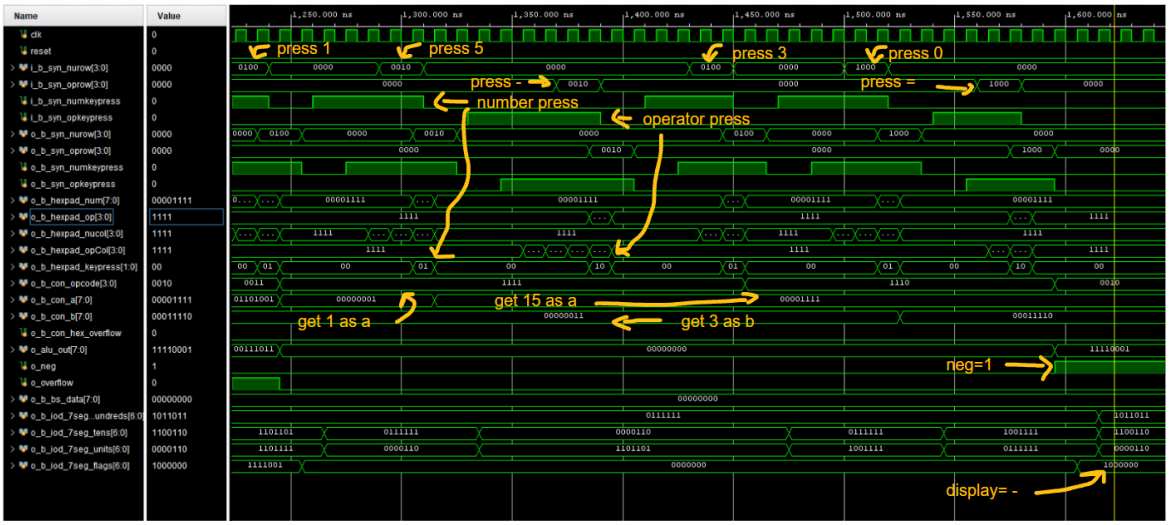


//test 3 (6<<3=48)

//test 4 (105*3=315,overflow)



//test 5 (15-30=-15,negative flag) // Simulate number key press


press 1
press 5
press 3
press 0
press -
press =
number press
operator press
get 1 as a
get 15 as a
get 3 as b
neg=1
display= -

4 Micro-Architecture Specification (Block level)

4.1 Synchronizer

4.1.1 Functionality

- The **SYN Module** is designed to synchronize asynchronous row signals from a keypad matrix to the system clock domain. The module uses multiple flip-flops to sample and stabilize the asynchronous input signals. It prevents metastability by ensuring that only stable signals are passed to the system, providing reliable data synchronization for key scanning operations.
- The**i_b_syn_nurow[3:0]** , **i_b_syn_oprow[3:0] ,i_b_syn_numkeypress and i_b_syn_opkeypress** signals are used to input the row signals of the keypad matrix during the scanning process. These signals are active when scanning different rows of the matrix.
- The **i_system_reset** signal initializes the synchronization process, ensuring that all internal states are cleared before synchronization begins. This helps prevent erroneous data during the initial stage of synchronization.
- The **i_b_syn_nurow[3:0]** , **i_b_syn_oprow[3:0] ,i_b_syn_numkeypress and i_b_syn_opkeypress** are the synchronized row signals that represent the stabilized rows of the keypad matrix, ensuring that the matrix rows are correctly identified and processed by the system clock.
- The **i_system_clk** signal is used as the system clock to synchronize the asynchronous input signals. All synchronization actions are aligned to the rising edge of this clock signal.

4.1.2 Features

Asynchronous Signal Synchronization:

The **SYN Module** synchronizes asynchronous row signals from the keypad matrix (**o_b_syn_nurow[3:0]**, **o_b_syn_oprow[3:0]**) to the system clock (**i_system_clk**), ensuring that all key scanning operations are synchronized.

Keypad Scanning Support:

The module supports keypad scanning by taking input row signals (**o_b_syn_nurow[3:0]**, **o_b_syn_oprow[3:0], i_b_syn_numkeypress, i_b_syn_opkeypress**) from the keypad matrix. It processes these signals and outputs synchronized row signals (**o_b_syn_nurow[3:0]**, **o_b_syn_oprow[3:0], i_b_syn_numkeypress, i_b_syn_opkeypress**).

Metastability Prevention:

The module prevents metastability by synchronizing the incoming asynchronous signals and ensuring that only stable signals are passed to the output. The **i_reset** signal clears the synchronization process if needed to maintain reliability.

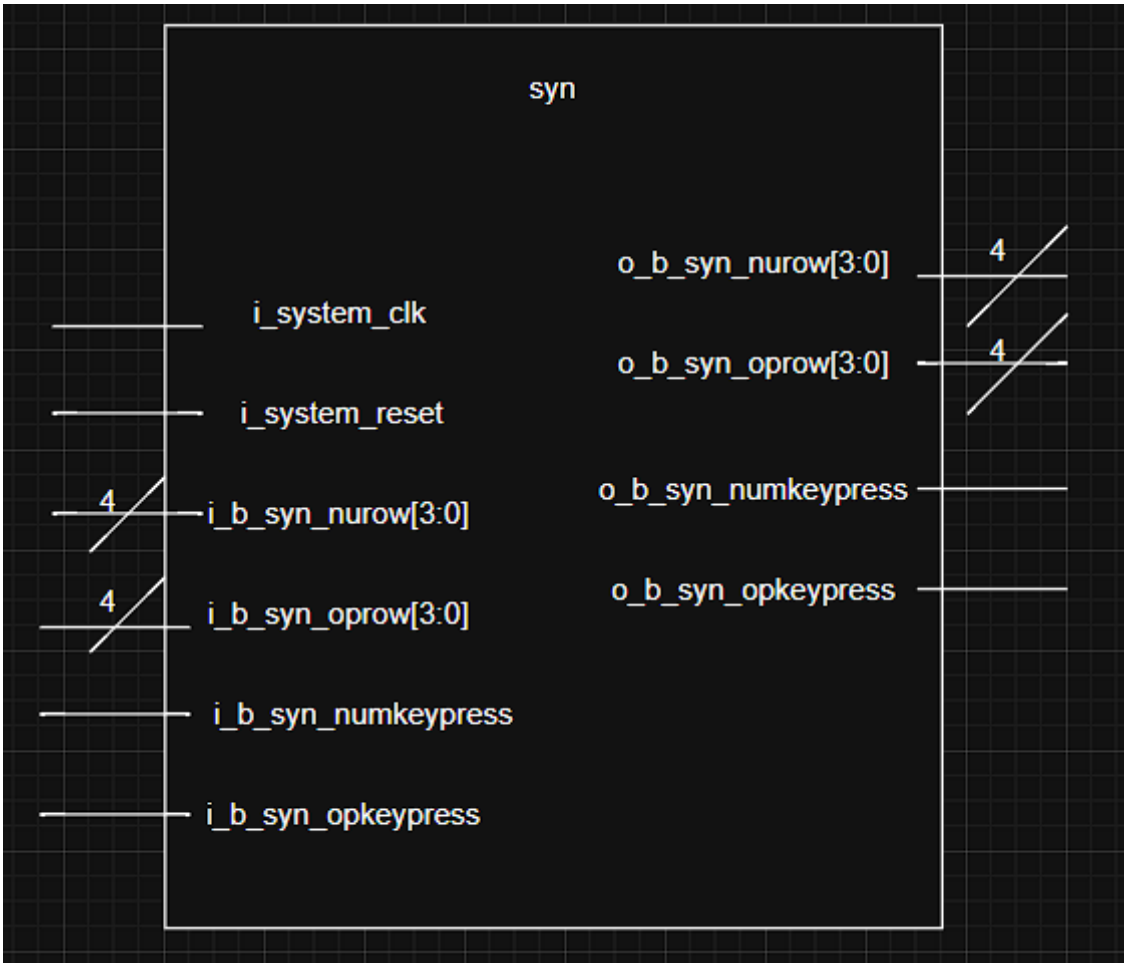
System Clock Synchronization:

The **i_system_clk** system clock ensures that the synchronization process occurs at the correct time, with all asynchronous inputs being synchronized to this clock domain.

Initialization with Reset:

The **i_system_reset** signal ensures that the synchronization process is initialized correctly, clearing any previous states and avoiding spurious data.

4.1.3 Block Interface



4.1.4 I/O Pin Description

Pin Name	**Pin Class**	**Pin Direction**	**Source to direction**	**Pin Function**
i_system_clock	Control	Input	System Clock to syn	Clock signal to synchronize input signals
i_system_reset	Control	Input	Reser b button to SYN	Asynchronous reset for internal registers
i_b_syn_nurow [3:0]	Data	Input	Keypad (number row) to SYN	Raw row data from numeric keypad
i_b_syn_oprow [3:0]	Data	Input	Keypad (operator row) to SYN	Raw row data from operator keypad
i_b_syn_numkeypress	Status/ Control	Input	hexpad module to SYN	Signal indicating numeric key was pressed

i_b_syn_opkeypress	Status/ Control	Input	hexpad module to SYN	Signal indicating opcode key was pressed
o_b_syn_nurow [3:0]	Data	Output	SYN to hexpad	Synchronized numeric keypad row output
o_b_syn_oprow [3:0]	Data	Output	SYN to hexpad	Synchronized operator keypad row output
o_b_syn_numkeypress	Status/ Control	Output	SYN to cub or FSM	Synchronized numeric key press status
o_b_syn_opkeypress	Status/ Control	Output	SYN → cub or FSM	Synchronized operator key press status

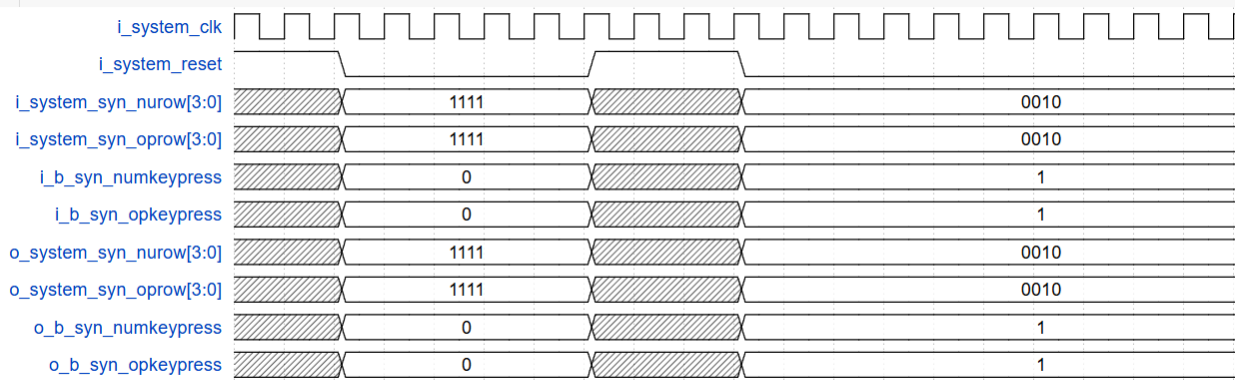
4.1.5 Internal operational

Clock signal (ip_sys_clk)	Input	Output	Synchronizer Function Description
0 → 1 (↑ rising edge)	0	0	Latches active-low key input as ‘pressed’ without uncertain state
0 → 1 (↑ rising edge)	1	1	Latches active-low key input as ‘not pressed’ without issues
0 → 1 (↑ rising edge)	X	X	Maintains undefined input if unstable
0 → 1 (↑ rising edge)	Z	Z	Handles high-impedance state as-is
1 → 0 (↓ falling edge)	Qn	Qn-1	No operation; output remains latched on rising edge only

Clock Signal (i_clk)	Input stable for 8 Cycle	Output (o_b_syn_*)
0 → 1 (↑ rising edge)	Yes	Updated with new value
0 → 1 (↑ rising edge)	No	Previous value held
1 → 0 (↓ falling edge)	N/A	No change

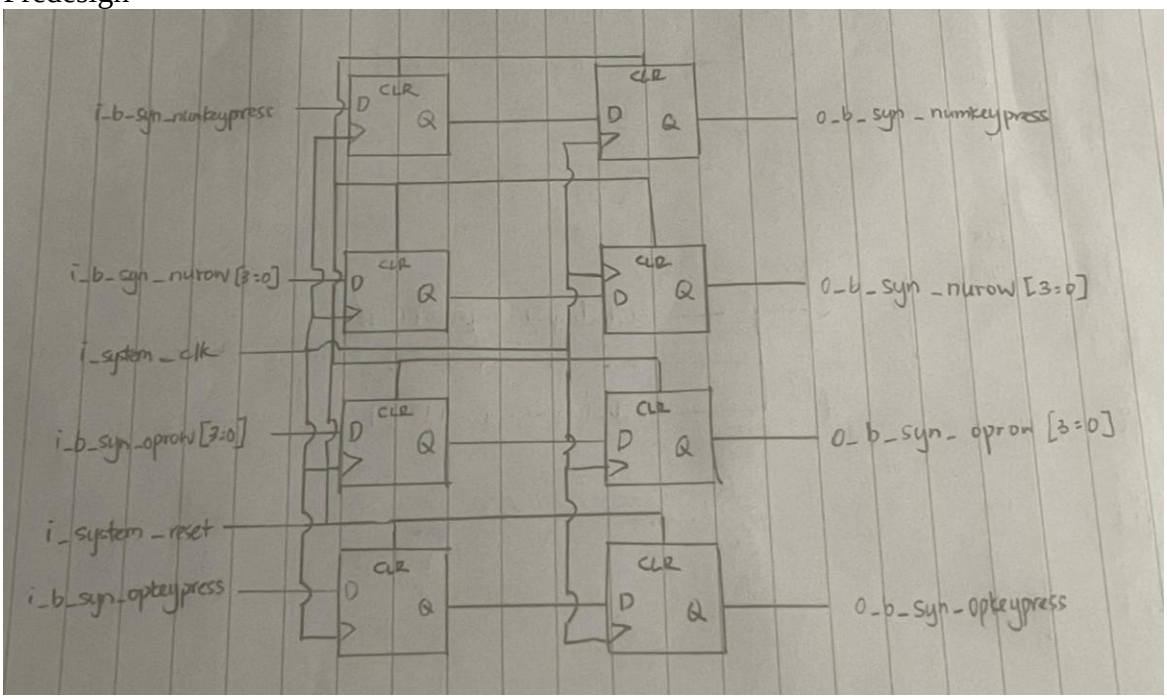
4.1.6 Timing Requirement

Synchronizer timing diagram

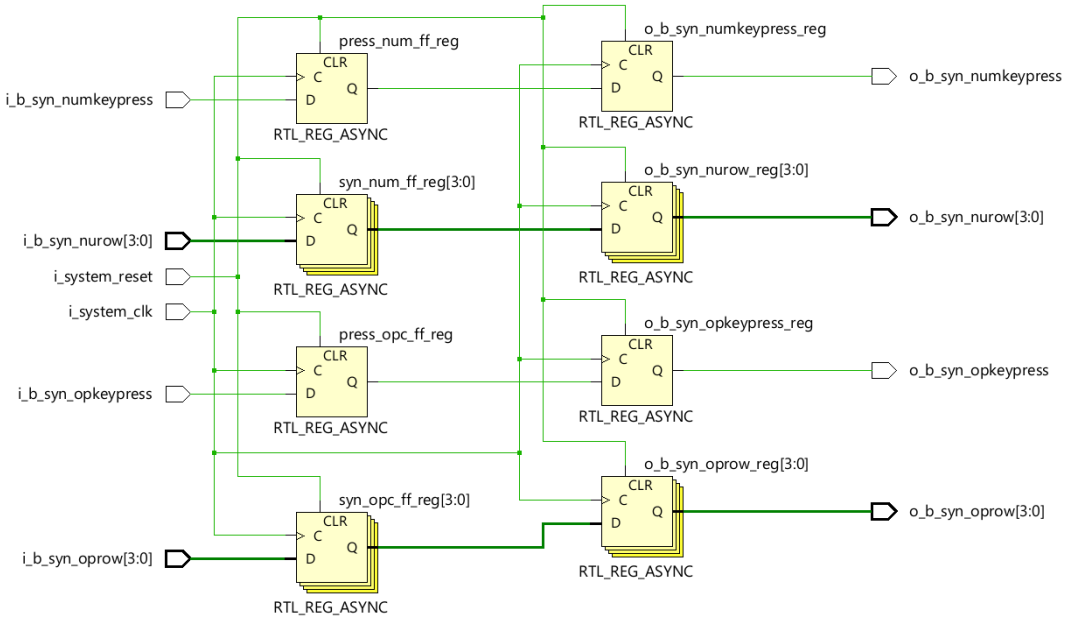


4.1.7 Schematic

Predesign



Postdesign



4.1.8 Vivado Model

```
//////////////////////////////////////////////////////////////////////////////////
 // Company: UTAR
 // Engineer: Cheah Lik Ding
 // Create Date: 05/05/2025
// Module Name: syn
// Project Name: syn
//////////////////////////////////////////////////////////////////////////////////

module SYN(
input i_system_clk,
input i_system_reset,
input [3:0] i_b_syn_nurow,
input [3:0] i_b_syn_oprow,
input logic i_b_syn_numkeypress,
input logic i_b_syn_opkeypress,
output logic [3:0] o_b_syn_nurow,
output logic [3:0] o_b_syn_oprow,
output logic o_b_syn_numkeypress,
output logic o_b_syn_opkeypress

);

logic [3:0] syn_num_ff;
logic [3:0] syn_opc_ff;
logic  press_num_ff;
logic  press_opc_ff;

always_ff @(posedge i_system_clk or posedge i_system_reset) begin
   if (i_system_reset) begin
       syn_num_ff <= 4'b0000;
       syn_opc_ff <= 4'b0000;
       press_num_ff <= 0;
       press_opc_ff <= 0;
       o_b_syn_nurow <= 4'b0000;
       o_b_syn_oprow <= 4'b0000;
       o_b_syn_numkeypress <=0;
       o_b_syn_opkeypress  <=0;

   end else begin
       syn_num_ff <= i_b_syn_nurow;
       syn_opc_ff <= i_b_syn_oprow;
       o_b_syn_nurow <= syn_num_ff;
       o_b_syn_oprow <= syn_opc_ff;
       press_num_ff<=i_b_syn_numkeypress;
       press_opc_ff<=i_b_syn_opkeypress;
      o_b_syn_numkeypress<= press_num_ff;
      o_b_syn_opkeypress<=  press_opc_ff;
   end
end


endmodule
```

4.1.9 Test Plan

No.	Test Feature	Test Vector	Expected Outputs	Pass / Fail
1	System Reset	reset= 1	o_b_syn_nurow = 4'b0000 o_b_syn_oprow = 4'b0000 o_b_syn_numkeypress = 1'b0 o_b_syn_opkeypress = 1'b0	pass

2	Valid Key Press (Debounced)	i_b_syn_nurow = 4'b1010;
		i_b_syn_oprow = 4'b1010;
		#100;
			o_b_syn_nurow = 4'b1010
			o_b_syn_oprow = 4'b1010
			o_b_syn_numkeypress = 1'b1
			o_b_syn_opkeypress = 1'b1
				pass

3	Unstable Input (Debounced Fail)	i_b_syn_nurow = 4'b1100; #5;	o_b_syn_nurow = 4'b1111; o_b_syn_oprow = 4'b1111;	pass

4.1.10 TestBench

```
//////////////////////////////////////////////////////////////////////////////////
 // Company: UTAR
 // Engineer: Cheah Lik Ding
 // Create Date: 05/05/2025
// Module Name: tb_syn
// Project Name: tb_syn
//////////////////////////////////////////////////////////////////////////////////
module tb_syn;
// Inputs
logic clk;
logic rst;
logic [3:0] i_b_syn_nurow;
logic [3:0] i_b_syn_oprow;
logic i_b_syn_numkeypress;
logic i_b_syn_opkeypress;

// Outputs
logic [3:0] o_b_syn_nurow;
logic [3:0] o_b_syn_oprow;
logic o_b_syn_numkeypress;
logic o_b_syn_opkeypress;

// Instantiate DUT
syn uut (
    .i_system_clk(clk),
    .i_system_reset(rst),
    .i_b_syn_nurow(i_b_syn_nurow),
    .i_b_syn_oprow(i_b_syn_oprow),
    .i_b_syn_numkeypress(i_b_syn_numkeypress),
```

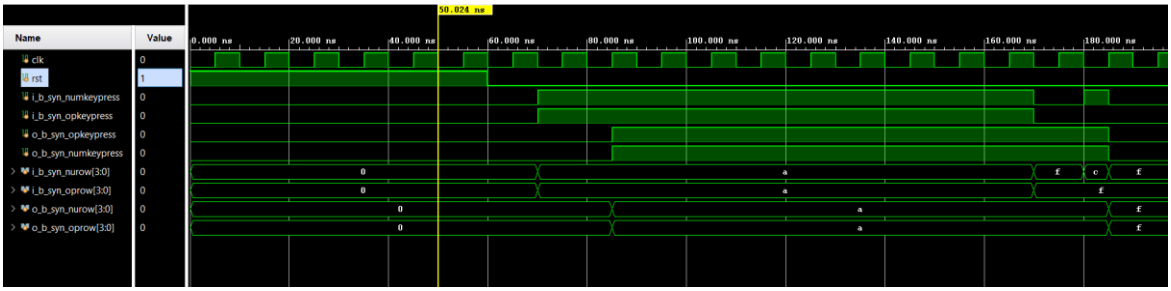
```
    .i_b_syn_opkeypress(i_b_syn_opkeypress),
    .o_b_syn_nurow(o_b_syn_nurow),
    .o_b_syn_oprow(o_b_syn_oprow),
    .o_b_syn_numkeypress(o_b_syn_numkeypress),
    .o_b_syn_opkeypress(o_b_syn_opkeypress)
);

// Clock generation
always #5 clk = ~clk;

// Stimulus
initial begin
  $display("==== Start syn_tb simulation ====");
    clk = 0;
    rst = 1;
    i_b_syn_nurow = 4'b0000;
    i_b_syn_oprow = 4'b0000;
    i_b_syn_numkeypress = 0;
    i_b_syn_opkeypress = 0;

    // Test 1: System Reset
    $display("Test 1: System Reset");
    #60;
    rst = 0;

    // Test 2: Valid Key Press (Debounced)
    $display("Test 2: Valid Key Press (Debounced)");
    #10;
    i_b_syn_nurow = 4'b1010;
    i_b_syn_oprow = 4'b1010;
    i_b_syn_numkeypress = 1;
    i_b_syn_opkeypress = 1;
    #100;
    i_b_syn_nurow = 4'b1111;
    i_b_syn_oprow = 4'b1111;
    i_b_syn_numkeypress = 0;
    i_b_syn_opkeypress = 0;

    // Test 3: Unstable Input (Debounce Fail)
    $display("Test 3: Unstable Input (Debounce Fail)");
    #10;
    i_b_syn_nurow = 4'b1100;
    i_b_syn_numkeypress = 1;
    #5;
    i_b_syn_nurow = 4'b1111;
    i_b_syn_oprow = 4'b1111;
    i_b_syn_numkeypress = 0;
    i_b_syn_opkeypress = 0;

    // Observe Outputs
    #20;
    $display("Output NuRow: %b", o_b_syn_nurow);
    $display("Output OpRow: %b", o_b_syn_oprow);
    $display("Output NumPress: %b", o_b_syn_numkeypress);
    $display("Output OpPress: %b", o_b_syn_opkeypress);
```

```
    $display("==== End syn_tb simulation ====");
    $stop;
end

endmodule
```

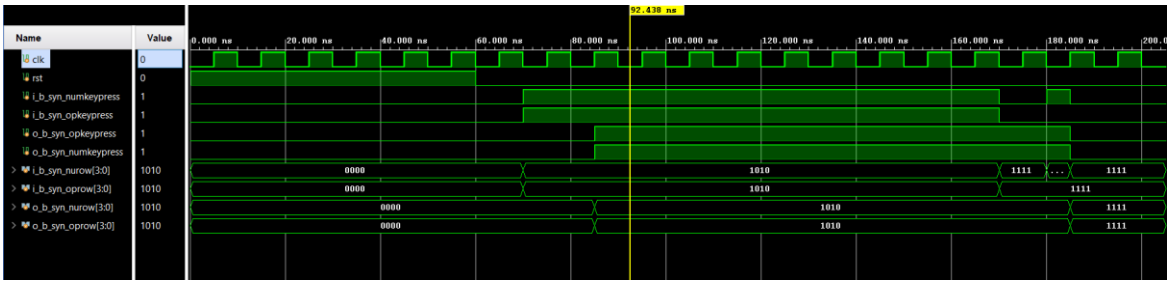
4.1.11 Test Result Timing Diagram

//Test 1: System reset
reset= 1
o_b_syn_nurow = 4'b0000
o_b_syn_oprow = 4'b0000
o_b_syn_numkeypress = 1'b0
o_b_syn_opkeypress = 1'b0

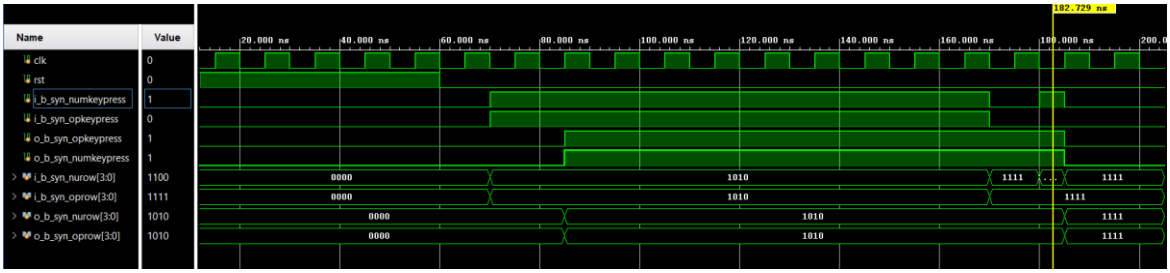
#60 i_system_reset = 0;



// Test 2: Valid Key Press (Debounced)
#10
i_b_syn_nurow = 4'b1010;
i_b_syn_oprow = 4'b1010;
#100;
o_b_syn_nurow = 4'b1010
o_b_syn_oprow = 4'b1010
o_b_syn_numkeypress = 1'b1
o_b_syn_opkeypress = 1'b1



// Test 3: Unstable Input (Debounce Fail)
i_b_syn_nurow = 4'b1100;
#5;
o_b_syn_nurow = 4'b1111;
o_b_syn_oprow = 4'b1111;



4.2 hex keypad code generator

4.2.1 Functionality

- Number Keys From 0 to 9

- Operation Keys of Addition, Subtraction, Multiplication, Division, Bitwise AND, OR , XOR, NOT, Barrel Shift Left and Right. The Enter Button for Continue the operation.
- Accepts 8-bit results from previous computations (e.g., ALU/shifter outputs).

