COE 181. 1 Lab Report

Lab Number: 7

Lab Title: Single-Cycle MIPS Control

Student Name: Jemuel Amporinguez

Student Email:jemuel.amporinguez@g.msuiit.edu.ph

Student Phone Number:09763708855

Objectives:

- Gain a comprehensive understanding of single-cycle processor architecture and its key components.
- Implement key datapath elements such as the ALU, data memory, instruction memory, and register file.
- Integrate all components into a fully functional single-cycle processor capable of executing basic instructions.
- Demonstrate the functionality of your processor by executing a predefined set of instructions and verifying correct outputs.

Procedures:

For Task #1:

```
1  `timescale 1ns / 1ps
2
3  module control (
4      input [31:0] instruction, // The full 32-bit instruction
5      output reg RegDst,        // Selects destination register
6      output reg ALUSrc,        // Selects ALU input source
7      output reg MemToReg,      // Selects data to write back to the register
8      output reg RegWrite,      // Enables writing to the register file
9      output reg MemRead,       // Enables memory read
10     output reg MemWrite,      // Enables memory write
11     output reg [5:0] ALUOp    // ALU operation code
12 );
13
14     // Extract opcode and function code
15     wire [5:0] opcode = instruction[31:26];
16     wire [5:0] funct = instruction[5:0];
17
18     // Opcodes
19     localparam LW   = 6'b100011;
20     localparam SW   = 6'b101011;
21     localparam ADDI = 6'b001000;
22
23     // R-type instructions share the same opcode
24     localparam R_TYPE = 6'b000000;
25
26     // Function codes for R-type instructions
27     localparam ADD  = 6'b100000;
28     localparam SUB  = 6'b100010;
29     localparam AND  = 6'b100100;
30     localparam OR   = 6'b100101;
31     localparam NOR  = 6'b100111;
32     localparam XOR  = 6'b100110;
33
34     // Control signal assignment
35     always @(*) begin
36         // Default values for control signals
37         RegDst   = 0;
38         ALUSrc   = 0;
39         MemToReg = 0;
40         RegWrite = 0;
41         MemRead  = 0;
42         MemWrite = 0;
43         ALUOp    = 6'b000000;
44
45         // Decode instruction based on opcode and function code
46         case (opcode)
47             LW: begin
48                 // Load Word
```

Figure 1: Code for Control Module Part 1

```
    localparam R_TYPE = 6'b000000;z

    // Function codes for R-type instructions
    localparam ADD  = 6'b100000;
    localparam SUB  = 6'b100010;
    localparam AND  = 6'b100100;
    localparam OR   = 6'b100101;
    localparam NOR  = 6'b100111;
    localparam XOR  = 6'b100110;

    // Control signal assignment
    always @(*) begin
        // Default values for control signals
        RegDst   = 0;
        ALUSrc   = 0;
        MemToReg = 0;
        RegWrite = 0;
        MemRead  = 0;
        MemWrite = 0;
        ALUOp    = 6'b000000;

        // Decode instruction based on opcode and function code
        case (opcode)
            LW: begin
                // Load Word
                RegDst   = 0;          // Destination register is rt
                ALUSrc   = 1;          // ALU takes immediate as input
                MemToReg = 1;          // Write memory data to register
                RegWrite = 1;          // Enable writing to the register fil
                MemRead  = 1;          // Enable memory read
                MemWrite = 0;          // Disable memory write
                ALUOp    = 6'b100000; // ALU performs ADD
            end
            SW: begin
                // Store Word
                RegDst   = 0;          // Don't care
                ALUSrc   = 1;          // ALU takes immediate as input
                MemToReg = 0;          // Don't care
                RegWrite = 0;          // Disable writing to the register fi
                MemRead  = 0;          // Disable memory read
                MemWrite = 1;          // Enable memory write
                ALUOp    = 6'b100000; // ALU performs ADD
            end
            ADDI: begin
                // Add Immediate
                RegDst   = 0;          // Destination register is rt
                ALUSrc   = 1;          // ALU takes immediate as input
                MemToReg = 0;          // Write ALU result to register
```

Figure 2: Code for Control Module Part 2

```
62                  RegWrite = 0;          // Disable writing to the register fi
63                  MemRead  = 0;          // Disable memory read
64                  MemWrite = 1;          // Enable memory write
65                  ALUOp    = 6'b100000; // ALU performs ADD
66              end
67              ADDI: begin
68                  // Add Immediate
69                  RegDst   = 0;          // Destination register is rt
70                  ALUSrc   = 1;          // ALU takes immediate as input
71                  MemToReg = 0;          // Write ALU result to register
72                  RegWrite = 1;          // Enable writing to the register fil
73                  MemRead  = 0;          // Disable memory read
74                  MemWrite = 0;          // Disable memory write
75                  ALUOp    = 6'b100000; // ALU performs ADD
76              end
77              R_TYPE: begin
78                  // R-Type instructions
79                  RegDst   = 1;          // Destination register is rd
80                  ALUSrc   = 0;          // ALU takes register as input
81                  MemToReg = 0;          // Write ALU result to register
82                  RegWrite = 1;          // Enable writing to the register fil
83                  MemRead  = 0;          // Disable memory read
84                  MemWrite = 0;          // Disable memory write
85                  case (funct)
86                      ADD: ALUOp = 6'b100000; // ALU performs ADD
87                      SUB: ALUOp = 6'b100010; // ALU performs SUB
88                      AND: ALUOp = 6'b100100; // ALU performs AND
89                      OR:  ALUOp = 6'b100101; // ALU performs OR
90                      NOR: ALUOp = 6'b100111; // ALU performs NOR
91                      XOR: ALUOp = 6'b100110; // ALU performs XOR
92                      default: ALUOp = 6'b000000; // Default no-op
93                  endcase
94              end
95              default: begin
96                  // Default case to avoid latches
97                  RegDst   = 0;
98                  ALUSrc   = 0;
99                  MemToReg = 0;
100                 RegWrite = 0;
101                 MemRead  = 0;
102                 MemWrite = 0;
103                 ALUOp    = 6'b000000;
104             end
105         endcase
106     end
107
108 endmodule
109
```

Figure 3: Code for Control Module Part 3

In this Verilog code, I code for the control module with the opcode such as LW, SW, ADDI and corresponding R type instructions such as ADD, SUB ,AND,OR ,NOR and XOR. By declaring some parameters which corresponds to the instruction that is being generated by the inst rom file or the memh file instructions. By using also the always function and making a first case to check if its opcode or Rtype and then each of them has also another case which breaks down the function of every MIPS Control mentioned above. So for example the instruction is 32'b000000_00000_00000_00000_00000_100000; this will read the [5:0] or the last bit until the 6th bit which is 100000 which we coded as ADD in the control module and the opcode which is in the position of [31:26] which is the 27th bit from the right to the left which in the example is

000000 in the control module I code this is an R type so to summarize on that instruction it is an Rtype case then after that case on to the ADD function so this is generated with the values of

```
RegDst   = 1;

ALUSrc   = 0;

MemToReg = 0;

RegWrite =1       ;

MemRead  = 0;

MemWrite = 0;

ALUOp    = 6'b100000;
```

This is done the same thing depends on the 32 bit input that is given on the memh file to process on inst rom.

For Task #2 and #3

```
1   `timescale 1ns / 1ps
2
3   module control_tb;
4       // Inputs
5       reg [31:0] instruction;
6       // Outputs
7       wire RegDst;
8       wire ALUSrc;
9       wire MemToReg;
10      wire RegWrite;
11      wire MemRead;
12      wire MemWrite;
13      wire [3:0] ALUOp;
14
15      // Instruction name string
16      reg [80*8:0] instr_name; // 80 characters to store instruction name
17
18      // Instantiate the control module
19      control uut (
20          .instruction(instruction),
21          .RegDst(RegDst),
22          .ALUSrc(ALUSrc),
23          .MemToReg(MemToReg),
24          .RegWrite(RegWrite),
25          .MemRead(MemRead),
26          .MemWrite(MemWrite),
27          .ALUOp(ALUOp)
28      );
29
30      // Monitor changes in inputs and outputs
31      initial begin
32          $monitor("Time: %0t | Instr: %h | %s | RegDst=%b | ALUSrc=%b | MemT
33                   $time, instruction, instr_name, RegDst, ALUSrc, MemToReg, I
34      end
35
36      // Test Instructions
37      initial begin
38          // Test R-type (ADD)
39          instruction = 32'b000000_00000_00000_00000_00000_100000; // ADD
40          instr_name = "R-type (ADD)";
41          #10;
```

Figure 4: Code for Control Testbench Module Part 1

```
25          .MemRead(MemRead),
26          .MemWrite(MemWrite),
27          .ALUOp(ALUOp)
28      );
29
30      // Monitor changes in inputs and outputs
31      initial begin
32          $monitor("Time: %0t | Instr: %b | RegDst=%b | ALUSrc=%b | MemToReg=%
33                   $time, instruction, RegDst, ALUSrc, MemToReg, RegWrite, Mem
34      end
35
36      // Test Instructions
37      initial begin
38          // Test R-type (ADD)
39          instruction = 32'b00000000000000000000000000100000; // ADD
40          instr_name = "R-type (ADD)";
41          #10;
42
43          // Test R-type (SUB)
44          instruction = 32'b00000000000000000000000000100010; // SUB
45          instr_name = "R-type (SUB)";
46          #10;
47
48          // Test R-type (AND)
49          instruction = 32'b00000000000000000000000000100100; // AND
50          instr_name = "R-type (AND)";
51          #10;
52
53          // Test R-type (OR)
54          instruction = 32'b00000000000000000000000000100101; // OR
55          instr_name = "R-type (OR)";
56          #10;
57
58          // Test I-type (LW)
59          instruction = 32'b10001100000000000000000000000000; // LW
60          instr_name = "I-type (LW)";
61          #10;
62
63          // Test I-type (SW)
64          instruction = 32'b10101100000000000000000000000000; // SW
65          instr_name = "I-type (SW)";
66          #10;
67
68          // Stop simulation
69          $stop;
70      end
71  endmodule
72
```

