Vietnam National University Ho Chi Minh City
Ho Chi Minh City University of Technology
Department of Electronics

EE3043: Computer Architecture

Laboratory Report

Student

Hien Nguyen Phuong — 2051039

Vy Nguyen Tuan— 2051027

Loc Le Quy — 2051144

Supervisor

Dr. Linh Tran

November 2023

Content

Milestone 2

Design of a Single-Cycle Processor

1. Objectives

In this milestone, we are required to :

• Review understanding of SystemVerilog

• Review understanding of RV32I instructions

• Design a single-cycle RV32I processor

To design a Single_Cycle Processor, we need to understand the structure of RISC V and how to run RISC V by assembly. Then we intend to design some modules such as Immem to store instruction, Dmem to store data, Alu to calculate some basic instruction, a Control Unit to control the Single-Cycle unit, and LSU to communicate with peripherals,…

2. Analyses

2.1. Instruction analyses

The instruction set is divided into groups of instructions based on the instr[6:2] bits in the red frame (because 2 bits inst[1:0] all the commands we are using are 11 so they can be omitted)

R format

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and

I calculate format

31 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai

I load format

31 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu

S format

31 25	24 20	19 15	14 12	11 7	6 0	
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw

B format

31 25	24 20	19 15	14 12	11 7	6 0	
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu

U format

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc

J format

31 25	24 20	19 15	14 12	11 7	6 0	
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr

Figure 1.1: RISCV R32I 37 instructions

Each instruction in the group is identified by the 3 bits instr[14:12] but in addition, there are several instructions. Especially with the same value instr[14:12], you need to use the instr[30] bit to determine the correct command. Specifically the commands: ADD vs SUB, SRL vs SRA, and SRLI vs SRAI. In command group U no need to use instr[14:12] to divide commands where instr[6:2] has already identified the command.

In terms of immediate, we divide them into these groups:

R format:

• Rs1: instr[19:15]

• Rs2: instr[24:20]

• Rd : instr[11:7]

I calculate format and I load format:

• Rs1: instr[19:15]

• Rd : instr[11:7]

• Imm: {{20{instr[31]}},instr[31:20]} (except slli,srli,srai)

S format

• Rs1: instr[19:15]

• Rs2: instr[24:20]

• Imm: {{20{instr[31]}},instr[31:25],instr[11:7]}6

B format

• Rs1: instr[19:15]

• Rs2: instr[24:20]

• Imm: {{20{instr_i[31]}},instr _i[7],instr_i[30:25], instr_i[11:8], 1'b0}

U format

• Rd : instr[11:7]

• Imm: {instr_i[31:12], 12'b0}

Jal

• Rd : instr[11:7]

• Imm: {{12{instr_i[31]}}, inst_i[19:12], instr_i[20], instr_i[30:21], 1'b0}

Jalr

• Rs1: instr[19:15]

• Rd : instr[11:7]

• Imm: {{20{instr_i[31]}},instr_i[11:0]}

2.2. ALU and Branch Comparison

ALU: The ALU in CPUs performs some sort of operation, depending on the ISA: ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, LUI, SLT, SLTU.

SLT and SLTU: These 2 instructions will need a comparison to compare data, however, we do not allow to use of bitwise so we will design a comparison.

SUB: We do not allow to use of the operator " - " so we will find another way to calculate the subtractor.

Branch Comparison: Only compares two data. We are also required not to use bit-wise, so we design a comparison. This module is used for these instructions: SLTI, SLTIU, SLT, SLTU, BEQ, BNE, BLT, BGE, BLTU, BGEU

Question:

- Convert this binary number to its hexadecimal representation:

16'b0010001110110011 => 0010 0011 1011 0011 (group into group of 4)

- 0010 corresponds to 2 in hexadecimal.

- 0011 corresponds to 3 in hexadecimal.

- 1011 corresponds to B in hexadecimal.

- 0011 corresponds to 3 in hexadecimal.

0010 0011 1011 0011 => 0x23B3.

- Convert this hexadecimal number to its binary representation:

16’hEECA => E = 1110 ; E = 1110 ; C = 1100 ; A = 1010 => 1110 1110 1100 1010

16’hEECA => 16'b1110111011001010

- Two’s complement hexadecimal number of this decimal number: 16’d3043

Convert the decimal number to binary: 3043 in decimal = 101111100011 in binary. Pad the binary representation to 16 bits if it's not already 16 bits long. In this case, we have a 12-bit binary number, so we need to pad it with zeros on the left: 0000101111100011.

Find the two's complement by inverting the bits and adding 1 to the least significant bit:

Inverting the bits: 1111010000011100

Adding 1 to the least significant bit: 1111010000011101

Convert the two's complement binary representation to hexadecimal: 1111 0100 0001 1101

- Two's complement hexadecimal representation of the decimal number 16'd3043 is 16'hF41D.

Two’s complement hexadecimal number of NEGATIVE decimal number: -16’d2022

Convert the absolute value of the negative decimal number to binary. The absolute value of -2022 is 2022, and its binary representation is: 2022 in decimal = 11111100110 in binary.

Pad the binary representation to 16 bits if it's not already 16 bits long. In this case, we have an 11-bit binary number, so we need to pad it with zeros on the left: 0000111111001100.

Find the two's complement by inverting the bits and adding 1 to the least significant bit:

Inverting the bits: 1111000000110011

Adding 1 to the least significant bit: 1111000000110100

Convert the two's complement binary representation to hexadecimal:

1111 0000 0011 0100

Two’s complement hexadecimal representation of the negative decimal number -16'd2022 is 16'hF034.

- Sign Extension

• Extend to a 16-bit two’s complement hexadecimal number: 8’h15

• Extend to a 16-bit two’s complement hexadecimal number: 8’hE9

For 8'h15: Binary representation of 8'h15: 00010101

- Sign bit (most significant bit) is 0 (positive).
- To extend to 16 bits, simply copy 0s on the left: 16'h0015

For 8'hE9: Binary representation of 8'hE9: 11101001

- Sign bit (most significant bit) is 1 (negative).
- To extend to 16 bits, simply copy 1s on the left: 16'hFFE9

- Addition and Subtraction

• Determine: 16’h5A78 + 16’h11FE

• Determine: 16’hEFB7 + 16’h6AA9

• Determine: 16’h7713 - 16’h3BC1

• Explain the concept of overflow and underflow.

Addition: 16'h5A78 + 16'h11FE

Convert the hexadecimal numbers to decimal:

16'h5A78 = 23160

16'h11FE = 4606

Now, we add the decimal values:

23160 + 4606 = 27766

Convert the result back to hexadecimal:

27766 in decimal is 0x6C16 in hexadecimal => 16'h5A78 + 16'h11FE = 16'h6C16.

Addition: 16'hEFB7 + 16'h6AA9

Convert the hexadecimal numbers to decimal:

16'hEFB7 = -4169 (msb = 1, negative number)

16'h6AA9 = 27305

Now, we add the decimal values:

-4169 + 27305 = 23136

Convert the result back to hexadecimal:

23136 in decimal is 0x5A80 in hexadecimal => 16'hEFB7 + 16'h6AA9 = 16'h5A80.

Subtraction: 16'h7713 - 16'h3BC1

Convert the hexadecimal numbers to decimal:

16'h7713 = 30483

16'h3BC1 = 15297

Now, we subtract the decimal values:

30483 - 15297 = 15186

Convert the result back to hexadecimal:

15186 in decimal is 0x3B52 in hexadecimal => 16'h7713 - 16'h3BC1 = 16'h3B52.

- About the concept of overflow and underflow:

Overflow:

Overflow occurs when the result of an arithmetic operation exceeds the maximum representable value for the data type being used. In the context of two's complement representation (commonly used for signed integers), overflow occurs when adding two positive numbers results in a negative number or adding two negative numbers results in a positive number. Overflow can also occur in subtraction, where subtracting a large number from a smaller number results in a positive number or subtracting a small number from a large number results in a negative number.

Underflow:

Underflow is less commonly discussed but can happen when the result of an arithmetic operation is smaller (in absolute value) than the minimum representable value for the data type. In two's complement representation, this can occur in subtraction when subtracting a positive number from a smaller positive number results in a value closer to zero than the minimum representable value. Underflow is typically a concern for very small values or in applications where the available range of values is limited. Both overflow and underflow can lead to unexpected or incorrect results in numerical computations, so it's important to be aware of these concepts when working with numeric data in computer programs.

Logic Operation

• Determine: 16’h5A78 and 16’h11FE

• Determine: 16’hEFB7 or 16’h6AA9

• Determine: 16’h7713 xor 16’h3BC1

*Bitwise AND: 16'h5A78 and 16'h11FE

Convert the hexadecimal numbers to binary:

16'h5A78 = 0101101001111000

16'h11FE = 0001000111111110

0101101001111000 & 0001000111111110 = 0001000001111000

=> 16'h5A78 AND 16'h11FE = 16'h1078.

*Bitwise OR: 16'hEFB7 or 16'h6AA9

Convert the hexadecimal numbers to binary:

16'hEFB7 = 1110111110110111

16'h6AA9 = 0110101010101001

1110111110110111 | 0110101010101001 = 1110111110111111

=> 16'hEFB7 OR 16'h6AA9 = 16'hEFDF.

*Bitwise XOR: 16'h7713 xor 16'h3BC1

Convert the hexadecimal numbers to binary:

16'h7713 = 0111011100010011

16'h3BC1 = 0011101111000001

0111011100010011 ^ 0011101111000001 = 0100110011010010

=> 16'h7713 XOR 16'h3BC1 = 16'h4C62.

Shift Operation

• Determine: 16’h5A78 << 5

• Determine: 16’hEFB7 >> 5

• Determine: 16’hF713 >>> 5

*Left Shift (16'h5A78 << 5):

Convert the hexadecimal number to binary: 16'h5A78 = 0101101001111000

Perform a left shift by 5 positions, which means shifting all bits to the left and filling in with zeros on the right: 0101101001111000 << 5 = 1001110000000000

=> 16'h5A78 << 5 = 16'h9C00.

*Right Shift (16'hEFB7 >> 5):

Convert the hexadecimal number to binary: 16'hEFB7 = 1110111110110111

Perform a right shift by 5 positions, which means shifting all bits to the right and filling in with the most significant bit (sign bit): 1110111110110111 >> 5 = 1111111111110111

=> 16'hEFB7 >> 5 = 16'hFFFF.

*Right Logical Shift (16'hF713 >>> 5):

Convert the hexadecimal number to binary: 16'hF713 = 1111011100010011

Perform a right logical shift by 5 positions, which means shifting all bits to the right and filling in with zeros on the left: 1111011100010011 >>> 5 = 0000011110110000

=> 16'hF713 >>> 5 = 16'h07B0.

2.3. Load-Store Unit

In actuality, a CPU needs to communicate with peripherals to read or export data. Designing an I/O system could finish this. Standard peripherals include things like switches, LCDs, and LEDs. In actuality, people view these peripherals as "memory." For instance, putting data to a 32-bit register that is allocated to 32 LEDs denotes controlling the LED array's status. One method for outlining the structure of memory

is memory mapping. various locations may perform a variety of functions in memory.