- Uses system clock to synchronize keypad inputs and avoid metastability.

- System reset initializes registers and outputs to default states.

- Flags overflow if the operand exceeds 8 bits.

- Binary Output:
  - Based on the scanned keys, the module outputs the corresponding binary values.

- Data Routing:
  - Directs operands to the ALU or Barrel Shifter based on the opcode
  - Routes computation results to the I/O conversion block for display.

- Finite State Machine (FSM) with 4 States:
  - State 00 (Value a): Stores value of a
  - State 01 (Operators): stores operator used
  - State 10 (Value b): Stores value of b
  - State 11 (Computation): Calculates the result

4.2.2 Features

Dual Keypad Sections:

- Numeric Section: Inputs values (0-9) via dedicated rows/columns.
- Operation Section: Inputs arithmetic/logic operations. via separate rows/columns.

Output Interfaces:

- 8-bit Operands: Send numeric inputs to ALU, Barrel Shifter, and IOD.
- 4-bit Opcode: Specifies operations

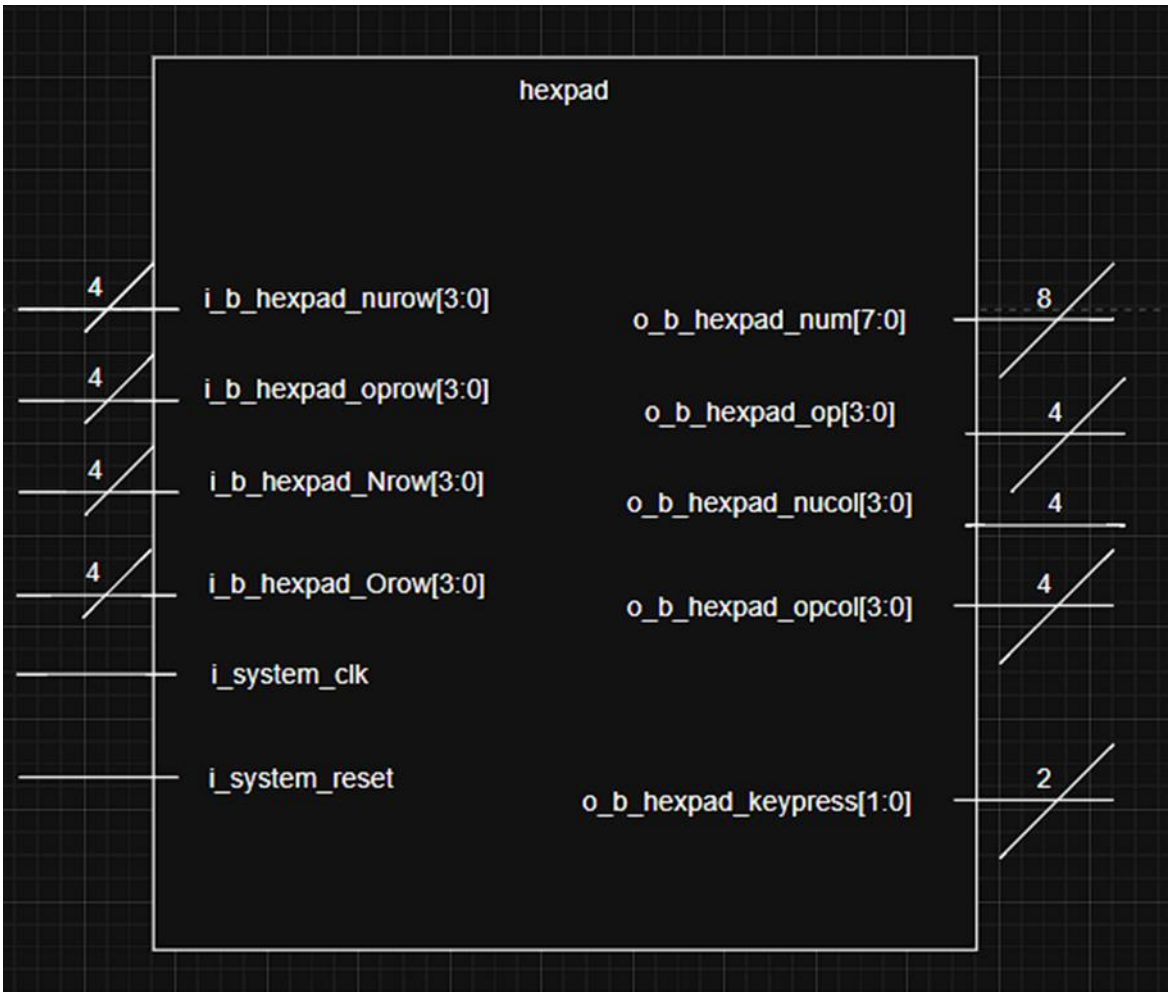
Compatibility

- Integrates with ALU, Barrel Shifter, and IOD for arithmetic operations, shifts, and display updates.

Low-Latency Design
- Processes inputs within 1 clock cycle for real-time responsiveness.

Synchronization:
- The clock signal is used to synchronize the keypad input signals with the system's clock, ensuring that the outputs are stable and reliable.

4.2.3 Block Interface



4.2.4 I/O Pin Description

Pin Name	Pin Class	Pin Direction	Source to direction	Pin Function
i_system_reset	Global, control	Input	Reset button to hexpad	Asynchronous reset to initialize state machine and outputs
i_system_clk	Global, control	Input	Clock to hexpad	System clock for FSM timing and sequential logic

i_b_hexpad_nurow [3:0]	data	Input	keypad to hexpad	Active-low row signal from number keypad (detect key press)
i_b_hexpad_oprow [3:0]	data	Input	keypad to hexpad	Active-low row signal from operator keypad
i_b_hexpad_Nrow[3:0]	Control	input	keypad scan enable to hexpad	Enable signal for number keypad row scanning
i_b_hexpad_Orow[3:0]	Control	input	keypad scan enable to hexpad	Enable signal for operator keypad row scanning
o_b_hexpad_num[7:0]	data	Output	Hexpad to CON	Outputs decoded number key value (0–9)
o_b_hexpad_op[3:0]	data	output	Hexpad to CON	Outputs decoded operator key value (e.g., +, -, *, etc.)
o_b_hexpad_nucol[3:0]	control	Output	Hexpad to keypad matrix	Active-low column scan signal for number keypad
o_b_hexpad_opCol [3:0]	control	Output	Hexpad to keypad matrix	Active-low column scan signal for operator keypad
o_b_hexpad_keypress [1:0]	Status	Output	Hexpad to controller	Indicates if a key is pressed: 01 = number, 10 = operator

4.2.5 Internal operation

Number key

Column (o_b_hexpad_nurow)	Row (i_b_hexpad_nucol)	Key	binary code
4’b0001	4’b0010	9	4’b1001
4’b0001	4’b0100	8	4’b1000
4’b0001	4’b1000	7	4’b0111
4’b0010	4’b0010	6	4’b0110
4’b0010	4’b0100	5	4’b0101
4’b0010	4’b1000	4	4’b0100
4’b0100	4’b0010	3	4’b0011
4’b0100	4’b0100	2	4’b0010
4’b0100	4’b1000	1	4’b0001
4’b1000	4’b0100	0	4’b0000

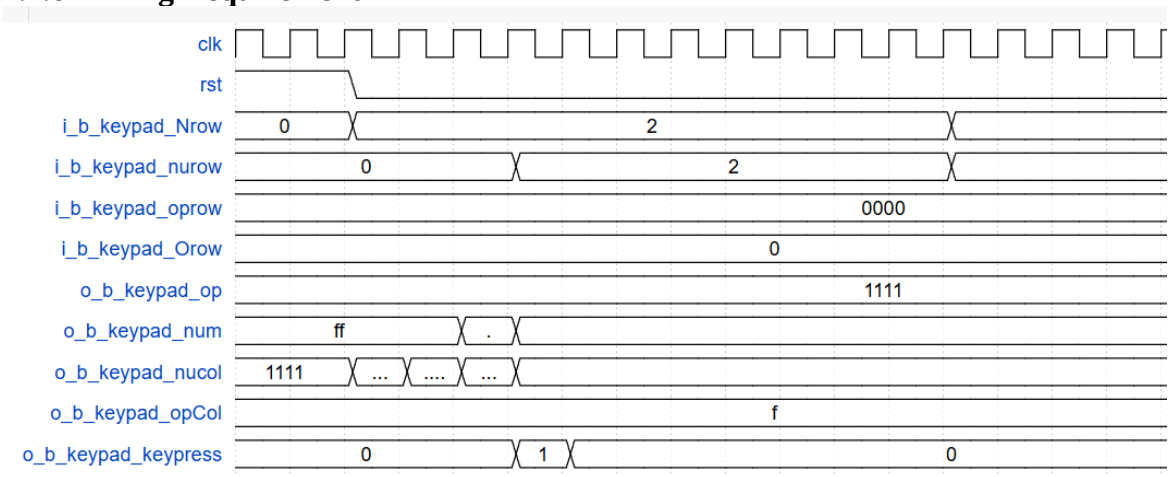
Default/Invalid	-	-	-

Column (o_b_hexpad_opcol)	Row (i_b_hexpad_oprow)	Operator	binary code
4’b0001	4’b0100	|	4’b0101
4’b0001	4’b1000	+	4’b0001
4’b0010	4’b0010	<<	4’b1001
4’b0010	4’b0100	&	4’b0110
4’b0010	4’b1000	-	4’b0010
4’b0100	4’b0010	>>	4’b1010
4’b0100	4’b0100	^	4’b0111
4’b0100	4’b1000	*	4’b0011
4’b1000	4’b0010	=	4’b1011
4’b1000	4’b0100	~	4’b1000
4’b1000	4’b1000	/	4’b0100
Default/Invalid	-	-	-

8bit result to binary

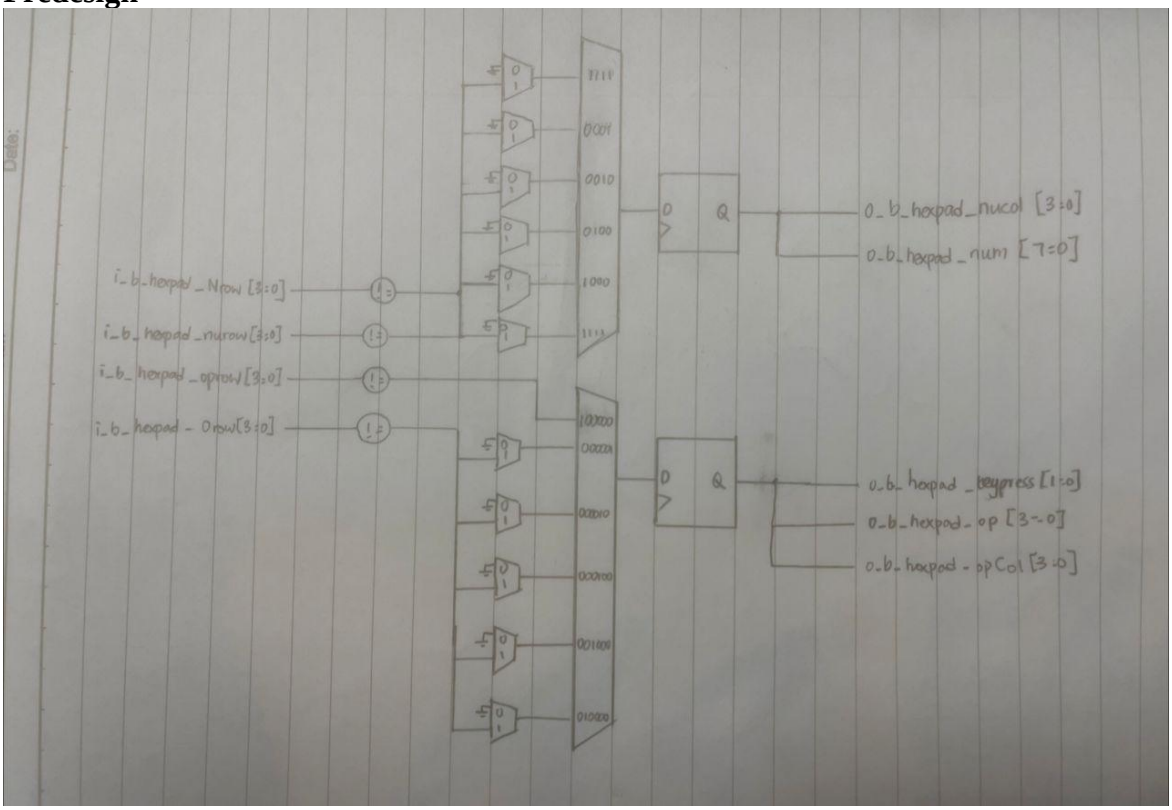
Result(i_b_hexpad_result)	binary code
8b’00000001	4’b0001
8b’00000010	4’b0010
8b’00000011	4’b0011
8b’00000100	4’b0100
8b’00000101	4’b0101
8b’00000110	4’b0110
8b’00000111	4’b0111
8b’00001000	4’b1000
8b’00001001	4’b1001
8b’00001010	4’b1010
8b’00001011	4’b1011
No key	4’b0000

4.2.6 Timing Requirement

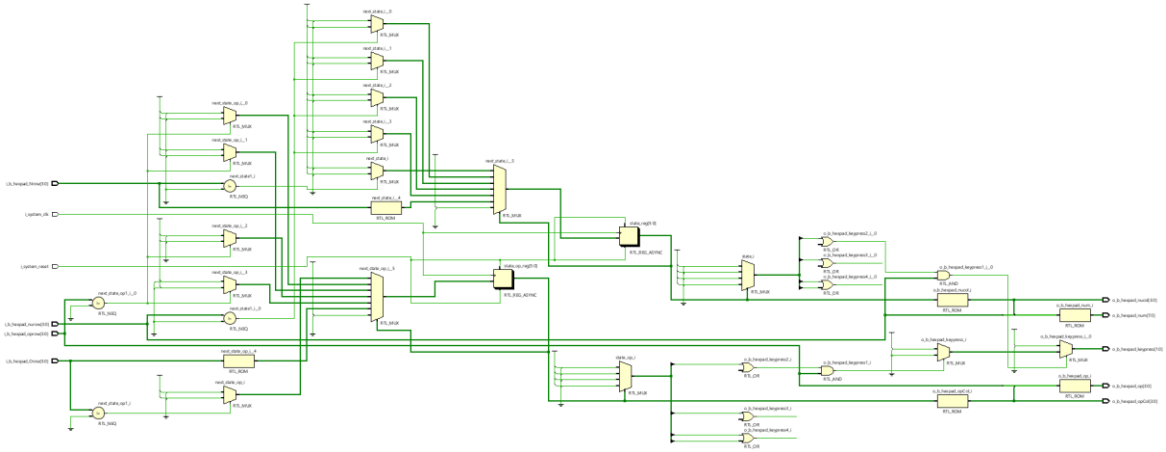


4.2.7 Schematic

Predesign



Postdesign



4.2.8 Vivado Model

```
//////////////////////////////////////////////////////////////////////////////////
 // Company: UTAR
 // Engineer: Cheah Lik Ding
 // Create Date: 05/05/2025
// Module Name: hexpad
// Project Name: hexpad
//////////////////////////////////////////////////////////////////////////////////


module hexpad (

 output logic [7:0] o_b_hexpad_num,

 output logic [3:0] o_b_hexpad_op,

output logic [3:0] o_b_hexpad_nucol,

output logic [3:0] o_b_hexpad_opCol,

output logic [1:0]o_b_hexpad_keypress,

 input logic [3:0] i_b_hexpad_nurow,

 input logic [3:0] i_b_hexpad_oprow,

 input logic i_b_hexpad_Nrow,

 input logic i_b_hexpad_Orow,

 input logic i_system_clk,

input logic i_system_reset

);

// ---------------- Number FSM ----------------

typedef enum

logic [5:0] { S_0 = 6'b000001, S_1 = 6'b000010, S_2 = 6'b000100, S_3 =
6'b001000, S_4 = 6'b010000, S_5 = 6'b100000 } state_t;

state_t state, next_state;

always_ff @(posedge i_system_clk or posedge i_system_reset) begin

 if (i_system_reset) state <= S_0;

 else state <= next_state;

end

always_comb begin
```

```
next_state = S_0;

o_b_hexpad_nucol = 4'b0000;

case (state)
   S_0: begin
      o_b_hexpad_nucol = 4'b1111;
      if (i_b_hexpad_Nrow != 0)
         next_state = S_1;
      else
         next_state = S_0;
   end
   S_1: begin
      o_b_hexpad_nucol = 4'b0001;
      if (i_b_hexpad_nurow != 4'b0000)
         next_state = S_5;
      else
         next_state = S_2;
   end
   S_2: begin
      o_b_hexpad_nucol = 4'b0010;
      if (i_b_hexpad_nurow != 4'b0000)
         next_state = S_5;
      else
         next_state = S_3;
   end
   S_3: begin
      o_b_hexpad_nucol = 4'b0100;
      if (i_b_hexpad_nurow != 4'b0000)
         next_state = S_5;
      else
         next_state = S_4;
   end
   S_4: begin
      o_b_hexpad_nucol = 4'b1000;
      if (i_b_hexpad_nurow != 4'b0000)
         next_state = S_5;
      else
         next_state = S_0;
   end
   S_5: begin
      o_b_hexpad_nucol = 4'b1111;
      if (i_b_hexpad_Nrow == 4'b0000)
         next_state = S_0;
      else
         next_state = S_5;
   end
   default: next_state = S_0;
endcase


end

// ---------------- Operator FSM ----------------

 typedef enum
```

```
logic [5:0] { S0_OP = 6'b000001, S1_OP = 6'b000010, S2_OP = 6'b000100, S3_OP
= 6'b001000, S4_OP = 6'b010000, S5_OP = 6'b100000 } state_op_t;

state_op_t state_op, next_state_op;

always_ff @(posedge i_system_clk or posedge i_system_reset) begin

if (i_system_reset) state_op <= S0_OP;

 else state_op <= next_state_op;

 end

always_comb begin

next_state_op = S0_OP;

o_b_hexpad_opCol = 4'b0000;

case (state_op)
   S0_OP: begin
      o_b_hexpad_opCol = 4'b1111;
      if (i_b_hexpad_Orow != 0)
         next_state_op = S1_OP;
      else
         next_state_op = S0_OP;
   end
   S1_OP: begin
      o_b_hexpad_opCol = 4'b0001;
      if (i_b_hexpad_oprow != 4'b0000)
         next_state_op = S5_OP;
      else
         next_state_op = S2_OP;
   end
   S2_OP: begin
      o_b_hexpad_opCol = 4'b0010;
      if (i_b_hexpad_oprow != 4'b0000)
         next_state_op = S5_OP;
      else
         next_state_op = S3_OP;
   end
   S3_OP: begin
      o_b_hexpad_opCol = 4'b0100;
      if (i_b_hexpad_oprow != 4'b0000)
         next_state_op = S5_OP;
      else
         next_state_op = S4_OP;
   end
   S4_OP: begin
      o_b_hexpad_opCol = 4'b1000;
      if (i_b_hexpad_oprow != 4'b0000)
         next_state_op = S5_OP;
      else
         next_state_op = S0_OP;
   end
   S5_OP: begin
      o_b_hexpad_opCol = 4'b1111;
```

```
        if (i_b_hexpad_Orow == 4'b0000)
            next_state_op = S0_OP;
        else
            next_state_op = S5_OP;
    end
    default: next_state_op = S0_OP;
endcase


end

always_comb begin

o_b_hexpad_keypress = 2'b00; // default: no key pressed
if (((state==S_1)||(state==S_2)||(state==S_3)||(state==S_4))&&
i_b_hexpad_nurow)
    //if (numRow != 4'b0000)
    o_b_hexpad_keypress = 2'b01; // Number key pressed
else if
(((state_op==S1_OP)||(state_op==S2_OP)||(state_op==S3_OP)||(state_op==S4_O
P))&& i_b_hexpad_oprow)

    //else if (opRow != 4'b0000)
    o_b_hexpad_keypress = 2'b10; // Operator key pressed
end

// ---------------- Decoder Logic ----------------
always_comb begin
case ({i_b_hexpad_nurow, o_b_hexpad_nucol})
8'b0001_0010: o_b_hexpad_num = 8'd9;
8'b0001_0100: o_b_hexpad_num = 8'd8;
8'b0001_1000: o_b_hexpad_num = 8'd7;
8'b0010_0010: o_b_hexpad_num = 8'd6;
8'b0010_0100: o_b_hexpad_num = 8'd5;
8'b0010_1000: o_b_hexpad_num = 8'd4;
8'b0100_0010: o_b_hexpad_num = 8'd3;
8'b0100_0100: o_b_hexpad_num = 8'd2;
8'b0100_1000: o_b_hexpad_num = 8'd1;
8'b1000_0100: o_b_hexpad_num = 8'd0;
default: o_b_hexpad_num = 8'b1111;
endcase
end

always_comb begin
case ({i_b_hexpad_oprow, o_b_hexpad_opCol})
8'b0001_0100: o_b_hexpad_op = 4'd5; // |
8'b0001_1000: o_b_hexpad_op = 4'd1; // +
8'b0010_0010: o_b_hexpad_op = 4'd9; // <<
8'b0010_0100: o_b_hexpad_op = 4'd6; // &
8'b0010_1000: o_b_hexpad_op = 4'd2; // -
8'b0100_0010: o_b_hexpad_op = 4'd10; // >>
8'b0100_0100: o_b_hexpad_op = 4'd7; // ^
8'b0100_1000: o_b_hexpad_op = 4'd3; // *
8'b1000_0010: o_b_hexpad_op = 4'd11; // =
8'b1000_0100: o_b_hexpad_op = 4'd8; // ~
8'b1000_1000: o_b_hexpad_op = 4'd4; // /
default: o_b_hexpad_op = 4'b1111;
```

```
endcase
end

endmodule
```

4.2.9 Test Plan

No.	Test Case	Description of Test Vector Generation	Expected Operation	Pass/fail
1	**Reset**	reset = 1	o_b_hexpad_nucol = 4'b1111 o_b_hexpad_opCol = 4'b1111 o_b_hexpad_num = 8'hFF o_b_hexpad_op = 4'hF o_b_hexpad_keypress = 2'b00	Pass

2	Press key 5	i_b_hexpad_Nrow = 1;
		wait for 3 clock cycle
		i_b_hexpad_nurow = 4'b0010;
			o_b_hexpad_nucol = 4'b0100;
			o_b_hexpad_num = 8'd5;
			o_b_hexpad_keypress = 2'b01;
				Pass

3	Press operator ‘+’	i_b_hexpad_Orow =1;
		Wait for 4 clock cycle
		i_b_hexpad_nurow = 4'b0010;
			o_b_hexpad_opCol = 4'b1000;
			o_b_hexpad_op = 4'd1;
			o_b_hexpad_keypress = 2'b10;
				Pass