Figure 5: Code for Control Testbench Module Part 2

In this task by finishing up the control module, I also made the testbench which is to test the values of the control.v code before jumping in to other modules that could affect the overall processor . I check either R type and I type instructions on this one which is ADD,SUB , AND , OR , LW and SW. First is by declaring the control module on the testbench with the corresponding inputs and outputs wires I made output for all the results on the control module to test and test each one every 10 ns , so that by running the testbench this will show each of the instructions results

For Task 4:

In the processor module, I declare all the inputs and outputs which needed for my testbench in order to get the most results as possible to compare, in this way I can test and debug which one are not being able to do on what supposed to.

```
1  module program_counter (
2      input clock,
3      input reset,
4      input [31:0] pc_in,    // External PC input (optional)
5      output reg [31:0] pc_out // Program counter output
6  );
7
8      always @(posedge clock ) begin
9          if (reset) begin
10             pc_out <= 32'h003FFFFC  ; // Reset value
11         end else begin
12             pc_out <= pc_out + 4; // Increment PC by 4 on every clock cycle
13         end
14     end
15 endmodule
16
```

Figure 6: Code for Program Counter

```
1   `timescale 1ns / 1ps
2
3   module processortest (
4       input clock,
5       input reset,
6       input [7:0] serial_in,          // Serial input
7       input serial_valid_in,          // Serial valid flag
8       input serial_ready_in,          // Serial ready flag
9       input [1:0] rom_selector,
10      output [7:0] serial_out,        // Serial output
11      output serial_rden_out,         // Serial read enable
12      output serial_wren_out,         // Serial write enable
13      output [31:0] pc_out,           // Program Counter output
14      output [31:0] instruction_out,  // Fetched instruction
15      output [31:0] alu_a_out,        // ALU input A
16      output [31:0] alu_b_out,        // ALU input B
17      output [31:0] alu_out_internal  // ALU result
18  );
19
20      wire [31:0] pc_next;
21      wire [31:0] alu_out;
22      wire [31:0] read_data1;
23      wire [31:0] read_data2;
24      wire [31:0] mem_data_out;
25      wire [31:0] sign_ext_imm;------|
26      wire [31:0] alu_b_in;
27      wire [31:0] write_data;
28      wire [5:0] opcode;
29      wire [5:0] funct;
30      wire [5:0] alu_op;
31      wire RegDst, ALUSrc, MemToReg, RegWrite, MemRead, MemWrite;
32
33      // Serial wires
34      wire [7:0] serial_out_wire;
35      wire serial_wren_out_wire;
36      wire serial_rden_out_wire;
37
38      // Program Counter
39      program_counter PC (
40          .clock(clock),
41          .reset(reset),
42          .pc_in(pc_next),
43          .pc_out(pc_out)
44      );
45
46       //Instruction Memory
47      inst_rom InstructionMemory (
48          .clock(clock),
49          .reset(reset),
50          .addr_in(pc_out),
51          .data_out(instruction_out)
52      );
53
```

Figure 7: Code for Processor to use with .memh file Part 1

In figure 7, this shows the code module for processortest. I made 2 modules with different inst rom , the other module named processor , one for using .memh and one for using the hardcoded similar instructions that is on the memh file. Since by trying this module to the FPGA , it only shows 0 values compare to the memh files that is running on the testbenches. The program counter module is created shown in Figure 6 which is just initialize as 0 and if reset is is shows as 003FFFC as instructed in the laboratory, and this increments as 4 every clock .

```
// Fetch instruction with optional endian correction
always @(*) begin
    if (reset) begin
        data_out = 32'h00000000; // Reset output to zero
    end else begin
        // Correct endian swapping to match SPIM's expected order
        data_out = {
            rom[addr_in[ADDR_WIDTH+1:2]][7:0],
            rom[addr_in[ADDR_WIDTH+1:2]][15:8],
            rom[addr_in[ADDR_WIDTH+1:2]][23:16],
            rom[addr_in[ADDR_WIDTH+1:2]][31:24]
        };
    end
end
```

Figure 8: Code for making the instructions in endian format

After getting the specific program counter which will be starting at 004000 after first clock on 003fffc instruction (the reset) this serves as another input to put on the inst_rom, which is to fetch the instructions to be get from the memh file or the hardcoded files, same process the difference is the hardcoded is on the code itself directly. To Breakdown the process on this from the instruction on memh file or hardcoded file, we must ensure that this is in endian format by shifting the positions of the 2 bits original to opposite sides, so for example we have an program counter value of 004000 as the clock positive edges, the rom is parameterizes by ADDR_WIDTH which is how many instructions it can store and this is crucial number for the FPGA which how many instruction only can handle or store within the memory. so since the ADDR_WIDTH=5,

2^5=32 instructions - instructions in rom are stored as 32-bit words , the code rearranges as it uses the code on Figure 8 which is to swap them via endian format,So this results as calculating the word addresses

Calculate Word Address:

- addr_in[ADDR_WIDTH+1:2] -translates the byte address 004000 to a word address.
- For ADDR_WIDTH = 5, ADDR_WIDTH+1 = 6. So the slice is [6:2].
- Extracting addr_in[6:2]:
    - Binary of 004000 (hex) = 0000 0000 0100 0000 0000 0000 (binary).
    - Bits [6:2] = 00000 (decimal 0).

on bits [6:2] this is 0 as decimal so we access the rom[0] that is register either on the inst rom that uses memh file or just the hardcoded inst rom.But on this stage we also make the swapping now as it arranges the endian format by reassigning each of the indexes on the 32 bit instruction

example if the instruction is 32'h12345678

this will result as to {rom[0][7:0], rom[0][15:8], rom[0][23:16], rom[0][31:24]} = 32'h78563412.

Resulting h’78563412 as an output of the module to as endian format from the original instruction from inst_rom.

```
52      );
53
54      // Adder for PC + 4
55      adder PCAdder (
56          .a(pc_out),
57          .b(32'd4),
58          .sum(pc_next)
59      );
60
61      // Control Unit
62      control Control (
63          .instruction(instruction_out),
64          .RegDst(RegDst),
65          .ALUSrc(ALUSrc),
66          .MemToReg(MemToReg),
67          .RegWrite(RegWrite),
68          .MemRead(MemRead),
69          .MemWrite(MemWrite),
70          .ALUOp(alu_op)
71      );
72
73      // Register File
74      register_file RegFile (
75          .clock(clock),
76          .reset(reset),
77          .read_reg1(instruction_out[25:21]),  // rs
78          .read_reg2(instruction_out[20:16]),  // rt
79          .write_reg(RegDst ? instruction_out[15:11] : instruction_out[20:16]), // rd or rt
80          .write_data(write_data),
81          .write_enable(RegWrite),
82          .read_data1(read_data1),
83          .read_data2(read_data2)
84      );
85
86      // Sign Extender
87      sign_extender SignExt (
88          .in(instruction_out[15:0]),
89          .out(sign_ext_imm)
90      );
91
92      // ALU
93      alu ALU (
94          .Func_in(alu_op),
95          .A_in(read_data1),
96          .B_in(ALUSrc ? sign_ext_imm : read_data2),
97          .O_out(alu_out),
98          .Branch_out(),
99          .Jump_out()
100     );
101
102     // Data Memory
103     data_memory DataMem (
104         .clock(clock),
105         .reset(reset)
```

Figure 9: Code for Processor to use with .memh file Part 2

Next is have adder module increments the program counter to process the next instruction on next pos edge, and this is output by pc_next which will be another input as the next pos edge occurs. And next will be on the control unit which is output now on the module created on the past Task, to output as RegDst, ALUSrc , MemtoReg,Regwrite , MemRead , MemWrite. ALUOP, which are what we needed on the register files and other next modules to process shown code on figure 9.

```
`timescale 1ns / 1ps

module register_file (
    input clock,
    input reset,
    input [4:0] read_reg1,
    input [4:0] read_reg2,
    input [4:0] write_reg,
    input [31:0] write_data,
    input write_enable,
    output [31:0] read_data1,
    output [31:0] read_data2
);

    // 32 registers of 32-bit width
    reg [31:0] registers [31:0];
    integer i;

    // Reset and write logic
    always @(posedge clock or posedge reset) begin
        if (reset) begin
            // Reset all registers to 0
            for (i = 0; i < 32; i = i + 1) begin
                registers[i] <= 32'b0;
            end
        end else if (write_enable && write_reg != 0) begin
            // Write to the register only if write_enable is high and write_reg is non-zero
            registers[write_reg] <= write_data;
        end
    end

    // Continuous assignment for asynchronous reads
    assign read_data1 = registers[read_reg1];
    assign read_data2 = registers[read_reg2];

endmodule
```