00FFFFh 00FF80h	Interrupt Vector
0243FFh 004004	Code Memory
0043FFh 002400h	RAM
0023FFh 00IC00h	USB Ram
0091FFh 001800H	Information Memory
0017FFh 00I000h	Boot Ladder Memory
000FFFh 0h	Peripherals

So module LSU will do this job. The memory map is in the following figure :

Figure 2.1: Data memory

2.4. Single-Cycle Processor

The complete processor is the module that contains all sub_modules such as Control Unit, and Immediate Generator, and then integrates the memory modules into your project. The standard processor in this course is described in this figure

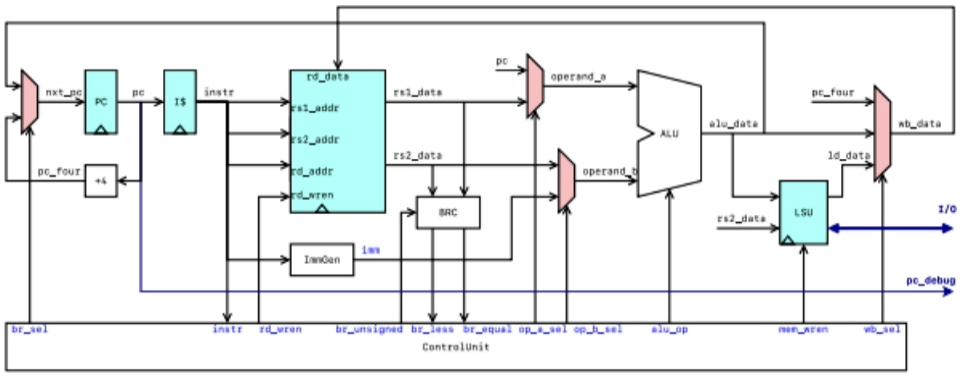


Figure 2.2: Singlecycle RTL

3. Design

3.1. ALU and Branch Comparison

3.1.1. ALU

alu_op	Description (R-type)	Description (I-type)
ADD	$rd = rs1 + rs2$	$rd = rs1 + imm$
SUB	$rd = rs1 - rs2$	n/a
SLT	$rd = (rs1 < rs2)?1 : 0$	$rd = (rs1 < imm)?1 : 0$
SLTU	$rd = (rs1 < rs2)?1 : 0$	$rd = (rs1 < imm)?1 : 0$
XOR	$rd = rs1 \oplus rs2$	$rd = rs1 \oplus imm$
OR	$rd = rs1 \vee rs2$	$rd = rs1 \vee imm$
AND	$rd = rs1 \wedge rs2$	$rd = rs1 \wedge imm$
SLL	$rd = rs1 << rs2[4:0]$	$rd = rs1 << imm[4:0]$
SRL	$rd = rs1 >> rs2[4:0]$	$rd = rs1 >> imm[4:0]$
SRA	$rd = rs1 >>> rs2[4:0]$	$rd = rs1 >>> imm[4:0]$

The module name is **alu**

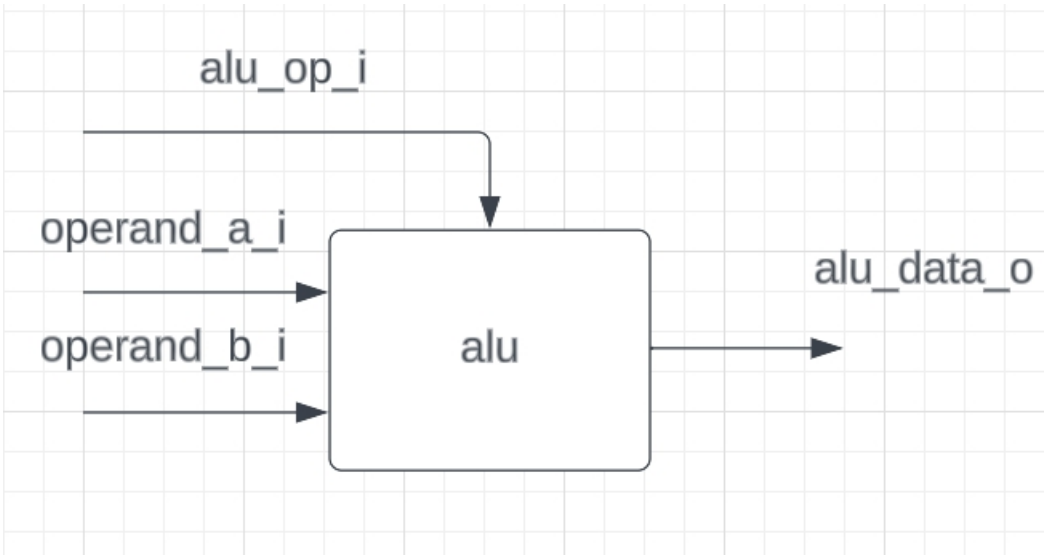


Figure 3.1: alu block diagram

Inputs:

- operand_a: the first operand — rs1.

- operand_b: the second operand — rs2.

- alu_op : an operation that the ALU has to perform.

If alu_op is equal to one of each 4-bit number, the ALU will execute that instruction.

```
A_ADD  = 4'b0000 ,
A_SUB  = 4'b0001 ,
A_SLL  = 4'b0010 ,
A_SLT  = 4'b0011 ,
A_SLTU = 4'b0100 ,
A_XOR  = 4'b0101 ,
A_SRL  = 4'b0110 ,
A_SRA  = 4'b0111 ,
A_OR   = 4'b1000 ,
A_AND  = 4'b1001 ,
A_LUI  = 4'b1010
```

Outputs

- alu_data: the result of the operation.

The picture below will give a straightforward idea for an ALU that we have designed.

```
always_comb begin
    case (alu_op_i)
            A_ADD   : begin
            alu_data_o = operand_a_i + operand_b_i;
        end
        A_SUB   : begin
            alu_data_o = operand_a_i + ~(operand_b_i) + 1;
        end
            A_SLT   : begin
            alu_data_o = {31'b0,blarger_s} ;
        end
            A_SLTU  : begin
            alu_data_o = {31'b0,blarger_u} ;
        end
            A_XOR   : begin
            alu_data_o = operand_a_i ^ operand_b_i ;
        end
            A_OR    : begin
            alu_data_o = operand_a_i | operand_b_i ;
        end
            A_AND   : begin
            alu_data_o = operand_a_i & operand_b_i ;
        end
            A_SLL   : begin
            alu_data_o = operand_a_i << shift_number ;
        end
            A_SRL   : begin
            alu_data_o = operand_a_i >> shift_number ;
        end

            A_SRA   : begin

            if(!operand_a_i[31]) begin
                alu_data_o      = operand_a_i >> shift_number;
            end
            else begin
                alu_data_o      = alu_data_temp2[31:0];
            end
        end
            A_LUI   : begin
            alu_data_o = operand_b_i ;
        end
        default: begin
            alu_data_o = 0;
        end
    endcase
```

As mentioned before we are not allowed to use these operators " -, <, > "

Then with the subtract instruction, we will do the same as the add instruction, however, we will inverse operand_b and then + 1, so it means that.

SUB = operand_a_i + ~ operand_b_i + 1

SLT, SLTU: These two instructions will be computed in branch comparison.

3.1.2 Branch comparison

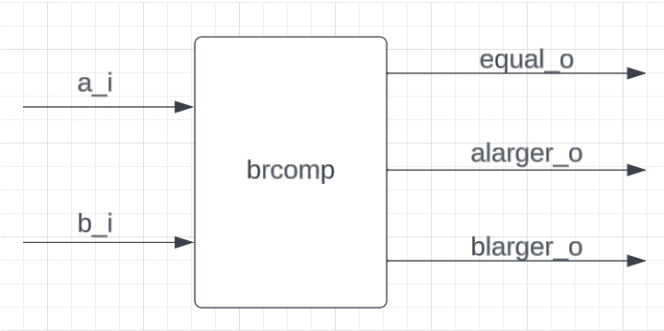


Figure 3.2: brcomp block diagram

Inputs:

a_I: the first operand — rs1 → A.

b_I: the first operand — rs2 → B.

Outputs

equal : 1 if A = B

alarger : 1 if A > B.

blarger 1: 1 if A < B.

To make it more challenging, we are not allowed to use the operator " -, <, > ". Therefore with the instruction that has comparison in it such as SLTI, SLTIU, SLT, SLTU, BEQ, BNE, BLT, BGE, BLTU, BGEU. We will design a comparison module (brcomp) :

INPUT		OUTPUT		
A	**B**	**A < B**	**A = B**	**A > B**
0	0	0	1	0
1	0	0	0	1
0	1	1	0	0
1	1	0	1	0

From this table, we will see that

Equal = A ~ ^ B; (XNOR)

A larger = A & (~ B);

B larger = (~A) & B;

From that logic then we can form a 4-bit comparison. It just follows this method

(A=B) = A3B3 . A2B2 . A1.B1 . A0.B0 = x3x2x1x0

(A>B) = A3B3′ + x3A2B2′ + x3x2A1B1′ + x3x2x1A0B0′

(A<B) = A3'B3 + x3A2'B2 + x3x2A1'B1 + x3x2x1A0'B0

Then from the logic of a 4-bit comparison, we can design an 8-bit comparison, a 16-bit comparison, and the final 32-bit comparison.

This is a 32-bit unsigned comparison, the logic here is the same as a 4-bit comparison.

```
assign equal_o    = &el_t;
assign alarger_o  = al_t[1]|(al_t[0]&el_t[1]);
assign blarger_o  = ~(equal_o|alarger_o);
/* verilator lint_off PINCONNECTEMPTY */
compare_16bit compare160(.a_i(a_i[15:0]),.b_i(b_i[15:0]),.equal_o(el_t[0]),.alarger_o(al_t[0]),.blarger_o());
compare_16bit compare161(.a_i(a_i[31:16]),.b_i(b_i[31:16]),.equal_o(el_t[1]),.alarger_o(al_t[1]),.blarger_o());
```

To design a 32-bit signed comparison, it is quite easy, we just divide it into 4 cases:

Case 1: both a and b are unsigned number

Case 2: a is unsigned and b is signed number

Case 3: a is signed and b is unsigned number

Case 4: both a and b are signed number

With case 4, we just inverse the input a and b then add 1 to each of them, then we compare a positive number, then to determine which number is larger, we just need to take the inverse of that signal.

```
assign a_t = ~(a_i)+1;
assign b_t = ~(b_i)+1;

compare_32bit_u compare32u_0(.a_i(a_i[31:0]),.b_i(b_i[31:0]),.equal_o(eq_t[0]),.alarger_o(al_t[0]),.blarger_o(bl_t[0]));
compare_32bit_u compare32u_1(.a_i(a_t[31:0]),.b_i(b_t[31:0]),.equal_o(eq_t[1]),.alarger_o(al_t[1]),.blarger_o(bl_t[1]));

always_comb begin
    case({a_i[31],b_i[31]})
    2'b00:  begin
        equal_o     =   eq_t[0];
        alarger_o   =   al_t[0];
        blarger_o   =   bl_t[0];
    end
    2'b01:  begin
        equal_o     =   1'b0;
        alarger_o   =   1'b1;
        blarger_o   =   1'b0;
    end
    2'b10:  begin
        equal_o     =   1'b0;
        alarger_o   =   1'b0;
        blarger_o   =   1'b1;
    end
    2'b11:  begin
        equal_o     =   eq_t[1];
        alarger_o   =   ~al_t[1];
        blarger_o   =   ~bl_t[1];
    end
    default: begin
    end
    endcase
end
```

3.2. Load-Store Unit

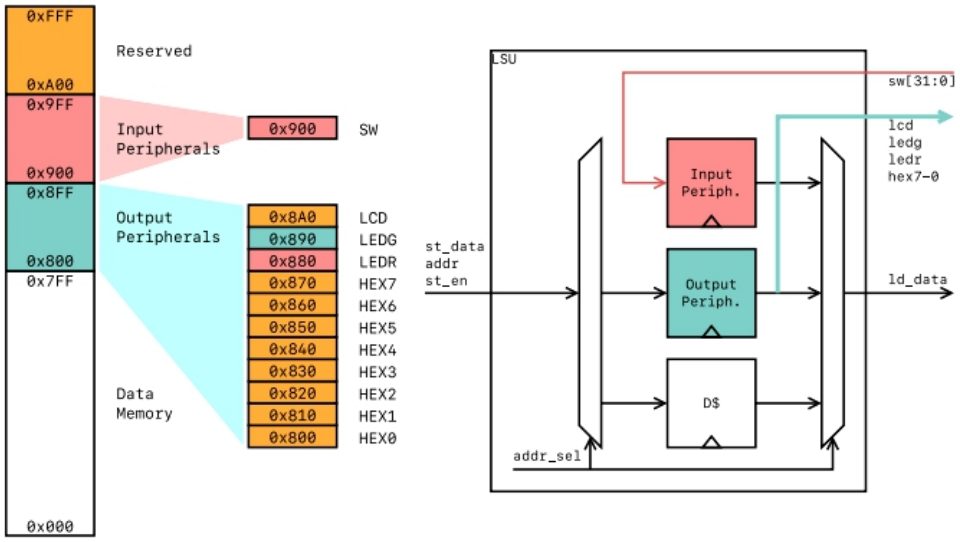


Figure 3.3: Data memory

The LSU is a module that controls peripherals such as LCD, LEDR, LEDG, and HEX7…..HEX0.

From the picture above, we see that each of the peripherals has their own address in the memory:

0x900 : SW

0x8A0 : LCD

……

0x800 : HEX0

So if we want to control peripherals, we just need to design a memory that has spaces for peripherals, so when we access to address of those peripherals, we can use it.