4.2.10 TestBench

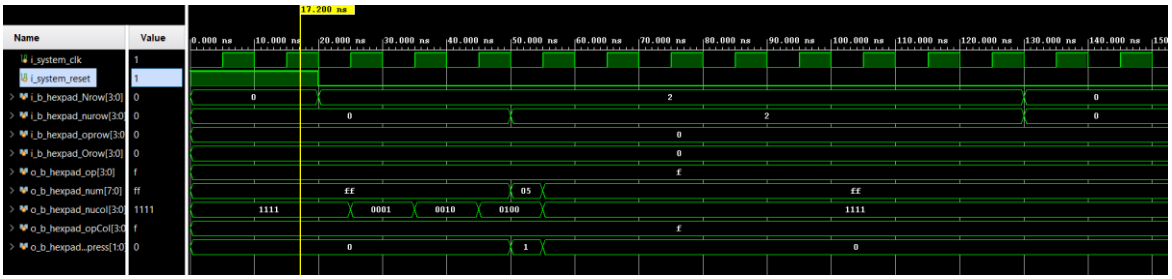
```
//////////////////////////////////////////////////////////////////////////////////
 // Company: UTAR
 // Engineer: Cheah Lik Ding
 // Create Date: 05/05/2025
// Module Name: tb_hexpad
// Project Name: tb_hexpad
//////////////////////////////////////////////////////////////////////////////////

module tb_hexpad;

logic [7:0] o_b_hexpad_num;
logic [3:0] o_b_hexpad_op;
logic [3:0] o_b_hexpad_nucol, o_b_hexpad_opCol;
logic [1:0] o_b_hexpad_keypress;

logic [3:0] i_b_hexpad_nurow, i_b_hexpad_oprow;
logic i_b_hexpad_Nrow, i_b_hexpad_Orow;
logic i_system_clk, i_system_reset;

// Instantiate DUT
hexpad dut (
    .o_b_hexpad_num(o_b_hexpad_num),
    .o_b_hexpad_op(o_b_hexpad_op),
    .o_b_hexpad_nucol(o_b_hexpad_nucol),
    .o_b_hexpad_opCol(o_b_hexpad_opCol),
    .o_b_hexpad_keypress(o_b_hexpad_keypress),
    .i_b_hexpad_nurow(i_b_hexpad_nurow),
    .i_b_hexpad_oprow(i_b_hexpad_oprow),
    .i_b_hexpad_Nrow(i_b_hexpad_Nrow),
    .i_b_hexpad_Orow(i_b_hexpad_Orow),
    .i_system_clk(i_system_clk),
    .i_system_reset(i_system_reset)
);

// Clock generation: 10ns period
initial i_system_clk = 0;
always #5 i_system_clk = ~i_system_clk;

initial begin
    // Test 1 Rest
    i_system_reset = 1;
    i_b_hexpad_nurow = 4'b0000;
    i_b_hexpad_oprow = 4'b0000;
    i_b_hexpad_Nrow  = 0;
    i_b_hexpad_Orow  = 0;

    #20;
    i_system_reset = 0;

    // Test 2 === Press key 5 ===
    i_b_hexpad_Nrow = 1;
    #30;
    i_b_hexpad_nurow = 4'b0010;
    #80;
    i_b_hexpad_nurow = 4'b0000;
```

```
    i_b_hexpad_Nrow  = 0;

    // Test 3 === Press operator '+' ===
    #50;
    i_b_hexpad_Orow = 1;
    #40;
    i_b_hexpad_oprow = 4'b0001;
    #80;
    i_b_hexpad_oprow = 4'b0000;
    i_b_hexpad_Orow  = 0;

    // === End simulation ===
    #100;
    $finish;
end

initial begin
    $display("Time\t NuRow NuCol Num OpRow OpCol Op KeyPress");
    $monitor("%0t\t %b  %b   %0d  %b   %b   %0d   %b",
             $time,
             i_b_hexpad_nurow,
             o_b_hexpad_nucol,
             o_b_hexpad_num,
             i_b_hexpad_oprow,
             o_b_hexpad_opCol,
             o_b_hexpad_op,
             o_b_hexpad_keypress);
end


endmodule
```

4.2.11 Test Result Timing Diagram

// Test 1: Reset test
reset = 1
All outputs should be at default values:

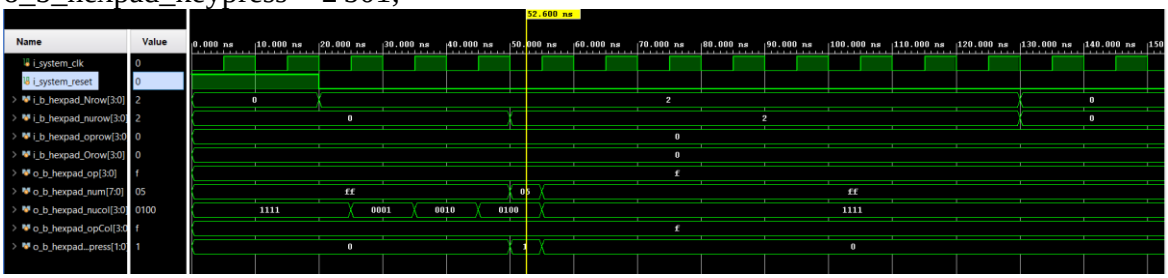
- o_b_hexpad_nucol = 4'b1111
- o_b_hexpad_opCol = 4'b1111
- o_b_hexpad_num = 8'hFF
- o_b_hexpad_op = 4'hF
- o_b_hexpad_keypress = 2'b00



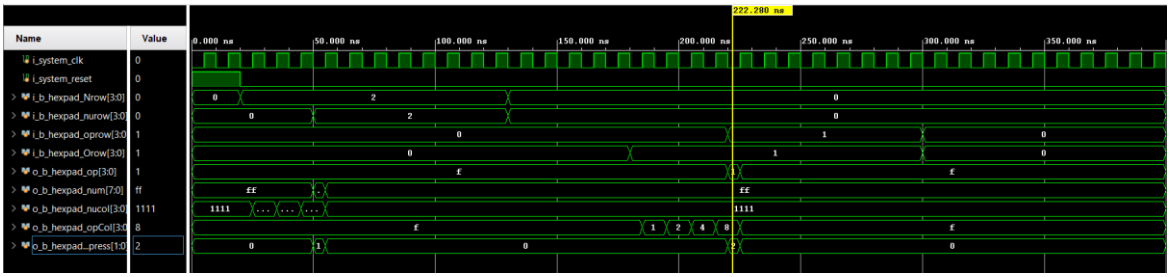
//Test 2: Press key 5
i_b_hexpad_Nrow = 1;

Wait for 3 clock cycle
i_b_hexpad_nurow = 4'b0010;

Expected output
o_b_hexpad_nucol = 4'b0100;
o_b_hexpad_num = 8'd5;
o_b_hexpad_keypress = 2'b01;



//Test 3: Press operator '+'
i_b_hexpad_Orow = 1;
Wait for 4 clock cycle
i_b_hexpad_nurow = 4'b0010;
Expected Output:
o_b_hexpad_opCol = 4'b1000;
o_b_hexpad_op = 4'd1;
o_b_hexpad_keypress = 2'b10;



4.3 Control Unit Block

4.3.1 Functionality

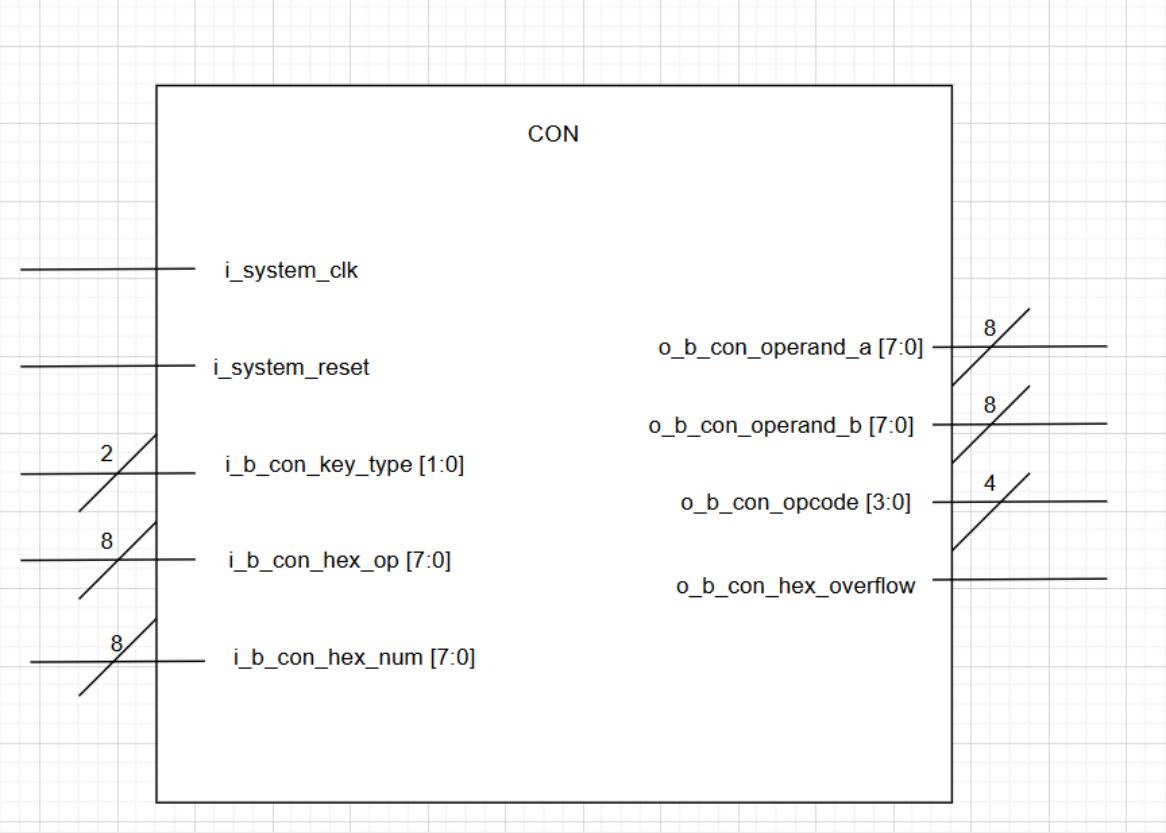
The cub module serves as the control unit that manages user input from a keypad for a simple calculator system. It interprets numeric and operator key inputs, organizes operand data, decodes operations, and prepares the appropriate output signals for arithmetic or logic computation.

1. **Operand Capture:**
   a. Accepts numeric key inputs and constructs multi-digit operands A and B.
   b. Supports up to 3-digit input for each operand using decimal entry logic (×10 accumulation).
2. **Operator Decoding:**
   a. Receives an operator key and decodes it into a 4-bit opcode.
   b. Recognizes arithmetic (+, -, *, /) and bitwise (&, |, ^, ~, <<, >>) operations.
3. **State Management:**
   a. Uses a finite state machine (FSM) to control input sequencing:
      IDLE → LOAD_A → LOAD_OP → LOAD_B → (optional COMPUTE) → RESET
   b. Prevents accidental multiple registrations using key debounce logic (prev_key latch).
4. **Overflow Detection:**
   a. Monitors operand values and sets an overflow flag if they exceed 255 (8-bit limit).

4.3.2 Features

- **Support for Two Operands**: Handles two separate 8-bit operands.
- **Multi-Digit Entry**: Allows users to input numbers digit by digit (e.g., pressing ‘1’, then ‘2’ results in 12).
- **Operator Mapping**: Decodes specific hex values to standard operations (+, -, etc.).
- **Shift Operation Detection**: Flags if the selected operation is a shift (<< or >>).
- **Overflow Flag**: Signals when an operand exceeds 8-bit capacity.
- **Reset Handling**: Fully resets all internal registers and state on reset.

4.3.3 Block Interface



4.3.4 I/O Pin Description

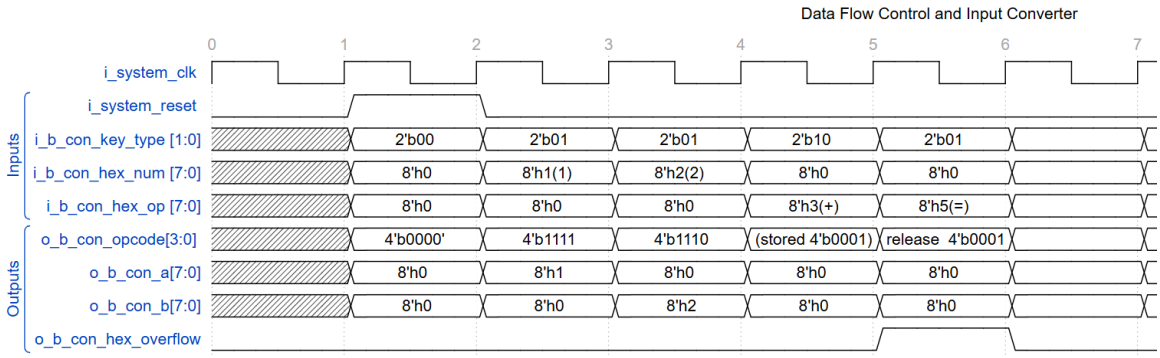
Pin Name	Source To Destination	Pin Direction	Pin Class	Pin Function
i_system_clk	Global Clock to CON	input	control	System clock used for synchronous logic
i_system_reset	Reset Button to CON	input	control	Active-high system reset that initializes the module
i_b_con_hex_num[7:0]	Keypad Decoder to CON	input	data	8-bit value representing the pressed number key
i_b_con_hex_opc[7:0]	Keypad Decoder to CON	input	data	8-bit value representing the pressed operator key
i_b_con_key_type [1:0]	Keypad Decoder to CON	input	data	Indicates the type of key pressed: number

				or operator
o_b_con_operand_a [7:0]	CON to ALU/Display	output	data	8-bit output of the first operand (A)
o_b_con_operand_ b[7:0]	CON to ALU/Display	output	data	8-bit output of the second operand (B)
o_b_con_hex_overf low	CON to Display	output	status	High if operand A or B exceeds 8-bit (overflow)
o_b_con_opcode[3: 0]	CON to ALU	output	data	4-bit decoded opcode based on the operator key

4.3.5 Internal operation

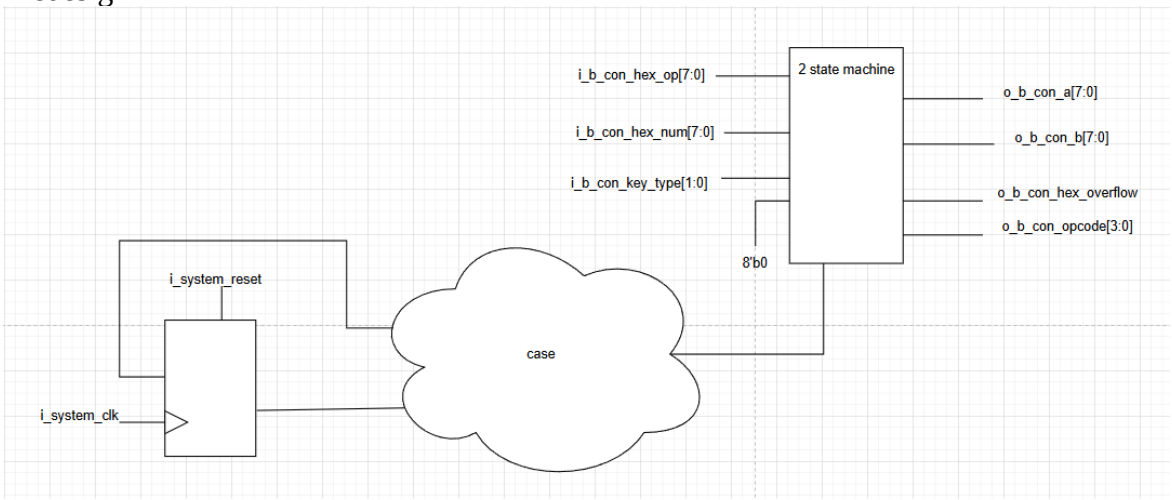
	reset	**A=1**	**B=2**	**Opcode='+'**	**Opcode='='**	**B=256**
i_system_reset	**1**	**0**	**0**	**0**	**0**	**0**
i_b_con_hex_num[7:0]	**8'h0**	**8'h1**	**8'h2**	**8'h0**	**8'h0**	**8'd2, 8'd5, 8'd6**
i_b_con_hex_opc[7:0]	**8'h0**	**8'h0**	**8'h0**	**8'h3**	**8'h5**	**8'h4**
i_b_con_key_type [1:0]	**2'b00**	**2'b01**	**2'b01**	**2'b10**	**2'b10**	**2'b01**
o_b_con_operand_a[7:0]	**8'h0**	**8'h1**	**8'h0**	**8'h0**	**8'h0**	**8'h0**
o_b_con_operand_b[7:0]	**8'h0**	**8'h0**	**8'h2**	**8'h0**	**8'h0**	**8'h2 8'h25 8'h256**
o_b_con_hex_overflow	**0**	**0**	**0**	**0**	**0**	**1**
o_b_con_opcode[3:0]	**4'h0**	**4'b1111**	**4'b1110**	**stored 4'b0001 and release after press(=)**	**4'b1011 opcode that stored before**	**4'b1110**

4.3.6 Timing Requirement

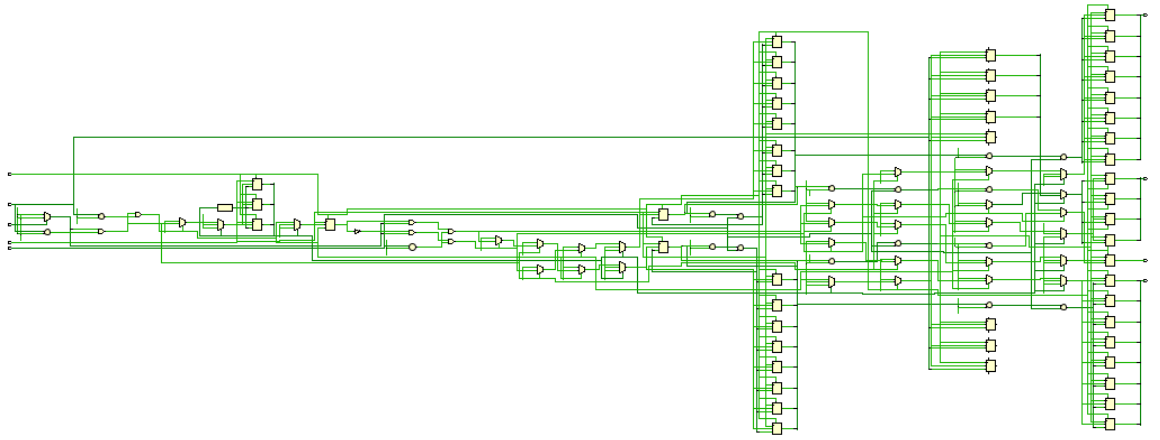


4.3.7 Schematic

Predesign



Postdesign



4.3.8 Vivado Model

```
//////////////////////////////////////////////////////////////////////////////////
 // Company: UTAR
 // Engineer: CHONG BING HONG
 // Create Date: 07/05/2025
```

```
// Module Name: CON
 // Project Name: CON
 //////////////////////////////////////////////////////////////////////////////////



module con (
input i_system_clock,
input i_system_reset,
input [1:0] i_b_con_key_type,
 input [7:0] i_b_con_hex_num,
input [7:0] i_b_con_hex_op,

 output logic [3:0] o_b_con_opcode,
output logic [7:0] o_b_con_a,
 output logic [7:0] o_b_con_b,
output logic o_b_con_hex_overflow );

logic [8:0] reg_a, reg_b;
 logic opc_load = 1'b0;
 logic [2:0] state = 3'b000;
logic [7:0] str_key_op;

always_ff @(posedge i_system_clock, posedge i_system_reset) begin
   if (i_system_reset) begin
      o_b_con_opcode      <= 4'hF;
      o_b_con_a           <= 8'h00;
      o_b_con_b           <= 8'h00;
      reg_a               <=0;
      reg_b               <=0;
      o_b_con_hex_overflow <= 1'b0;
      opc_load            <= 1'b0;
      state               <= 3'b000;
   end
   else begin
      case (state)
         3'b000: begin
            if ((i_b_con_key_type == 2'b10) && (i_b_con_hex_op != 4'hF)&&
(i_b_con_hex_op != 4'hB)) begin
               str_key_op <= i_b_con_hex_op;
               opc_load       <= 1'b1;
            end
            else if ((i_b_con_key_type == 2'b10)&&(i_b_con_hex_op != 4'hF)) begin
             state     <= 3'b100;
            end
            else if ((!opc_load) && (i_b_con_key_type == 2'b01)  && (i_b_con_hex_num
<= 8'h09))  begin
               reg_a<=reg_a*10+i_b_con_hex_num;
               o_b_con_opcode <= 4'hF;
               o_b_con_a <= reg_a*10+i_b_con_hex_num;
               o_b_con_hex_overflow   <= ((reg_a*10+i_b_con_hex_num) > 255);
```

```
                end
            else if (opc_load && (i_b_con_key_type == 2'b01) && (i_b_con_hex_num <=
8'h09)) begin
                reg_b<=reg_b*10+i_b_con_hex_num;
                o_b_con_opcode <= 4'hE;
                o_b_con_b <= reg_b*10+i_b_con_hex_num;
                 o_b_con_hex_overflow   <= ((reg_b*10+i_b_con_hex_num) > 255);
            end
          end

          3'b100: begin
             o_b_con_opcode <=str_key_op;
             opc_load<=0;
             state     <= 3'b000;
          end
          default: state <= 3'b000;
       endcase
   end
end

endmodule
```

4.3.9 Test Plan

No.	Test Case	Description of Test Vector Generation	Expected Operation	Pass/fail

1	System Reset	//Wait for 10 clock cycle,
		tb_system_reset = 1;
		//wait for 10 clock cycle,
		tb_system_reset = 0;
		//wait for 10 clock cycle
			i_b_con_hex_num=11111111
			i_b_con_hex_op=11111111
			o_b_con_a=00000000
			o_b_con_b=00000000
			o_b_con_opcode=1111
			o_b_con_hex_overflow=0
				pass

2	Load Operand A (135)	//A=1
		i_b_con_key_type = 2'b01;
		i_b_con_hex_num = 8'h01;
		//Wait for10 clock cycle,
		i_b_con_key_type = 2'b00;
		// release
		i_b_con_hex_num = 8'hFF;
		//Wait for10 clock cycle,
		
		//A=3
		i_b_con_key_type =
			i_b_con_key_type=2’b01
			o_b_con_a=8’d1
			o_b_con_opcode=4’b1111
			
			After 10 clock cycle,
			i_b_con_key_type=2’b01
			I_b_con_a=8’d13
			o_b_con_opcode=4’b1111
			
			After 10 clock cycle,
			i_b_con_key_type=2’b01
			I_b_con_a=8’d135
			o_b_con_opcode=4’b1111
				pass

		2'b01;
		i_b_con_hex_num =8'h03;
		//Wait for10 clock cycle,
		i_b_con_key_type = 2'b00;
		// release
		i_b_con_hex_num =8'hFF;
		//Wait for10 clock cycle,
		
		//A=5
		i_b_con_key_type = 2'b01;
		i_b_con_hex_num =8'h05;
		//Wait for10 clock cycle,
		i_b_con_key_type = 2'b00;
		// release
		i_b_con_hex_num =8'hFF;
		//Wait for10 clock cycle,		

3	Load Operator (+)	i_b_con_key_type = 2'b10; i_b_con_hex_op = 8'h01;
		
		// +
		//Wait for10 clock cycle,
		i_b_con_key_type = 2'b00; i_b_con_hex_op = 8'hFF;
		//Wait for10 clock cycle,
			i_b_con_key_type = 2'b10
			i_b_con_hex_op = 8'h01
				pass

4	Load Operand B(4)	//B=4
		i_b_con_key_type = 2'b01;
		i_b_con_hex_num = 8'h04;
		//Wait for10 clock cycle,
		i_b_con_key_type =
			i_b_con_key_type = 2'b01
			o_b_con_b= 8'h4
			o_b_con_opcode=4'b1110
				pass

		2'b00;
		i_b_con_hex_num = 8'hFF;
		//Wait for10 clock cycle,		

5	Load (=)	//hex_op =(=)
		i_b_con_key_type = 2'b10; i_b_con_hex_op = 8'h0B;
		
		//Wait for10 clock cycle,
		i_b_con_key_type = 2'b00; i_b_con_hex_op = 8'hFF;
		
		//Wait for20 clock cycle,
			i_b_con_key_type = 2'b10
			i_b_con_hex_op = 8'hB
			o_b_con_opcode=4'0001
				pass

6	Load operator(>>)	// operator = >>
		i_b_con_key_type = 2'b10; i_b_con_hex_op = 8'h10;
		
		//Wait for10 clock cycle,
		i_b_con_key_type = 2'b00; i_b_con_hex_op = 8'hFF;
		
		//Wait for10 clock cycle,
			i_b_con_key_type = 2'b10
			i_b_con_hex_op = 8'h10
				pass

7	Overflow(256)	//reset previous result
		//Wait for10 clock cycle,
		i_system_reset = 1;
		//Wait for10 clock cycle,
		i_system_reset = 0;
		//Wait for10 clock cycle,
		
