Military College of Signals

National University of Sciences & Technology

DIGITAL SYSTEM DESIGN

Asst. Prof. Dr. Hussain Ali

OPEN-ENDED LAB REPORT

4-Bit ALU Design with LCD Interfacing on Spartan-3E FPGA

Submitted To: Lab Engr. Muhammad Hammad

Submission Date:	31st May, 2023

Group Member Details:

S. No	Names	Section
1.	Fareeha Zaidi – 359343	57B
2.	Ayesha Rauf – 334270	
3.	Obaidullah Ahmed – 334328	

EC-418 Digital System Design

RUBRICS for Practical Implementation of Digital Circuits

Student Name	Fareeha Zaidi	Ayesha Rauf	Obaidullah Ahmed	
R1 (3)				
R2 (3)				
R3 (3)				
R4 (3)				
R5 (3)				
Total (15)				

RUBRICS for Design presentation (Report) and viva

R1 (3)				
R2 (3)				
R3 (3)				
R4 (3)				
Total (12)				
Grand Total (27)				

Digital System Design

Open-Ended Lab Report

4-Bit ALU Design with LCD Interfacing on Spartan-3E FPGA

Introduction:

The purpose of this lab project was to design and implement a 4-bit Arithmetic Logic Unit (ALU) on the Spartan 3E Field-Programmable Gate Array (FPGA) using the Verilog hardware description language. The ALU was intended to perform various arithmetic and logical operations on two 4-bit inputs, denoted as A and B, and produce an 8-bit output. The output was then displayed simultaneously on both an LCD display and the on-board LEDs of the FPGA.

One of the challenges encountered in this project was the **limited availability of slide switches** for input. To overcome this limitation, a solution was devised by designing registers for each input, allowing the use of a reduced number of input buttons to accommodate all necessary inputs. By implementing these registers, we were able to utilize the available resources efficiently and maintain functionality despite the constraints.

In this report, we will provide a detailed explanation of the design and implementation of the 4-bit ALU, Verilog code, and the functionality of the registers. Additionally, the results and observations obtained from testing the ALU will be presented and analyzed. By investigating the performance and functionality of the implemented design, we can evaluate the effectiveness of our approach and discuss potential areas for improvement.

This project allowed us to delve into the intricacies of FPGA programming and provided valuable insights into the challenges and considerations involved in designing hardware systems with limited input resources.

Introduction to Spartan 3E FPGA Starter Board:

The Spartan 3E FPGA Starter Board is a versatile and user-friendly development platform designed to facilitate the exploration and implementation of digital circuits and systems.

This FPGA offers a generous amount of programmable logic cells, dedicated digital signal processing (DSP) slices, and abundant block RAM, allowing for the implementation of complex digital systems. The Spartan-3E FPGA is known for its affordability, making it an excellent choice for educational settings and projects with budget constraints.

The starter board features a variety of input and output interfaces, including switches, buttons, LEDs, seven-segment displays, VGA connectors, and an LCD display. These interfaces provide convenient means of interacting with the FPGA and showcasing the outputs or results of implemented designs.

The Spartan 3E FPGA Starter Board is accompanied by a comprehensive set of development tools and resources. These include a user-friendly integrated development environment (IDE), such as Xilinx ISE which provides a graphical interface for designing, simulating, and programming the FPGA.

Design Approach:

The design of the 4-bit ALU on the Spartan 3E FPGA involved a systematic approach that consisted of three key modules: the register memory module, the mid-level ALU module, and the top-level module for interfacing with the LCD display.

The first step in the design process was the creation of the register memory module. This module was responsible for storing the inputs, denoted as A and B, by utilizing registers. By implementing registers for each input, we were able to overcome the limitation of having only four slide switches available for input. The register memory module facilitated the storage and retrieval of the input values during the ALU operations.

The mid-level ALU module was the core component of the design and performed all the necessary calculations on the inputs stored in the register modules. Using the Verilog hardware description language, we implemented the required arithmetic and logical operations, such as addition, subtraction, AND, OR, and XOR. The mid-level ALU module utilized the inputs from the register memory module and generated the 8-bit output based on the selected opcode.

To visualize the inputs, opcode, and results, we created a top-level module that initialized and interfaced the 2x16 LCD display with the FPGA. This module facilitated the communication between the ALU and the LCD display, ensuring that the relevant information was accurately displayed. By integrating the LCD display into the design, we provided a convenient visual representation of the inputs and outputs, enhancing the overall usability and functionality of the 4-bit ALU.

By following this design approach, we were able to successfully implement a functional 4-bit ALU on the Spartan 3E FPGA. This modular approach allowed for easier development, testing, and troubleshooting, as each module had distinct responsibilities and could be individually verified before integration. The use of Verilog as the hardware description language enabled us to precisely define the behavior and interactions of the modules, resulting in a robust and efficient design.