For example, the address at 0x800 is a memory for HEX0, so if you want to control this peripheral, we just need to access this memory and then write data into it but based on sel_mod BYE, HWORD, or WORD, your data will be 8 bit, 16 bit or 32 bit.

```
always_ff @(posedge clk_i) begin
    if (dmux_oper_st_en) begin
        case (addr_i[15:0])
            16'h0800 : begin
                if (sel_mod[1:0] == BYTE) begin
                    output_per_reg[0][0] <= st_data_i[7:0];
                    output_per_reg[0][1] <= output_per_reg[0][1];
                    output_per_reg[0][2] <= output_per_reg[0][2] ;
                    output_per_reg[0][3] <= output_per_reg[0][3]  ;
                end
                else if (sel_mod[1:0] == HWORD) begin
                    output_per_reg[0][0] <= st_data_i[7:0];
                    output_per_reg[0][1] <= st_data_i[15:8];
                    output_per_reg[0][2] <= output_per_reg[0][2] ;
                    output_per_reg[0][3] <= output_per_reg[0][3] ;
                end
                else if (sel_mod[1:0] == WORD) begin
                    output_per_reg[0][0] <= st_data_i[7:0];
                    output_per_reg[0][1] <= st_data_i[15:8];
                    output_per_reg[0][2] <= st_data_i[23:16];
                    output_per_reg[0][3] <= st_data_i[31:24];
                end
                else begin
                    output_per_reg[0][0] <= output_per_reg[0][0];
                    output_per_reg[0][1] <= output_per_reg[0][1];
                    output_per_reg[0][2] <= output_per_reg[0][2] ;
                    output_per_reg[0][3] <= output_per_reg[0][3]  ;
                end
            end
```

The module name is **lsu.**

Input:

- clk_i : positive clock.

- rst_ni : low negative reset.

- addr : the address for both reading and writing.

- st_data: the store data.

- st_en : 1 if write, 0 if read.

- io_sw : 32-bit from switches.

Outputs:

- ld_data : the load data.

- io_lcd : 32-bit data to drive LCD.

- io_ledg : 32-bit data to drive green LEDs.

- io_ledr : 32-bit data to drive red LEDs.

- io_hex0..7 : 32-bit data to drive 7-segment LEDs from hex0 to hex 7.

3.3 Control unit

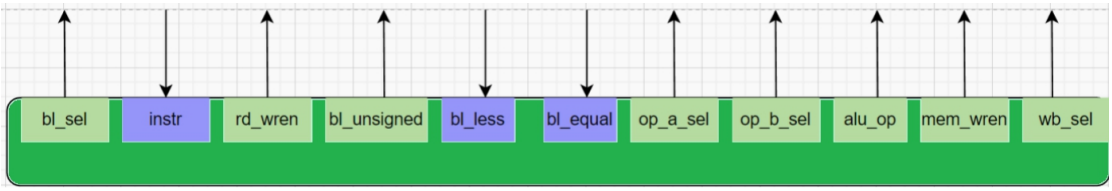


Figure 3.4: control unit block diagram

The module name is **ctrl_unit.**

Inputs:

instr : the 32-bit instruction.

br_less : data from Branch Comparison, 1 if A < B.

br_equal: data from Branch Comparison, 1 if A = B.

Outputs:

br_sel : select PC source: 0 if P C + 4, 1 if computed in ALU.

br_unsigned: 1 if the two operands are unsigned.

rd_wren : 1 if the instruction writes data into regfile.

mem_wren : 1 if the instruction writes data into LSU.

op_a_sel : select operand A source: 0 if rs1, 1 if P C.

op_b_sel : select operand B source: 0 if rs2, 1 if imm.

wb_sel : select data to write into regfile: 0 if alu_data, 1 if ld_data, and 2 or 3 if pc_four

3.4 Single-Cycle Processor

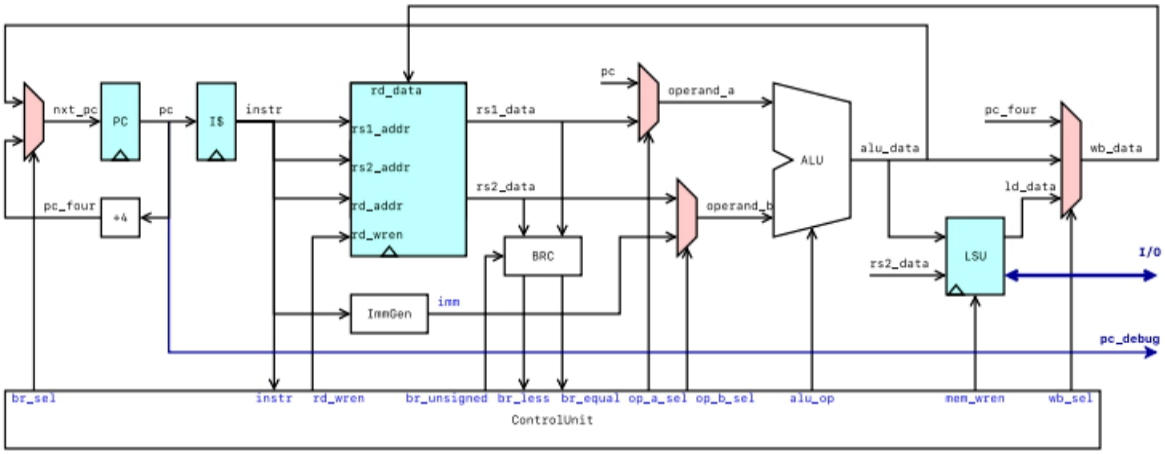


Figure 3.5: Singlecycle RTL

This module works as a top module that combines all sub_modules together to make a completely RISCV CPU.

The module name is singlecycle.

Inputs:

- clk_i : positive clock.

- rst_ni :low negative reset.

- io_sw_i: 32-bit from switches.

Outputs:

- pc_debug_o: PC counter.

- io_lcd_o : 32-bit data to drive LCD.

- io_ledg_o : 32-bit data to drive green LEDs.

- io_ledr_o : 32-bit data to drive red LEDs.

- io_hex0..7_o : 32-bit data to drive 7-segment LEDs

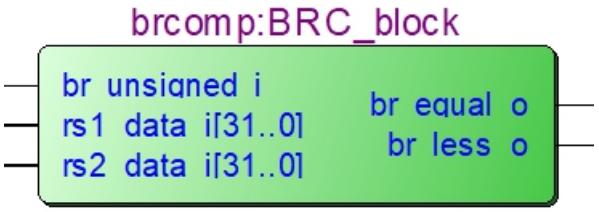
This module is a top module that contains all the sub_files, and it is a complete module of a RISCV CPU.

4. Results

4.1. ALU and Branch Comparison

This is an RTL of ALU:

This is an RTL of Branch comparison



Now we will check if the alu is running correctly or not. At first, we will initialize the value 0 for the register, then perform some basic calculations that are designed in alu

and x1, x1, x0	addi x11, x11, -5
and x2, x2, x0	addi x2, x2, 0x400
and x3, x3, x0	addi x1, x5, 9
and x4, x4, x0	xori x3, x5, 9
and x5, x5, x0	ori x4, x5, 9
and x6, x6, x0	andi x15, x5, 7
and x7, x7, x0	slli x7, x5, 2
and x8, x8, x0	slti x8, x5, -2
and x9, x9, x0	sltiu x9, x5, -2
and x10, x10, x0	srli x10, x5, 1
and x11, x11, x0	srai x14, x11, 2
addi x5, x5, 5	sw, x10, 0(x2)
addi x6, x6, 6	lw, x6, 0(x2)

At first, we ran it on Venus to see the result

zero 0x00000000 ra (x1) 0x0000000E sp (x2) 0x00000400 gp (x3) 0x0000000C tp (x4) 0x0000000D t0 (x5) 0x00000005 t1 (x6) 0x00000002 t2 (x7) 0x00000014 s0 (x8) 0x00000000 s1 (x9) 0x00000001 a0 (x10) 0x00000002 a1 (x11) 0xFFFFFFFB a2 (x12) 0x00000000 a3 (x13) 0x00000000 a4 (x14) 0xFFFFFFFE a5 (x15) 0x00000005	At first, we initialize x5 with 5, x6 with 6, x11 with -5, and x2 with 0x400. Then based on the result on Venus we have this: x1 = 14 x2 = 1024 x3 = 12 x4 = 13 x7 = 20 x8 = 0 x9 = 1 x10 = 2 x14 = -2 X15 = 5

Then we test our hardware simulation on Ubuntu

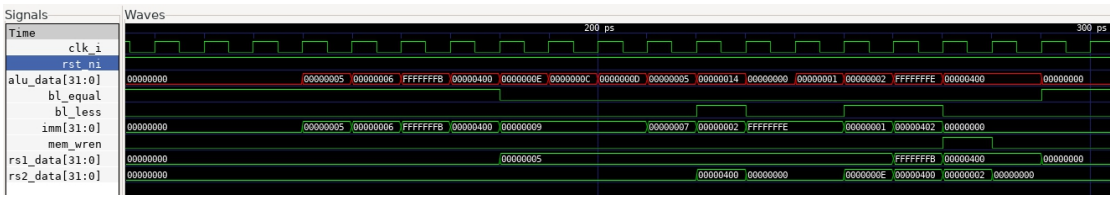


Figure 4.1: Ubuntu waveform

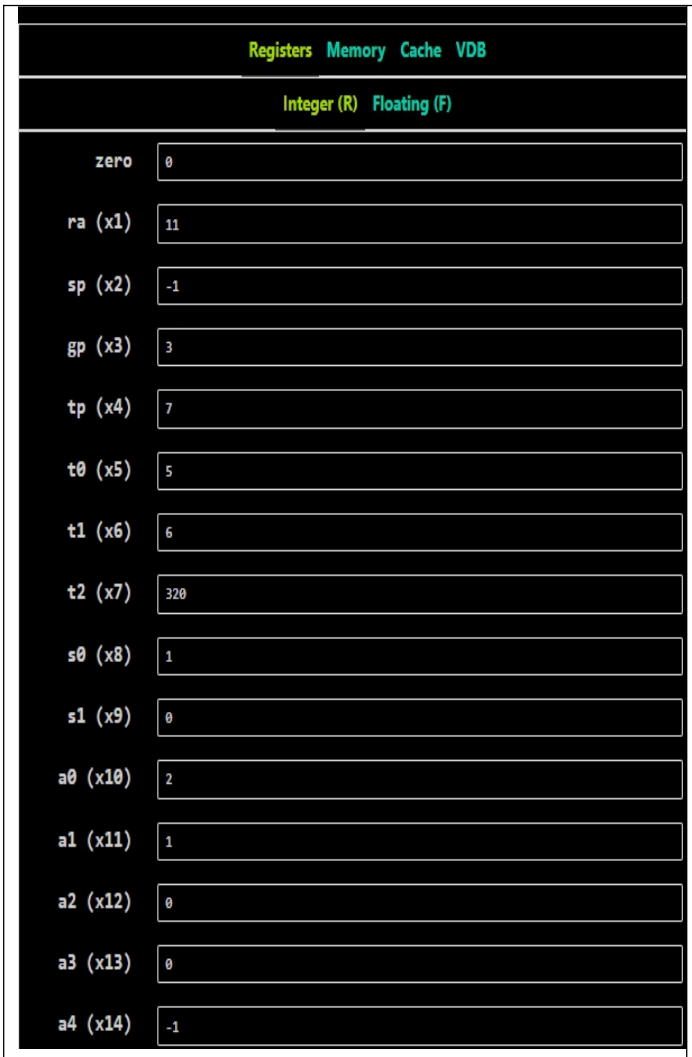
Let's have a look at alu_data (output of alu module) from the waveform.

alu_data (base 10)	description
5	addi x5, x5, x5
6	addi x6, x6, 6
-5	addi x11, x11, -5
1024 (400_{16})	addi x2, x2, 0x400
14	addi x1, x5, 9
12	xori x3, x5, 9
13	ori x4, x5, 9
5	andi x15, x5, 7
20	slli x7, x5, 2
0	slti x8, x5, -2
1	sltiu x9, x5, -2
2	srli x10, x5, 1
-2	srai x14, x11, 2
1024	sw, x10, 0(x2)
1024	lw, x6, 0(x2)

=> We see that the result of alu and branch compare module from the waveform are the same as the result that we ran on Venus

Then we check with another instruction. This code will check if the alu is running correctly or not. At first, we will initialize value 0 for the register, then perform some basic calculation

and x1, x1, x0
and x2, x2, x0
and x3, x3, x0
and x4, x4, x0
and x5, x5, x0
and x6, x6, x0
and x7, x7, x0
and x8, x8, x0
and x9, x9, x0
and x10, x10, x0
and x11, x11, x0
addi x11, x11, 1
	addi x5, x5, 5
	addi x6, x6, 6
	add x1, x5, x6
	sub x2, x5, x6
	xor x3, x5, x6
	or x4, x5, x6
	sll x7, x5, x6
	slt x8, x2, x6
	sltu x9, x2, x6
	srl x10, x5, x11
	sra x14, x2, x11



At first, we initialize x5 with 5, x6 with 6, x11 with -5, and x2 with 0x400.