		//A=2
		i_b_con_key_type = 2'b01;
		i_b_con_hex_num =8'h02;
		//Wait for10 clock
			i_b_con_key_type=2'b01
			o_b_con_a=8'd2
			o_b_con_opcode=4'b1111
			
			After 10 clock cycle,
			i_b_con_key_type=2'b01
			I_b_con_a=8'd25
			o_b_con_opcode=4'b1111
			
			After 10 clock cycle,
			i_b_con_key_type=2'b01
			I_b_con_a=8'd0
			o_b_con_opcode=4'b1111
			o_b_con_overflow=1
				pass

		cycle,
		i_b_con_key_type = 2'b00;
		// release
		i_b_con_hex_num =8'hFF;
		//Wait for10 clock cycle,
		
		//A=5
		i_b_con_key_type = 2'b01;
		i_b_con_hex_num =8'h05;
		//Wait for10 clock cycle,
		i_b_con_key_type = 2'b00;
		// release
		i_b_con_hex_num =8'hFF;
		//Wait for10 clock cycle,
		
		//A=6
		i_b_con_key_type = 2'b01;
		i_b_con_hex_num =8'h06; //Wait for10 clock cycle,
		i_b_con_key_type = 2'b00;
		// release
		i_b_con_hex_num =8'hFF;
		//Wait for10 clock cycle,		

4.3.10 TestBench

```
//////////////////////////////////////////////////////////////////////////////////
 // Company: UTAR
 // Engineer: CHONG BING HONG
 // Create Date: 07/05/2025
// Module Name: tb_con
 // Project Name: tb_con
 //////////////////////////////////////////////////////////////////////////////////


` timescale 1ns / 1ps
```

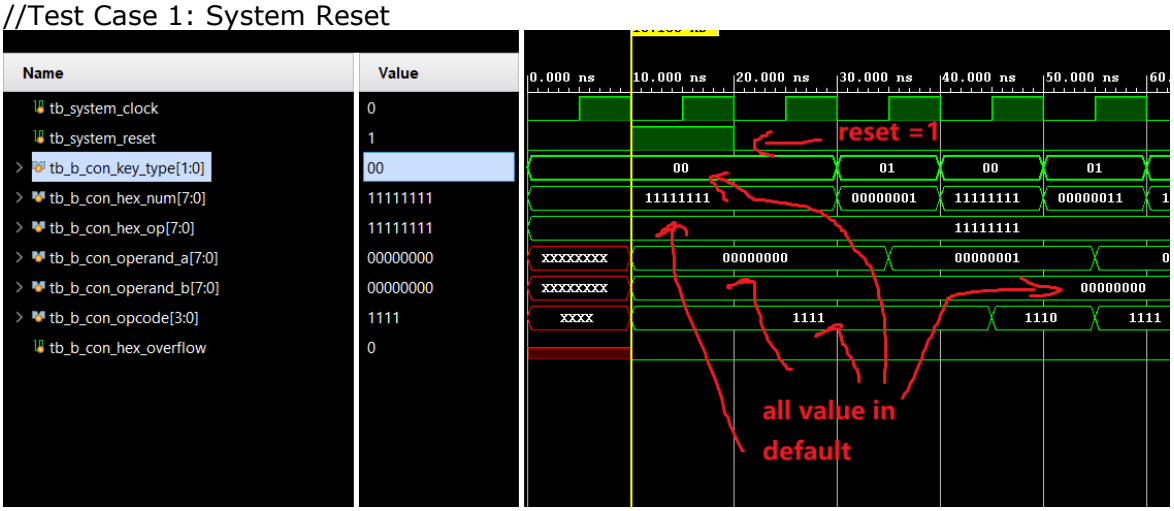
```
` define PERIOD_CLK 5

module tb_con();

// Inputs
logic       tb_system_clock;
logic       tb_system_reset;
logic [1:0]  tb_b_con_key_type;
logic [7:0]  tb_b_con_hex_num;
logic [7:0]  tb_b_con_hex_op;

// Outputs
logic [7:0]  tb_b_con_operand_a;
logic [7:0]  tb_b_con_operand_b;
logic [3:0]  tb_b_con_opcode;
logic       tb_b_con_hex_overflow;

// DUT
con dut (
    .i_system_clock(tb_system_clock),
    .i_system_reset(tb_system_reset),
  .i_b_con_key_type(tb_b_con_key_type),
.i_b_con_hex_num(tb_b_con_hex_num),
    .i_b_con_hex_op(tb_b_con_hex_op),
    .o_b_con_operand_a(tb_b_con_operand_a),
    .o_b_con_operand_b(tb_b_con_operand_b),
    .o_b_con_opcode(tb_b_con_opcode),
    .o_b_con_hex_overflow(tb_b_con_hex_overflow)
);

// Clock generation
initial tb_system_clock = 0;
always #(`PERIOD_CLK) tb_system_clock = ~tb_system_clock;

initial begin
    $display("--- Start tb_cub ---");

 // Default init
          tb_system_reset = 0;
    tb_b_con_key_type = 2'b00;
    tb_b_con_hex_num = 8'hFF;
    tb_b_con_hex_op = 8'hFF;

    //Test Case 1: System Reset
    #10; tb_system_reset = 1;
    #10; tb_system_reset = 0;
    #10;

    //Test Case 2: Load Operand A (1?3?5)
    tb_b_con_key_type = 2'b01;
    tb_b_con_hex_num  = 8'h01;
    #10;
    tb_b_con_key_type = 2'b00; // release
    tb_b_con_hex_num  = 8'hFF;
    #10;
```

```
tb_b_con_key_type = 2'b01;
tb_b_con_hex_num  = 8'h03;
#10;
tb_b_con_key_type = 2'b00; // release
tb_b_con_hex_num  = 8'hFF;
#10;

 tb_b_con_key_type = 2'b01;
tb_b_con_hex_num  = 8'h05;
#10;
tb_b_con_key_type = 2'b00; // release
tb_b_con_hex_num  = 8'hFF;
#10;

//Test Case 3: Load Operator (+)
tb_b_con_key_type = 2'b10;
tb_b_con_hex_op  = 8'h01; // +
#10;
tb_b_con_key_type = 2'b00;
tb_b_con_hex_op  = 8'hFF;
#10;

//Test Case 4: Load Operand B (4)
tb_b_con_key_type = 2'b01;
tb_b_con_hex_num  = 8'h04;
#10;
tb_b_con_key_type = 2'b00;
tb_b_con_hex_num  = 8'hFF;
#10;

//Test Case 5: Trigger Compute by Loading ("=")
tb_b_con_key_type = 2'b10;
tb_b_con_hex_op  = 8'h0B;
#10;
tb_b_con_key_type = 2'b00;
tb_b_con_hex_op  = 8'hFF;
#20;

//Test Case 6: Trigger Barrel Shifter by Loading (">>")
tb_b_con_key_type = 2'b10;
tb_b_con_hex_op  = 8'h10; // >>
#10;
tb_b_con_key_type = 2'b00;
tb_b_con_hex_op  = 8'hFF;
#10;

//Test Case 7: overflow (2，5，6)
//reset
#10; tb_system_reset = 1;
#10; tb_system_reset = 0;
#10;

tb_b_con_key_type = 2'b01;
tb_b_con_hex_num  = 8'h02;
#10;
tb_b_con_key_type = 2'b00; // release
```

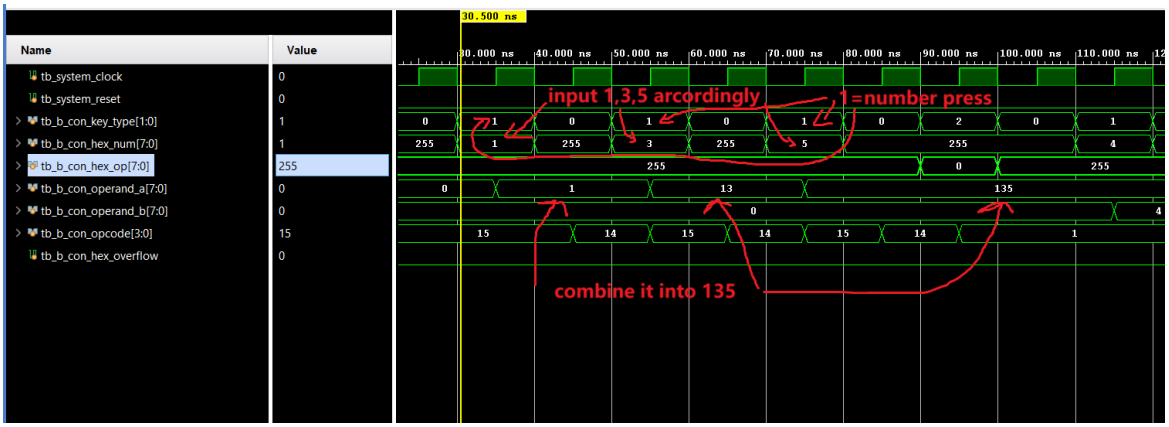
```
    tb_b_con_hex_num  = 8'hFF;
    #10;

    tb_b_con_key_type = 2'b01;
    tb_b_con_hex_num  = 8'h05;
    #10;
    tb_b_con_key_type = 2'b00; // release
    tb_b_con_hex_num  = 8'hFF;
    #10;

     tb_b_con_key_type = 2'b01;
    tb_b_con_hex_num  = 8'h06;
    #10;
    tb_b_con_key_type = 2'b00; // release
    tb_b_con_hex_num  = 8'hFF;
    #10;
    // End simulation
    #(`PERIOD_CLK * 5);
    $display("--- End tb_con ---");
    $finish;
end


endmodule
```

4.3.11 Result Simulation

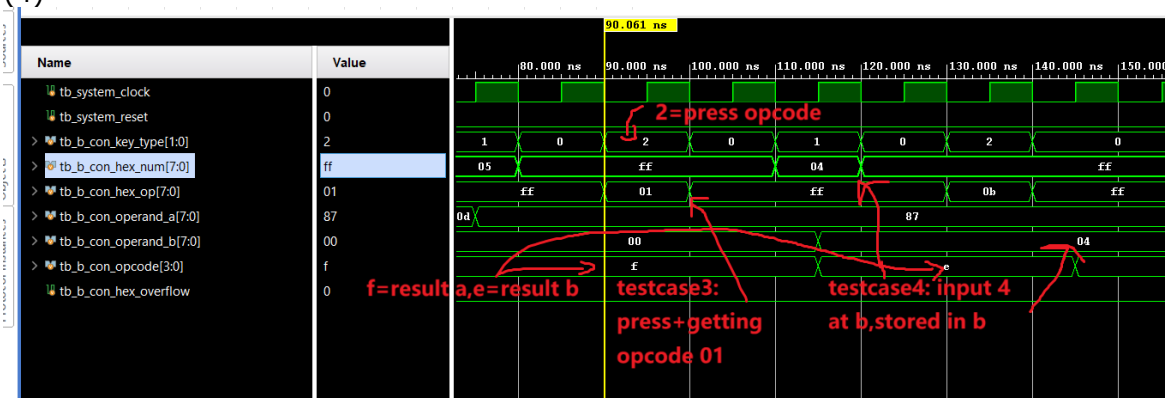
//Test Case 1: System Reset



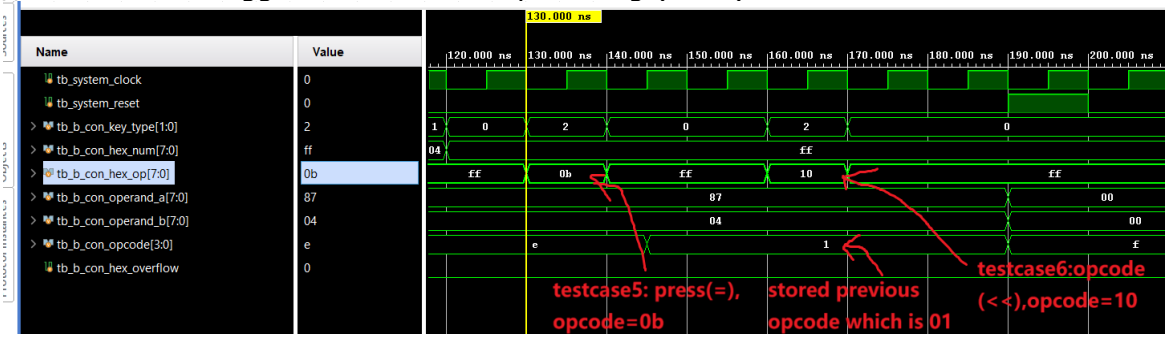
//Test Case 2: Load Operand A (1,3,5)



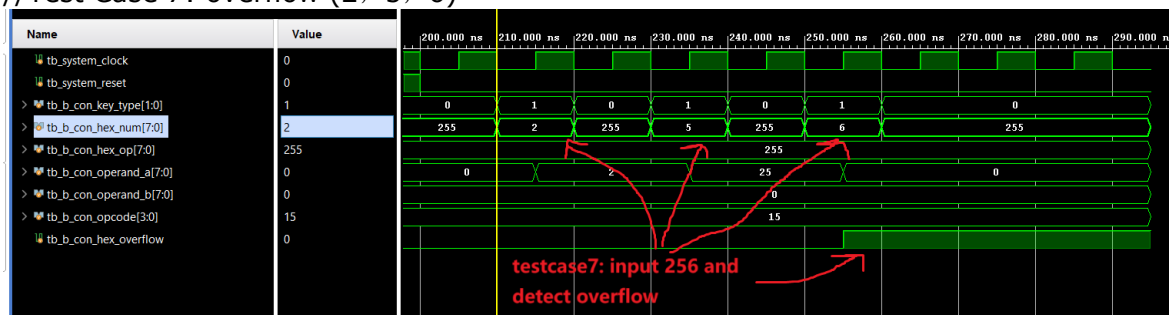
//Test Case 3: Load Operator (+)
//Test Case 4: Load Operand B
(4)



//Test Case 5: Trigger Compute by Loading ("=")
//Test Case 6: Trigger Barrel Shifter by Loading (">>")



//Test Case 7: overflow (2，5，6)



4.4 ALU

4.4.1 Functionality

- Addition of two 8-bit integers
- Subtraction of two 8-bit integers

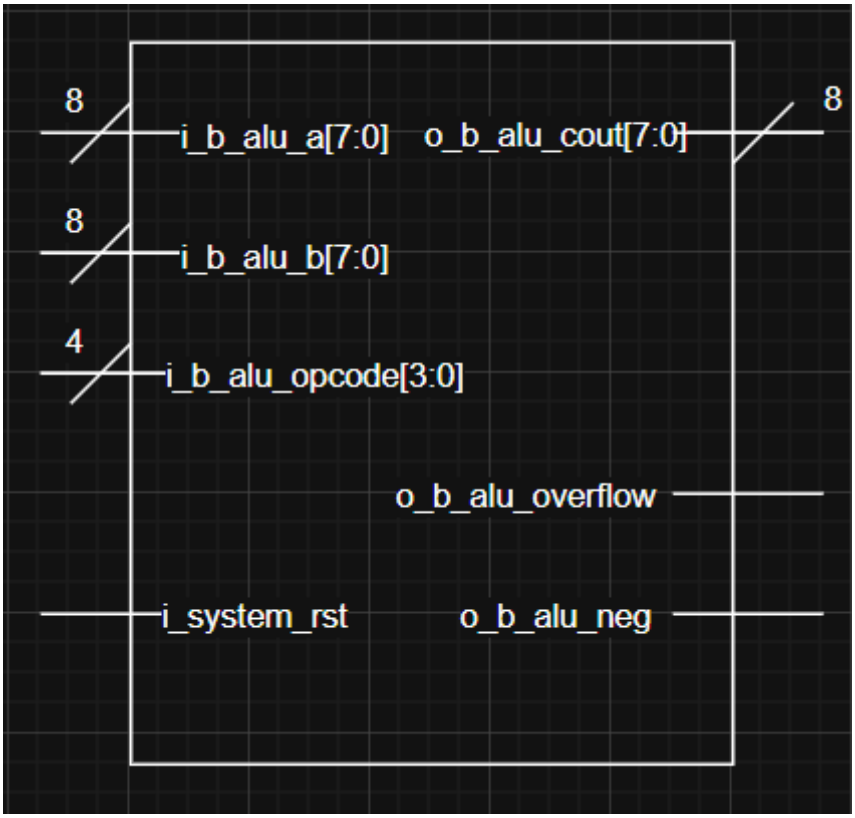
- Multiplication of two 8-bit integers
- Division of two 8-bit integers
- Bitwise AND of two 8-bit integers
- Bitwise OR of two 8-bit integers
- Bitwise XOR of two 8-bit integers
- Bitwise NOT of an 8-bit integer

4.4.2 Features

- Detects overflow for Addition, Multiplication and Division
- Detects negative of Subtraction

4.4.3 Block Interface

ALU Block Interface:



4.4.4 I/O Pin Description

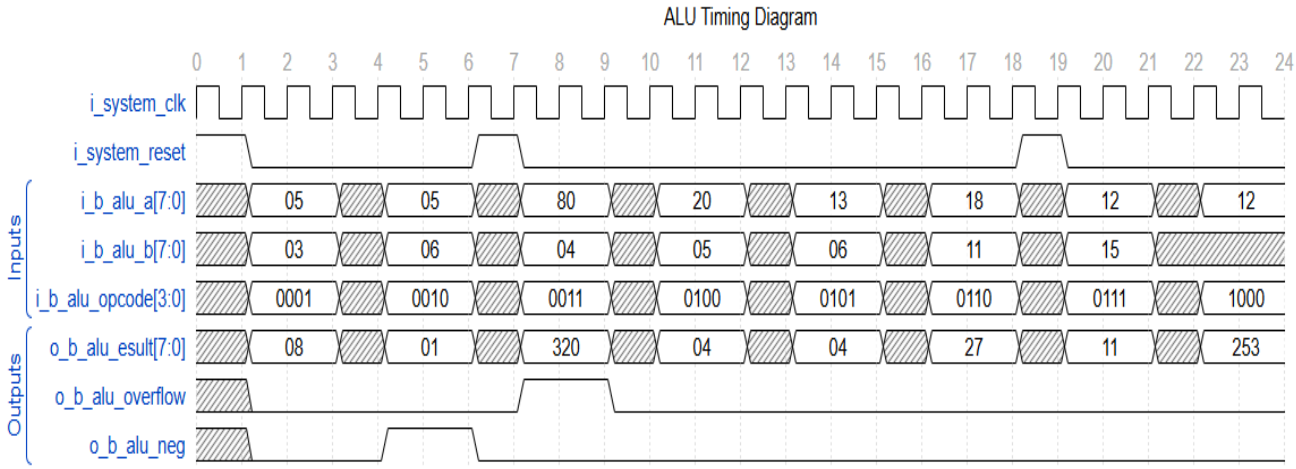
Pin Name	Source To Destination	Pin Direction	Pin Class	Pin Function
i_b_alu_a[7:0]	CON to ALU	Input	data	Transmit the first 8-bit data to ALU
i_b_alu_b [7:0]	CON to ALU	Input	data	Transmit the second 8-bit data to ALU
i_b_alu_opcode [3:0]	CON to ALU	Input	data	Transmit the operation code to the ALU for determine the operation

i_system_rst	RESET button to ALU	Input	Global, control	To reset system
o_b_alu_cout [7:0]	ALU to IOD	Output	data	Transmit the 8-bit result to Display
o_b_alu_overflow	ALU to IOD	Output	data	Transmit the Overflow flag to Display
o_b_alu_neg	ALU to IOD	Output	data	Transmit the Negative flag to Display

4.4.5 Internal operation

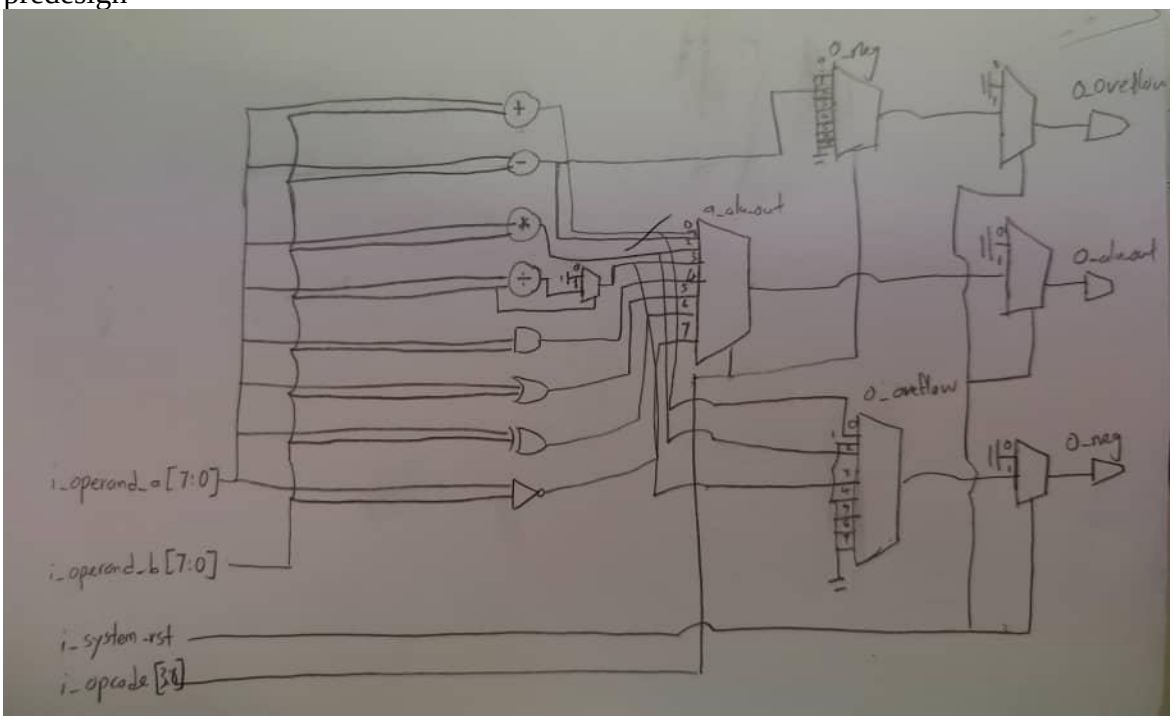
i_b_alu_opcode [3:0]	ALU Function	ALU Implication	Notes
0001	Addition	A + B	If result > 255, overflow=1
0010	Subtraction	A – B	if A < B, neg = 1
0011	Multiplication	A * B	If result > 255, overflow=1
0100	Division	A / B	If B == 0, overflow =1
0101	Bitwise AND	A & B	
0110	Bitwise OR	A | B	
0111	Bitwise XOR	A ^ B	
1000	Bitwise NOT	~A	