Step 1: Designing Register Memory Module

```
module register_mem(clk, clr, ld, D_in, D_out);
input clk, clr, ld;
input [3:0] D_in;
output reg [3:0] D_out;

always@(posedge clk) begin
if(clr) begin
D_out <= 0;
end
else if (ld) begin
D_out <= D_in;
end
end

endmodule

```

The module **register_mem** has five ports:

- **clk** is the clock input, used to synchronize the operations within the module.
- **clr** is the clear input, which when asserted (active high) clears the register.
- **ld** is the load input, which when asserted (active high) loads the input data into the register.
- **D_in** is a 4-bit input, representing the data to be loaded into the register.
- **D_out** is a 4-bit output, representing the data stored in the register.

Inside the module, there is an always block, which is executed whenever there is a positive edge of the clk signal. The purpose of this block is to define the behavior of the module based on the input conditions.

- If **clr** is asserted (high), the **D_out** register is cleared to 0.
- If **ld** is asserted (high), the value of **D_in** is loaded into the **D_out** register.

In other words, the module acts as a simple 4-bit register with a synchronous reset (clr) and a load signal (ld). It stores the data present at D_in on the rising edge of the clock signal (clk) when the ld signal is active. The stored data is then provided as output through D_out. When the clr signal is active, the register is cleared to 0.

This module can be instantiated and used as a component in other Verilog designs to create larger circuits or systems that require the storage of data values.

Step 2: Designing Arithmetic Logic Unit Module

```
module ALU_board(clk, data, clr_common, O, but_A, but_B, but_op, A, B, opcode);

input clk, clr_common, but_A, but_B, but_op;
input [3:0] data;

output reg [3:0] A, B = 0;
output reg [3:0] opcode = 0;
wire [3:0] out1, out2;
wire [3:0] out3;
output reg [7:0] O;

parameter   add_op = 4'b0000,
            sub_op = 4'b0001,
            mul_op = 4'b0010,
            div_op = 4'b0011,
            and_op = 4'b0100,
            or_op = 4'b0101,
            xor_op = 4'b0110,
            xnor_op = 4'b0111,
            C = 2'b10;
register_mem r1(clk, clr_common, but_A, data, out1);
register_mem r2(clk, clr_common, but_B, data, out2);
register_mem r3(clk, clr_common, but_op, data, out3);

always@(*) begin
   A <= out1;

   B <= out2;

   opcode <= out3;
end

always@(*) begin

   if(clr_common) begin
   O <= 8'b0;
   end else begin
   case(opcode)
   add_op: O <= A + B;
   sub_op: O <= A - B;
   mul_op: O <= A * B;
   div_op: O <= A/C;
   and_op: O <= A & B;
   or_op: O <= A | B;
   xor_op: O <= A ^ B;
   xnor_op: O <= A ^~ B;
   default: O <= 8'b11111111;
endcase
end
end
```

Inputs:

- clk: Clock input used for synchronizing the operations within the module.
- data: 4-bit input representing the data received by the ALU.
- clr_common: Clear input that, when asserted (active high), clears the ALU and resets its outputs.

- but_A, but_B, but_op: Inputs for the button signals associated with the respective operations.

Outputs:

- A, B: 4-bit outputs representing the values stored in registers r1 and r2 respectively.
- opcode: 4-bit output representing the value stored in register r3 indicating the opcode of the desired operation.
- O: 8-bit output representing the result of the operation performed by the ALU.

register_mem Instances:

- The module instantiates three instances of the register module: r1, r2, and r3.
- These instances are responsible for storing the values received via data based on the respective button inputs (but_A, but_B, but_op).
- The stored values are provided as outputs through out1, out2, and out3 respectively.

Combining Outputs:

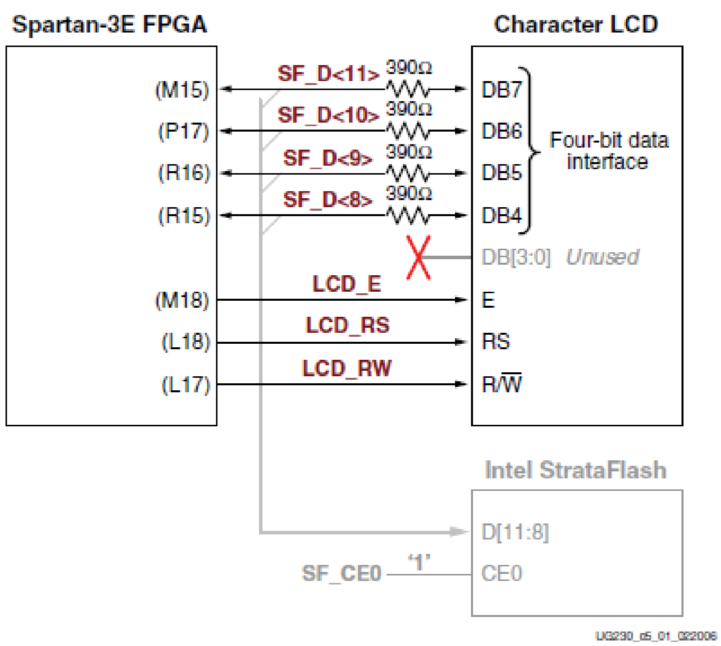
The always @(*) block combines the outputs of the register memory modules and assigns them to A, B, and opcode respectively.