Figure 10: Code for Register File

```
module sign_extender (
    input [15:0] in,
    output [31:0] out
);

    assign out = {{16{in[15]}}, in}; // Sign-extend the 16-bit input
endmodule
```

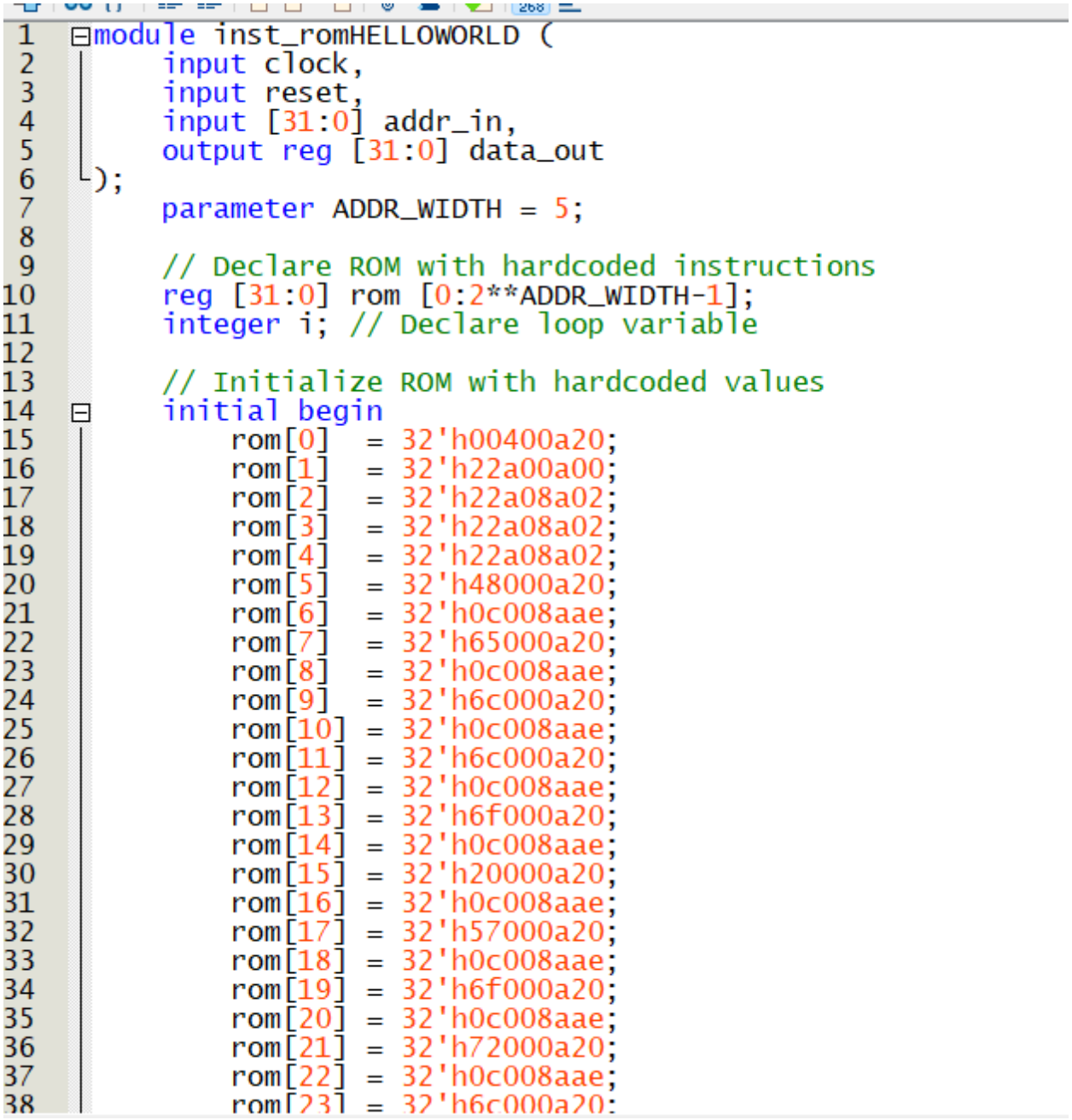
Figure 11: Code for Sign Extender

In figure above , this is the code for sign extender where we extend then 16 bit to 32 bit so in this case we extend the [15:0] of the instruction to turn into [31:0] by calling this module. This is the next step on the processor as it used on the alu module for ALU part.

```
78          .read_reg2(instruction_out[20:16]),  // rt
79          .write_reg(RegDst ? instruction_out[15:11] : instruction_out[20:16]), // rd or rt
80          .write_data(write_data),
81          .write_enable(RegWrite),
82          .read_data1(read_data1),
83          .read_data2(read_data2)
84      );
85
86      // Sign Extender
87      sign_extender SignExt (
88          .in(instruction_out[15:0]),
89          .out(sign_ext_imm)
90      );
91
92      // ALU
93      alu ALU (
94          .Func_in(alu_op),
95          .A_in(read_data1),
96          .B_in(ALUSrc ? sign_ext_imm : read_data2),
97          .O_out(alu_out),
98          .Branch_out(),
99          .Jump_out()
100     );
101
102     // Data Memory
103     data_memory DataMem (
104         .clock(clock),
105         .reset(reset),
106         .addr_in(alu_out),
107         .writedata_in(read_data2),
108         .re_in(MemRead),
109         .we_in(MemWrite),
110         .size_in(2'b11),
111         .readdata_out(mem_data_out),
112         .serial_in(serial_in),
113         .serial_ready_in(serial_ready_in),
114         .serial_valid_in(serial_valid_in),
115         .serial_out(serial_out_wire),
116         .serial_rden_out(serial_rden_out_wire),
117         .serial_wren_out(serial_wren_out_wire)
118     );
119
120     // Connect outputs
121     assign serial_out = serial_out_wire;
122     assign serial_rden_out = serial_rden_out_wire;
123     assign serial_wren_out = serial_wren_out_wire;
124     assign alu_a_out = read_data1;
125     assign alu_b_out = ALUSrc ? sign_ext_imm : read_data2;
126     assign alu_out_internal = alu_out;
127     assign write_data = MemToReg ? mem_data_out : alu_out;
128
129 endmodule
130
```

Figure 12: Code for Processor to use with .memh file Part 3

By using the ALU module on laboratory 6 this process the aluop which is output on the control module, the readdata1 which is the value of the first register that is being declared in the registerfile, then if its alusrc it will use the signextender output but if not it will use the readdata2 or the 2nd register file declared, then it will do the process on the alu module based on that instruction this will result to aluout which is the result if example addition its the sum of two registers and store into aluout.



```
module inst_romHELLOWORLD (
    input clock,
    input reset,
    input [31:0] addr_in,
    output reg [31:0] data_out
);
    parameter ADDR_WIDTH = 5;

    // Declare ROM with hardcoded instructions
    reg [31:0] rom [0:2**ADDR_WIDTH-1];
    integer i; // Declare loop variable

    // Initialize ROM with hardcoded values
    initial begin
        rom[0]  = 32'h00400a20;
        rom[1]  = 32'h22a00a00;
        rom[2]  = 32'h22a08a02;
        rom[3]  = 32'h22a08a02;
        rom[4]  = 32'h22a08a02;
        rom[5]  = 32'h48000a20;
        rom[6]  = 32'h0c008aae;
        rom[7]  = 32'h65000a20;
        rom[8]  = 32'h0c008aae;
        rom[9]  = 32'h6c000a20;
        rom[10] = 32'h0c008aae;
        rom[11] = 32'h6c000a20;
        rom[12] = 32'h0c008aae;
        rom[13] = 32'h6f000a20;
        rom[14] = 32'h0c008aae;
        rom[15] = 32'h20000a20;
        rom[16] = 32'h0c008aae;
        rom[17] = 32'h57000a20;
        rom[18] = 32'h0c008aae;
        rom[19] = 32'h6f000a20;
        rom[20] = 32'h0c008aae;
        rom[21] = 32'h72000a20;
        rom[22] = 32'h0c008aae;
        rom[23] = 32'h6c000a20;
```

Figure 13: Code for Hardcoded Inst rom for Hello World Instructions Part 1

```
30          rom[15] = 32'h20000a20;
31          rom[16] = 32'h0c008aae;
32          rom[17] = 32'h57000a20;
33          rom[18] = 32'h0c008aae;
34          rom[19] = 32'h6f000a20;
35          rom[20] = 32'h0c008aae;
36          rom[21] = 32'h72000a20;
37          rom[22] = 32'h0c008aae;
38          rom[23] = 32'h6c000a20;
39          rom[24] = 32'h0c008aae;
40          rom[25] = 32'h64000a20;
41          rom[26] = 32'h0c008aae;
42          rom[27] = 32'h00000000;
43
44
45          // Initialize remaining memory with zeros
46          for (i = 28; i < 2**ADDR_WIDTH; i = i + 1) begin
47              rom[i] = 32'h00000000;
48          end
49      end
50
51      // Fetch instruction with endian correction
52      always @(*) begin
53          if (reset) begin
54              data_out <= 32'h00000000; // Reset output to zero
55          end else begin
56              // Correct endian swapping to match SPIM's expected order
57              data_out <= {
58                  rom[addr_in[ADDR_WIDTH+1:2]][7:0],
59                  rom[addr_in[ADDR_WIDTH+1:2]][15:8],
60                  rom[addr_in[ADDR_WIDTH+1:2]][23:16],
61                  rom[addr_in[ADDR_WIDTH+1:2]][31:24]
62              };
63          end
64      end
65  endmodule
66
```