4.4.6 Timing Requirement

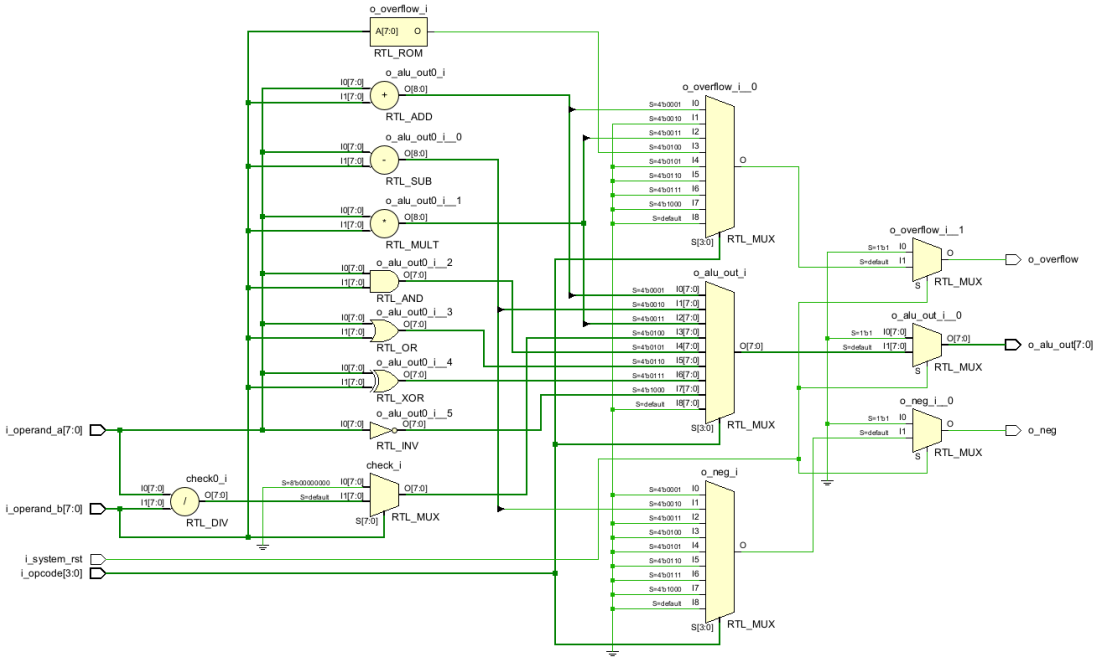


4.4.7 Schematic

predesign



postdesign



4.4.8 Vivado Model

```
//////////////////////////////////////////////////////////////////////////////////
 // Company: UTAR
 // Engineer: SHU MEE ANG
 // Create Date: 05/05/2025
// Module Name: ALU
 // Project Name: ALU
 //////////////////////////////////////////////////////////////////////////////////
module ALU (
output logic [7:0] o_alu_out,
output logic o_neg,
 output logic o_overflow,
 input logic [7:0] i_operand_a,
 input logic [7:0] i_operand_b,
 input logic [3:0] i_opcode,
input logic i_system_rst
 );

logic [7:0] bitwise;
logic [8:0] check;

always_comb begin
    if (i_system_rst) begin
        o_alu_out = 8'b0;
        o_neg = 1'b0;
        o_overflow = 1'b0;
    end else begin
        case (i_opcode)
            4'b0001: begin // A + B
                check = i_operand_a + i_operand_b;
                {o_overflow, o_alu_out} = check;
                o_neg = 1'b0;
            end
            4'b0010: begin // A - B
                check = i_operand_a - i_operand_b;
                {o_neg, o_alu_out} = check;
                o_overflow = 1'b0;
            end
            4'b0011: begin // A * B
                check = i_operand_a * i_operand_b;
                {o_overflow, o_alu_out} = check;
                o_neg = 1'b0;
            end
            4'b0100: begin // A / B
                if (i_operand_b == 0) begin
                    check = 9'b0;
                    o_overflow = 1'b1;
                end else begin
                    check = i_operand_a / i_operand_b;
                    o_overflow = 1'b0;
                end
                o_alu_out = check;
                o_neg = 1'b0;
            end
```

```
            4'b0101: begin // A & B
               bitwise = i_operand_a & i_operand_b;
               o_alu_out = bitwise;
               o_neg = 1'b0;
               o_overflow = 1'b0;
            end
            4'b0110: begin // A | B
               bitwise = i_operand_a | i_operand_b;
               o_alu_out = bitwise;
               o_neg = 1'b0;
               o_overflow = 1'b0;
            end
            4'b0111: begin // A ^ B
               bitwise = i_operand_a ^ i_operand_b;
               o_alu_out = bitwise;
               o_neg = 1'b0;
               o_overflow = 1'b0;
            end
            4'b1000: begin // ~A
               bitwise = ~i_operand_a;
               o_alu_out = bitwise;
               o_neg = 1'b0;
               o_overflow = 1'b0;
            end
            default: begin
               o_alu_out = 8'b0;
               o_neg = 1'b0;
               o_overflow = 1'b0;
            end
         endcase
      end
   end

   endmodule
```

4.4.9 Test Plan

No.	Test Feature / Functionality	Test Vector Generator	Expected Outputs	Pass/ Fail
1	Addition (Normal: 10 + 20)	i_b_alu_a[7:0] = 8'd10, i_b_alu_b[7:0] = 8'd20, i_b_alu_opcode[3:0] = 4'b0001, Wait for 1 clock cycle	o_b_alu_cout[7:0] = 8'd30, o_b_alu_overflow = 0, o_b_alu_neg = 0	pass
2	Addition (Overflow: 200 + 100)	i_b_alu_a[7:0] = 8'd200, i_b_alu_b[7:0] = 8'd100, i_b_alu_opcode[3:0] = 4'b0001, Wait for 1 clock cycle	o_b_alu_cout[7:0] = 8'd44, o_b_alu_overflow = 1, o_b_alu_neg = 0	pass
3	Subtraction (Normal: 50 - 30)	i_b_alu_a[7:0] = 8'd50, i_b_alu_b[7:0] = 8'd30, i_b_alu_opcode[3:0] = 4'b0010, Wait for 1 clock cycle	o_b_alu_cout[7:0] = 8'd20, o_b_alu_overflow = 0, o_b_alu_neg = 0	pass
4	Subtraction (Negative: 30 - 50)	i_b_alu_a[7:0] = 8'd30, i_b_alu_b[7:0] = 8'd50, i_b_alu_opcode[3:0] = 4'b0010, Wait for 1 clock cycle	o_b_alu_cout[7:0] = 8'd236, o_b_alu_overflow = 0, o_b_alu_neg = 1	pass
5	Multiplication (Normal: 10 * 20)	i_b_alu_a[7:0] = 8'd10, i_b_alu_b[7:0] = 8'd20, i_b_alu_opcode[3:0] = 4'b0011, Wait for 1 clock cycle	o_b_alu_cout[7:0] = 8'd200, o_b_alu_overflow = 0, o_b_alu_neg = 0	pass
6	Multiplication (Overflow: 16 * 16)	i_b_alu_a[7:0] = 8'd16, i_b_alu_b[7:0] = 8'd16, i_b_alu_opcode[3:0] = 4'b0011, Wait for 1 clock cycle	o_b_alu_cout[7:0] = 8'd0, o_b_alu_overflow = 1, o_b_alu_neg = 0	pass
7	Division (Normal: 100 /	i_b_alu_a[7:0] = 8'd100, i_b_alu_b[7:0] = 8'd5, i_b_alu_opcode[3:0] =	o_b_alu_cout[7:0] = 8'd20, o_b_alu_overflow = 0,	pass

	5)	4'b0100, Wait for 1 clock cycle	o_b_alu_neg = 0	
8	Division (By Zero: 100 / 0)	i_b_alu_a[7:0] = 8'd100, i_b_alu_b[7:0] = 8'd0, i_b_alu_opcode[3:0] = 4'b0100, Wait for 1 clock cycle	o_b_alu_cout[7:0] = 8'd0, o_b_alu_overflow = 1, o_b_alu_neg = 0	pass
9	Bitwise AND	i_b_alu_a[7:0] = 8'b10101010, i_b_alu_b[7:0] = 8'b01010101, i_b_alu_opcode[3:0] = 4'b0101, Wait for 1 clock cycle	o_b_alu_cout[7:0] = 8'b00000000, o_b_alu_overflow = 0, o_b_alu_neg = 0	pass
10	Bitwise OR	i_b_alu_a[7:0] = 8'b10101010, i_b_alu_b[7:0] = 8'b01010101, i_b_alu_opcode[3:0] = 4'b0110, Wait for 1 clock cycle	o_b_alu_cout[7:0] = 8'b11111111, o_b_alu_overflow = 0, o_b_alu_neg = 0	pass
11	Bitwise XOR	i_b_alu_a[7:0] = 8'b10101010, i_b_alu_b[7:0] = 8'b01010101, i_b_alu_opcode[3:0] = 4'b0111, Wait for 1 clock cycle	o_b_alu_cout[7:0] = 8'b11111111, o_b_alu_overflow = 0, o_b_alu_neg = 0	pass
12	Bitwise NOT	i_b_alu_a[7:0] = 8'b10101010, i_b_alu_b[7:0] = 8'd0, i_b_alu_opcode[3:0] = 4'b1000, Wait for 1 clock cycle	o_b_alu_cout[7:0] = 8'b01010101, o_b_alu_overflow = 0, o_b_alu_neg = 0	pass
13	Addition (Max Value: 255 + 0)	i_b_alu_a[7:0] = 8'd255, i_b_alu_b[7:0] = 8'd0, i_b_alu_opcode[3:0] = 4'b0001, Wait for 1 clock	o_b_alu_cout[7:0] = 8'd255, o_b_alu_overflow = 0, o_b_alu_neg = 0	pass

		cycle		
14	Subtraction (Min Value: 0 - 0)	i_b_alu_a[7:0] = 8'd0, i_b_alu_b[7:0] = 8'd0, i_b_alu_opcode[3:0] = 4'b0010, Wait for 1 clock cycle	o_b_alu_cout[7:0] = 8'd0, o_b_alu_overflow = 0, o_b_alu_neg = 0	pass
15	Multiplication (Zero: 0 * 100)	i_b_alu_a[7:0] = 8'd0, i_b_alu_b[7:0] = 8'd100, i_b_alu_opcode[3:0] = 4'b0011, Wait for 1 clock cycle	o_b_alu_cout[7:0] = 8'd0, o_b_alu_overflow = 0, o_b_alu_neg = 0	pass
16	Division (Zero Num: 0 / 5)	i_b_alu_a[7:0] = 8'd0, i_b_alu_b[7:0] = 8'd5, i_b_alu_opcode[3:0] = 4'b0100, Wait for 1 clock cycle	o_b_alu_cout[7:0] = 8'd0, o_b_alu_overflow = 0, o_b_alu_neg = 0	pass
17	Reset Test	i_system_rst = 1, Hold for 1 clock cycle	o_b_alu_cout[7:0] = 8'd0, o_b_alu_overflow = 0, o_b_alu_neg = 0	pass

4.4.10 TestBench

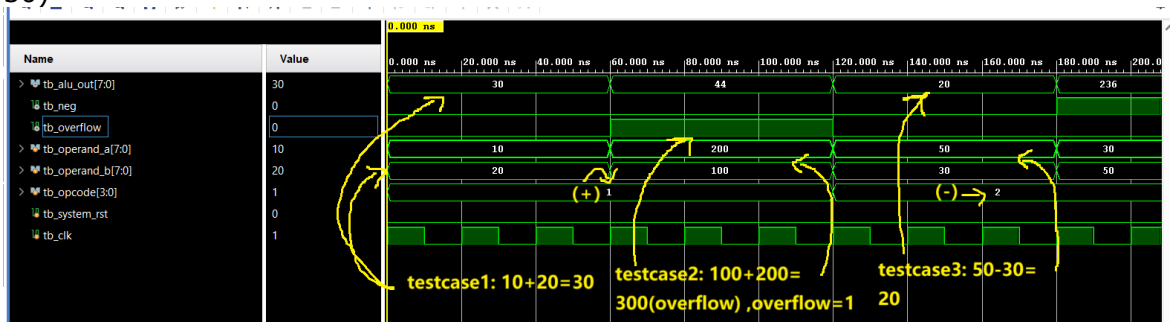
```
//////////////////////////////////////////////////////////////////////////////////
// Company: UTAR
// Engineer: SHU MEE ANG
// Create Date: 05/07/2025
 // Module Name: tbALU
// Project Name: tbALU
//////////////////////////////////////////////////////////////////////////////////

module tb_alu8 ();
wire [7:0] tb_alu_out;
wire tb_neg;
wire tb_overflow;

logic [7:0] tb_operand_a;
logic [7:0] tb_operand_b;
logic [3:0] tb_opcode;
logic tb_system_rst;

ALU
dut_alu
(.o_alu_out (tb_alu_out),
 .o_neg (tb_neg),
 .o_overflow (tb_overflow),
 .i_operand_a (tb_operand_a),
 .i_operand_b (tb_operand_b),
 .i_opcode (tb_opcode),
 .i_system_rst (tb_system_rst));

bit tb_clk = 0;
initial tb_clk <= 1;
always #10 tb_clk = ~tb_clk;

initial begin
    // Initialize inputs
    tb_operand_a <= 8'b0;
    tb_operand_b <= 8'b0;
    tb_opcode <= 4'b0;
    tb_system_rst <= 1'b0;

    // Test 1: Addition (Normal: 10 + 20)
    @(posedge tb_clk);
    tb_operand_a <= 8'd10;
    tb_operand_b <= 8'd20;
    tb_opcode <= 4'b0001;
    repeat(2) @(posedge tb_clk);

    // Test 2: Addition (Overflow: 200 + 100)
    @(posedge tb_clk);
    tb_operand_a <= 8'd200;
    tb_operand_b <= 8'd100;
    tb_opcode <= 4'b0001;
    repeat(2) @(posedge tb_clk);
```

```
// Test 3: Subtraction (Normal: 50 - 30)
@(posedge tb_clk);
tb_operand_a <= 8'd50;
tb_operand_b <= 8'd30;
tb_opcode <= 4'b0010;
repeat(2) @(posedge tb_clk);

// Test 4: Subtraction (Negative: 30 - 50)
@(posedge tb_clk);
tb_operand_a <= 8'd30;
tb_operand_b <= 8'd50;
tb_opcode <= 4'b0010;
repeat(2) @(posedge tb_clk);

// Test 5: Multiplication (Normal: 10 * 20)
@(posedge tb_clk);
tb_operand_a <= 8'd10;
tb_operand_b <= 8'd20;
tb_opcode <= 4'b0011;
repeat(2) @(posedge tb_clk);

// Test 6: Multiplication (Overflow: 16 * 16)
@(posedge tb_clk);
tb_operand_a <= 8'd16;
tb_operand_b <= 8'd16;
tb_opcode <= 4'b0011;
repeat(2) @(posedge tb_clk);

// Test 7: Division (Normal: 100 / 5)
@(posedge tb_clk);
tb_operand_a <= 8'd100;
tb_operand_b <= 8'd5;
tb_opcode <= 4'b0100;
repeat(2) @(posedge tb_clk);

// Test 8: Division (By Zero: 100 / 0)
@(posedge tb_clk);
tb_operand_a <= 8'd100;
tb_operand_b <= 8'd0;
tb_opcode <= 4'b0100;
repeat(2) @(posedge tb_clk);

// Test 9: Bitwise AND
@(posedge tb_clk);
tb_operand_a <= 8'b10101010;
tb_operand_b <= 8'b01010101;
tb_opcode <= 4'b0101;
repeat(2) @(posedge tb_clk);

// Test 10: Bitwise OR
@(posedge tb_clk);
tb_operand_a <= 8'b10101010;
tb_operand_b <= 8'b01010101;
tb_opcode <= 4'b0110;
repeat(2) @(posedge tb_clk);
```

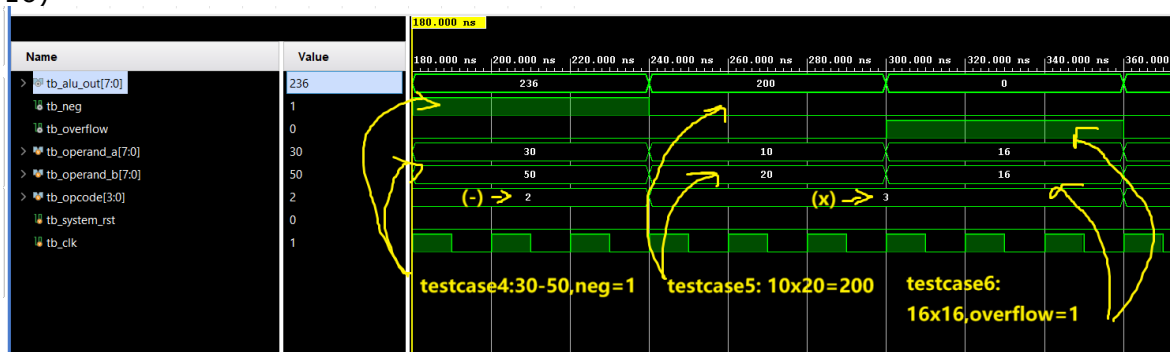
```
    // Test 11: Bitwise XOR
    @(posedge tb_clk);
    tb_operand_a <= 8'b10101010;
    tb_operand_b <= 8'b01010101;
    tb_opcode <= 4'b0111;
    repeat(2) @(posedge tb_clk);

    // Test 12: Bitwise NOT
    @(posedge tb_clk);
    tb_operand_a <= 8'b10101010;
    tb_operand_b <= 8'd0;
    tb_opcode <= 4'b1000;
    repeat(2) @(posedge tb_clk);

    // Test 13: Addition (Max Value: 255 + 0)
    @(posedge tb_clk);
    tb_operand_a <= 8'd255;
    tb_operand_b <= 8'd0;
    tb_opcode <= 4'b0001;
    repeat(2) @(posedge tb_clk);

    // Test 14: Subtraction (Min Value: 0 - 0)
    @(posedge tb_clk);
    tb_operand_a <= 8'd0;
    tb_operand_b <= 8'd0;
    tb_opcode <= 4'b0010;
    repeat(2) @(posedge tb_clk);

    // Test 15: Multiplication (Zero: 0 * 100)
    @(posedge tb_clk);
    tb_operand_a <= 8'd0;
    tb_operand_b <= 8'd100;
    tb_opcode <= 4'b0011;
    repeat(2) @(posedge tb_clk);

    // Test 16: Division (Zero Num: 0 / 5)
    @(posedge tb_clk);
    tb_operand_a <= 8'd0;
    tb_operand_b <= 8'd5;
    tb_opcode <= 4'b0100;
    repeat(2) @(posedge tb_clk);

    // Test 17: Reset Test
    @(posedge tb_clk);
    tb_system_rst <= 1'b1;
    repeat(2) @(posedge tb_clk);
    tb_system_rst <= 1'b0;

    // Delay to settle outputs
    repeat(10) @(posedge tb_clk);
    $stop;
end

endmodule
```

4.4.11 Result Simulation

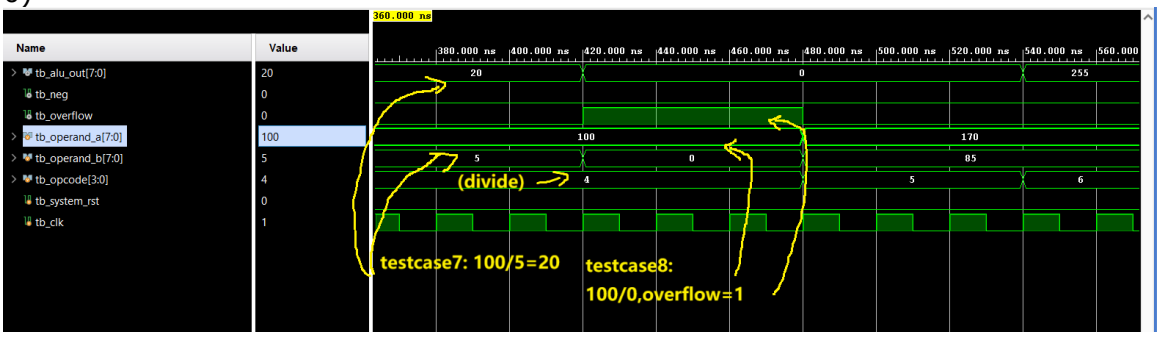
// Test 1: Addition (Normal: 10 + 20)
// Test 2: Addition (Overflow: 200 + 100)
// Test 3: Subtraction (Normal: 50 -
30)



// Test 4: Subtraction (Negative: 30 - 50)
// Test 5: Multiplication (Normal: 10 * 20 ）
// Test 6: Multiplication (Overflow: 16 *
16)

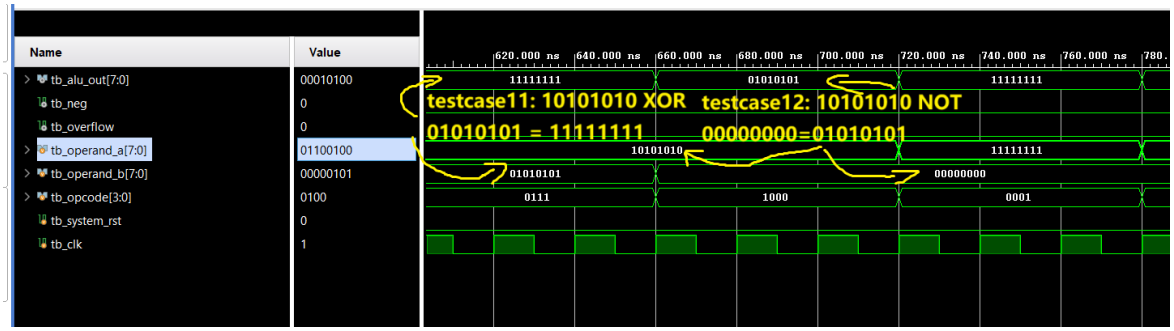


// Test 7: Division (Normal: 100 / 5)
// Test 8: Division (By Zero: 100 /
0)

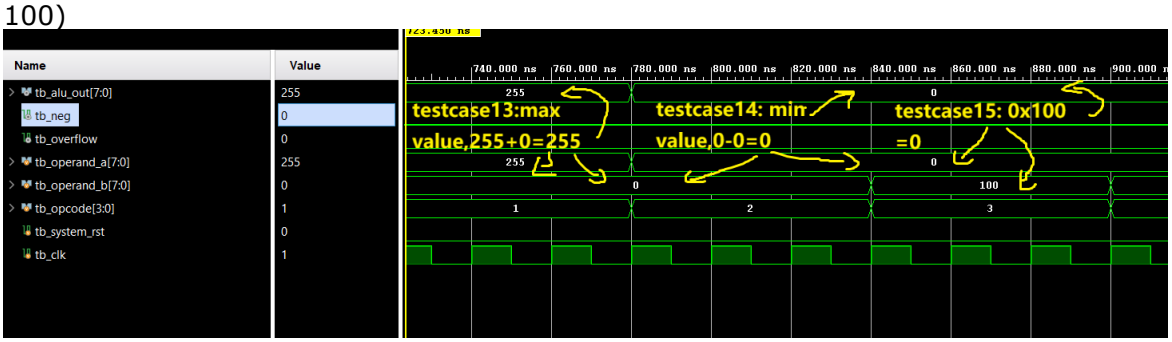


// Test 9: Bitwise AND
// Test 10: Bitwise
OR

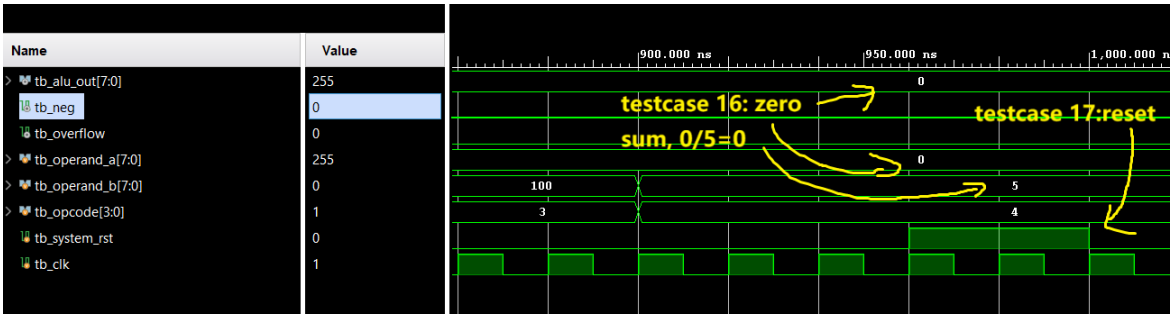
// Test 11: Bitwise XOR
// Test 12: Bitwise
NOT



// Test 13: Addition (Max Value: 255 + 0)
// Test 14: Subtraction (Min Value: 0 - 0)
// Test 15: Multiplication (Zero: 0 *
100)



// Test 16: Division (Zero Num: 0 / 5)
// Test 17: Reset
Test



4.5 Barrel Shifter

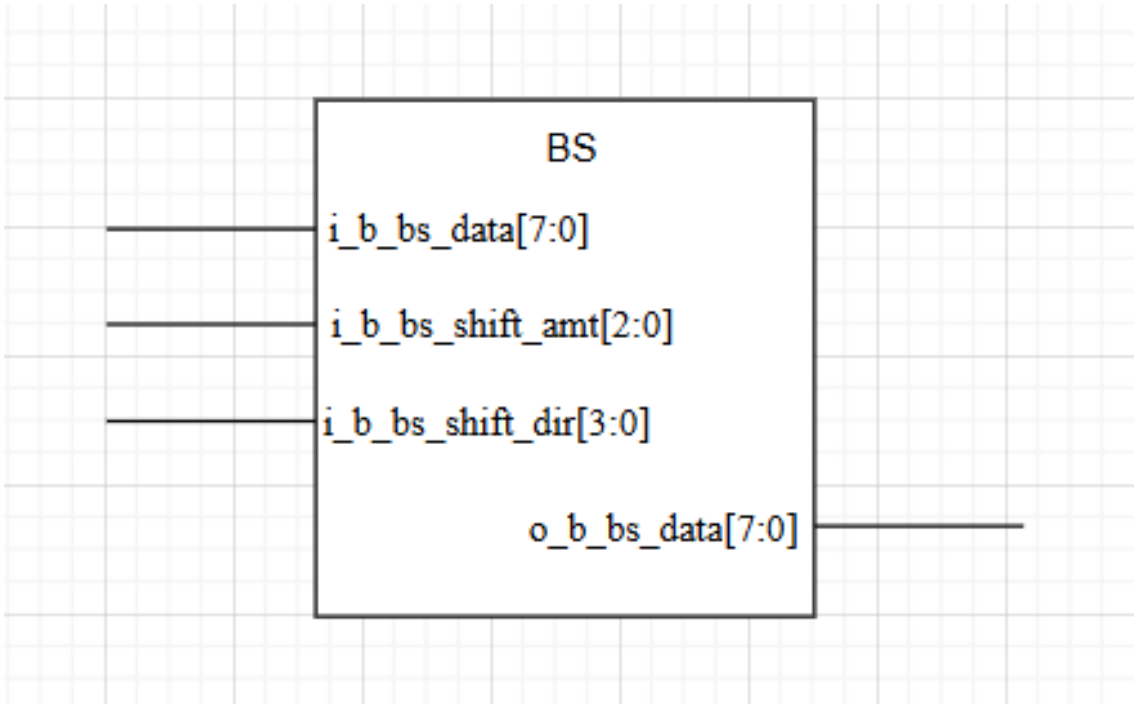
4.5.1 Functionality

- Accepts an 8-bit input word and a 3-bit shift amount.
- Uses a single combinational pass (one “cycle” of logic) to produce the result.
- When shift to the left, moves each bit toward the MSB and fills vacated LSB positions with 0.
- When shift to the right, moves each bit toward the LSB and fills vacated MSB positions with 0.
- Does not use any built-in shift operators
- Implements shifts via a tree of multiplexers.