ALU Operations:

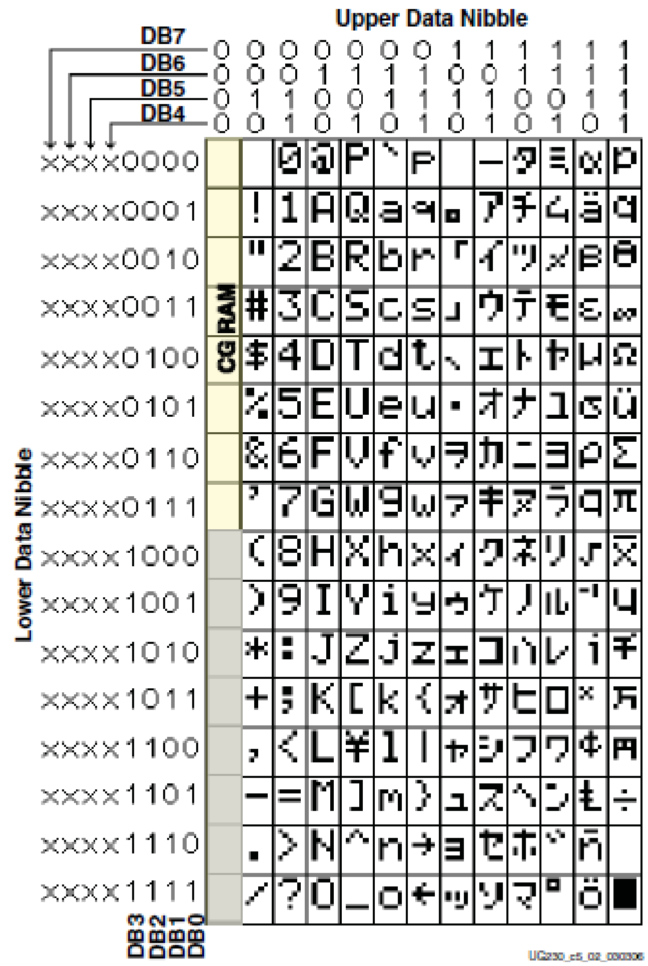
- The second always @(*) block performs the ALU operations based on the stored opcode and values.
- If clr_common is asserted, the output O is cleared to 8'b0.
- Otherwise, based on the value of opcode, a specific operation is selected and the result is assigned to O.
- The supported operations include addition, subtraction, multiplication, and division, logical AND, logical OR, XOR, and XNOR. **Verilog does not support division operator with a user-defined divider due to complex circuitry. Hence, we fixed a parameter C as 2 to enable division of A by 2.**
- If the opcode does not match any of the defined values, the output O is set to 8'b11111111.

Overall, this module implements an ALU that can perform various arithmetic and logical operations based on the values stored in registers and the specified opcode. It leverages the register_mem module to store the inputs, combines the outputs, and calculates the result accordingly.