Then based on the result on Venus we have this:

x1 = 11

x2 = -1

x3 = 3

x4 = 7

x7 = 320

x8 = 1

x9 = 0

x10 = 2

x14 = -1

Then we run our hardware simulation on Ubuntu

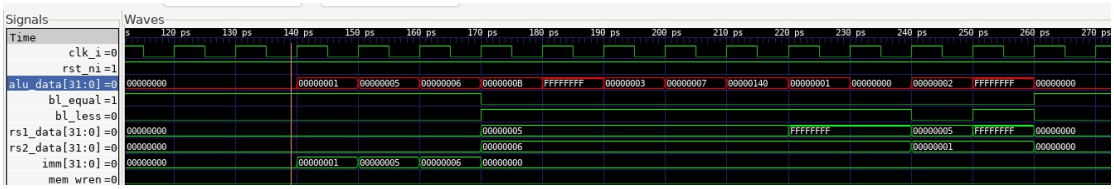


Figure 4.2: Ubuntu waveform

Let's have a look at alu_data (output of alu module) from the waveform

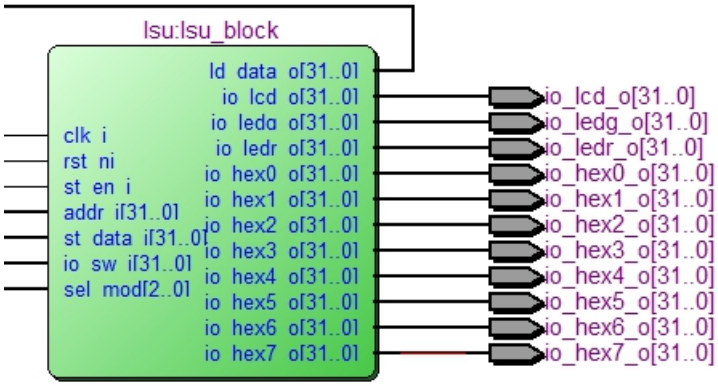
alu_data	description
1	addi x11, x11, 1
5	addi x5, x5, 5
6	addi x6, x6, 6
11	add x1, x5, x6
-1	sub x2, x5, x6
3	xor x3, x5, x6
7	or x4, x5, x6
320	sll x7, x5, x6
1	slt x8, x2, x6
0	sltu x9, x2, x6
2	srl x10, x5, x11
-1	sra x14, x2, x11

=> We see that the result of alu and branch compared from the waveform are the same as the result that we ran on Venus

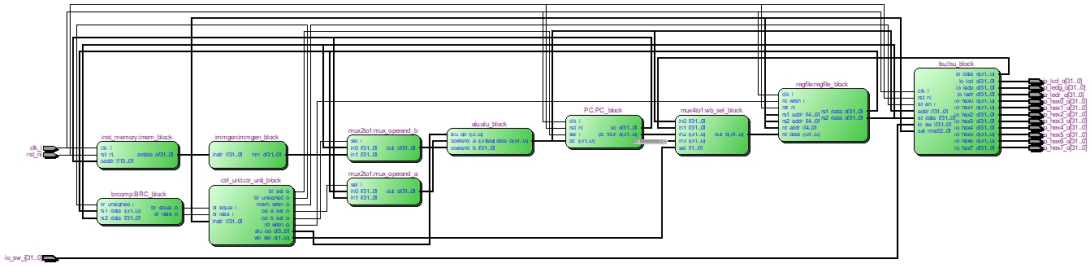
4.2. Load-Store Unit

We will test this part later in the verification part.

Here is our RTL of LSU module:



4.3. Single-Cycle Processor



This is an RTL of our module

We run our project on Ubuntu. We check lint, then make sim and make the wave

```
tom1710@LAPTOP-7HRREBED:~/lab1/tb$ make lint
<> LINT CHECK ----------------
tom1710@LAPTOP-7HRREBED:~/lab1/tb$ make sim
<> SIMULATING ----------------
tom1710@LAPTOP-7HRREBED:~/lab1/tb$ make wave
<> WAVEFORMS ----------------
```

Figure 4.3: Check lint, make sim, make wave on Ubuntu

4.3.1. Testbench

Now we write a testbench to check our hardware by testing 37 instructions. We will write a program for each sub-group of instructions and observe the output values if they match with our expectations or not.

ADD, SUB, ADDI instructions

```
Check instructions : ADD, SUB, ADDI
All intrsuctions have PASSED
Check instructions : ADD, SUB, ADDI
All intrsuctions have PASSED
Check instructions : ADD, SUB, ADDI
All intrsuctions have PASSED
tom1710@QUYLOC1710:~/lab1/tb$
```

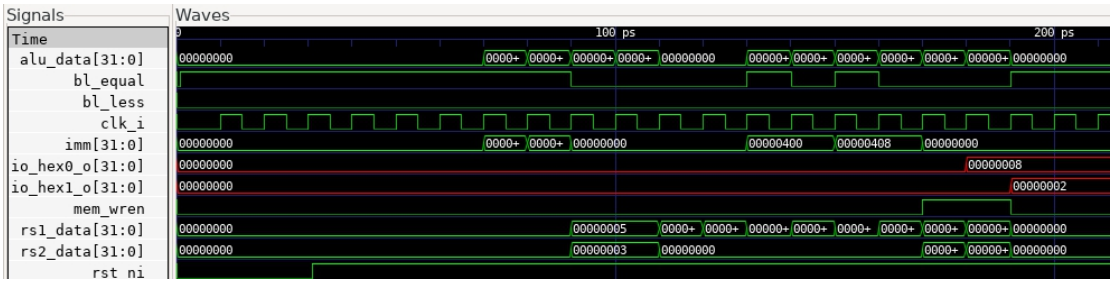


Figure 4.4: Waveform of ADD, SUB, ADDI instructions

In this part we add immediate numbers 3 and 5 to a register, then add them together then we have 8, we also subtract it then we get 2. Finally, we store the result at addresses 0x800 and 0x810. From the waveform, the result is the same as what we have calculated by hand. So our hardware is correct for these instructions.

AND, OR, XOR , ANDI , ORI, XORI instructions

```
Check instructions : AND, OR, XOR, ANDI, ORI, XORI
All intrsuctions have PASSED
Check instructions : AND, OR, XOR, ANDI, ORI, XORI
All intrsuctions have PASSED
Check instructions : AND, OR, XOR, ANDI, ORI, XORI
All intrsuctions have PASSED
tom1710@QUYLOC1710:~/lab1/tb$
```

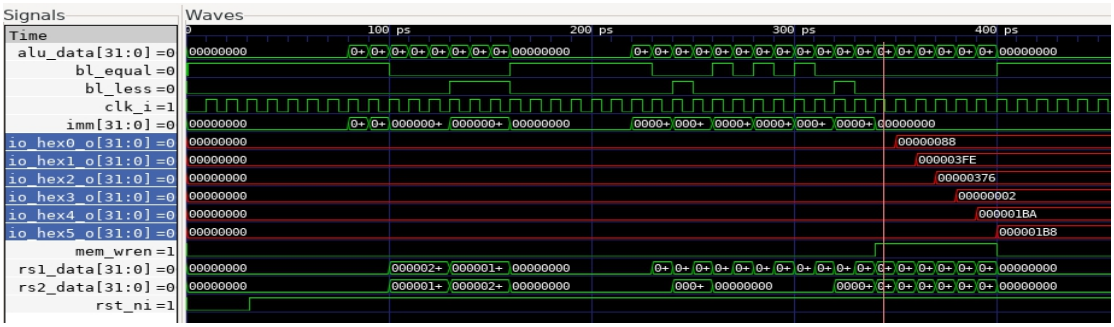


Figure 4.5: Waveform of AND, OR, XOR , ANDI , ORI, XORI instructions

In this part, we initialize 2 values: 0x1BA and 0x2CC, then use AND, OR, XOR, ANDI, ORI, and XORI instructions with these 2 values. Then we see that the waveform gives a correct value as Venus.

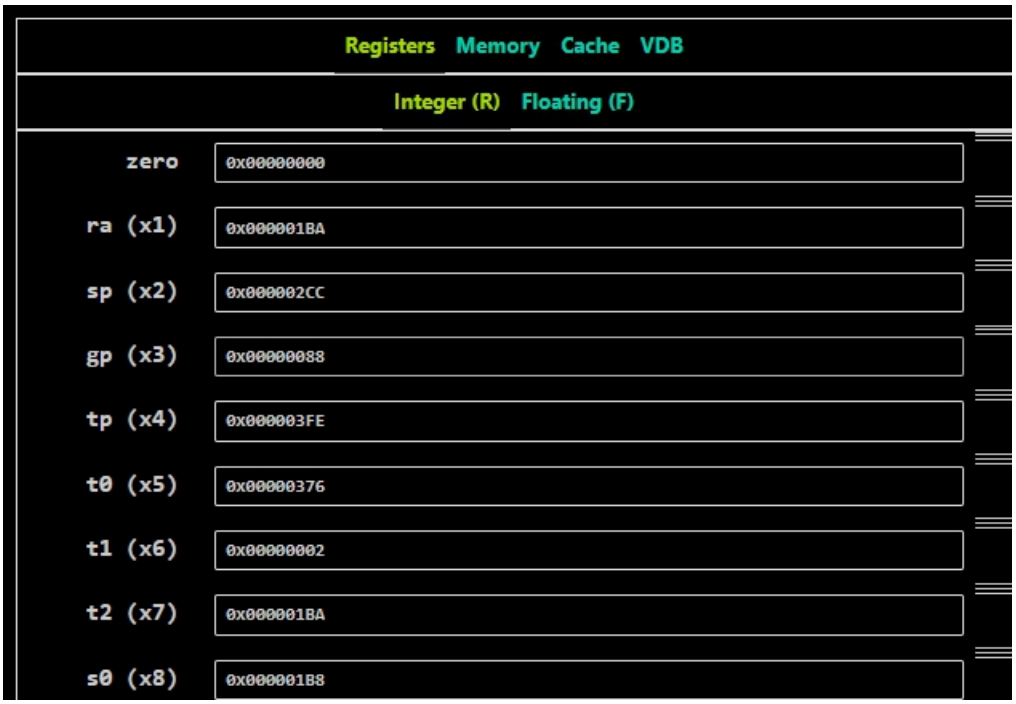


Figure 4.6: Result on Venus of AND, OR, XOR , ANDI , ORI, XORI instructions

SW, SH, SB, LW, LH, LB, LHU, LBU instructions

```
Check instructions : SW, SH, SB, LW, LH, LB, LHU, LBU
All intrsuctions have PASSED
Check instructions : SW, SH, SB, LW, LH, LB, LHU, LBU
All intrsuctions have PASSED
Check instructions : SW, SH, SB, LW, LH, LB, LHU, LBU
All intrsuctions have PASSED
tom1710@QUYLOC1710:~/lab1/tb$
```

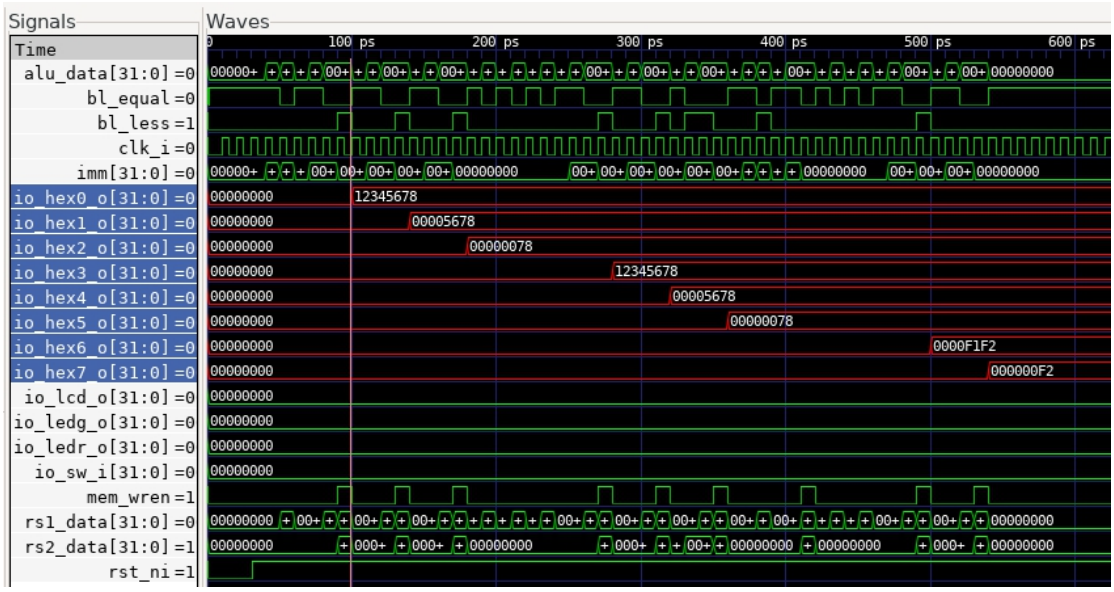


Figure 4.7: Waveform of SW, SH, SB, LW, LH, LB, LHU, LBU instructions

We store a value of 32-bit number 0x12345678 to a register, then we use SW, SH, and SB to store a value in memory at addresses 0x800, 0x810, and 0x820. Next, we use LW, LH, and LB to load out data at address 0x800, we use SW to store these values again in memory at addresses 0x830, 0x840, and 0x850. Then we try to store a 32-bit negative number 0xFFFFF1F2 into a register, then we use LHU and LBU to load out them and then store them into memory at addresses 0x860 and 0x870. From the waveform, we see that the result is the same as our expectation so these instructions have passed.