4.5.2 Features

- Bidirectional: supports both logical left and logical right shifts.
- Variable amount: shift distance from 0 to 7 via a 3-bit control.
- Zero-fill: always inserts zeros into the newly vacated bit positions.
- Purely combinational: no flip-flops or latches—outputs change immediately with inputs.
- Single-cycle latency: the entire shift completes in one layer of logic.
- Mux-tree implementation: uniform, predictable delay across shifting stages.

4.5.3 Block Interface



4.5.4 I/O Pin Description

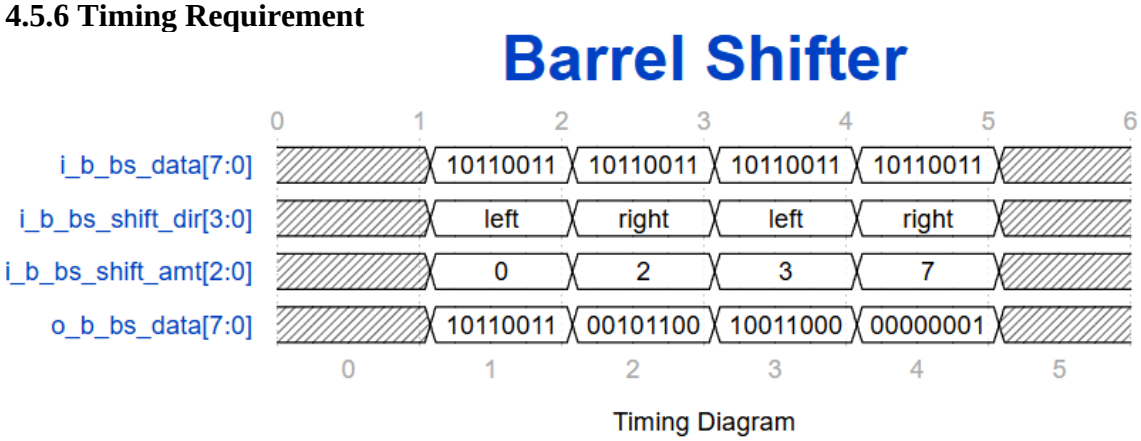
Pin Name	Pin Class	Pin Direction	Source to Destination	Pin Function
i_b_bs_data[7:0]	data	input	CON to BS	8-bit input data to be shifted
i_b_bs_shift_amt[2:0]	data	input	CON to BS	Shift amount of input data
i_b_bs_shift_dir [3:0]	control	input	CON to BS	Shift direction of input data
o_b_bs_data[7:0]	data	output	BS to IOD	8-bit shifted output data

4.4.5 Internal Operation

i_b_bs_data[7:0]	i_b_bs_shift_a mt[2:0]	i_b_bs_shif t_dir[3:0]	o_b_bs_data[7:0]	Function Description
8'b00011111	3'b000 (0 bit)	4'b1001 (left)	8'b00011111	Logical left shift by 0 (No Shift)
8'b00011111	3'b001 (1 bit)	4'b1001	8'b00111110	Logical left shift by 1
8'b00011111	3'b011 (3 bit)	4'b1001	8'b11111000	Logical left shift by 3
8'b00011111	3'b111 (7 bit)	4'b1001	8'b10000000	Logical left shift by 7
8'b11111000	3'b000 (0 bit)	4'b1010 (right)	8'b11111000	Logical right shift by 0 (No Shift)
8'b11111000	3'b001 (1 bit)	4'b1010	8'b01111100	Logical right shift by 1
8'b11111000	3'b011 (3 bit)	4'b1010	8'b00011111	Logical right shift by 3

8’b11111000	3’b111 (7 bit)	4’b1010	8’b00000001	Logical right shift by 7

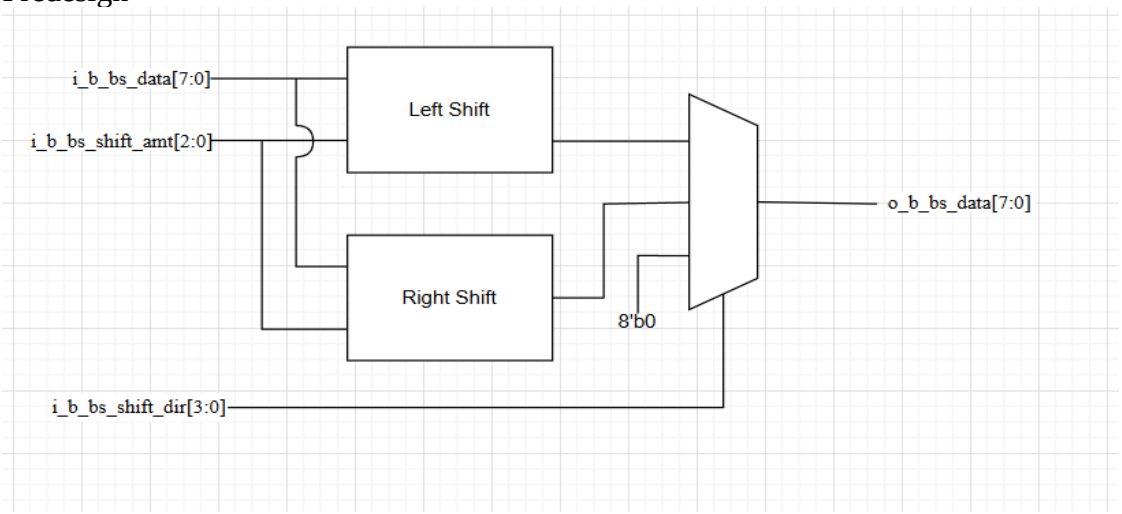
4.5.6 Timing Requirement



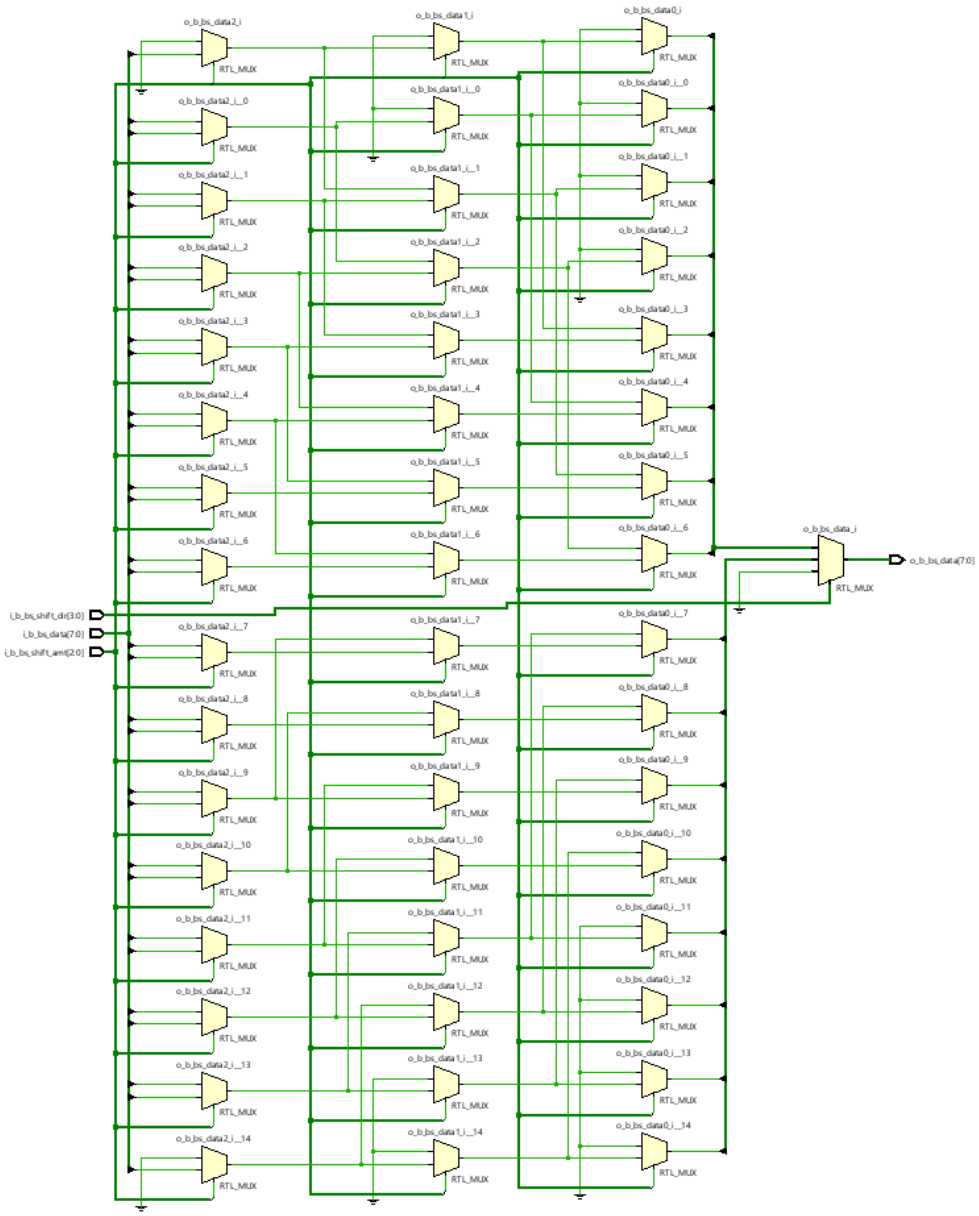
Timing Diagram

4.5.7 Schematic

Predesign



Postdesign



4.5.8 Vivado Model

```
//////////////////////////////////////////////////

//company : UTAR

//Engineer: JAYSON CHEN MIN XIANG

 // CREATE DATE :07/05/2025

//MODULE NAME: BS

//PROJECT NAME :BS

//////////////////////////////////////////////////
module BS
(
   input logic[7:0]    i_b_bs_data,
   input logic[2:0]    i_b_bs_shift_amt,
   input logic[3:0]    i_b_bs_shift_dir,
   output logic[7:0]   o_b_bs_data
);

localparam left = 4'b1001; //the opcode for logical left shift
localparam right = 4'b1010; //the opcode for logical left shift

logic [7:0] temp_left [0:3];
logic [7:0] temp_right [0:3];

// Left shift logic
assign temp_left[0] = i_b_bs_data;
always_comb begin
```

```
    for (int col = 0; col < 3; col++) begin
        for (int row = 0; row < 8; row++) begin
            if (row < (2 ** col)) begin
                temp_left[col+1][row] = i_b_bs_shift_amt[col] ? 1'b0 :
temp_left[col][row];
            end else begin
                temp_left[col+1][row] = i_b_bs_shift_amt[col] ? temp_left[col][row - (2
** col)] : temp_left[col][row];
            end
        end
    end
end

// Right shift logic
assign temp_right[0] = i_b_bs_data;
always_comb begin
    for (int col = 0; col < 3; col++) begin
        for (int row = 0; row < 8; row++) begin
            if (row >= 8 - (2 ** col)) begin
                temp_right[col+1][row] = i_b_bs_shift_amt[col] ? 1'b0 :
temp_right[col][row];
            end else begin
                temp_right[col+1][row] = i_b_bs_shift_amt[col] ? temp_right[col][row
+ (2 ** col)] : temp_right[col][row];
            end
        end
    end
end



always_comb begin
        case (i_b_bs_shift_dir)
```

```
            left:  o_b_bs_data = temp_left[3];
            right: o_b_bs_data = temp_right[3];
            default: o_b_bs_data = 8'b0; // for invalid direction
        endcase
end


endmodule
```

4.5.9 Test Plan

No	Test Case	Description of Test Vector Generation	Expected Operation	Status
1	System initialization	i_b_bs_data[7:0]: 4’b00000000 i_b_bs_shift_amt[2:0]: 3’b000 i_b_bs_shift_dir[3:0]: 4’b0000	-o_b_bs_data[7:0]: 8’b00000000	pass
2	Left shift by 1	i_b_bs_data[7:0]: 4’b00000001 i_b_bs_shift_amt[2:0]: 3’b001 i_b_bs_shift_dir[3:0]: 4’b1001 (left)	o_b_bs_data[7:0]: 8’b00000010	pass
3	Left shift by 7	i_b_bs_data[7:0]: 8’b00000001 i_b_bs_shift_amt[2:0]: 3’b111 i_b_bs_shift_dir[3:0]: 4’b1001 (left)	o_b_bs_data[7:0]: 8’b10000000	pass
4	Right shift by 1	i_b_bs_data[7:0]: 8’b10000000 i_b_bs_shift_amt[2:0]: 3'b111 i_b_bs_shift_dir[3:0]:4’b1010(right)	o_b_bs_data[7:0]: 8’b01000000	pass
5	Right shift by 7	i_b_bs_data[7:0]: 8’b10000000 i_b_bs_shift_amt[2:0]: 3'b111 i_b_bs_shift_dir[3:0]:4’b1010(right)	o_b_bs_data[7:0]: 8’b00000001	pass
6	Pattern left shift by 2	i_b_bs_data[7:0]:8’b10101010 i_b_bs_shift_amt[2:0]: 3'b010 i_b_bs_shift_dir[3:0]:4’b1001(left)	o_b_bs_data[7:0]: 8’b10101000	pass
7	Pattern right shift by 3	i_b_bs_data[7:0]:8’b10101010 i_b_bs_shift_amt[2:0]: 3'b011 i_b_bs_shift_dir[3:0]:4’b1010(right)	o_b_bs_data[7:0]: 8’b00010101	pass
8	No shift	i_b_bs_data[7:0]:8’b11111111 i_b_bs_shift_amt[2:0]: 3'b000	o_b_bs_data[7:0]: 8’b11111111	pass

		i_b_bs_shift_dir[3:0]:4’b1001(right) //wait for 1 clock cycle		
9	Invalid shift direction	i_b_bs_data[7:0]:8’b10101010 i_b_bs_shift_amt[2:0]: 3'b010 i_b_bs_shift_dir[3:0]:4’b1000	o_b_bs_data[7:0]: 8’b00000000	pass
10	No shift and invalid direction	i_b_bs_data[7:0]:8’b10101010 i_b_bs_shift_amt[2:0]: 3'b000 i_b_bs_shift_dir[3:0]:4’b1000	o_b_bs_data[7:0]: 8’b00000000	pass
11	Maximum left shift test	i_b_bs_data[7:0]:8’b11111111 i_b_bs_shift_amt[2:0]: 3'b111 i_b_bs_shift_dir[3:0]:4’b1001(left)	o_b_bs_data[7:0]: 8’b10000000	pass
12	Maximum right shift test	i_b_bs_data[7:0]:8’b11111111 i_b_bs_shift_amt[2:0]: 3'b111 i_b_bs_shift_dir[3:0]:4’b1010(right)	o_b_bs_data[7:0]: 8’b00000001	pass

4.5.10 TestBench

```
//////////////////////////////////////////////////

//company : UTAR

//Engineer: JAYSON CHEN MIN XIANG

 // CREATE DATE :07/05/2025

//MODULE NAME: tb_BS

//PROJECT NAME :tb_BS

//////////////////////////////////////////////////
`define PERIOD_CLK 10

module BS_tb;
```

```
// Inputs
logic [3:0]   i_b_bs_shift_dir;
logic [7:0]   i_b_bs_data;
logic [2:0]   i_b_bs_shift_amt;



// Outputs
wire [7:0]    o_b_bs_data;

// Instantiate DUT
BS dut (
   .i_b_bs_shift_dir  (i_b_bs_shift_dir),
   .i_b_bs_data       (i_b_bs_data),
   .i_b_bs_shift_amt  (i_b_bs_shift_amt),
   .o_b_bs_data       (o_b_bs_data)
);

// Test stimulus

initial begin

// Default inputs

i_b_bs_shift_dir = 4'b0000;

i_b_bs_data = 8'h00;

i_b_bs_shift_amt = 3'b000;

// Wait out reset
```

```
#(`PERIOD_CLK * 5);

// 2) Left shift by 1
i_b_bs_shift_dir = 4'b1001;
i_b_bs_data      = 8'b00000001;
i_b_bs_shift_amt = 3'b001;
#(`PERIOD_CLK * 2);

// 3) Left shift by 7
i_b_bs_data      = 8'b00000001;
i_b_bs_shift_amt = 3'b111;
#(`PERIOD_CLK * 2);

// 4) Right shift by 1
i_b_bs_shift_dir = 4'b1010;
i_b_bs_data      = 8'b10000000;
i_b_bs_shift_amt = 3'b001;
#(`PERIOD_CLK * 2);

// 5) Right shift by 7
i_b_bs_data      = 8'b10000000;
i_b_bs_shift_amt = 3'b111;
#(`PERIOD_CLK * 2);

// 6) Pattern left shift by 2
i_b_bs_shift_dir = 4'b1001;
i_b_bs_data      = 8'b10101010;
i_b_bs_shift_amt = 3'b010;
#(`PERIOD_CLK * 2);

// 7) Pattern right shift by 3
```

```
i_b_bs_shift_dir = 4'b1010;
i_b_bs_shift_amt = 3'b011;
#(`PERIOD_CLK * 2);

// 8) No shift
i_b_bs_shift_dir = 4'b1001;
i_b_bs_shift_amt = 3'b000;
i_b_bs_data      = 8'hFF;
#(`PERIOD_CLK * 2);

// 9) invalid shift direction
i_b_bs_shift_dir = 4'b1000;
i_b_bs_data      = 8'b10101010;
i_b_bs_shift_amt = 3'b010;
#(`PERIOD_CLK * 2);

// 10) no shift and invalid direction
i_b_bs_shift_dir = 4'b1000;
i_b_bs_data      = 8'b10101010;
i_b_bs_shift_amt = 3'b000;
#(`PERIOD_CLK * 2);

// 11) Maximum shift test (Left and Right)
i_b_bs_shift_dir = 4'b1001;
i_b_bs_data      = 8'b11111111;
i_b_bs_shift_amt = 3'b111;
#(`PERIOD_CLK * 2);

i_b_bs_shift_dir = 4'b1010;
i_b_bs_data      = 8'b11111111;
i_b_bs_shift_amt = 3'b111;
```

```
#(`PERIOD_CLK * 2);


$finish;


end


endmodule
```

4.5.11 Test Result Timing Diagram

Each figure provided represents a test case that is provided in the **4.4.9 test plan.** The figures provide the name, their values and waveform of the input and output of their respective test case.