Figure 14: Code for Hardcoded Inst rom for Hello World Instructions Part 2

The only difference of instrom between the processor and processortest module is the inst rom , which shown on figure 13 above is the code for the hard coded instructions which is similar values on the memh file and can be changed also based on what will be the instructions to read,In the code there is addr_width=5 which determines the number of addressable entries in the ROM. The total number of addressable entries is 2^5, which equals 32 entries in this case. And this intialize every instruction each index and on the last part that is shown on Figure 13, this also make endian format for the final registration of the instruction roms

```
module data_memory(
    input clock,
    input reset,
    input [31:0] addr_in,
    input [31:0] writedata_in,
    input re_in, we_in,
    input [1:0] size_in,
    output reg [31:0] readdata_out,
    input [7:0] serial_in,
    input serial_ready_in, serial_valid_in,
    output [7:0] serial_out,
    output serial_rden_out, serial_wren_out
);

    parameter INIT_PROGRAM0 = "C:/intelFPGA_lite/18.1/lab7_tests/nbhelloworld.data_ram0.memh";
    parameter INIT_PROGRAM1 = "C:/intelFPGA_lite/18.1/lab7_tests/nbhelloworld.data_ram1.memh";
    parameter INIT_PROGRAM2 = "C:/intelFPGA_lite/18.1/lab7_tests/nbhelloworld.data_ram2.memh";
    parameter INIT_PROGRAM3 = "C:/intelFPGA_lite/18.1/lab7_tests/nbhelloworld.data_ram3.memh";

    wire [31:0] data_readdata_serial, data_readdata_data, data_readdata_stack;

    // Select the correct memory segment based on the address
    always @(*) begin
        case (addr_in[31:16])
            16'h1000: readdata_out <= data_readdata_data;
            16'h7fff: readdata_out <= data_readdata_stack;
            16'hffff: readdata_out <= data_readdata_serial;
            default:  readdata_out <= 32'b0;
        endcase
    end

    // Data Segment
    async_memory #(
        .MEM_ADDR(16'h1000),
        .DO_INIT(1),
        .INIT_PROGRAM0(INIT_PROGRAM0),
        .INIT_PROGRAM1(INIT_PROGRAM1),
        .INIT_PROGRAM2(INIT_PROGRAM2),
```

Figure 15: Data Memory Module Code Part 1

```
41          .clock(clock),
42          .reset(reset),
43          .addr_in(addr_in),
44          .size_in(size_in),
45          .data_out(data_readdata_data),
46          .re_in(re_in),
47          .data_in(writedata_in),
48          .we_in(we_in)
49      );
50
51      // Stack Segment
52      async_memory #(
53          .MEM_ADDR(16'h7fff)
54      ) stack_seg (
55          .clock(clock),
56          .reset(reset),
57          .addr_in(addr_in),
58          .size_in(size_in),
59          .data_out(data_readdata_stack),
60          .re_in(re_in),
61          .data_in(writedata_in),
62          .we_in(we_in)
63      );
64
65      // Serial MMIO
66      serial_buffer #(
67          .MEM_ADDR(16'hffff)
68      ) ser (
69          .clock(clock),
70          .reset(reset),
71          .addr_in(addr_in),
72          .data_out(data_readdata_serial),
73          .re_in(re_in),
74          .data_in(writedata_in),
75          .we_in(we_in),
76          .s_data_valid_in(serial_valid_in),
77          .s_data_ready_in(serial_ready_in),
78          .s_data_in(serial_in),
```

Figure 16: Data Memory Module Code Part 2

```
47          .data_in(writedata_in),
48          .we_in(we_in)
49      );
50
51      // Stack Segment
52      async_memory #(
53          .MEM_ADDR(16'h7fff)
54      ) stack_seg (
55          .clock(clock),
56          .reset(reset),
57          .addr_in(addr_in),
58          .size_in(size_in),
59          .data_out(data_readdata_stack),
60          .re_in(re_in),
61          .data_in(writedata_in),
62          .we_in(we_in)
63      );
64
65      // Serial MMIO
66      serial_buffer #(
67          .MEM_ADDR(16'hffff)
68      ) ser (
69          .clock(clock),
70          .reset(reset),
71          .addr_in(addr_in),
72          .data_out(data_readdata_serial),
73          .re_in(re_in),
74          .data_in(writedata_in),
75          .we_in(we_in),
76          .s_data_valid_in(serial_valid_in),
77          .s_data_ready_in(serial_ready_in),
78          .s_data_in(serial_in),
79          .s_rden_out(serial_rden_out),
80          .s_data_out(serial_out),
81          .s_wren_out(serial_wren_out)
82      );
83  endmodule
84
```

Figure 17: Data Memory Module Code Part 3

Figure 15-17 shows the datamemory code which this is to manages three memory segments: a data segment, a stack segment, and a serial MMIO segment, determined by the higher 16 bits of the address (addr_in). It supports reading and writing operations, as well as serial communication through a serial_buffer interface. The data segment (16'h1000) uses an async_memory module with optional initialization from external files (INIT_PROGRAMx), allowing predefined values during synthesis or simulation. The stack segment (16'h7fff) operates

similarly but is used for stack operations. The serial MMIO segment (16'hffff), managed by the serial_buffer module, handles serial input (serial_in) and output (serial_out), enabling communication with external devices. An always @(*) block selects the active memory segment based on the address input, assigning the corresponding read data to readdata_out. If the address doesn't match any segment, the output defaults to zero. This module is ideal for systems requiring segmented memory and integrated serial communication.

For Task 5:

```
1   `timescale 1ns / 1ps
2
3   module processor_tb;
4
5       // Input wires
6       reg clock;
7       reg reset;
8       reg [7:0] serial_in;
9       reg serial_valid_in;
10      reg serial_ready_in;
11
12      // Output wires
13      wire [7:0] serial_out;
14      wire serial_rden_out;
15      wire serial_wren_out;
16      wire [31:0] pc_out;
17      wire [31:0] instruction_out;
18      wire [31:0] alu_a_out;
19      wire [31:0] alu_b_out;
20      wire [31:0] alu_out_internal;
21
22      // Internal wires for observation
23      wire [31:0] pc_next;
24      wire [31:0] alu_out;
25      wire [31:0] read_data1;
26      wire [31:0] read_data2;
27      wire [31:0] mem_data_out;
28      wire [31:0] sign_ext_imm;
29      wire [31:0] alu_b_in;
30      wire [31:0] write_data;
31      wire [5:0] opcode;
32      wire [5:0] funct;
33      wire [5:0] alu_op;
34      wire RegDst, ALUSrc, MemToReg, RegWrite, MemRead, MemWrite;
35
36      // Instantiate the processor
37      processor uut (
38          .clock(clock),
```

Figure 18: Code for processor testbench for Task 5 Part 1

```
37      processor uut (
38          .clock(clock),
39          .reset(reset),
40          .serial_in(serial_in),
41          .serial_valid_in(serial_valid_in),
42          .serial_ready_in(serial_ready_in),
43          .serial_out(serial_out),
44          .serial_rden_out(serial_rden_out),
45          .serial_wren_out(serial_wren_out),
46          .pc_out(pc_out),
47          .instruction_out(instruction_out),
48          .alu_a_out(alu_a_out),
49          .alu_b_out(alu_b_out),
50          .alu_out_internal(alu_out_internal)
51      );
52
53      // Clock generation
54      always #5 clock = ~clock; // 10ns period (100 MHz)
55
56      initial begin
57          // Initialize inputs
58          clock = 0;
59          reset = 1;           // Assert reset
60          serial_in = 8'h00;
61          serial_valid_in = 0;
62          serial_ready_in = 0;
63
64          // Wait 20ns, then de-assert reset
65          #20 reset = 0;
66
67          // Test stimulus
68          #50 serial_in = 8'hAA;  // Example serial data input
69          serial_valid_in = 1;
70          serial_ready_in = 1;
71
72          #100 serial_valid_in = 0;
73          serial_ready_in = 0;
74
```

Figure 19: Code for processor testbench for Task 5 Part 2

```
68          #50 serial_in = 8'hAA;  // Example serial data input
69          serial_valid_in = 1;
70          serial_ready_in = 1;
71
72          #100 serial_valid_in = 0;
73          serial_ready_in = 0;
74
75          // Wait for 1us (1,000ns)
76          #1000 $finish;
77      end
78
79      // Observe internal signals
80      assign pc_next = uut.pc_next;
81      assign alu_out = uut.alu_out;
82      assign read_data1 = uut.read_data1;
83      assign read_data2 = uut.read_data2;
84      assign mem_data_out = uut.mem_data_out;
85      assign sign_ext_imm = uut.sign_ext_imm;
86      assign alu_b_in = uut.alu_b_in;
87      assign write_data = uut.write_data;
88      assign opcode = uut.opcode;
89      assign funct = uut.funct;
90      assign alu_op = uut.alu_op;
91      assign RegDst = uut.RegDst;
92      assign ALUSrc = uut.ALUSrc;
93      assign MemToReg = uut.MemToReg;
94      assign RegWrite = uut.RegWrite;
95      assign MemRead = uut.MemRead;
96      assign MemWrite = uut.MemWrite;
97
98      // Add $monitor for real-time observation
99      initial begin
100         $monitor("Time: %0t | PC: %h | Instruction: %h | ALU A: %h | ALU B:
101                  $time, pc_out, instruction_out, alu_a_out, alu_b_out, alu_
102     end
103
104 endmodule
105
```

Figure 20: Code for processor testbench for Task 5 Part 3

In figure 18-20 shows the processor testbench to show all the wires with waveforms and print also for monitor values, this I get all the possible values for a processor including the pc_next ,alu_out,read_data1,read_data2, mem_dataout, sign_ext_imm, alu_b_in ,write_data ,opcode, funct,alu_op,RegDst, ALUSrc,MemToReg,RegWrite,MemRead, and MemWrite. I added also monitor values code to show this on the modelsim.