Step 3: LCD Interfacing Module



Signal Name	FPGA Pin	Function	
SF_D<11>	M15	Data bit DB7	Shared with StrataFlash pins SF_D<11:8>
SF_D<10>	P17	Data bit DB6	
SF_D<9>	R16	Data bit DB5	
SF_D<8>	R15	Data bit DB4	
LCD_E	M18	Read/Write Enable Pulse 0: Disabled 1: Read/Write operation enabled	
LCD_RS	L18	Register Select 0: Instruction register during write operations. Busy Flash during read operations 1: Data for read or write operations	
LCD_RW	L17	Read/Write Control 0: WRITE, LCD accepts data 1: READ, LCD presents data	




```
module lcd(clk, sf_e, e, rs, rw, d, c, b, a, data, but_A, but_B, but_op, clr_common,L);

   (* LOC = "C9" *) input clk; // pin C9 is the 50-MHz on-board clock
                    input [3:0] data;
                    input but_A, but_B, but_op, clr_common;
                     wire [7:0] w1;
                     wire [3:0] A, B, opcode;
                     output reg [7:0] L;
   (* LOC = "D16" *) output reg sf_e; // 1 LCD access (0 StrataFlash access)
   (* LOC = "M18" *) output reg e; // enable (1)
   (* LOC = "L18" *) output reg rs; // Register Select (1 data bits for R/W)
   (* LOC = "L17" *) output reg rw; // Read/Write, 1/0
   (* LOC = "M15" *) output reg d; // 4th data bit (to form a nibble)
   (* LOC = "P17" *) output reg c; // 3rd data bit (to form a nibble)
   (* LOC = "R16" *) output reg b; // 2nd data bit (to form a nibble)
   (* LOC = "R15" *) output reg a; // 1st data bit (to form a nibble)

   parameter    add_op = 4'b0000, // mapping opcodes to parameter names
                sub_op = 4'b0001,
                mul_op = 4'b0010,
                div_op = 4'b0011,
                and_op = 4'b0100,
                or_op = 4'b0101,
                xor_op = 4'b0110,
                xnor_op = 4'b0111;

```

```
ALU_board A1(clk, data, clr_common, w1, but_A, but_B, but_op, A, B, opcode);


   reg [ 26 : 0 ] count = 0;  // 27-bit count, 0-(128M-1), over 2 secs
   reg [ 5 : 0 ] code;        // 6-bit different signals to give out
   reg refresh;               // refresh LCD rate @ about 25Hz

   always @ (posedge clk) begin

      assign L = w1;              // assign ALU output to LEDs

      count <= count +1;
      case ( count[ 26 : 21 ] )  // as top 6 bits change

         0: code <= 6'h03;        // power-on init sequence
         1: code <= 6'h03;        // this is needed at least once
         2: code <= 6'h03;        // when LCD's powered on
         3: code <= 6'h02;        // it flickers existing char display

//  Function Set
// send 00 and upper nibble 0010, then 00 and lower nibble 10xx
         4: code <= 6'h02;        // Function Set, upper nibble 0010
         5: code <= 6'h08;        // lower nibble 1000 (10xx)

//  Entry Mode
// send 00 and upper nibble 0000, then 00 and lower nibble 0 1 I/D S
// last 2 bits of lower nibble: I/D bit (Incr 1, Decr 0), S bit (Shift 1, 0 no)
         6: code <= 6'h00;       // see table, upper nibble 0000, then lower nibble:
         7: code <= 6'h06;       //  0110: Incr, Shift disabled

//  Display On/Off
// send 00 and upper nibble 0000, then 00 and lower nibble 1DCB:
// D: 1, show char represented by code in DDR, 0 don't, but code remains
// C: 1, show cursor, 0 don't
// B: 1, cursor blinks (if shown), 0 don't blink (if shown)
         8: code <= 6'h00;       // Display On/Off, upper nibble 0000
         9: code <= 6'h0C;       // lower nibble 1100 (1 D C B)

//  Clear Display, 00 and upper nibble 0000, 00 and lower nibble 0001
         10: code <= 6'h00;      // Clear Display, 00 and upper nibble 0000
         11: code <= 6'h01;      // then 00 and lower nibble 0001

//  Write Data to DD RAM (or CG RAM)
// Characters are then given out, 1st line
// send 10 and upper nibble 0100, then 10 and lower nibble 1000

         12: code <= 6'b100100;     // A upper nibble
         13: code <= 6'b100001;     // A lower nibble
         14: code <= 6'b100011;     // equals to sign upper nibble
         15: code <= 6'b101101;     // equals to sign lower nibble
         16: code <= 6'b100011;     // upper nibble for all 1-9
         17: code <= {2'b10,A[3:0]}; // lower nibble for 1-9 as selected in register of A

         18: code <= 6'b100010;     // space
         19: code <= 6'b100000;

         20: code <= 6'b100100;     // B upper nibble and so on
         21: code <= 6'b100010;
         22: code <= 6'b100011;
         23: code <= 6'b101101;
         24: code <= 6'b100011;
         25: code <= {2'b10,B[3:0]};

         26: code <= 6'b100010;     // space
         27: code <= 6'b100000;
```

```
        28: code <= 6'b100100;     // O for operator upper nibble and so on
        29: code <= 6'b101111;
        30: code <= 6'b100011;
        31: code <= 6'b101101;
        32: code <= 6'b100010;     // same upper nibble for all symbols as per ASCII table
        33:
        begin
           case(opcode)            // select lower nibble for symbol as per opcode
           add_op: code <= 6'b101011;
           sub_op: code <= 6'b101101;
           mul_op: code <= 6'b101010;
           div_op: code <= 6'b101111;
           and_op: code <= 6'b100110;
           or_op: code <= 6'b100001;
           xor_op: code <= 6'b101110;
           xnor_op: code <= 6'b100010;
           endcase
           end

// Set DD RAM (DDR) Address
// position the cursor onto the start of the 2nd line
         34: code <= 6'b001100;  // pos cursor to 2nd line upper nibble h40 (...)
         35: code <= 6'b000000;  // lower nibble: h0
// Characters are then given out
         36: code <= 6'b100100;
         37: code <= 6'b100001;
         38: code <= 6'b100110;
         39: code <= 6'b101110;
         40: code <= 6'b100111;
         41: code <= 6'b100011;
         42: code <= 6'b100011;
         43: code <= 6'b101010;
         44: code <= 6'b100010;
         45: code <= 6'b100000;

         46: code <= 6'b100011; // same upper nibble for 1-9
         47: code <= {2'b10,w1[3:0]}; // lower nibble as per ALU output
// Read Busy Flag and Address
// send 01 BF (Busy Flag) x x x, then 01xxxx
// idling
         default: code <= 6'h10; // the rest un-used time
      endcase
// refresh (enable) the LCD when
         refresh <= count[ 20 ]; // flip rate almost 25 (50Mhz / 2^21-2M)
         sf_e <= 1;
         { e, rs, rw, d, c, b, a } <= { refresh, code };
   end // always block

endmodule
```