SLT, SLTU, SLI, SLTIU instructions

```
Check instructions : SLT, SLTU, SLI, SLTIU
All intrsuctions have PASSED
Check instructions : SLT, SLTU, SLI, SLTIU
All intrsuctions have PASSED
Check instructions : SLT, SLTU, SLI, SLTIU
All intrsuctions have PASSED
tom1710@QUYLOC1710:~/lab1/tb$
```

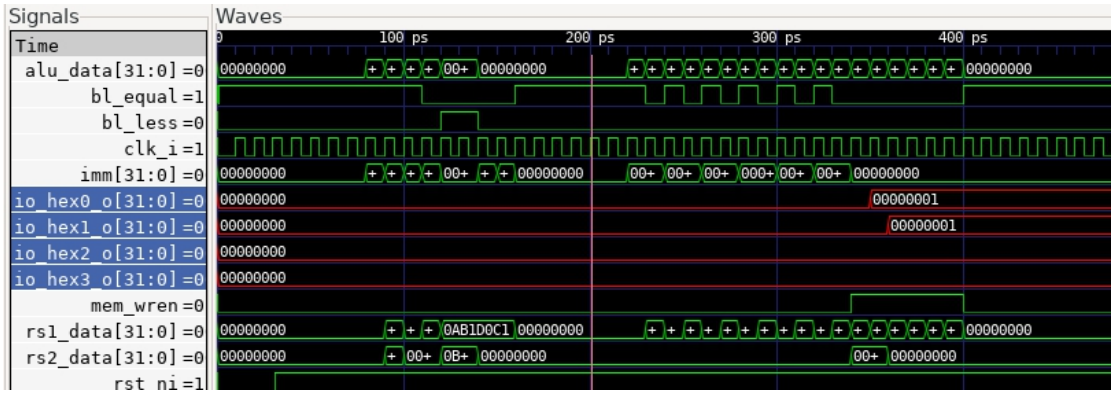


Figure 4.8: Waveform of SLT, SLTU, SLI, SLTIU instructions

In this section we will compare 2 values, first, we compare 0x0AB1D0C1 with 0x0B1FC02C, then 0x0AB1D0C1 is less than, so the result will be 1 for slt and sltu instruction. Then we compare immediate and compare immediate unsigned 0x0AB1D0C1 with 0xFF and 0xB4, then, of course, the result will be 0. The waveform shows that our instructions are run correctly.

SLL, SRL, SRA, SLLI, SRLI, SRAI instructions

```
Check instructions : SLL, SRL, SRA, SLLI, SRLI, SRAI
All intrsuctions have PASSED
Check instructions : SLL, SRL, SRA, SLLI, SRLI, SRAI
All intrsuctions have PASSED
Check instructions : SLL, SRL, SRA, SLLI, SRLI, SRAI
All intrsuctions have PASSED
tom1710@QUYLOC1710:~/lab1/tb$
```

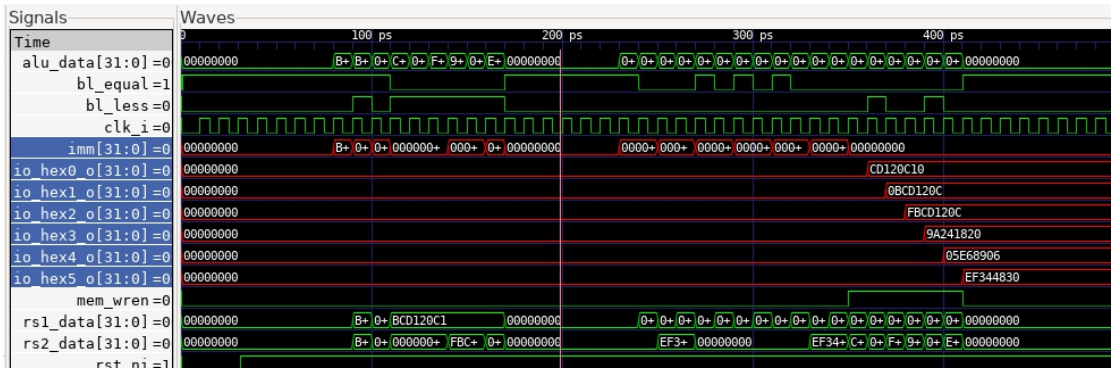


Figure 4.9: Waveform of SLL, SRL, SRA, SLLI, SRLI, SRAI instructions

In this part we will store value 0xBCD120C1 into a register and then shift it, you see that if we shift 4 bit: SLL, the result will be CD120C10; SRL, the result will be 0BCD120C; SRA, the result will be FBCD120C; then SRLI 5 bit, the result is 9A241820, SRLI 5 bit, the result is 05E68906 and SRAI 5 bit, EF344830 is the result of this instruction. So all of the result is the same as the result that we calculated by Venus:

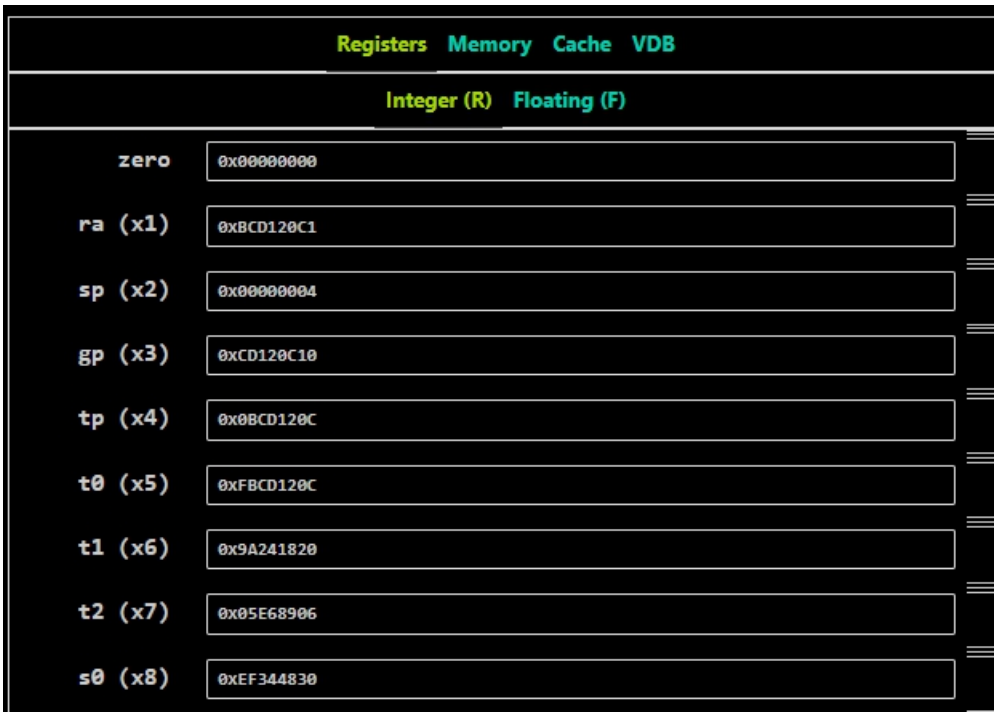


Figure 4.10: Result on Venus of SLL, SRL, SRA, SLLI, SRLI, SRAI instructions

LUI, AUIPC instructions

```
Check instructions : LUI, AUIPC
All intrsuctions have PASSED
Check instructions : LUI, AUIPC
All intrsuctions have PASSED
Check instructions : LUI, AUIPC
All intrsuctions have PASSED
tom1710@QUYLOC1710:~/lab1/tb$
```

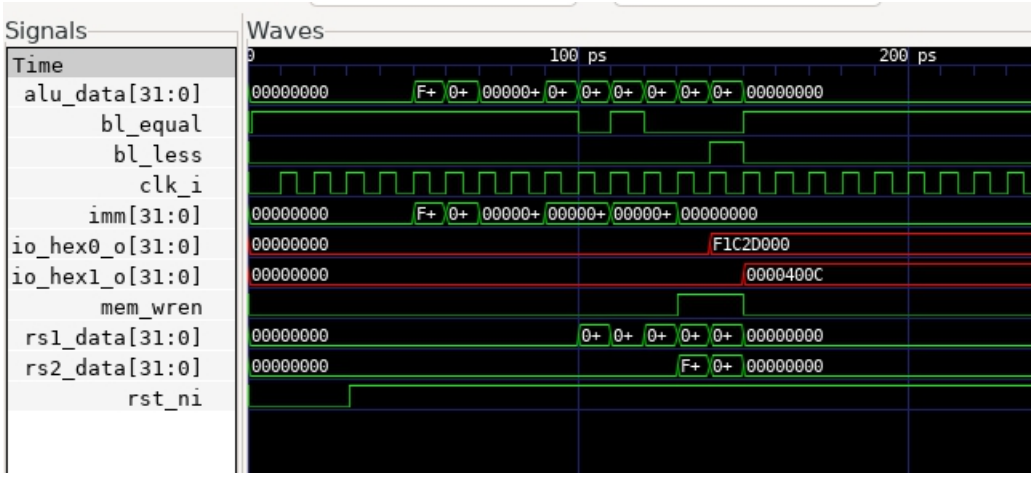
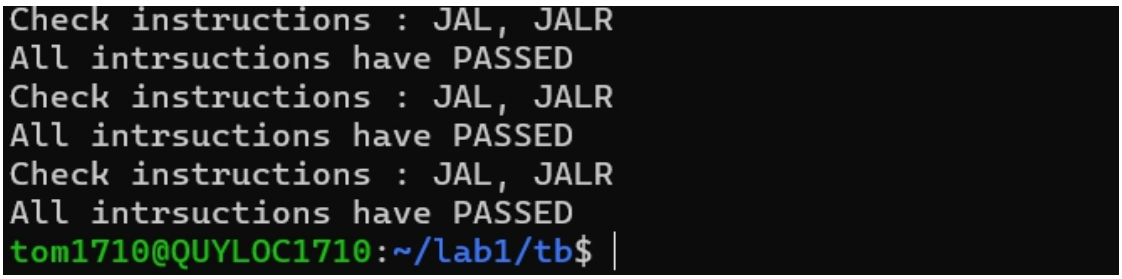


Figure 4.11: Waveform of LUI, AUIPC instructions

It is clear that if we LUI 0xF1C2D then the result will be 0xF1C2D000, and the auipc instruction is executed when PC = 00C, then the result will be 400C. Show the waveform shows that our hardware runs correctly with these instructions.