For FPGA Part:

```
module Outputshow(
    input [2:0] controlinput,       // 3-bit control input
    input [31:0] programcounter,
    input [31:0] instr,
    input [31:0] ALU_A,
    input [31:0] ALU_B,
    input [31:0] ALU_OUT,
    input [31:0] Serial_OUT,
    output reg [31:0] Finalshow     // 32-bit output for display
);

always @(*) begin
    case (controlinput)
        3'b000: Finalshow = 32'b0;          // Default value (if needed)
        3'b001: Finalshow = programcounter; // Output program counter
        3'b010: Finalshow = instr;          // Output instruction
        3'b011: Finalshow = ALU_A;          // Output ALU_A
        3'b100: Finalshow = ALU_B;          // Output ALU_B
        3'b101: Finalshow = ALU_OUT;        // Output ALU result
        3'b110: Finalshow = Serial_OUT;     // Output Serial data
        default: Finalshow = 32'b111111;    // Default case for undefined control inputs
    endcase
end

endmodule
```

Figure 21: Code for Outputshow module

In figure 21, it shows the output show module to be able to control which output to show on the FPGA, and I assign them from 1-5 to be switch from the SW, these values are also 8 bit hex so I made an finalshow variable to to able to store them as the module calls.

```
1   `timescale 1ns / 1ps
2   module wrapper(
3       input [9:0] SW,
4       input [3:0] KEY,
5
6       output [6:0] hex_display0, // 7-segment display for digit 0 (LSB)
7       output [6:0] hex_display1, // 7-segment display for digit 1
8       output [6:0] hex_display2, // 7-segment display for digit 2
9       output [6:0] hex_display3, // 7-segment display for digit 3
10      output [6:0] hex_display4, // 7-segment display for digit 4
11      output [6:0] hex_display5,
12      output reg[3:0] led1,
13      output reg[3:0] led2,
14      output [32:0] modedf);
15
16      reg clock;
17      reg reset;
18
19     // Assuming KEY is an input, use always block to assign
20
21
22      // Outputs to monitor from the processor
23      wire [31:0] pc_out;            // Program Counter
24      wire [31:0] instruction_out;   // Fetched instruction
25      wire [31:0] alu_a_out;         // ALU input A
26      wire [31:0] alu_b_out;         // ALU input B
27      wire [31:0] alu_out_internal; // ALU result
28      wire [7:0] serial_out;         // Serial output data
29      wire serial_wren_out;          // Serial write enable
30
31      // Instantiate processor
32      processor uut (
33          .clock(KEY[0]),
34          .reset(KEY[1]),
35          .serial_in(8'b0),              // No serial input for this test
36          .serial_valid_in(1'b0),        // No valid input for this test
37          .serial_ready_in(1'b1),        // Serial is always ready to accept wr
38          .serial_out(serial_out),       // Serial output
```

Figure 22: Code for Wrapper Module Part 1

```
55
56
57      hexTo7Seg hex_display_inst0 (
58          .x(outputon[3:0]),      // Least significant nibble
59          .z(hex_display0)        // Output to 7-segment display
60      );
61
62      hexTo7Seg hex_display_inst1 (
63          .x(outputon[7:4]),      // Next nibble
64          .z(hex_display1)
65      );
66
67      hexTo7Seg hex_display_inst2 (
68          .x(outputon[11:8]),     // Next nibble
69          .z(hex_display2)
70      );
71
72      hexTo7Seg hex_display_inst3 (
73          .x(outputon[15:12]),    // Next nibble
74          .z(hex_display3)
75      );
76
77      hexTo7Seg hex_display_inst4 (
78          .x(outputon[19:16]),    // Next nibble
79          .z(hex_display4)
80      );
81
82      hexTo7Seg hex_display_inst5 (
83          .x(outputon[23:20]),    // Most significant nibble
84          .z(hex_display5)
85      );
86      always @* begin
87          led1 = outputon[27:24];
88          led2 = outputon[31:28];
89      end
90
91  endmodule
92
```

Figure 23: Code for Wrapper Module Part 2

```
40          .serial_wren_out(serial_wren_out), // Serial write enable
41          .pc_out(pc_out),                   // Program Counter output
42          .instruction_out(instruction_out), // Fetched instruction
43          .alu_a_out(alu_a_out),             // ALU input A
44          .alu_b_out(alu_b_out),             // ALU input B
45          .alu_out_internal(alu_out_internal) // ALU result
46      );
47
48
49
50
51      wire [32:0] outputon;
52
53
54      Outputshow outshoww (SW[2:0],pc_out,instruction_out,alu_a_out,alu_b_out,
55
56
57      hexTo7Seg hex_display_inst0 (
58          .x(outputon[3:0]),     // Least significant nibble
59          .z(hex_display0)       // Output to 7-segment display
60      );
61
62      hexTo7Seg hex_display_inst1 (
63          .x(outputon[7:4]),     // Next nibble
64          .z(hex_display1)
65      );
66
67      hexTo7Seg hex_display_inst2 (
68          .x(outputon[11:8]),    // Next nibble
69          .z(hex_display2)
70      );
71
72      hexTo7Seg hex_display_inst3 (
73          .x(outputon[15:12]),   // Next nibble
74          .z(hex_display3)
75      );
76
77      hexTo7Seg hex_display_inst4 (
```

Figure 24: Code for Wrapper Module Part 3

In the figures 22-24 , this shows the wrapper module and this module set as the top hierarchy that the pin planner do to show outputs on the FPGA, I assign on the six 7 segments , and also the remaining 2 bits for leds since the output is an 8 bit hex, since the output of the outputshow is a 32 bit this assign each of the 7 segments 4 bits and also the 8 leds for the remaining, to be able to show it properly the outputs

KEY[3]	Input	PIN_Y16	3B	B3B_N0	2.5 V ...fault)		12mA ...ault)						
KEY[2]	Input	PIN_W15	3B	B3B_N0	2.5 V ...fault)		12mA ...ault)						
KEY[1]	Input	PIN_AA15	3B	B3B_N0	2.5 V ...fault)		12mA ...ault)						
KEY[0]	Input	PIN_AA14	3B	B3B_N0	2.5 V ...fault)		12mA ...ault)						
SW[9]	Input	PIN_AE12	3A	B3A_N0	2.5 V ...fault)		12mA ...ault)						
SW[8]	Input	PIN_AD10	3A	B3A_N0	2.5 V ...fault)		12mA ...ault)						
SW[7]	Input	PIN_AC9	3A	B3A_N0	2.5 V ...fault)		12mA ...ault)						
SW[6]	Input	PIN_AE11	3A	B3A_N0	2.5 V ...fault)		12mA ...ault)						
SW[5]	Input	PIN_AD12	3A	B3A_N0	2.5 V ...fault)		12mA ...ault)						
SW[4]	Input	PIN_AD11	3A	B3A_N0	2.5 V ...fault)		12mA ...ault)						
SW[3]	Input	PIN_AF10	3A	B3A_N0	2.5 V ...fault)		12mA ...ault)						
SW[2]	Input	PIN_AF9	3A	B3A_N0	2.5 V ...fault)		12mA ...ault)						
SW[1]	Input	PIN_AC12	3A	B3A_N0	2.5 V ...fault)		12mA ...ault)						
SW[0]	Input	PIN_AB12	3A	B3A_N0	2.5 V ...fault)		12mA ...ault)						
hex_display0[6]	Output	PIN_AH28	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display0[5]	Output	PIN_AG28	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display0[4]	Output	PIN_AF28	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display0[3]	Output	PIN_AG27	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display0[2]	Output	PIN_AE28	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display0[1]	Output	PIN_AE27	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display0[0]	Output	PIN_AE26	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display1[6]	Output	PIN_AD27	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display1[5]	Output	PIN_AF30	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display1[4]	Output	PIN_AF29	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display1[3]	Output	PIN_AG30	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display1[2]	Output	PIN_AH30	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					