The code uses a state machine to control the behavior of an LCD (Liquid Crystal Display) module. Here is a breakdown of the different states and their corresponding actions:

- **State 0 to 3: Initialization sequence:** These states set up the LCD module when it is powered on. The code sends specific commands to configure the LCD's function, entry mode, display on/off settings, and clears the display.

- **State 4 and 5: Function Set:** These states send commands to set the function of the LCD, including the number of display lines and character font.

- State 6 and 7: Entry Mode: These states configure the entry mode of the LCD, determining the cursor movement direction and whether the display should shift.

- State 8 and 9: Display On/Off: These states control the display and cursor settings. The code enables or disables the display, shows or hides the cursor, and determines if the cursor should blink.

- State 10 and 11: Clear Display: These states send commands to clear the display.

- State 12 to 17: Write Data to DDR: These states send commands and data to write characters to the Display Data RAM (DDR) or Character Generator RAM (CG RAM). The characters displayed include digits 1-9, the equals sign, and space.

- State 18 to 27: Write Data to DDR (Continued): These states continue writing characters to the DDR. Characters displayed include letters A and B, the operator "O," and symbols based on the opcode received from an ALU (Arithmetic Logic Unit) module.

- State 28 and 29: Set DDR Address: These states set the DDR address to position the cursor on the start of the second line of the display.

- State 30 to 45: Write Data to DDR (Continued): These states continue writing characters to the DDR for the second line. Characters displayed include additional letters, numbers, and symbols.

- State 46 and 47: Write Data to DDR (Continued): These states write the lower nibble of the ALU output (w1) to the DDR.

The code uses a counter (count) to transition between the states based on the count value. The ALU_board module is also instantiated within this module and receives various inputs such as clk, data, clr_common, and button inputs to perform arithmetic operations.

The code controls the various control lines (sf_e, e, rs, rw, d, c, b, a) of the LCD module to send commands and data to display characters and control the behavior of the LCD. The ALU output (w1) is assigned to the LEDs (L) for display.

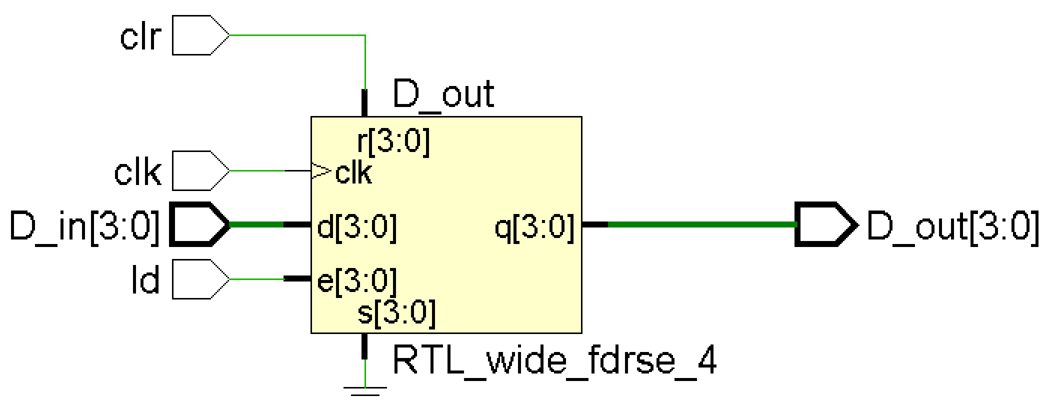
Please note that this code represents only the interface and control logic for the LCD module, and the actual display content and behavior may depend on the external inputs, ALU calculations, and other factors not mentioned in the code snippet.