JAL, JALR instructions



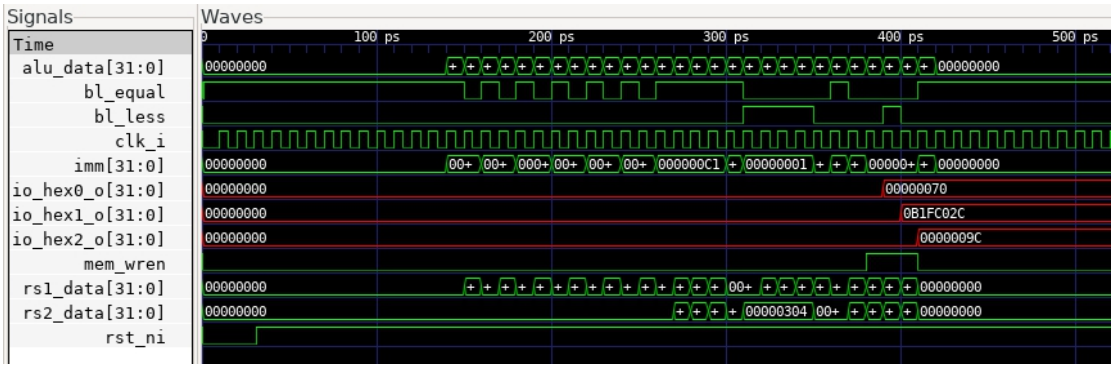
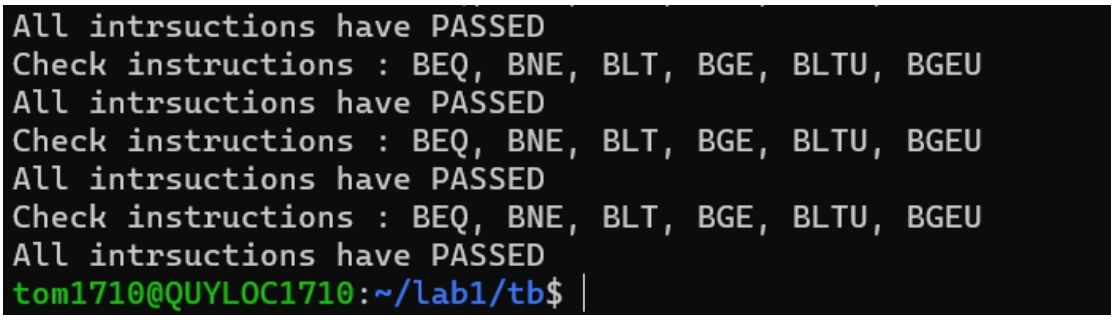


Figure 4.12: Waveform of JAL, JALR instructions

First, we reset registers and set the address value of Hex0,1,2. I randomly added x1 with C1 4 times so the value in x1 now is 0x304 and PC now is 0x48. Jal instruction is taken and x2 is register store the return address x2 = 0x48+4 = 0x4C and it jumps to

label add1. We add x3 with 1 for 4 times and place instruction jalr x5,x2,0. In this instruction, the current PC is 0x74, x5 stores the return address which is 0x74 + 4 = 0x78, x2 which is 0x4C and offset 0 so it jumps back to the next instruction right behind jal (PC = 0x4C). We use lui, addi to fill x4 with value 0x0B1FC02C for checking whether the jump is correct or not. We store the value to hex0,1,2 for checking waveform.

BEQ, BNE, BLT, BGE, BLTU, BGEU instructions



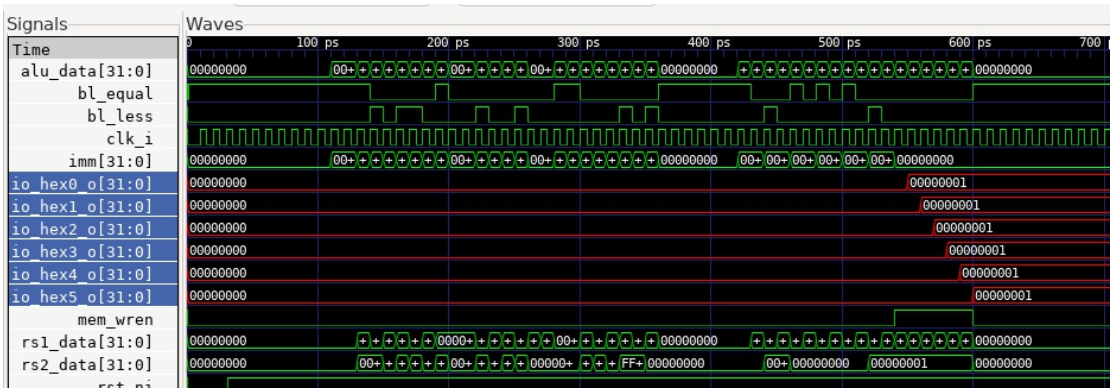


Figure 4.13: Waveform of BEQ, BNE, BLT, BGE, BLTU, BGEU instructions

First, each time a branch instruction is executed correctly, we increase by 1 for x3,4,5,6,7,8 to note for 6 branch instructions. We see that x1 = x2 = 30 then beq will branch into label equal, I increase x3 to note that it branches correctly. Next, I sub the value of x1 by -10, so it is clear that x1 != x2 so bne is taken to label notequal and x4 increases 1 as usual. Add x1 with 5 so x1 < x2, blt is taken to label lessthan, increase x5. Adding x1 with value 50 makes it bigger than x2, so bge is taken to label greater than, increase x6. I reset x1,x2, set x1 = -30, x2 = 15. Because we use bgeu which treats numbers as unsigned so value of x1 is 0xFFFFFFE2, and x2 is 0x0000000F, obviously x1 > x2 and increase x7. For the last check for bltu x2,x1, we add x2 with -74 so the value in x2 now is 0xFFFFFFC4 still less than 0xFFFFFFE2, so the branch

less than unsigned is taken and increases x8. Note that between labels, we already added some instructions that verify whether it branches or not.

After verifying our hardware by test bench. We are going to test our design again with a program. We store a 32-bit value to the register then take the 8 first bits as output and split it into hundreds, tens, and ones and store it in memory at locations 0x200, 0x210, and 0x220 respectively.

```
addi x4,x0,0x64
addi x5,x0,0xa
addi x6,x0,0x220
addi x7,x0,0x210
addi x8,x0,0x200

addi x9,x0,0x300
addi x14,x0 0x310
addi x17, x0, 0x320

lui  x15, 0x12345
addi x15,x15,0x678

sw x15,0(x9)
sh x15,0(x14)
sb x15, 0(x17)

addi x11,x0,0x00
addi x12,x0,0x00

lbu x13,0(x9)

jal compare2
sub1: sub x13,x13,x4
 addi x11,x11,0x01
compare2: bge x13,x4,sub1
 jal compare1
sub2: sub x13,x13,x5
 addi x12,x12,0x1
compare1: bge x13,x5,sub2
sw x11, 0(x6)
sw x12, 0(x7)
sw x13, 0(x8)
```

Figure 4.14: Assembly code

At first I store number 0x12345678 in to register x15, then the last 8-bit will be $78_{16} = 120_{10}$

So in the memory, you see that 1 is stored at 0x220, 2 is stored at 0x210 and 0 is stored at 0x200.

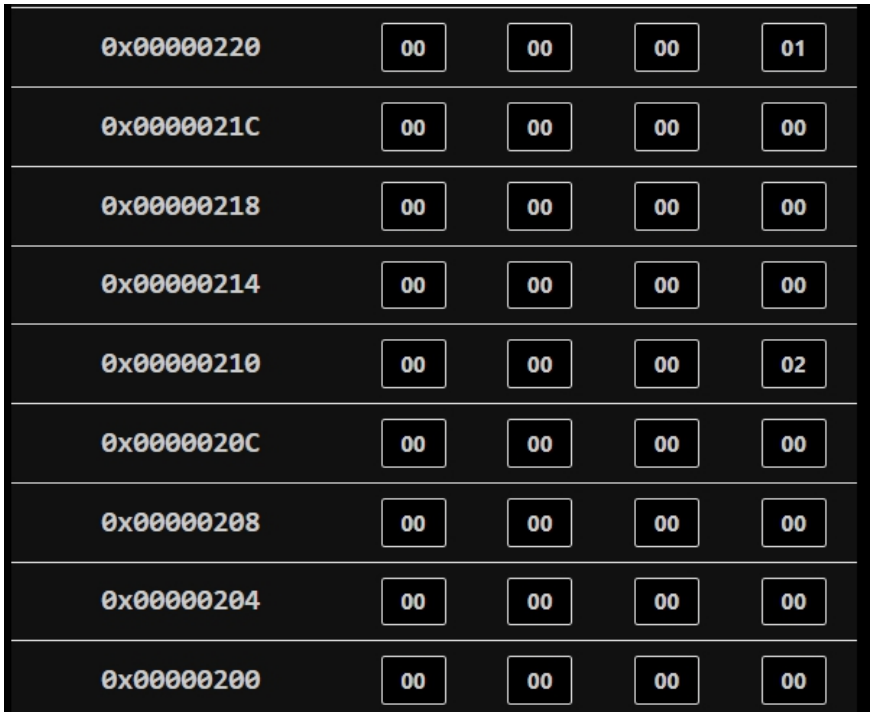


Figure 4.15: Venus result

After we pour the code into immem then we have this waveform

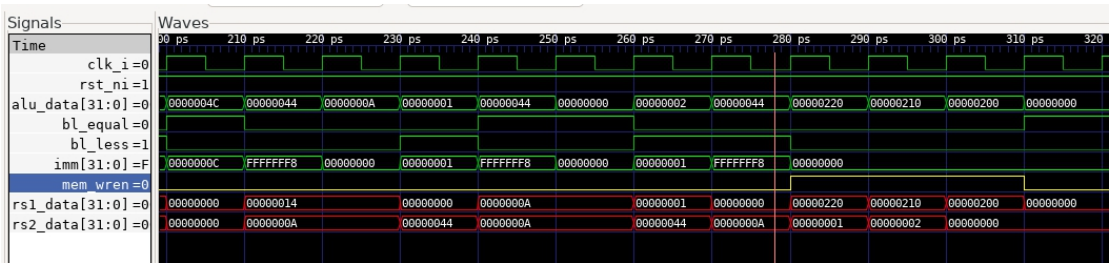
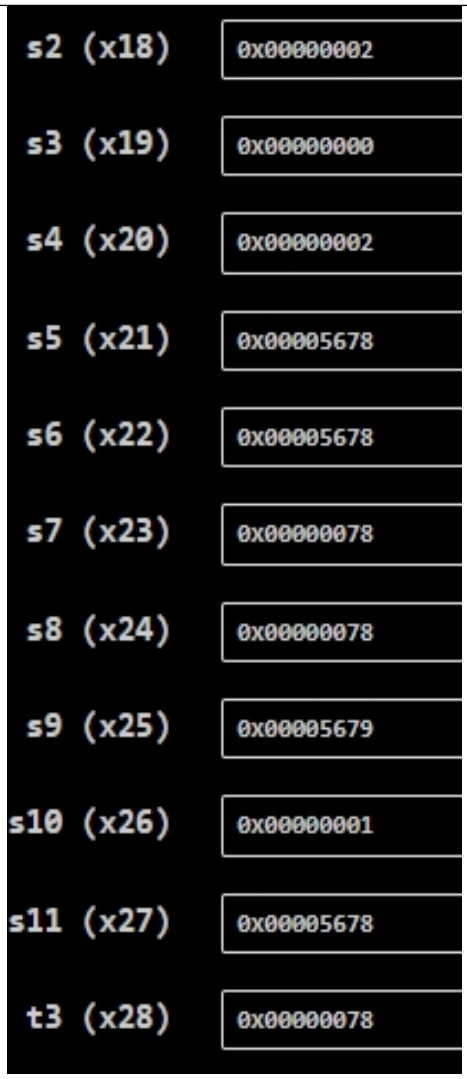


Figure 4.16: Waveform

It is clear to see that to the right of the vertical orange bar, the waveform in red color shows us that 01 is stored at 0x220, 02 is stored at 0x210 and 00 is stored at 0x200. At that time, mem_wren (yellow) will turn on to store data in memory. So Our waveform result is the same as the result that we ran on Venus.

Then our code continue with this part:

```
35 lw x18, 0(x7)
36 lw x19, 0(x8)
37 lw x27, 0(x14)
38 lw x28, 0(x17)
39 add x20, x18, x19
40
41 lh x21, 0(x9)
42 add x22, x21, x19
43
44 lb x23, 0(x9)
45 add x24, x23, x19
46
47 beq x22, x21, equal
48 addi x25, x22, 2
49 j break
50 equal:
51 addi x25, x22, 1
52
53 bne x25, x22, notequal
54 addi x25, x22, 1
55 j break
56 notequal:
57 sub x26, x25, x22
58 break:
```



Now, we load out the result and take some branch conditions in order to check our hardware.