Figure 25: Pin Planner for FPGA Part 1

Node Name	Direction	Location	I/O Bank	VREF Group	I/O Standard	Reserved	...rrent Streng	Slew Rate	...ifferential Pai	Analog Settir	...(B/VCCT_GXE	...er I/O Pin Ter	...icated Refclk	...mmon Mode	...tter Slew
hex_display1[3]	Output	PIN_AG30	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display1[2]	Output	PIN_AH30	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display1[1]	Output	PIN_AH29	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display1[0]	Output	PIN_AJ29	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display2[6]	Output	PIN_AC30	5B	B5B_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display2[5]	Output	PIN_AC29	5B	B5B_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display2[4]	Output	PIN_AD30	5B	B5B_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display2[3]	Output	PIN_AC28	5B	B5B_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display2[2]	Output	PIN_AD29	5B	B5B_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display2[1]	Output	PIN_AE29	5B	B5B_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display2[0]	Output	PIN_AB23	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display3[6]	Output	PIN_AB22	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display3[5]	Output	PIN_AB25	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display3[4]	Output	PIN_AB28	5B	B5B_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display3[3]	Output	PIN_AC25	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display3[2]	Output	PIN_AD25	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display3[1]	Output	PIN_AC27	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display3[0]	Output	PIN_AD26	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display4[6]	Output	PIN_W25	5B	B5B_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display4[5]	Output	PIN_V23	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display4[4]	Output	PIN_W24	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display4[3]	Output	PIN_W22	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display4[2]	Output	PIN_Y24	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display4[1]	Output	PIN_Y23	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display4[0]	Output	PIN_AA24	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							
hex_display5[6]	Output	PIN_AA25	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)							

Figure 22: Pin Planner for FPGA Part 2

hex_display5[6]	Output	PIN_AA25	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display5[5]	Output	PIN_AA26	5B	B5B_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display5[4]	Output	PIN_AB26	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display5[3]	Output	PIN_AB27	5B	B5B_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display5[2]	Output	PIN_Y27	5B	B5B_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display5[1]	Output	PIN_AA28	5B	B5B_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
hex_display5[0]	Output	PIN_V25	5B	B5B_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
led1[3]	Output	PIN_V18	4A	B4A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
led1[2]	Output	PIN_V17	4A	B4A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
led1[1]	Output	PIN_W16	4A	B4A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
led1[0]	Output	PIN_V16	4A	B4A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
led2[3]	Output	PIN_W20	5A	B5A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
led2[2]	Output	PIN_Y19	4A	B4A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
led2[1]	Output	PIN_W19	4A	B4A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					
led2[0]	Output	PIN_W17	4A	B4A_N0	2.5 V ...fault)		12mA ...ault)	1 (default)					

Figure 26: Pin Planner for FPGA Part 3

Figure 24-26 shows the pin assignment for each of the 7 segments and led displays, each are declared to be able to gather and show each of the bits of an 8 bit output. This is reference from the de1-soc documentation that is given by the professor.

```
module hexTo7Seg(
    input [3:0] x,  // 4-bit input to represent hexadecimal digits
    output reg [6:0] z  // 7-bit output for 7-segment display
);

always @(*) begin
    case (x)
        4'b0000: z = 7'b1000000; // 0
        4'b0001: z = 7'b1111001; // 1
        4'b0010: z = 7'b0100100; // 2
        4'b0011: z = 7'b0110000; // 3
        4'b0100: z = 7'b0011001; // 4
        4'b0101: z = 7'b0010010; // 5
        4'b0110: z = 7'b0000010; // 6
        4'b0111: z = 7'b1111000; // 7
        4'b1000: z = 7'b0000000; // 8
        4'b1001: z = 7'b0010000; // 9
        4'b1010: z = 7'b0001000; // A
        4'b1011: z = 7'b0000011; // b
        4'b1100: z = 7'b1000110; // C
        4'b1101: z = 7'b0100001; // d
        4'b1110: z = 7'b0000110; // E
        4'b1111: z = 7'b0001110; // F
        default: z = 7'b1111111; // Blank display for invalid input
    endcase
end

endmodule
```

Figure 27: Code for Mapping on to 7 segment display -Hexto7Seg

Figure 27 shows the mapping of values to whatever the values to show on 7 segment. this shows the value of 0-F in the 7 segment display

Results and Analysis:

(Depends on the tasks given)

Task 1-3:

```
#          File in use by: user  Hostname: LAPTOP-TR3A51I9  ProcessID: 27436
#          Attempting to use alternate WLF file "./wlft84z26t".
# ** Warning: (vsim-WLF-5001) Could not open WLF file: vsim.wlf
#          Using alternate file: ./wlft84z26t
VSIM 6> run -all
# Time: 0 | Instr: 00000000000000000000000000100000 | RegDst=1 | ALUSrc=0 | MemToReg=0 | RegWrite=1 | MemRead=0 | MemWrite=0 | ALUOp=0000
# Time: 10000 | Instr: 00000000000000000000000000100010 | RegDst=1 | ALUSrc=0 | MemToReg=0 | RegWrite=1 | MemRead=0 | MemWrite=0 | ALUOp=0010
# Time: 20000 | Instr: 00000000000000000000000000100100 | RegDst=1 | ALUSrc=0 | MemToReg=0 | RegWrite=1 | MemRead=0 | MemWrite=0 | ALUOp=0100
# Time: 30000 | Instr: 00000000000000000000000000100101 | RegDst=1 | ALUSrc=0 | MemToReg=0 | RegWrite=1 | MemRead=0 | MemWrite=0 | ALUOp=0101
# Time: 40000 | Instr: 10001100000000000000000000000000 | RegDst=0 | ALUSrc=1 | MemToReg=1 | RegWrite=1 | MemRead=1 | MemWrite=0 | ALUOp=0000
# Time: 50000 | Instr: 10101100000000000000000000000000 | RegDst=0 | ALUSrc=1 | MemToReg=0 | RegWrite=0 | MemRead=0 | MemWrite=1 | ALUOp=0000
# ** Note: $stop    : C:/intelFPGA_lite/18.1/control_tb.v(69)
```

Figure 28: ModelSim Output Values for Task 1-3

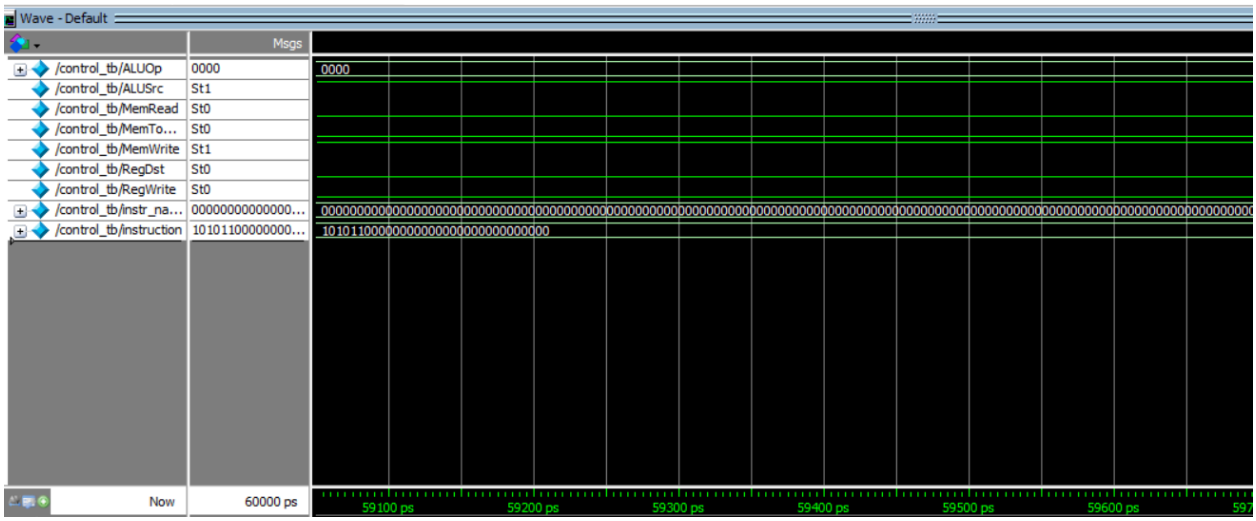


Figure 29: ModelSim Waveform for Task 1-3

Task 4:

```
See the file README for a full copyright notice.
C:\Users\user\Desktop\College\New folder\1st Sem 2024-2025\COE181.1-Computer Organization and Architecture Laboratory\Lab 7\lab7_tests\nbhelloworld.
[0x00400000]    0x200a4000  addi $10, $0, 16384             ; 18: addi  $10,$0,0x4000
[0x00400004]    0x000aa022  sub $20, $0, $10                ; 19: sub   $20,$0,$10
[0x00400008]    0x028aa022  sub $20, $20, $10               ; 20: sub   $20,$20,$10             # $20 should now contain 0xfffc000
[0x0040000c]    0x028aa022  sub $20, $20, $10               ; 21: sub   $20,$20,$10             # $20 should now contain 0xffff8000
[0x00400010]    0x028aa022  sub $20, $20, $10               ; 22: sub   $20,$20,$10             # $20 should now contain 0xffff0000
[0x00400014]    0x200a0048  addi $10, $0, 72                ; 24: addi  $10,$0,0x48             # $10 is now 'H'
[0x00400018]    0xae8a000c  sw $10, 12($20)                 ; 25: sw    $10,12($20)             # write word
[0x0040001c]    0x200a0065  addi $10, $0, 101               ; 27: addi  $10,$0,0x65             # 'e'
[0x00400020]    0xae8a000c  sw $10, 12($20)                 ; 28: sw    $10,12($20)             # write word
[0x00400024]    0x200a006c  addi $10, $0, 108               ; 30: addi  $10,$0,0x6C             # 'l'
```