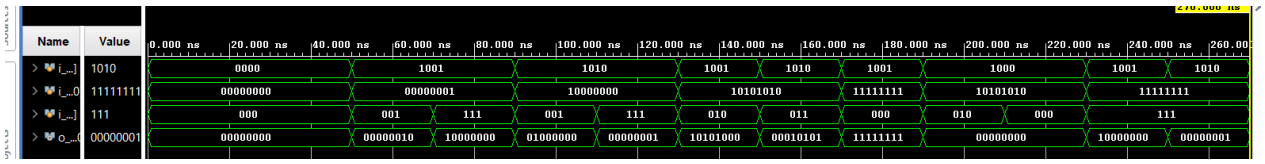


Figure 4.4.11.1: The Entire overall Timing Diagram

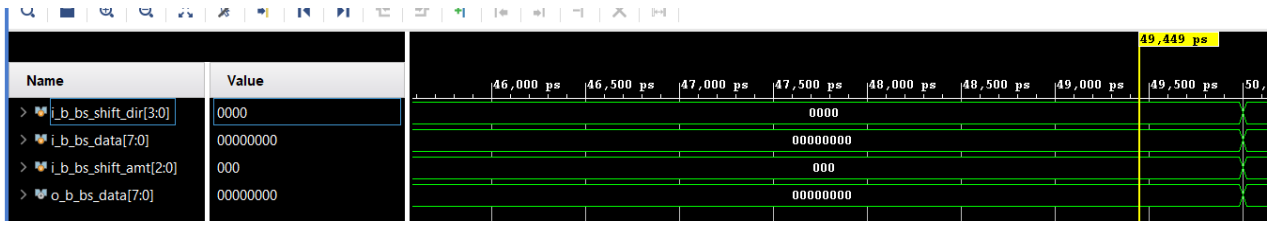


Figure 4.4.11.2: System Initialization

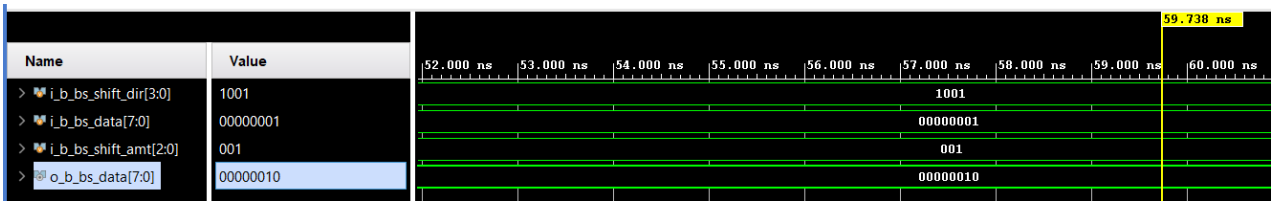


Figure 4.4.11.3: left shift by 1

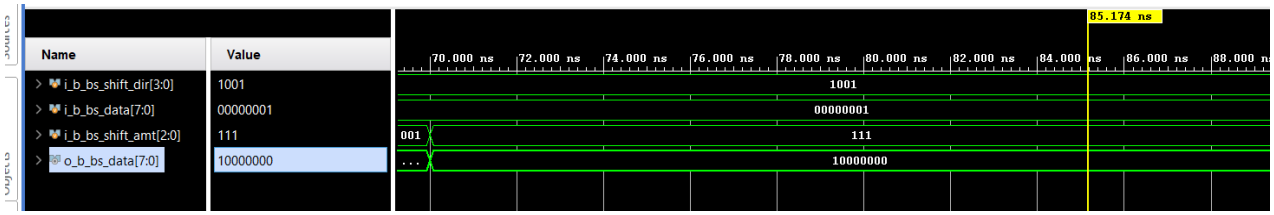


Figure 4.4.11.4: left shift by 7

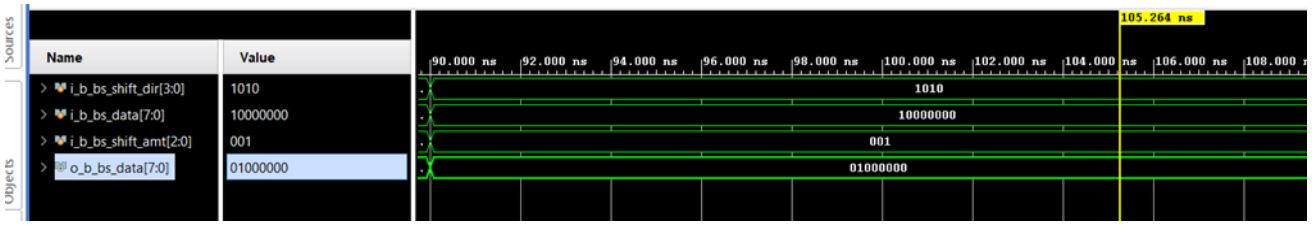


Figure 4.4.11.5: right shift by 1

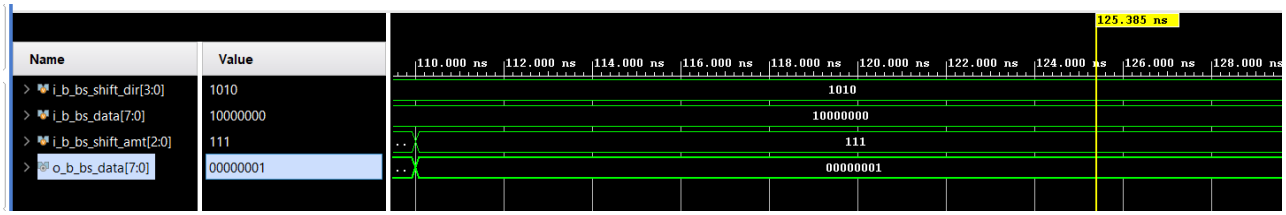


Figure 4.4.11.6: right shift by 7

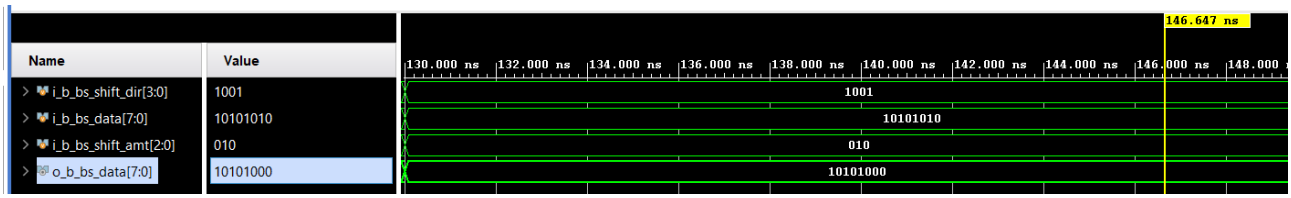


Figure 4.4.11.7: pattern left shift by 2

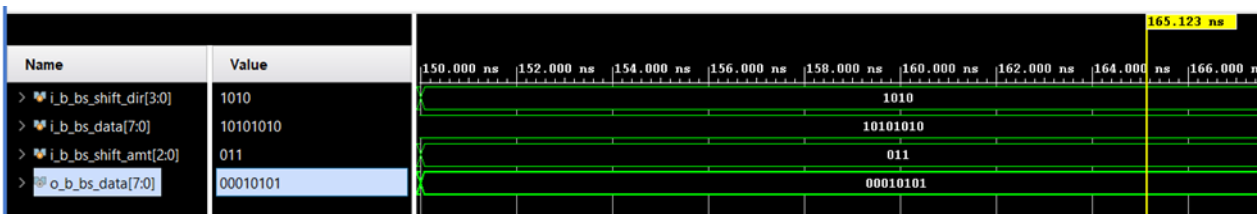


Figure 4.4.11.8: pattern right shift by 3

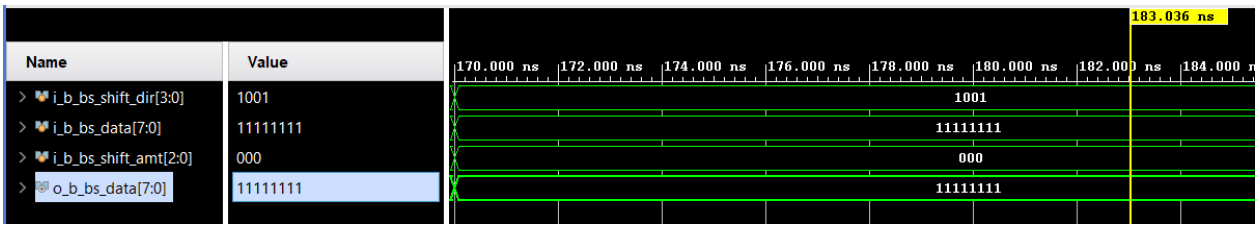


Figure 4.4.11.9: no shift

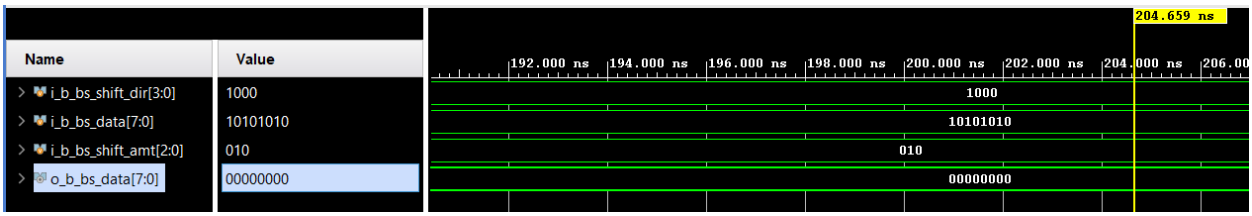


Figure 4.4.11.10: invalid shift direction

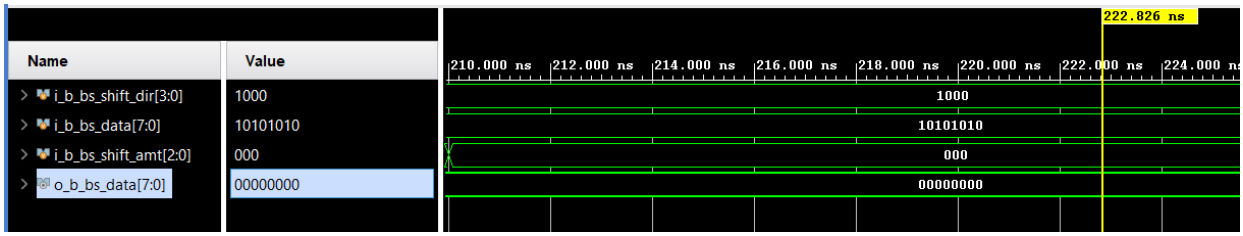


Figure 4.4.11.11: no shift and invalid shift direction

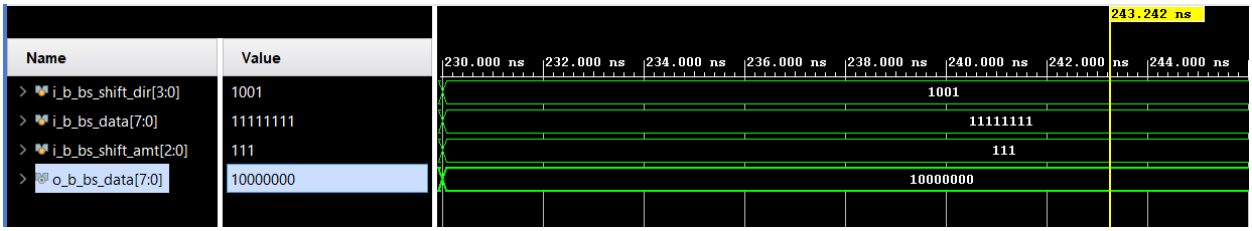


Figure 4.4.11.12: maximum left shift

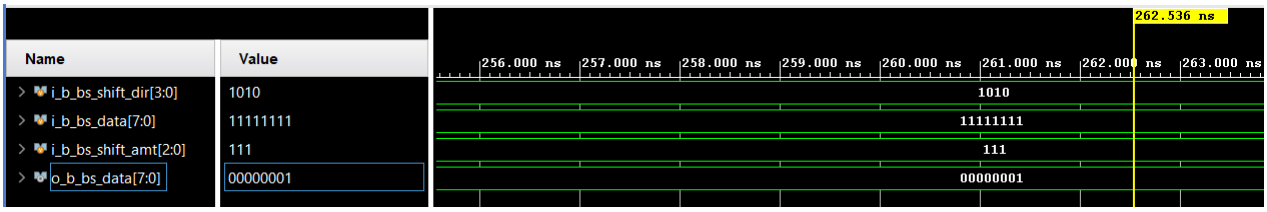


Figure 4.4.11.13: maximum right shift

4.6 Input / Output Conversion

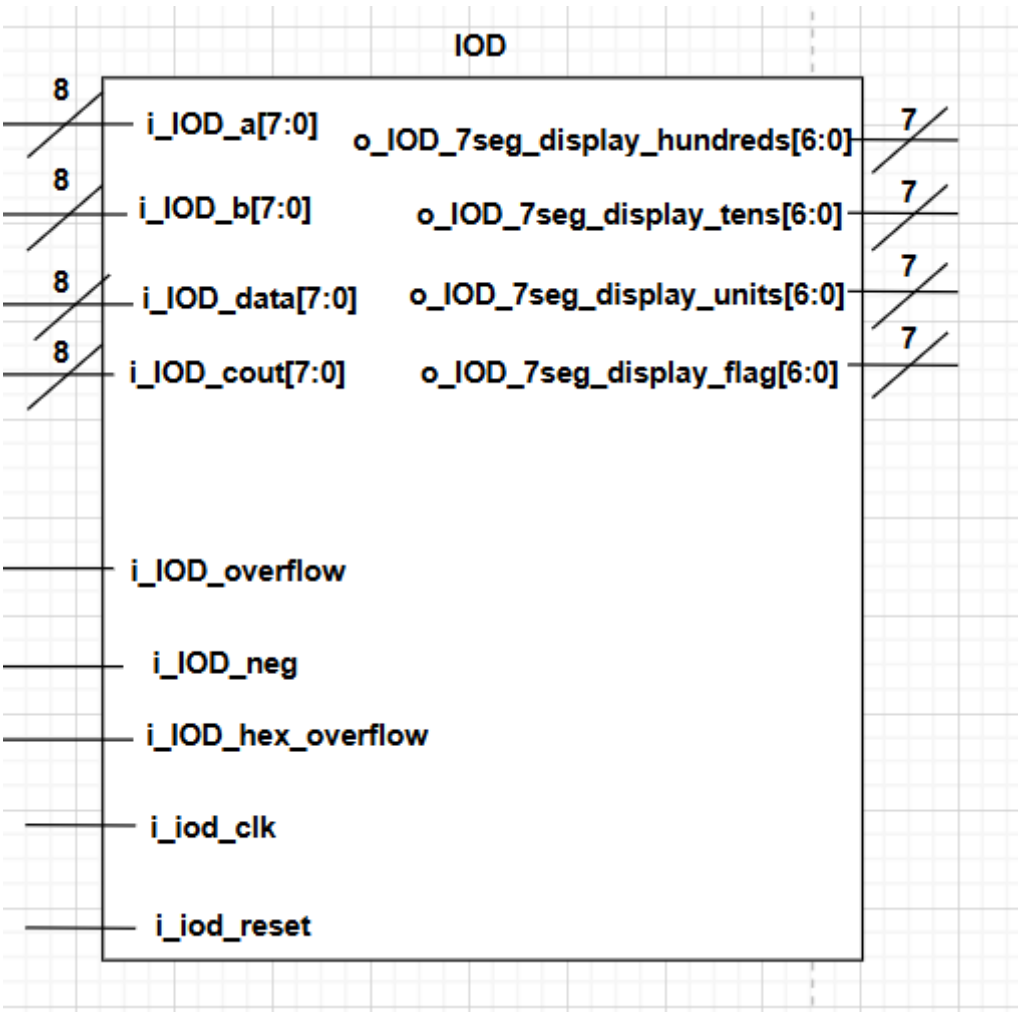
4.6.1 Functionality

- Display Input Digit by Digit
- convert Binary into BCD
- Convert BCD into 7Segment Display Decoder
- Error Handling for Results
- Real-Time Synchronization
- Reset Mechanism

4.6.2 Features

- Operates with 8-bit Integers
- Supports Input and Output via Hex Keypad and 7-Segment Display
- Binary Conversion Capabilities
- Error Handling
- Sign Handling
- Screen Clear Mechanism

4.6.3 Block Interface



4.6.4 I/O Pin Description

Pin Name	Pin Class	Pin Direction	Source to direction	Pin Function
i_b_iod_a	data	input	CON to IOD	Carry 8-bit binary input from the hexpad
I_b_iod_b	data	input	CON to IOD	Carry 8-bit binary inpu from hexpad
i_b_iod_cout	data	input	ALU to IOD	Computation result from the ALU
I_b_iod_data	data	input	BS to IOD	Computation result from the barrier shifter
i_b_iod_opcode	data	input	CON to IOD	Determine to display result from ALU or display result from barrier shifter
i_b_iod_overflow	data	input	ALU to IOD	Overflow flag from the ALU indicating arithmetic overflow.
i_b_iod_neg	data	input	ALU to IOD	Negative result flag from the ALU indicating a subtraction result is negative.
i_b_iod_hex_overflow	control	input	CON to IOD	Overflow flag from the hexpad indicating arithmetic overflow.
i_system_clk	global, control	input	Clock to IOD	Clock signal for synchronizing operations.
i_system_reset	global, control	input	Reset button to IOD	Asynchronous reset signal to clear stored results and reset the system.
o_b_iod_7seg_hundreds	data	output	IOD to 7-segment display	Display signal for the hundreds digit on the 7-segment display.
o_b_iod_7seg_tens	data	output	IOD to 7-segment display	Display signal for the tens digit on the 7-segment display.
o_b_iod_7seg_units	data	output	IOD to 7-segment display	Display signal for the units digit on the 7-segment display.
o_b_iod_7seg_flags	data	output	IOD to 7-segment display	Signal to indicate errors (e.g., overflow, negative symbol, etc.).

4.6.5 Internal operation

Binary code from hexpad

Key	binary code
+	4’b0001
-	4’b0010
*	4’b0011
/	4’b0100
&	4’b0101
|	4’b0110
^	4’b0111
~	4’b1000
<<	4’b1001
>>	4’b1010
First key in value	4’b1111
Second key in value	4’b1110
No key	4’b0000

7-segment display table

7-Segment Output (7-bit)	**Display**
0111111	0
0000110	1
1011011	2
1001111	3
1100110	4
1101101	5
1111101	6
0000111	7
1111111	8
1101111	9
1000000	-
1111001	E
0000000	default

Example table

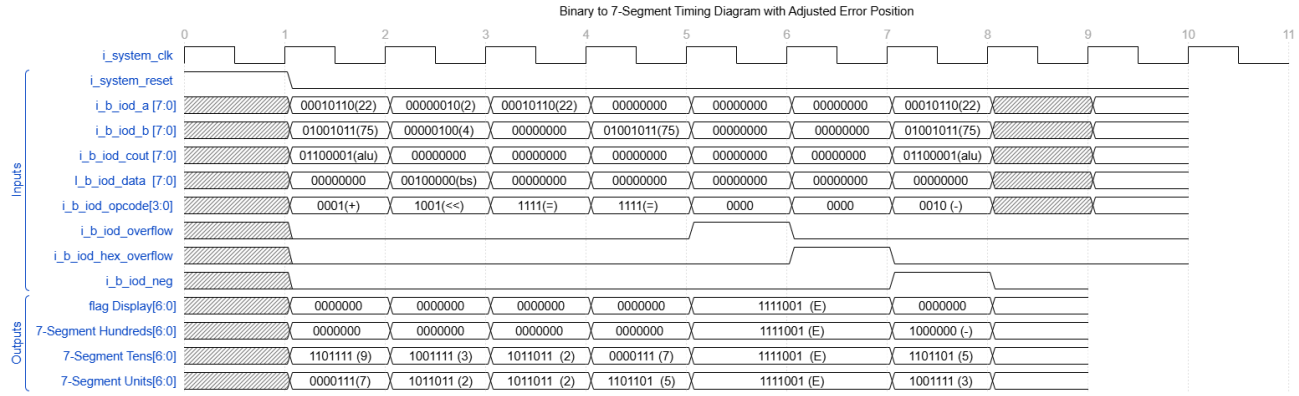
	22+75	**2<<4**	**Display a(22)**	**Display b(75)**
i_b_iod_a	8b’00010110(22)	8b’00000010(2)	8b’00010110(22)	8b’0000000
I_b_iod_b	8b’01001011(75)	8b’00000100(4)	8b'00000000	8b’01001011(75)
i_b_iod_cout	8b’01100001(97)	8b'00000000	8b'00000000	8b'00000000
I_b_iod_data	8b'00000000	8b’00100000(32)	8b'00000000	8b'00000000
i_b_iod_opcode	4’b0001(+)	4’b1001(<<)	4’b1111	4’b1110

i_b_iod_overflow	0	0	0	0
i_b_iod_neg	0	0	0	0
i_b_iod_hex_overflow	0	0	0	0
i_system_reset	0	0	0	0
o_b_iod_7seg_flags	0000000	0000000	0000000	0000000
o_b_iod_7seg_hundreds	0111111(0)	0111111(0)	0111111(0)	0111111(0)
o_b_iod_7seg_tens	1101111 (9)	1001111 (3)	1011011 (2)	0000111 (7)
o_b_iod_7seg_units	0000111(7)	1011011 (2)	1011011 (2)	1101101 (5)

Example table

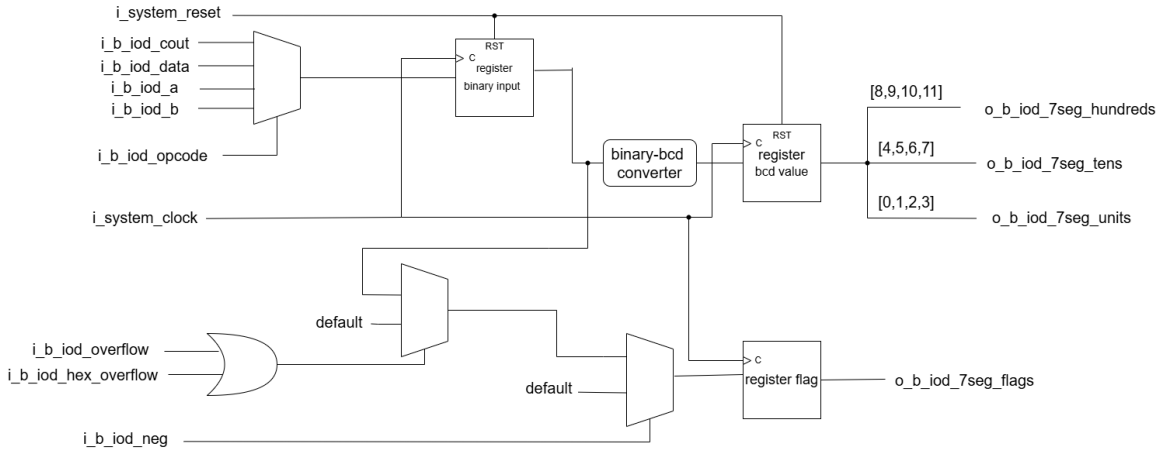
	Hex_overflow	**overflow**	**Neg(22-75)**	**reset**
i_b_iod_a	8b'00000000	8b'00000000	8b’00010110(22)	8b'00000000
I_b_iod_b	8b'00000000	8b'00000000	8b’01001011(75)	8b'00000000
i_b_iod_cout	8b'00000000	8b'00000000	8b’01100001 (53)	8b'00000000
I_b_iod_data	8b'00000000	8b'00000000	8b'00000000	8b'00000000
i_b_iod_opcode	4’b0000	4’b0000	4’b 0010(-)	4’b0000
i_b_iod_overflow	0	1	0	0
i_b_iod_neg	0	0	1	0
i_b_iod_hex_overflow	1	0	0	0
i_system_reset	0	0	0	1
o_b_iod_7seg_flags	1111001(E)	1111001(E)	1000000(-)	0000000
o_b_iod_7seg_hundreds	0111111(0)	0111111 (0)	0111111 (0)	0000000
o_b_iod_7seg_tens	0111111 (0)	0111111 (0)	1101101 (5)	0000000
o_b_iod_7seg_units	0111111 (0)	0111111 (0)	1001111(3)	0000000

4.6.6 Timing Requirement

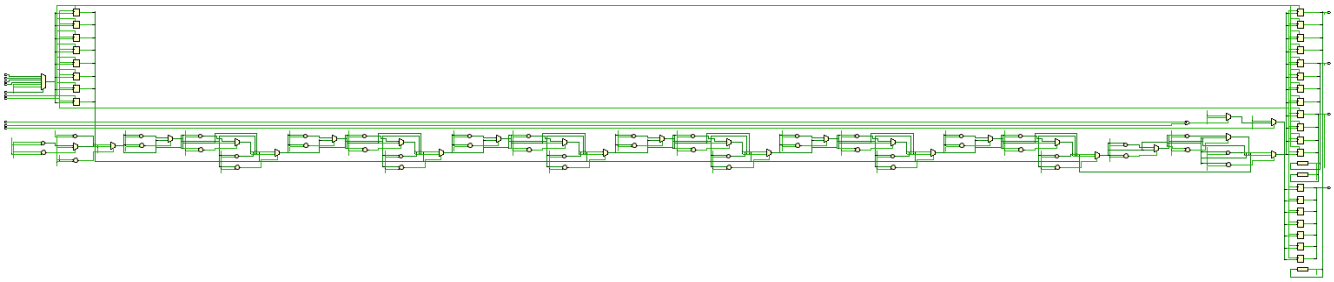


4.6.7 Schematic

Pre schematic



Post schematic



4.6.8 Vivado Model

```
//////////////////////////////////////////////////////
//company : UTAR
//Engineer: CHONG BING HONG
 // CREATE DATE :07/05/2025
//MODULE NAME:IOD
//PROJECT NAME :IOD
//////////////////////////////////////////////////////

module iod(
```

```
//input
input i_system_clk,
input i_system_reset,
input i_b_iod_overflow,
input i_b_iod_neg,
input i_b_iod_hex_overflow,
input [7:0] i_b_iod_a,
input [7:0] i_b_iod_b,
input [7:0] i_b_iod_data,
input [7:0] i_b_iod_cout,
input [3:0] i_b_iod_opcode,

//output
output logic[6:0] o_b_iod_7seg_hundreds,
output logic[6:0] o_b_iod_7seg_tens,
output logic[6:0] o_b_iod_7seg_units,
output [6:0] o_b_iod_7seg_flags);

//reg
logic [7:0] binary_input;
logic [19:0] shift_reg;
logic [6:0] flag_display;
logic [11:0] bcd_value;

// select a, b, result ALU, result barrirer
// Select computation result based on opcode

always_ff @(posedge i_system_clk) begin
        if (i_system_reset) binary_input <= 8'h0;
else begin

        case (i_b_iod_opcode)
                4'b0001, 4'b0010, 4'b0011, 4'b0100, 4'b0101,
                4'b0110, 4'b0111, 4'b1000: binary_input <= i_b_iod_cout;
                4'b1001, 4'b1010: binary_input <= i_b_iod_data;
                4'b1111: binary_input <= i_b_iod_a;
                4'b1110: binary_input <= i_b_iod_b;
                4'b0000: binary_input <= 8'b00000000;
                default: binary_input <= 8'b00000000;
        endcase
        end
end

//binary to bcd
always_ff @(posedge i_system_clk) begin
if (i_system_reset) begin
bcd_value <= 12'b000000000000;
flag_display <= 7'b0000000; end else begin

// Initialize variables
integer i;
shift_reg = {12'b000000000000, binary_input};

        // Shift & Add-3 algorithm for Binary to BCD conversion
        for (i = 0; i < 8; i = i + 1) begin
```

```
        // Correct BCD digits BEFORE shifting
          if (shift_reg[19:15] >= 5)
            shift_reg[19:15] = shift_reg[19:15] + 3;
          if (shift_reg[15:12] >=5 )
            shift_reg[15:12] = shift_reg[15:12] + 3;
          if (shift_reg[11:8] >= 5)
            shift_reg[11:8] = shift_reg[11:8] + 3;

        // Shift left by 1 bit while retaining the LSB
          shift_reg = {shift_reg[18:0], binary_input[7-i]};
    end

    // Assign final BCD values
    bcd_value <= shift_reg[19:8];

end
    // Handling Negative and Overflow
    if (i_b_iod_neg)
        flag_display <= 7'b1000000; // "-" sign
    else if (i_b_iod_overflow || i_b_iod_hex_overflow)
        flag_display <= 7'b1111001; // "E" error signal
    else
        flag_display <= 7'b0000000; // Default no error
end

always_comb begin

//7-seg output
 case (bcd_value[11:8])
4'b0000: o_b_iod_7seg_hundreds = 7'b0111111; // "0"
4'b0001: o_b_iod_7seg_hundreds = 7'b0000110; // "1"
4'b0010: o_b_iod_7seg_hundreds = 7'b1011011; // "2"
4'b0011: o_b_iod_7seg_hundreds = 7'b1001111; // "3"
4'b0100: o_b_iod_7seg_hundreds = 7'b1100110; // "4"
 4'b0101: o_b_iod_7seg_hundreds = 7'b1101101; // "5"
4'b0110: o_b_iod_7seg_hundreds = 7'b1111101; // "6"
4'b0111: o_b_iod_7seg_hundreds = 7'b0000111; // "7"
4'b1000: o_b_iod_7seg_hundreds = 7'b1111111; // "8"
4'b1001: o_b_iod_7seg_hundreds = 7'b1101111; // "9"
default: o_b_iod_7seg_hundreds = 7'b0000000; // Blank display
endcase

case (bcd_value[7:4])
    4'b0000: o_b_iod_7seg_tens = 7'b0111111;
    4'b0001: o_b_iod_7seg_tens = 7'b0000110;
    4'b0010: o_b_iod_7seg_tens = 7'b1011011;
    4'b0011: o_b_iod_7seg_tens = 7'b1001111;
    4'b0100: o_b_iod_7seg_tens = 7'b1100110;
    4'b0101: o_b_iod_7seg_tens = 7'b1101101;
    4'b0110: o_b_iod_7seg_tens = 7'b1111101;
    4'b0111: o_b_iod_7seg_tens = 7'b0000111;
    4'b1000: o_b_iod_7seg_tens = 7'b1111111;
    4'b1001: o_b_iod_7seg_tens = 7'b1101111;
    default: o_b_iod_7seg_tens = 7'b0000000;
endcase
```

```
case (bcd_value[3:0])
   4'b0000: o_b_iod_7seg_units = 7'b0111111;
   4'b0001: o_b_iod_7seg_units = 7'b0000110;
   4'b0010: o_b_iod_7seg_units = 7'b1011011;
   4'b0011: o_b_iod_7seg_units = 7'b1001111;
   4'b0100: o_b_iod_7seg_units = 7'b1100110;
   4'b0101: o_b_iod_7seg_units = 7'b1101101;
   4'b0110: o_b_iod_7seg_units = 7'b1111101;
   4'b0111: o_b_iod_7seg_units = 7'b0000111;
   4'b1000: o_b_iod_7seg_units = 7'b1111111;
   4'b1001: o_b_iod_7seg_units = 7'b1101111;
   default: o_b_iod_7seg_units = 7'b0000000;
endcase

end

//assign value(not necessarily)
assign o_b_iod_7seg_hundreds = bcd_value[11:8];
assign o_b_iod_7seg_tens = bcd_value[7:4];
assign o_b_iod_7seg_units = bcd_value[3:0];
 assign o_b_iod_7seg_flags = flag_display;


endmodule
```