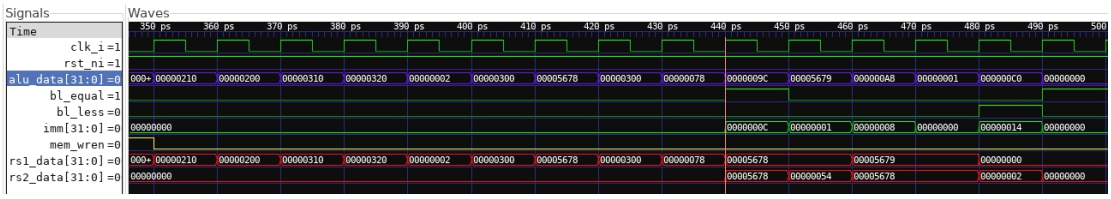


Figure 4.17: Waveform

From the waveform, we see that our hardware loads the correct data that we stored in memory, and branch instruction runs correctly. For example, next to the vertical orange bar, it is a beq instruction, you see that rs1_data = rs2_data, then bl_equal = 1, then it will jump to label and execute the instruction here which is addi, then the result (alu_data - 450ps) is 5679 which is 5678 + 1. Then we compare 0x5678 with 0x0x5679 by bne, and if it is not equal then we subtract 0x5678 by 0x5679, and the result (alu_data at 470 ps) is 1.

4.3.3 Application

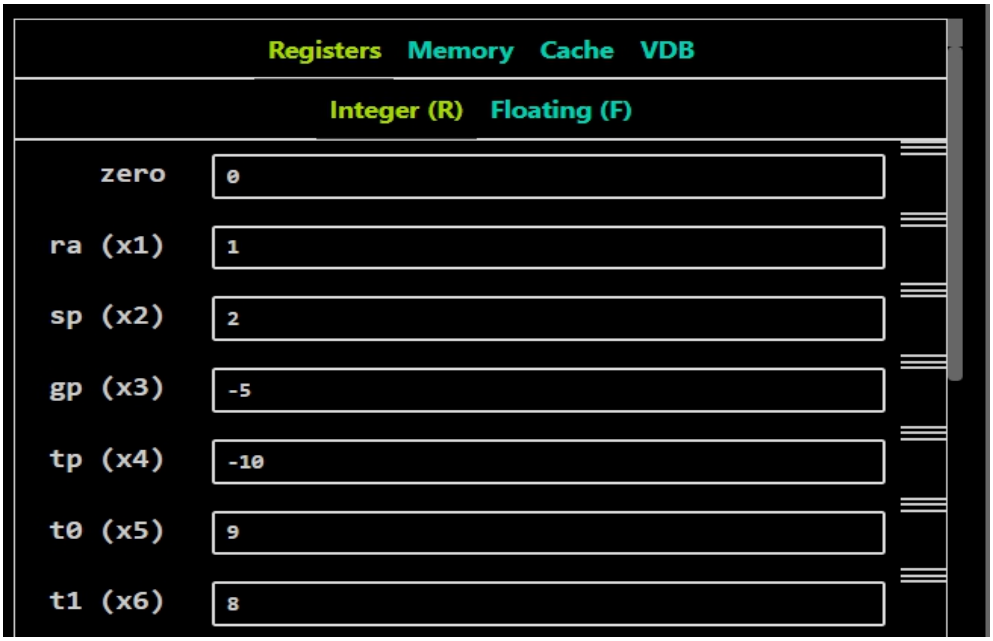
Input 3 2-D coordinates of A, B, and C. Determine which point, A or B, is closer to C using LCD as the display.

In this part, we also test our lsu module which is used for control peripherals. Overview For this part, we will write an assembly code that will calculate which point is closer to point C. We also have access to the data memory to control the peripherals such as lcd, ledr, and 7_segment.

For lcd we need to store data at address 0x

Case 1: AC < AB

With x1, x2 stand for x, y of point A(0, 1); x3,x4 stand for x,y of point B(-5, -10); x5,x6 stand for point C(9,8)



We can easily see that point A is closer to point C than point B

Figure 4.18: Result on Venus Case 1: AC < AB

So at the register, 0x8A0 will save the ASCII code of the letter “A” which is 41

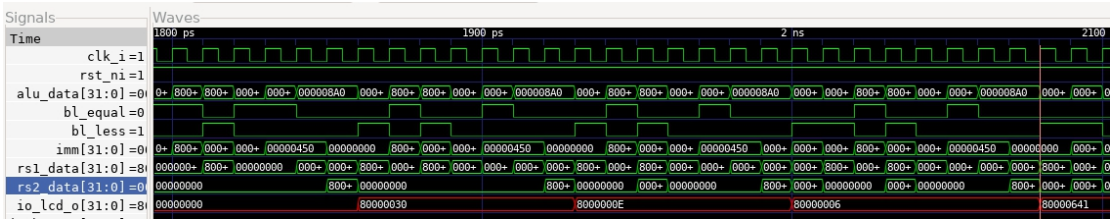


Figure 4.19: Waveform case1

You see that on the waveform the signal io_lcd_o which is the address 0x8A0 will have the result 80000641 (To the right of the orange vertical bar), 41 here is letter A in Ascii.

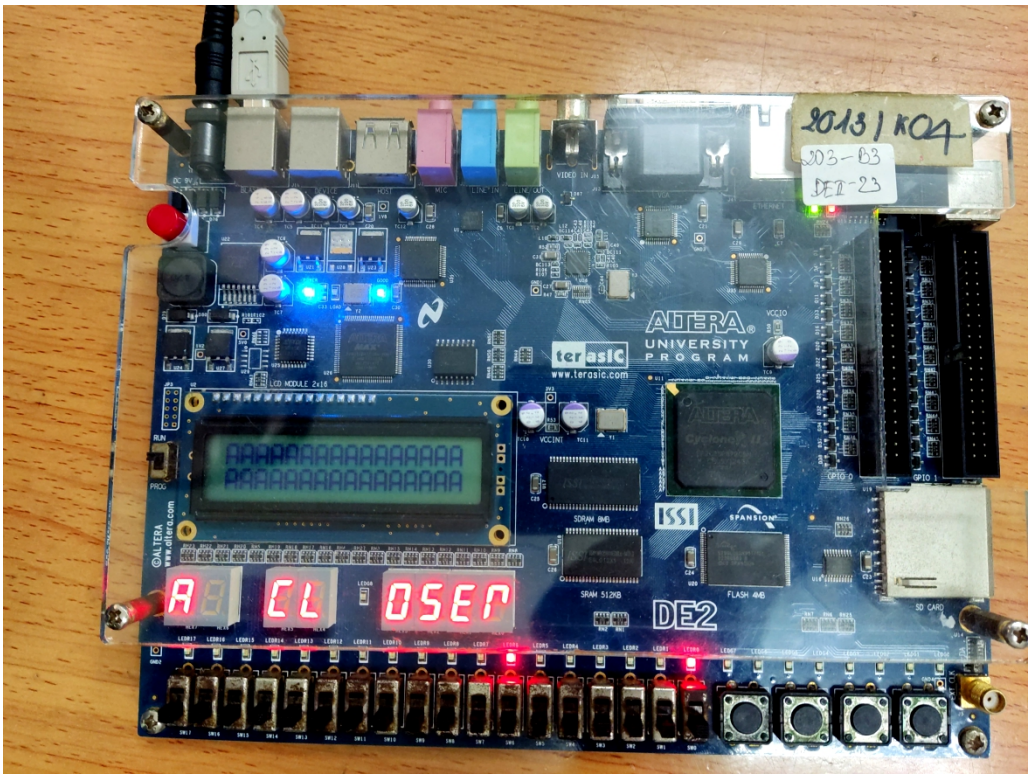


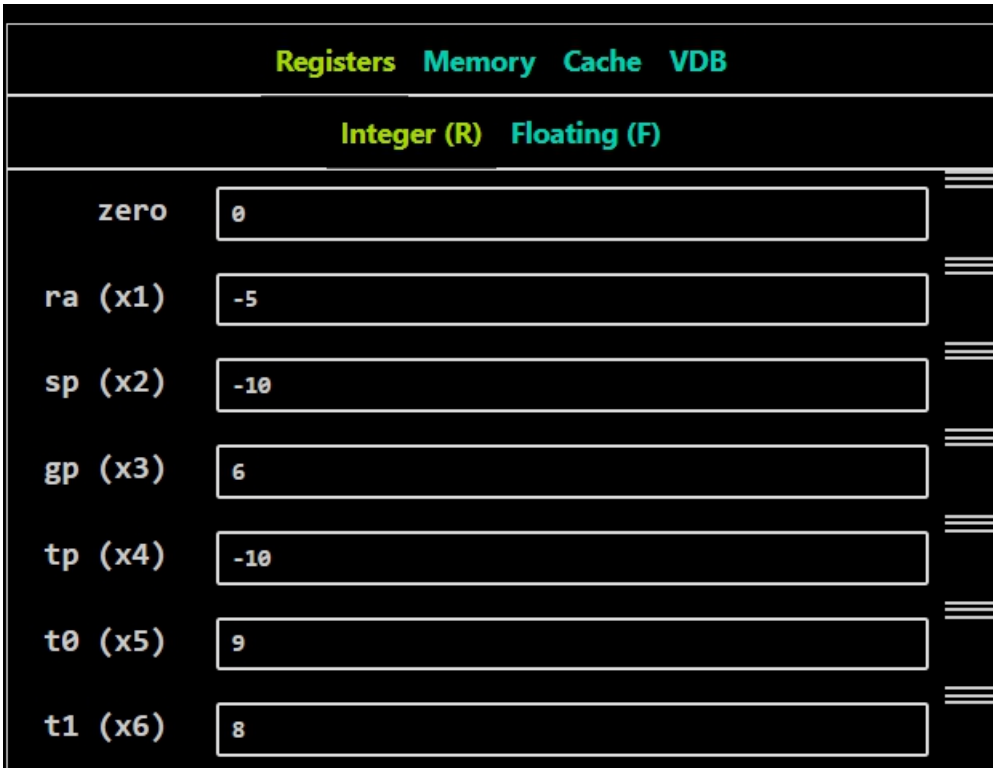
Figure 4.20: Implement on Kit DE2

You can see that the letter A is shown on the LCD because point A is closer to point C than point B.

We also write a sentence “A CLOSER” on 7_segment, and ledr0 and ledr6 turn on which is stand for $01000001_2 = 41_{16}$ which is ASCII code for letter “A”

Case 2: AC > AB

With x1, x2 stand for x, y of point A(-5, -10); x3,x4 stand for x,y of point B(6, -10); x5,x6 stand for point C(9,8)



Now we see that point B is closer to point C than point A

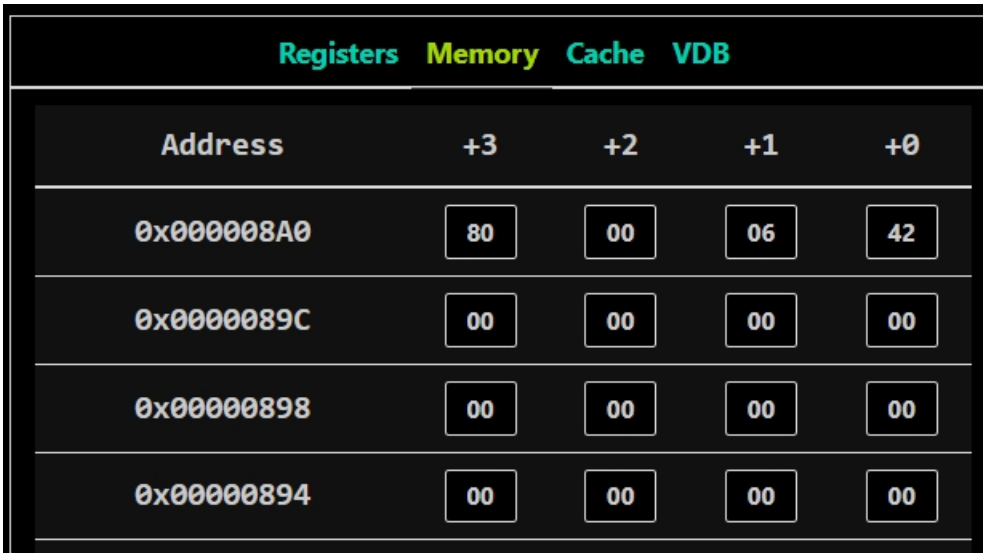


Figure 4.21: Result on Venus Case 2: AC > AB

So at the register, 0x8A0 will save the ASCII code of the letter “B” which is 42

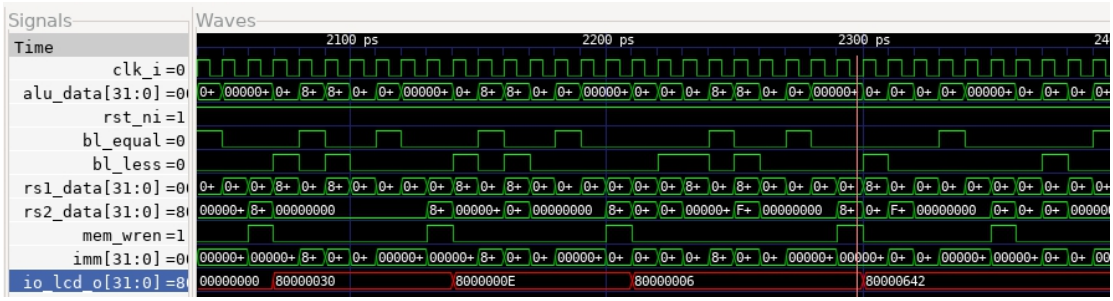


Figure 4.22: Waveform case2

You see that on the waveform the signal io_lcd_o which is the address 0x8A0 will have the result 80000642 (To the right of the orange vertical bar), 41 here is letter B in ASCII.