I/O Planning:

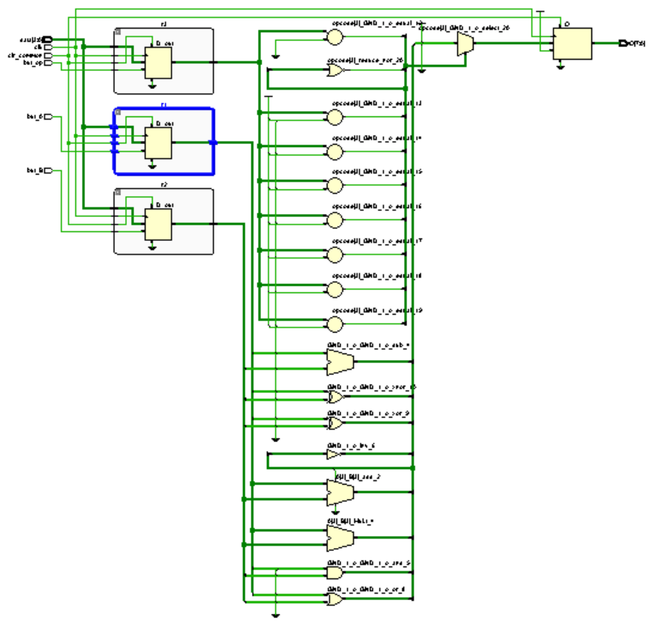
Name	Direction	Neg Diff Pair	Site	Fixed	Bank	I/O Std	Vcco	Vref	Drive Stre...	Slew Type	Pull Type
All ports (25)											
data (4)	Input				1	default (LVCMOS25)	2.500				NONE
data[3]	Input		N17	☑	1	default (LVCMOS25)	2.500				NONE
data[2]	Input		H18	☑	1	default (LVCMOS25)	2.500				NONE
data[1]	Input		L14	☑	1	default (LVCMOS25)	2.500				NONE
data[0]	Input		L13	☑	1	default (LVCMOS25)	2.500				NONE
L (8)	Output				0	default (LVCMOS25)	2.500		12	SLOW	NONE
L[7]	Output		F9	☑	0	default (LVCMOS25)	2.500		12	SLOW	NONE
L[6]	Output		E9	☑	0	default (LVCMOS25)	2.500		12	SLOW	NONE
L[5]	Output		D11	☑	0	default (LVCMOS25)	2.500		12	SLOW	NONE
L[4]	Output		C11	☑	0	default (LVCMOS25)	2.500		12	SLOW	NONE
L[3]	Output		F11	☑	0	default (LVCMOS25)	2.500		12	SLOW	NONE
L[2]	Output		E11	☑	0	default (LVCMOS25)	2.500		12	SLOW	NONE
L[1]	Output		E12	☑	0	default (LVCMOS25)	2.500		12	SLOW	NONE
L[0]	Output		F12	☑	0	default (LVCMOS25)	2.500		12	SLOW	NONE
Scalar ports (13)											
a	Output		R15	☑	1	default (LVCMOS25)	2.500		12	SLOW	NONE
b	Output		R16	☑	1	default (LVCMOS25)	2.500		12	SLOW	NONE
but_A	Input		H13	☑	1	default (LVCMOS25)	2.500				NONE
but_B	Input		D18	☑	1	default (LVCMOS25)	2.500				NONE
but_op	Input		K17	☑	1	default (LVCMOS25)	2.500				NONE
c	Output		P17	☑	1	default (LVCMOS25)	2.500		12	SLOW	NONE
clk	Input		C9	☑	0	default (LVCMOS25)	2.500				NONE
clr_common	Input		V4	☑	2	default (LVCMOS25)	2.500				NONE
d	Output		M15	☑	1	default (LVCMOS25)	2.500		12	SLOW	NONE
e	Output		M18	☑	1	default (LVCMOS25)	2.500		12	SLOW	NONE
rs	Output		L18	☑	1	default (LVCMOS25)	2.500		12	SLOW	NONE
rw	Output		L17	☑	1	default (LVCMOS25)	2.500		12	SLOW	NONE
sf_e	Output		D16	☑	1	default (LVCMOS25)	2.500		12	SLOW	NONE

RTL Design:

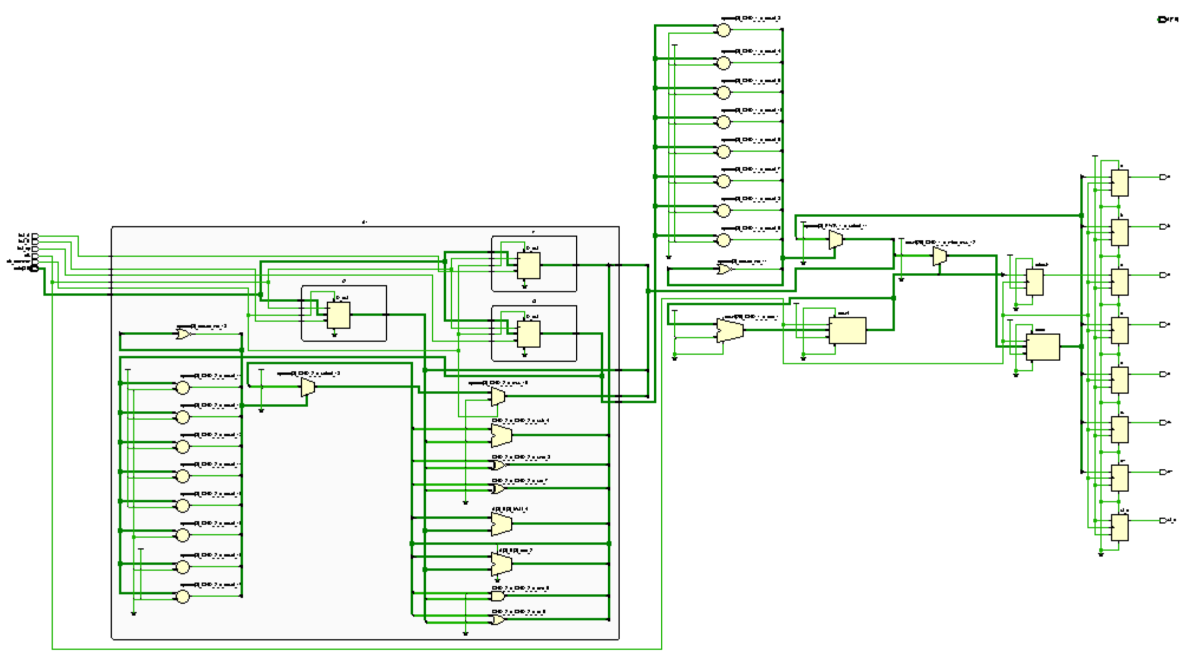
Register Memory:



Arithmetic Logic Unit:



LCD Interface:



Output:

The LCD module will display characters based on the commands and data sent to it through the control lines (sf_e, e, rs, rw, d, c, b, a). The code specifies various states and corresponding commands to write specific characters to the display, including numbers, letters, symbols, and operators. The characters will be shown on the LCD module's screen.

Additionally, the ALU output (w1) is assigned to the LEDs (L) using the line **assign L = w1**. This means that the values produced by the ALU module will be displayed on the 8 onboard LEDs. The LEDs will represent the binary value of the ALU output, providing a visual representation of the arithmetic calculations performed by the ALU.

Thus, we will be able to observe the output both on the LCD module's display and the 8 onboard LEDs.

Limitation of Design and Room for Improvement:

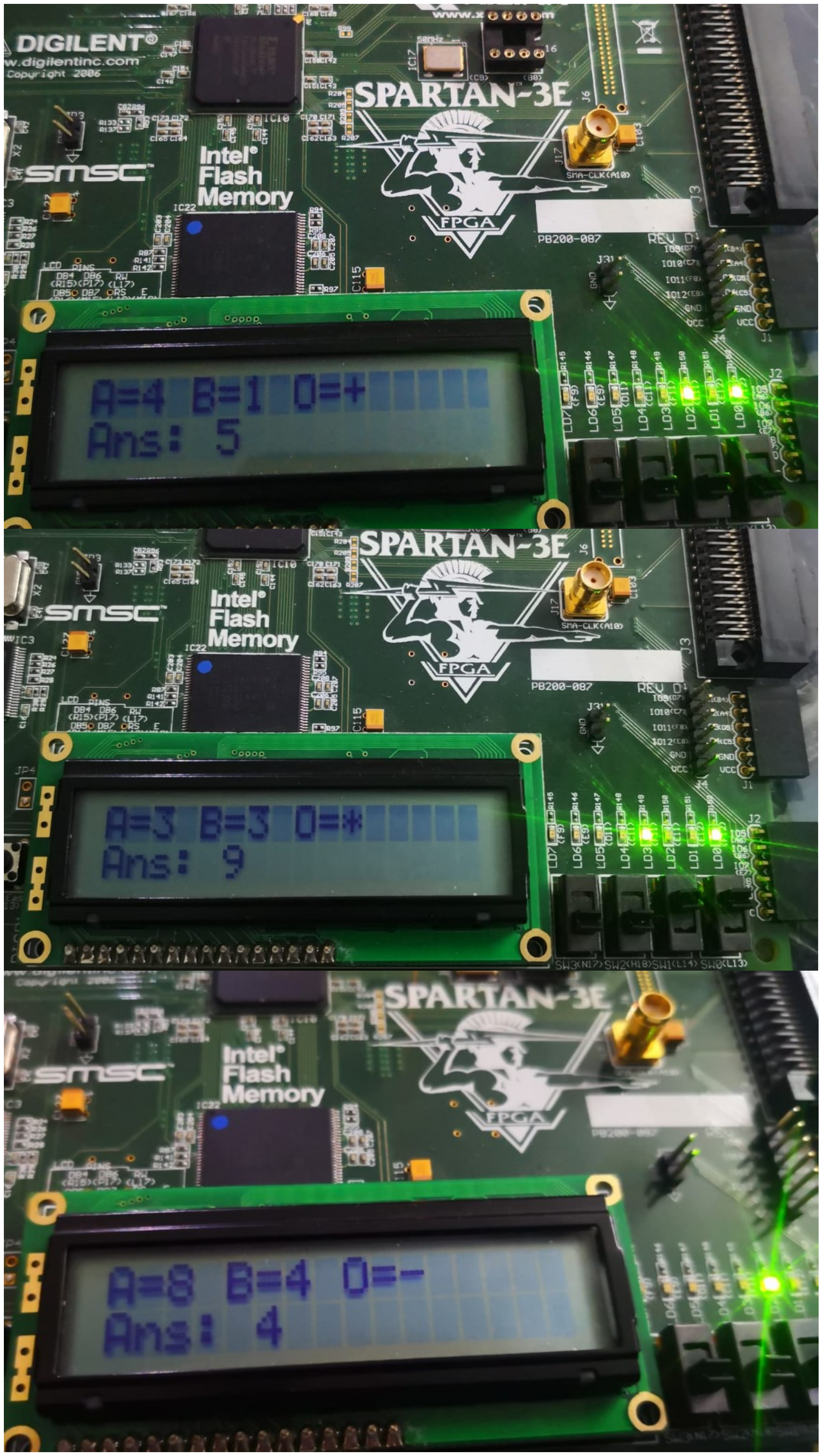
The limitation of using the LCD interface in this design is that it can display characters based on the ASCII table, which includes numerical digits from 0 to 9, as well as letters, symbols, and special characters. As a result, the LCD module alone cannot directly display the full range of 8-bit binary values accurately.

We can overcome this limitation by using additional registers and implementing a mechanism to shift individual digits from the most significant bit (MSB) to the least significant bit (LSB) of the answer value. By doing so, we can split the 8-bit number into separate digits and then display each digit on the LCD module sequentially.

This approach allows us to display any 8-bit number accurately on the LCD module by representing each digit using the available characters from the ASCII table. By shifting and updating the digits based on the changing value of the answer, we can create the illusion of displaying an 8-bit number on the LCD module, albeit with multiple characters.

By leveraging the available resources and implementing the necessary logic, we can extend the functionality of the LCD interface to display the full range of 8-bit binary values, providing a more comprehensive representation of the ALU output on the LCD module.

Results/Verification:



Conclusion:

In conclusion, the project involved the design and implementation of a 4-bit Arithmetic Logic Unit (ALU) on the Spartan 3E FPGA using Verilog. The ALU was capable of performing various arithmetic and logical operations on two 4-bit inputs, with an 8-bit output. The inputs were received through slide switches, and the output was displayed simultaneously on an on-board LCD display and the on-board LEDs.

To overcome the limitation of having only 4 slide switches for inputs, registers were implemented for each input. This allowed the limited input buttons to be used efficiently for all the required inputs, enabling the ALU to perform calculations accurately.

The design approach involved creating separate modules for register memory, the ALU itself, and interfacing with the LCD display. The register memory module stored the input values, while the ALU module performed calculations based on the stored inputs and the specified opcode. The top-level module initialized and controlled the LCD display to show the inputs, opcode, and results.

Although the ALU produced accurate 8-bit results on the on-board LEDs, the limitation of the LCD interface was identified. Due to the ASCII table limitation, the LCD could only display numerical digits from 0 to 9 accurately. To overcome this limitation, it was suggested to use additional registers and implement a mechanism to shift individual digits of the result for display on the LCD.

Overall, the project successfully implemented a functional 4-bit ALU on the Spartan 3E FPGA, providing the ability to perform arithmetic and logical operations. The project also highlighted the challenges and limitations of the LCD interface and proposed a solution to display the full range of 8-bit results accurately.