4.6.9 Test Plan

No	Test Case	Description of Test Vector Generation	Expected Operation	Status
1	**Reset**	i_b_iod_reset = 1, Wait for 4 clock cycle, I_b_iod_reset=0	//display o_iod_7seg_flags=7'b0000000, o_iod_7seg_hundreds=7'b0000000, o_iod_7seg_tens=7'b0000000, o_iod_7seg_units=7'b0000000	pass
2	Display first number	/a=110, I_b_opcode= 4'b1111, I_b_iod_a= 8'b01101110, //Wait for 10 clock cycle	//display o_b_iod_7seg_flags=7'b0000000, o_b_iod_7seg_hundreds=7'b0000110, o_b_iod_7seg_tens =7'b0000110, o_b_iod_7seg_units =7'b0111111	pass

3	Display second number	//b=28,
		
		i_b_iod_opcode = 4'b1110;
		i_b_iod_b=8'b00011100;
		
		//Wait for 10 clock cycle
			//display
			o_b_iod_7seg_flags=7'b0000000,
			o_b_iod_7seg_hundreds=7'b0111111,
			o_b_iod_7seg_tens =7'b1011011,
			o_b_iod_7seg_units =7'b1111111,
				pass

		i_b_iod_b=8'b00000000;		

4	ALU operation	//cout=125,opcode=-
		
		tb_b_iod_opcode = 4'b0010,
		
		tb_b_iod_cout= 8'b1111101,
		
		Wait for 10 clock cycle,
		
		i_b_iod_opcode = 4'b0000; i_b_iod_cout= 8'b00000000;
			//display
			o_b_iod_7seg_flags=7'b0000000,
			o_b_iod_7seg_hundreds=7'b0000110,
			o_b_iod_7seg_tens =7'b1011011,
			o_b_iod_7seg_units =7'b1101101,
				pass

5	Barrier shifter operation	//data=114, opcode=<<
		
		i_b_iod_opcode =4'b1001; i_b_iod_data= 8'b1110010;
		
		//Wait for 10 clock cycle
		
		i_b_iod_opcode = 4'b0000; i_b_iod_data= 8'b00000000;
			//display
			o_b_iod_7seg_flags=7'b0000000,
			o_b_iod_7seg_hundreds=7'b0000110,
			o_iod_7seg_tens =7'b0000110,
			o_iod_7seg_units =7'b1100110,
				pass

6	**Negative Result**	//neg=1,
		
		//opcode=-,cout=50
		
		i_b_iod_neg =1;
		i_b_iod_opcode = 4'b0010; i_b_iod_cout= 8'b00110010;
		
		//Wait for 2 clock cycle
		
		i_b_iod_neg =0;
		i_b_iod_opcode = 4'b0000; i_b_iod_cout= 8'b00000000;
			//display
			o_b_iod_7seg_flags=7'b1000000,
			o_b_iod_7seg_hundreds=7'b0000110,
			o_b_iod_7seg_tens =7'b0000110,
			o_b_iod_7seg_units =7'b1100110,
				pass

7	**Hexpad**	i_b_iod_hex_overflow=1	//display 'E'	pass

	overflow	//Wait for 2 clock cycle
		i_b_iod_hex_overflow=0
			o_b_iod_7seg_flags=7'b1111001,
			o_b_iod_7seg_hundreds=7'b0111111,
			o_iod_7seg_tens = 7'b0111111,
			o_iod_7seg_units = 7'b0111111	

8	**Overflow**	i_b_iod_overflow = 1
		//Wait for 2 clock cycle
		i_b_iod_overflow = 0
			//display 'E'
			o_b_iod_7seg_flags=7'b1111001,
			o_b_iod_7seg_hundreds=7'b0111111,
			o_iod_7seg_tens = 7'b011111,
			o_iod_7seg_units = 7'b0111111
				pass

4.6.10 TestBench

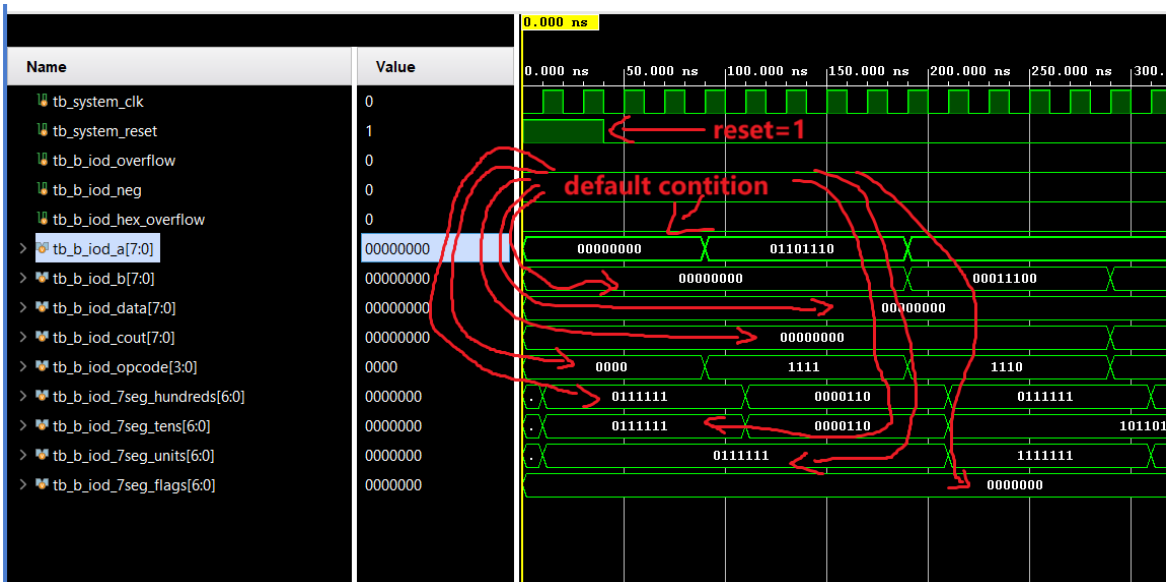
```
//////////////////////////////////////////////////
//COMPANY:UTAR
//ENGINEER:CHONG BING HONG
 //CREATE DATE: 07/05/2025
//MODULE NAME:tb_iod
//PROJECT NAME:tb_iod
//////////////////////////////////////////////////

`define PERIOD_CLK 10

module tb_iod();

logic tb_system_clk;
logic tb_system_reset;
logic tb_b_iod_overflow;
logic tb_b_iod_neg;
logic tb_b_iod_hex_overflow;
logic [7:0] tb_b_iod_a;
logic [7:0] tb_b_iod_b;
logic [7:0] tb_b_iod_data;
logic [7:0] tb_b_iod_cout;
logic [3:0] tb_b_iod_opcode;
logic [6:0] tb_b_iod_7seg_hundreds;
logic [6:0] tb_b_iod_7seg_tens;
logic [6:0] tb_b_iod_7seg_units;
logic [6:0] tb_b_iod_7seg_flags;

iod
dut_iod
(

.i_system_clk(tb_system_clk),
.i_system_reset(tb_system_reset),
.i_b_iod_overflow(tb_b_iod_overflow),
.i_b_iod_neg(tb_b_iod_neg),
.i_b_iod_hex_overflow(tb_b_iod_hex_overflow),
.i_b_iod_a(tb_b_iod_a),
.i_b_iod_b(tb_b_iod_b),
.i_b_iod_data(tb_b_iod_data),
.i_b_iod_cout(tb_b_iod_cout),
.i_b_iod_opcode(tb_b_iod_opcode),
.o_b_iod_7seg_hundreds(tb_b_iod_7seg_hundreds), .o_b_iod_7seg_tens(tb_b_iod_7
seg_tens),
.o_b_iod_7seg_units(tb_b_iod_7seg_units),
```

```
.o_b_iod_7seg_flags(tb_b_iod_7seg_flags)
);

initial tb_system_clk=0;
always #(`PERIOD_CLK) tb_system_clk <= ~tb_system_clk;

initial begin
//default tb_system_reset = 0;
tb_b_iod_overflow = 0;
tb_b_iod_neg = 0;
tb_b_iod_hex_overflow=0;
tb_b_iod_a=8'b00000000;
tb_b_iod_b=8'b00000000;
tb_b_iod_data=8'b00000000;
tb_b_iod_cout=8'b00000000;
tb_b_iod_opcode=4'b0000;
tb_b_iod_7seg_hundreds=7'b0000000;
tb_b_iod_7seg_tens=7'b0000000;
tb_b_iod_7seg_units=7'b0000000;
tb_b_iod_7seg_flags=7'b0000000;

//testcase1, reset
tb_system_reset = 1;
#(` PERIOD_CLK * 4);
tb_system_reset = 0;
#(` PERIOD_CLK * 5);

//testcase2, A=(110）
tb_b_iod_opcode = 4'b1111;
tb_b_iod_a=8'b01101110;
#(`PERIOD_CLK * 10);
tb_b_iod_a=8'b00000000;

//testcase3, B=(28）
tb_b_iod_opcode = 4'b1110;
tb_b_iod_b=8'b00011100;
#(`PERIOD_CLK * 10);
tb_b_iod_b=8'b00000000;

//testcase4, ALU result(125), opcode(-)
tb_b_iod_opcode = 4'b0010;
tb_b_iod_cout= 8'b1111101;
#(`PERIOD_CLK * 10);
tb_b_iod_opcode = 4'b0000;
tb_b_iod_cout= 8'b00000000;

//testcase5, shift result(114),opcode=(<<)
tb_b_iod_opcode =4'b1001;
tb_b_iod_data= 8'b1110010;
#(`PERIOD_CLK * 10);
tb_b_iod_opcode = 4'b0000;
tb_b_iod_data= 8'b00000000;

//testcase6, neg(1),ALU result(50),opcode(-)

tb_b_iod_neg =1;
tb_b_iod_opcode = 4'b0010;
```

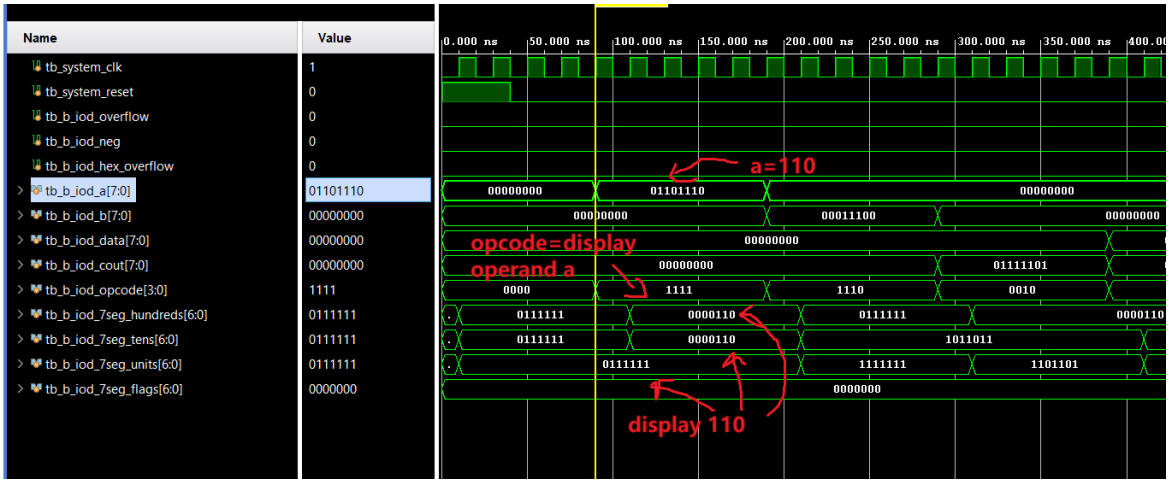
```
tb_b_iod_cout= 8'b00110010;
#(`PERIOD_CLK * 2);
tb_b_iod_neg =0;
tb_b_iod_opcode = 4'b0000;
tb_b_iod_cout= 8'b00000000;

//testcase7, overflow
tb_b_iod_hex_overflow = 1;
#(`PERIOD_CLK * 2);
tb_b_iod_hex_overflow = 0;

//testcase8, ALU overflow
tb_b_iod_overflow = 1;
#(`PERIOD_CLK * 2);
tb_b_iod_overflow = 0;

//End simulation
#(`PERIOD_CLK*100);
$finish;
end
endmodule
```

4.6.11 Test Result Timing Diagram

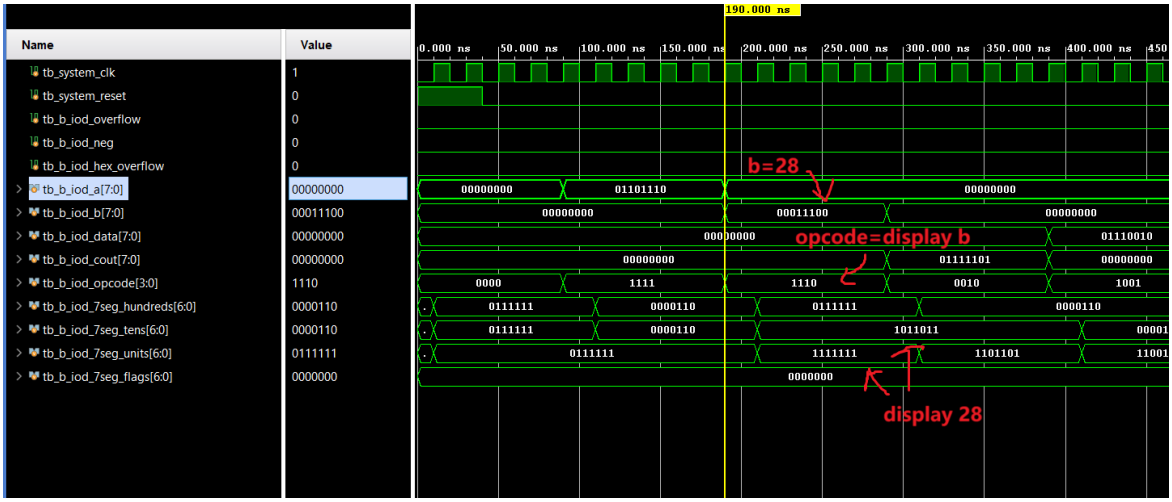
//testcase1, reset



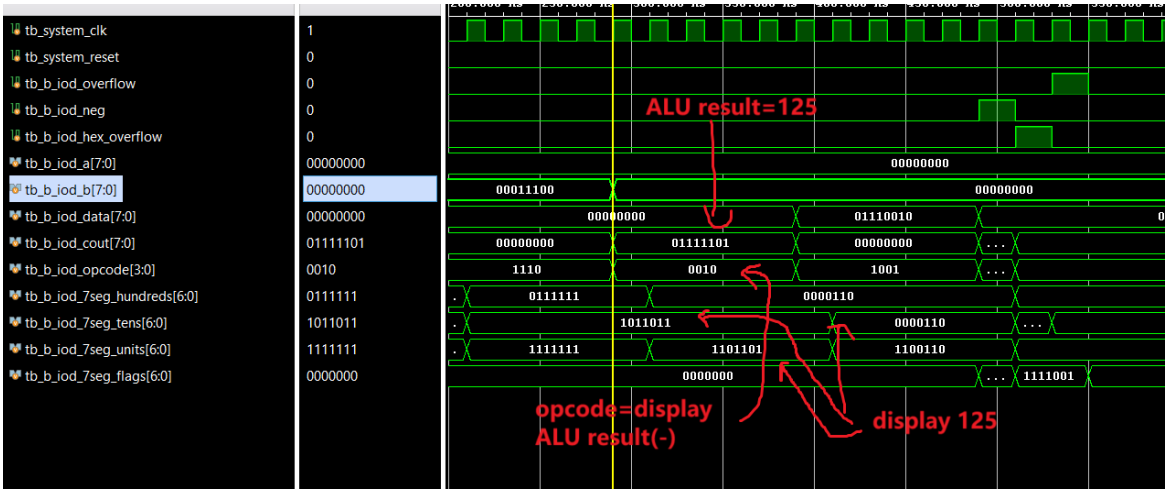
//testcase2, A=(110）



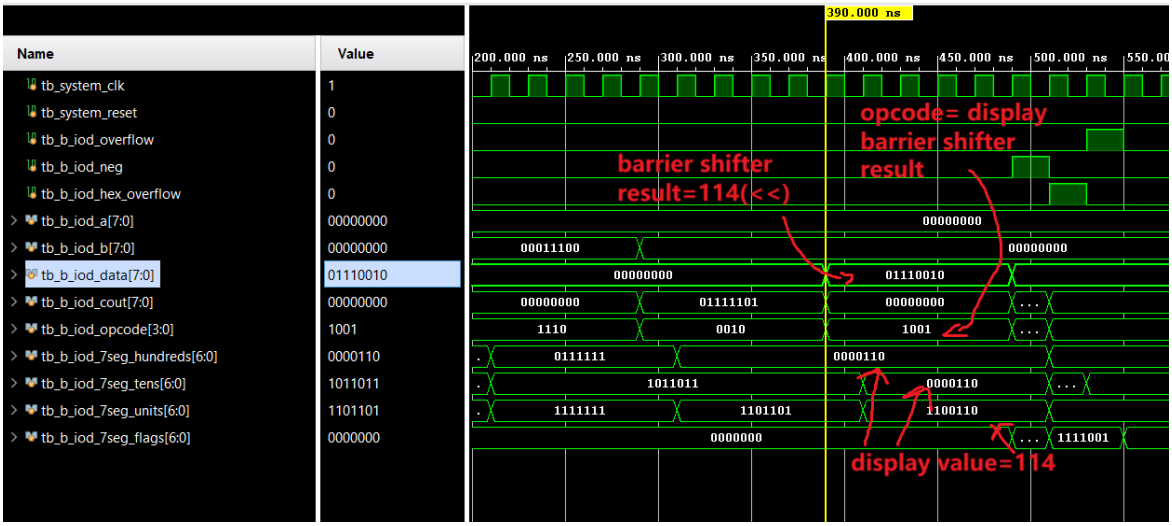
//testcase3, B=(28）



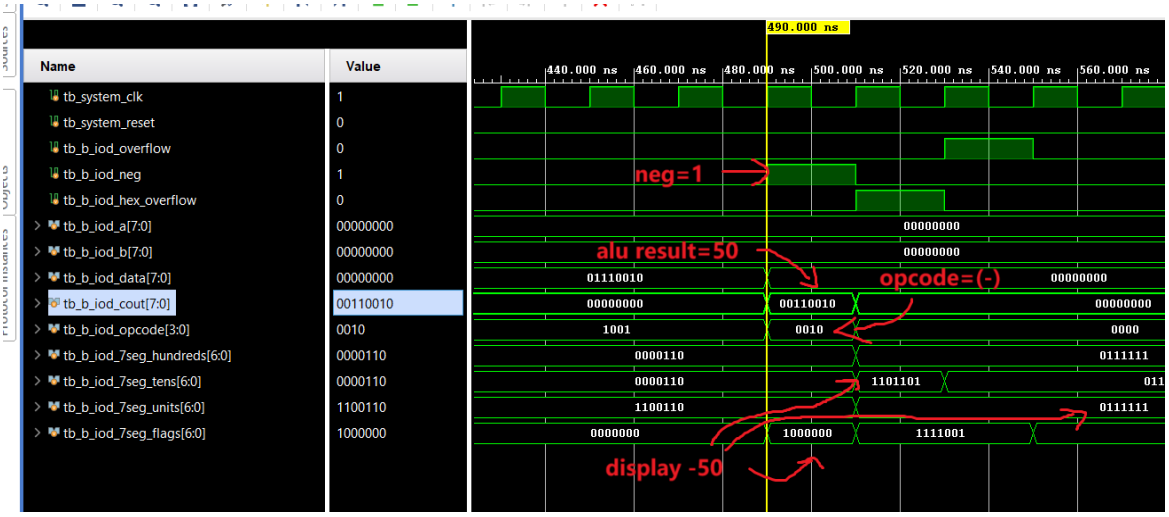
//testcase4, ALU result(125), opcode(-)



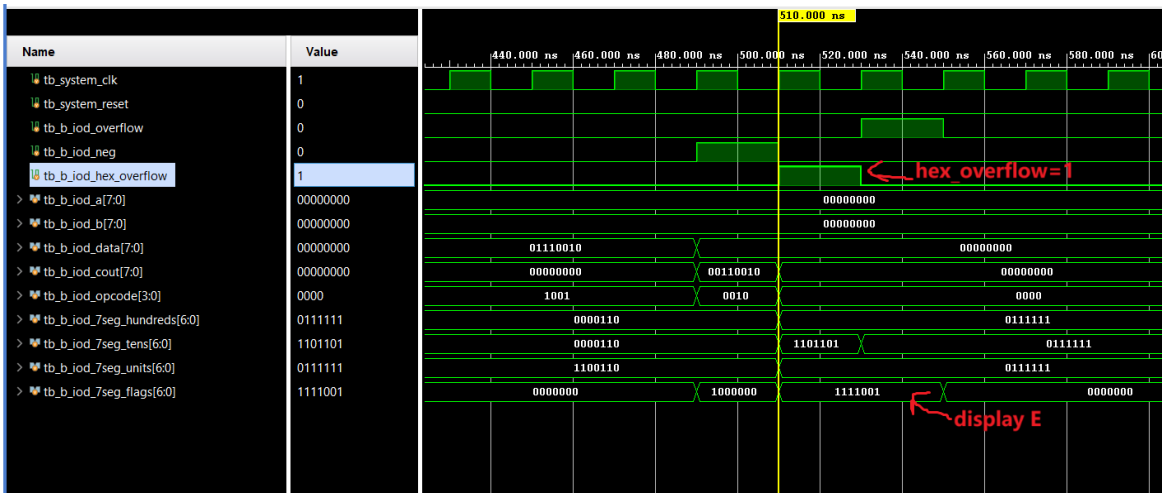
//testcase5, shift result(114),opcode=(<<)



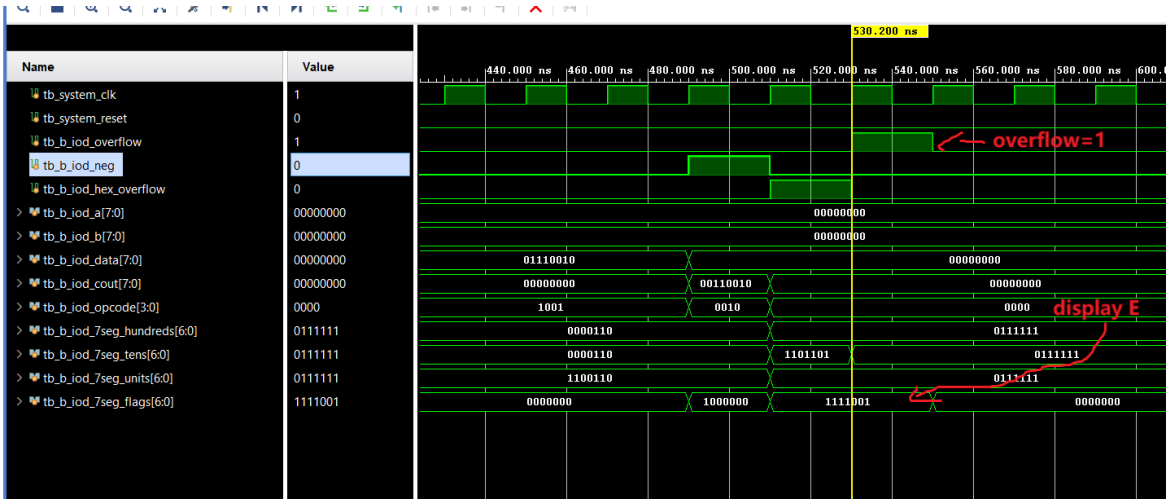
//testcase6, neg(1),ALU result(50),opcode(-)



//testcase7, overflow



//testcase8, ALU overflow


Name
Value
tb_system_clk
tb_system_reset
tb_b_iod_overflow
tb_b_iod_neg
tb_b_iod_hex_overflow
tb_b_iod_a[7:0]
tb_b_iod_b[7:0]
tb_b_iod_data[7:0]
tb_b_iod_cout[7:0]
tb_b_iod_opcode[3:0]
tb_b_iod_7seg_hundreds[6:0]
tb_b_iod_7seg_tens[6:0]
tb_b_iod_7seg_units[6:0]
tb_b_iod_7seg_flags[6:0]
1
0
1
0
0
00000000
00000000
00000000
00000000
0000
0111111
0111111
0111111
1111001
530.200 ns
440.000 ns
460.000 ns
480.000 ns
500.000 ns
520.000 ns
540.000 ns
560.000 ns
580.000 ns
overflow=1
display E