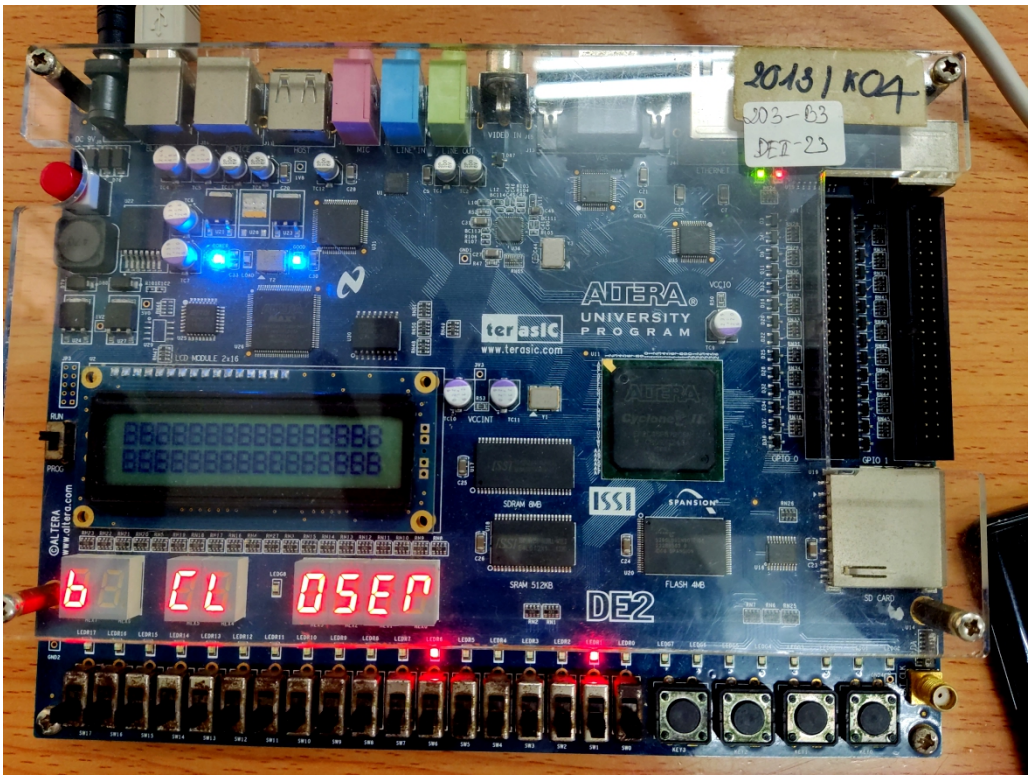


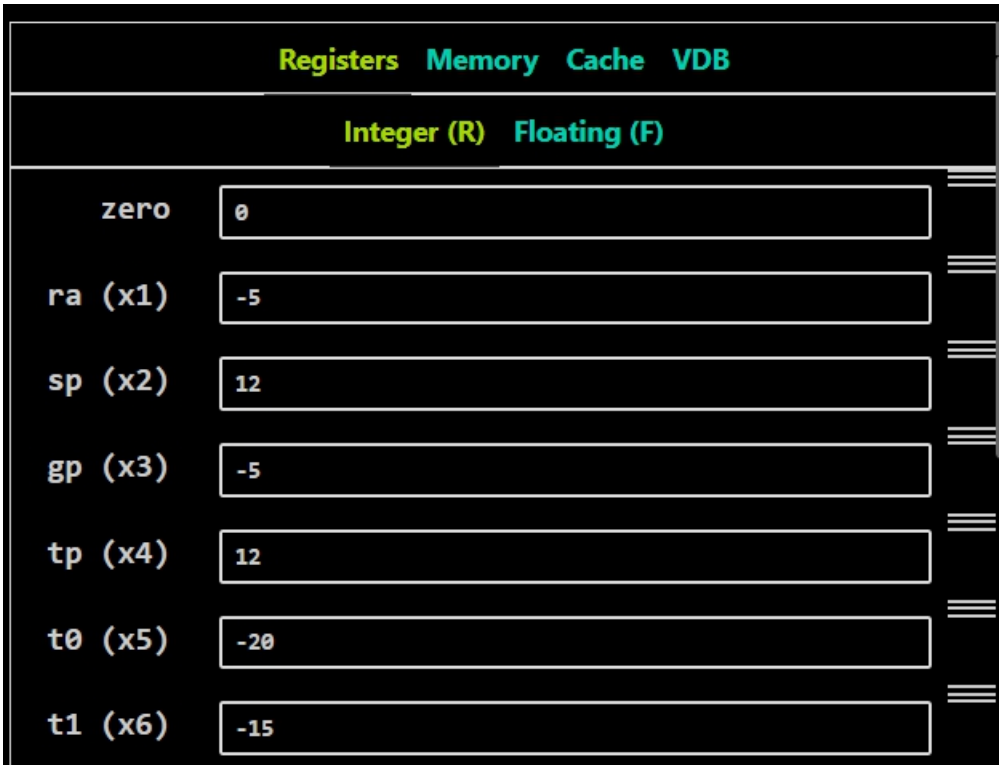
Figure 4.23: Implement on Kit DE2

You can see that the letter B is shown on the LCD because point B is closer to point C than point A.

We also write a sentence “b CLOSER” on 7_segment, and ledr1 and ledr6 turn on which stands for $01000010_2 = 42_{16}$ which is the ASCII code for the letter “B”.

Case 3: AC = AB

With x1, x2 stand for x, y of point A(-5, 12); x3,x4 stand for x,y of point B(-5, 12); x5,x6 stand for point C(-20,-15)



We can easily see that point A is closer to point C than point B

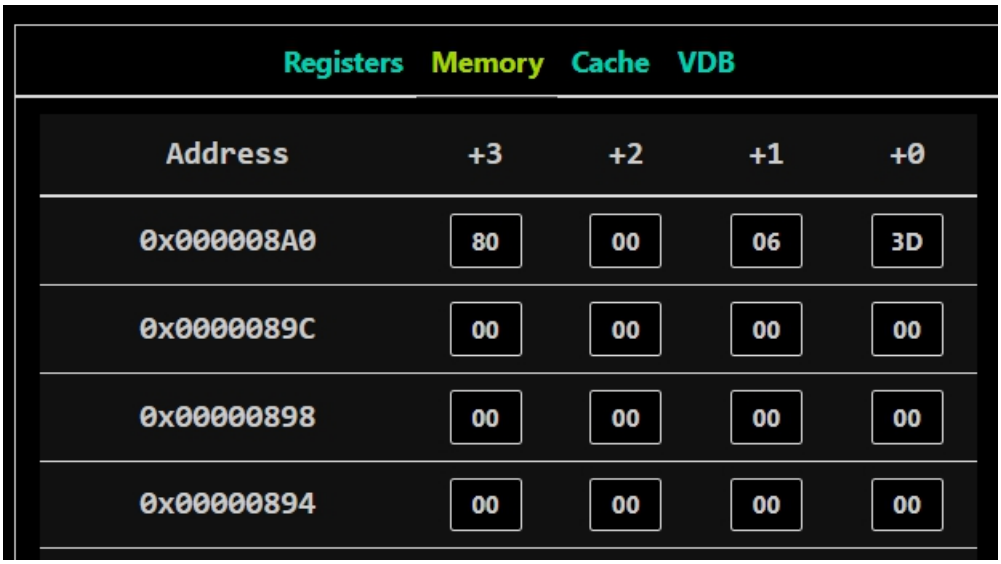


Figure 4.24: Result on Venus Case 3: AC = AB

So at the register, 0x8A0 will save the ASCII code of operation "=" which is 3D

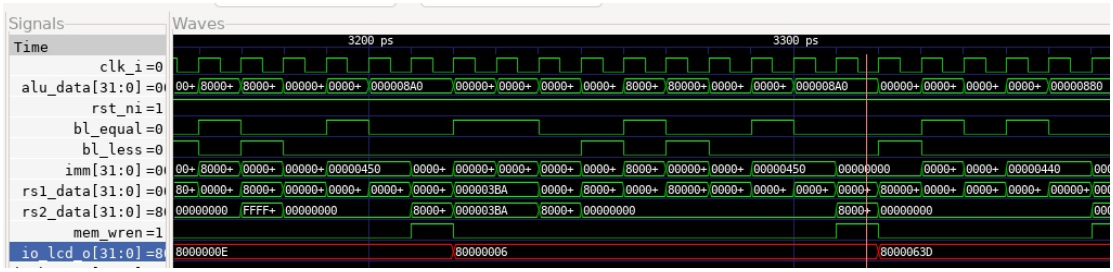


Figure 4.25: Waveform case3

You see that on the waveform the signal io_lcd_o which is the address 0x8A0 will have the result 8000063D (To the right of the orange vertical bar), 3D here is an operator = in ASCII.

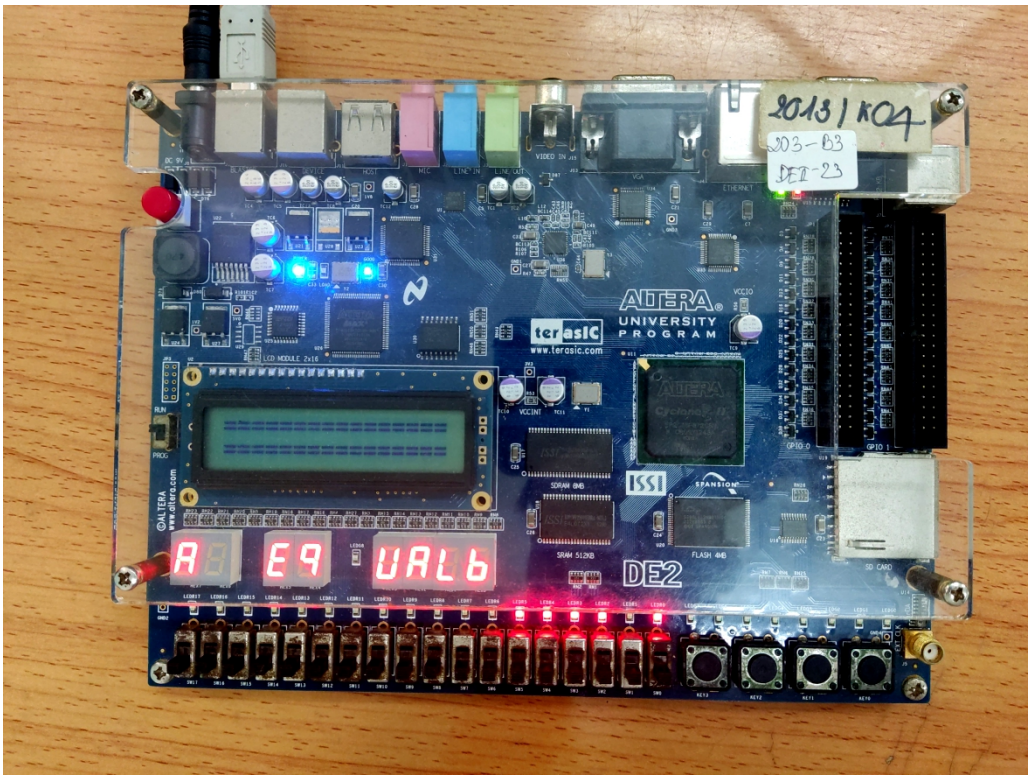


Figure 4.26: Implement on Kit DE2

You can see that operator = is shown on the LCD because point A and point B have the same distance to point.

We also write a sentence “A EqUALb” on 7_segment, and ledr0, ledr2,ledr3, ledr4, and ledr5 turn on which stands for $00111101_2 = 3D_{16}$ which is the ASCII code for operator “ = ”.

5. Discussion

In this project, we learned how to write System Verilog, design a RISC V CPU and its instructions. Throughout the project, we manage to fix bugs and implement it on kit DE2, run some programs and an application. We did not have an alternative design because it took a lot of time to complete the standard RISC V CPU. We hope to improve it in the future.