Figure 30: SPIM Output for Hello World.memh Part 1

```
[0x00400014]    0x200a0048  addi $10, $0, 72                ; 24: addi  $10,$0,0x48             # $10 is now 'H'
[0x00400018]    0xae8a000c  sw $10, 12($20)                 ; 25: sw    $10,12($20)             # write word
[0x0040001c]    0x200a0065  addi $10, $0, 101               ; 27: addi  $10,$0,0x65             # 'e'
[0x00400020]    0xae8a000c  sw $10, 12($20)                 ; 28: sw    $10,12($20)             # write word
[0x00400024]    0x200a006c  addi $10, $0, 108               ; 30: addi  $10,$0,0x6C             # 'l'
[0x00400028]    0xae8a000c  sw $10, 12($20)                 ; 31: sw    $10,12($20)             # write word
[0x0040002c]    0x200a006c  addi $10, $0, 108               ; 33: addi  $10,$0,0x6C             # 'l'
[0x00400030]    0xae8a000c  sw $10, 12($20)                 ; 34: sw    $10,12($20)             # write word
[0x00400034]    0x200a006f  addi $10, $0, 111               ; 36: addi  $10,$0,0x6F             # 'o'
[0x00400038]    0xae8a000c  sw $10, 12($20)                 ; 37: sw    $10,12($20)             # write word
[0x0040003c]    0x200a0020  addi $10, $0, 32                ; 39: addi  $10,$0,0x20             # ' '
```

Figure 31: SPIM Output for Hello World.memh Part 2

```
[0x0040003c]    0x200a0020  addi $10, $0, 32              ; 39: addi  $10,$0,0x20          # ' '
[0x00400040]    0xae8a000c  sw $10, 12($20)              ; 40: sw    $10,12($20)          # write word
[0x00400044]    0x200a0057  addi $10, $0, 87              ; 42: addi  $10,$0,0x57          # 'W'
[0x00400048]    0xae8a000c  sw $10, 12($20)              ; 43: sw    $10,12($20)          # write word
[0x0040004c]    0x200a006f  addi $10, $0, 111             ; 45: addi  $10,$0,0x6F          # 'o'
[0x00400050]    0xae8a000c  sw $10, 12($20)              ; 46: sw    $10,12($20)          # write word
[0x00400054]    0x200a0072  addi $10, $0, 114             ; 48: addi  $10,$0,0x72          # 'r'
[0x00400058]    0xae8a000c  sw $10, 12($20)              ; 49: sw    $10,12($20)          # write word
[0x0040005c]    0x200a006c  addi $10, $0, 108             ; 51: addi  $10,$0,0x6C          # 'l'
[0x00400060]    0xae8a000c  sw $10, 12($20)              ; 52: sw    $10,12($20)          # write word
[0x00400064]    0x200a0064  addi $10, $0, 100             ; 54: addi  $10,$0,0x64          # 'd'
```

Figure 32: SPIM Output for Hello World.memh Part 3

Task 5:

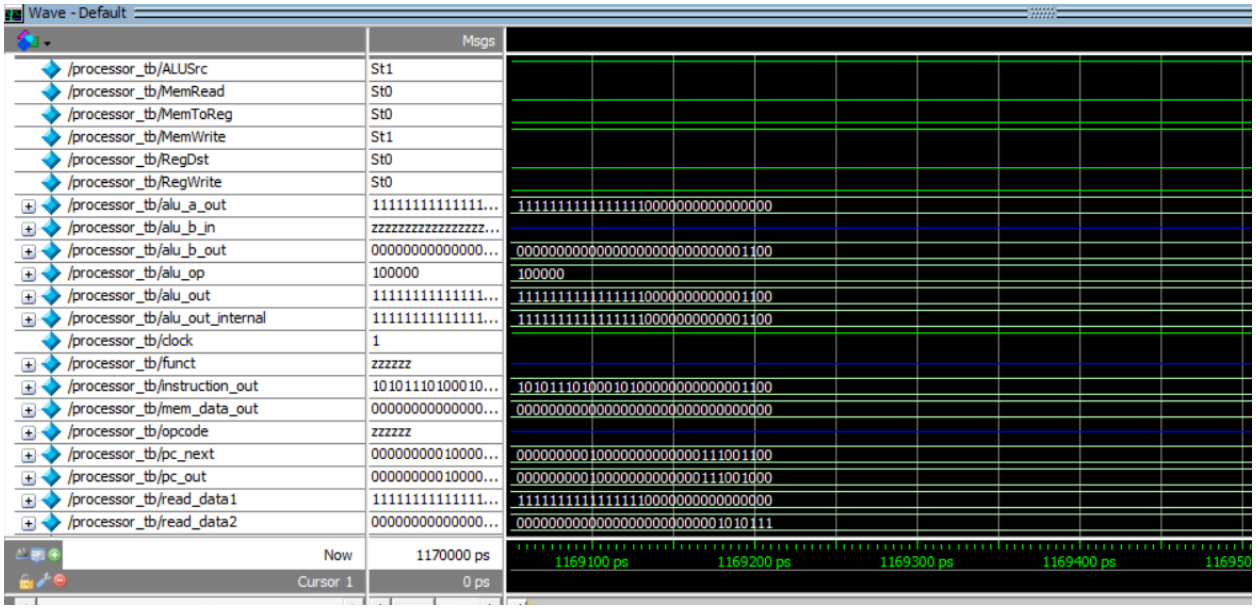


Figure 33: Waveform for processor Part 1

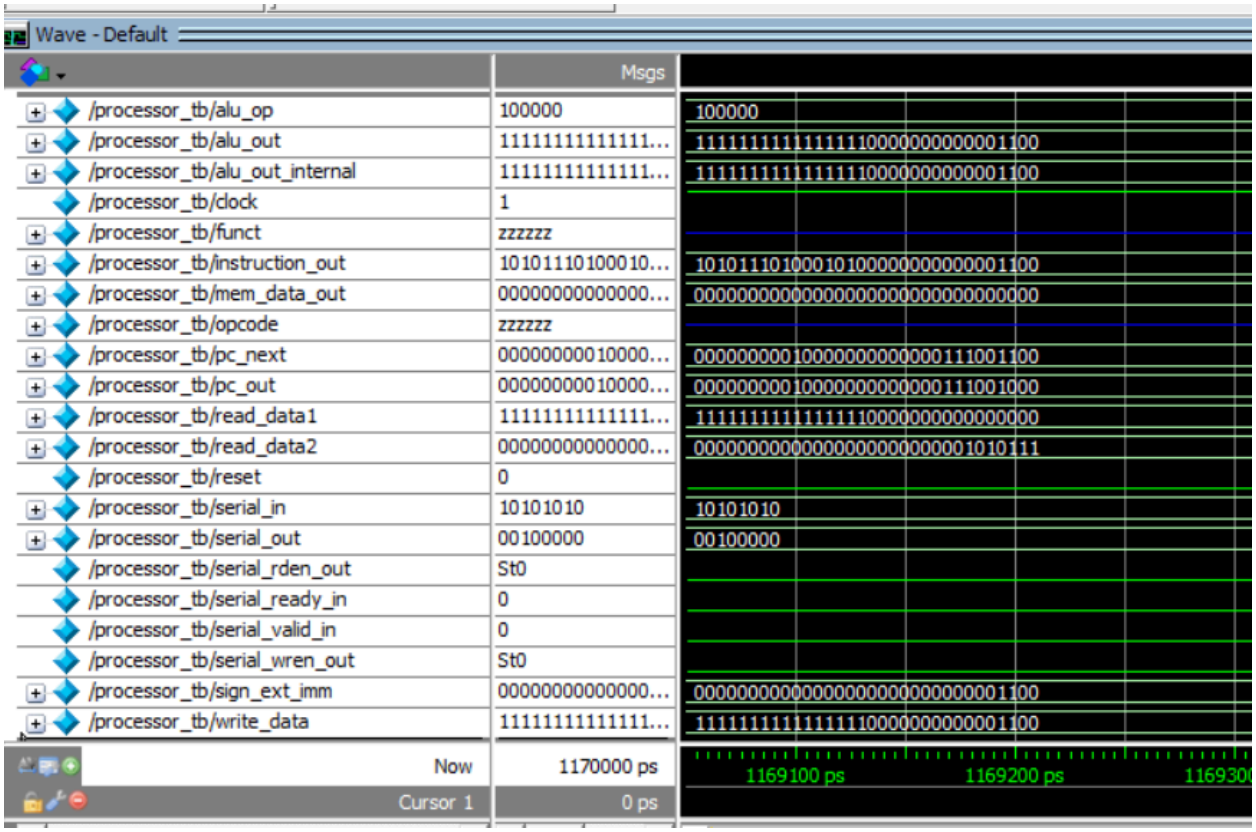


Figure 34: Waveform for processor Part 2

```
VSIM 8> run -all
# Time: 0 | PC: xxxxxxxx | Instr: 00000000 | ALU A: 00000000 | ALU B: 00000000 | ALU Out: 00000000 | Serial Out:  | Serial WREN: 0
# Time: 5000 | PC: 003ffffc | Instr: 00000000 | ALU A: 00000000 | ALU B: 00000000 | ALU Out: 00000000 | Serial Out:  | Serial WREN: 0
# Time: 20000 | PC: 003ffffc | Instr: xxxxxxxx | ALU A: xxxxxxxx | ALU B: xxxxxxxx | ALU Out: 00000000 | Serial Out:  | Serial WREN: 0
# Time: 25000 | PC: 00400000 | Instr: 200a4000 | ALU A: 00000000 | ALU B: 00004000 | ALU Out: 00004000 | Serial Out:  | Serial WREN: 0
# Time: 35000 | PC: 00400004 | Instr: 000aa022 | ALU A: 00000000 | ALU B: 00004000 | ALU Out: ffffc000 | Serial Out:  | Serial WREN: 0
# Time: 45000 | PC: 00400008 | Instr: 028aa022 | ALU A: ffffc000 | ALU B: 00004000 | ALU Out: ffff8000 | Serial Out:  | Serial WREN: 0
# Time: 55000 | PC: 0040000c | Instr: 028aa022 | ALU A: ffff8000 | ALU B: 00004000 | ALU Out: ffff4000 | Serial Out:  | Serial WREN: 0
# Time: 65000 | PC: 00400010 | Instr: 028aa022 | ALU A: ffff4000 | ALU B: 00004000 | ALU Out: ffff0000 | Serial Out:  | Serial WREN: 0
# Time: 75000 | PC: 00400014 | Instr: 200a0048 | ALU A: 00000000 | ALU B: 00000048 | ALU Out: 00000048 | Serial Out:  | Serial WREN: 0
# Time: 85000 | PC: 00400018 | Instr: ae8a000c | ALU A: ffff0000 | ALU B: 0000000c | ALU Out: ffff000c | Serial Out:  | Serial WREN: 0
# Time: 95000 | PC: 0040001c | Instr: 200a0065 | ALU A: 00000000 | ALU B: 00000065 | ALU Out: 00000065 | Serial Out: H | Serial WREN: 1
# Time: 105000 | Serial Write: H (ASCII: 48)
# Time: 105000 | PC: 00400020 | Instr: ae8a000c | ALU A: ffff0000 | ALU B: 0000000c | ALU Out: ffff000c | Serial Out: H | Serial WREN: 0
# Time: 115000 | PC: 00400024 | Instr: 200a006c | ALU A: 00000000 | ALU B: 0000006c | ALU Out: 0000006c | Serial Out: e | Serial WREN: 1
# Time: 125000 | Serial Write: e (ASCII: 65)
# Time: 125000 | PC: 00400028 | Instr: ae8a000c | ALU A: ffff0000 | ALU B: 0000000c | ALU Out: ffff000c | Serial Out: e | Serial WREN: 0
# Time: 135000 | PC: 0040002c | Instr: 200a006c | ALU A: 00000000 | ALU B: 0000006c | ALU Out: 0000006c | Serial Out: l | Serial WREN: 1
# Time: 145000 | Serial Write: l (ASCII: 6c)
# Time: 145000 | PC: 00400030 | Instr: ae8a000c | ALU A: ffff0000 | ALU B: 0000000c | ALU Out: ffff000c | Serial Out: l | Serial WREN: 0
# Time: 155000 | PC: 00400034 | Instr: 200a006f | ALU A: 00000000 | ALU B: 0000006f | ALU Out: 0000006f | Serial Out: l | Serial WREN: 1
# Time: 165000 | Serial Write: l (ASCII: 6c)
# Time: 165000 | PC: 00400038 | Instr: ae8a000c | ALU A: ffff0000 | ALU B: 0000000c | ALU Out: ffff000c | Serial Out: l | Serial WREN: 0
# Time: 175000 | PC: 0040003c | Instr: 200a0020 | ALU A: 00000000 | ALU B: 00000020 | ALU Out: 00000020 | Serial Out: o | Serial WREN: 1
# Time: 185000 | Serial Write: o (ASCII: 6f)
# Time: 185000 | PC: 00400040 | Instr: ae8a000c | ALU A: ffff0000 | ALU B: 0000000c | ALU Out: ffff000c | Serial Out: o | Serial WREN: 0
# Time: 195000 | PC: 00400044 | Instr: 200a0057 | ALU A: 00000000 | ALU B: 00000057 | ALU Out: 00000057 | Serial Out:   | Serial WREN: 1
# Time: 205000 | Serial Write:   (ASCII: 20)
# Time: 205000 | PC: 00400048 | Instr: ae8a000c | ALU A: ffff0000 | ALU B: 0000000c | ALU Out: ffff000c | Serial Out:   | Serial WREN: 0
# Time: 215000 | PC: 0040004c | Instr: 200a006f | ALU A: 00000000 | ALU B: 0000006f | ALU Out: 0000006f | Serial Out: W | Serial WREN: 1
# Time: 225000 | Serial Write: W (ASCII: 57)
# Time: 225000 | PC: 00400050 | Instr: ae8a000c | ALU A: ffff0000 | ALU B: 0000000c | ALU Out: ffff000c | Serial Out: W | Serial WREN: 0
# Time: 235000 | PC: 00400054 | Instr: 200a0072 | ALU A: 00000000 | ALU B: 00000072 | ALU Out: 00000072 | Serial Out: o | Serial WREN: 1
# Time: 245000 | Serial Write: o (ASCII: 6f)
# Time: 245000 | PC: 00400058 | Instr: ae8a000c | ALU A: ffff0000 | ALU B: 0000000c | ALU Out: ffff000c | Serial Out: o | Serial WREN: 0
# Time: 255000 | PC: 0040005c | Instr: 200a006c | ALU A: 00000000 | ALU B: 0000006c | ALU Out: 0000006c | Serial Out: r | Serial WREN: 1
Wave | Dataflow | testbench.v | Objects | Processes | Transcript | Library | Memory List | sim
```

Figure 35: Output Values for Hello World instruction in Model Sim

Figure 33-35 shows the waveform and output values for the processor , this shows all the wires and the serial output which is hello world instructions that is from the memh files

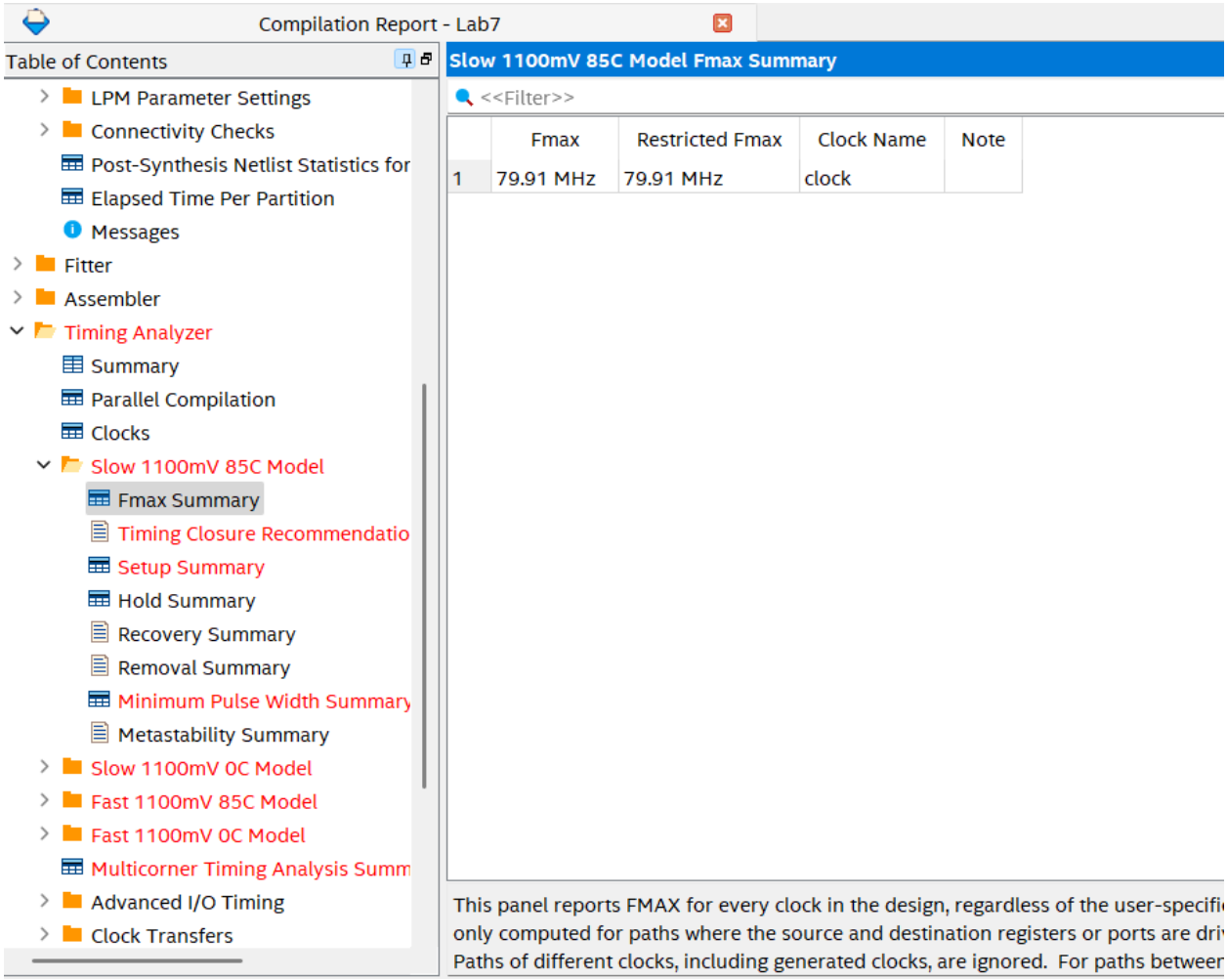


Figure 36: Total Fmax on Processor Module

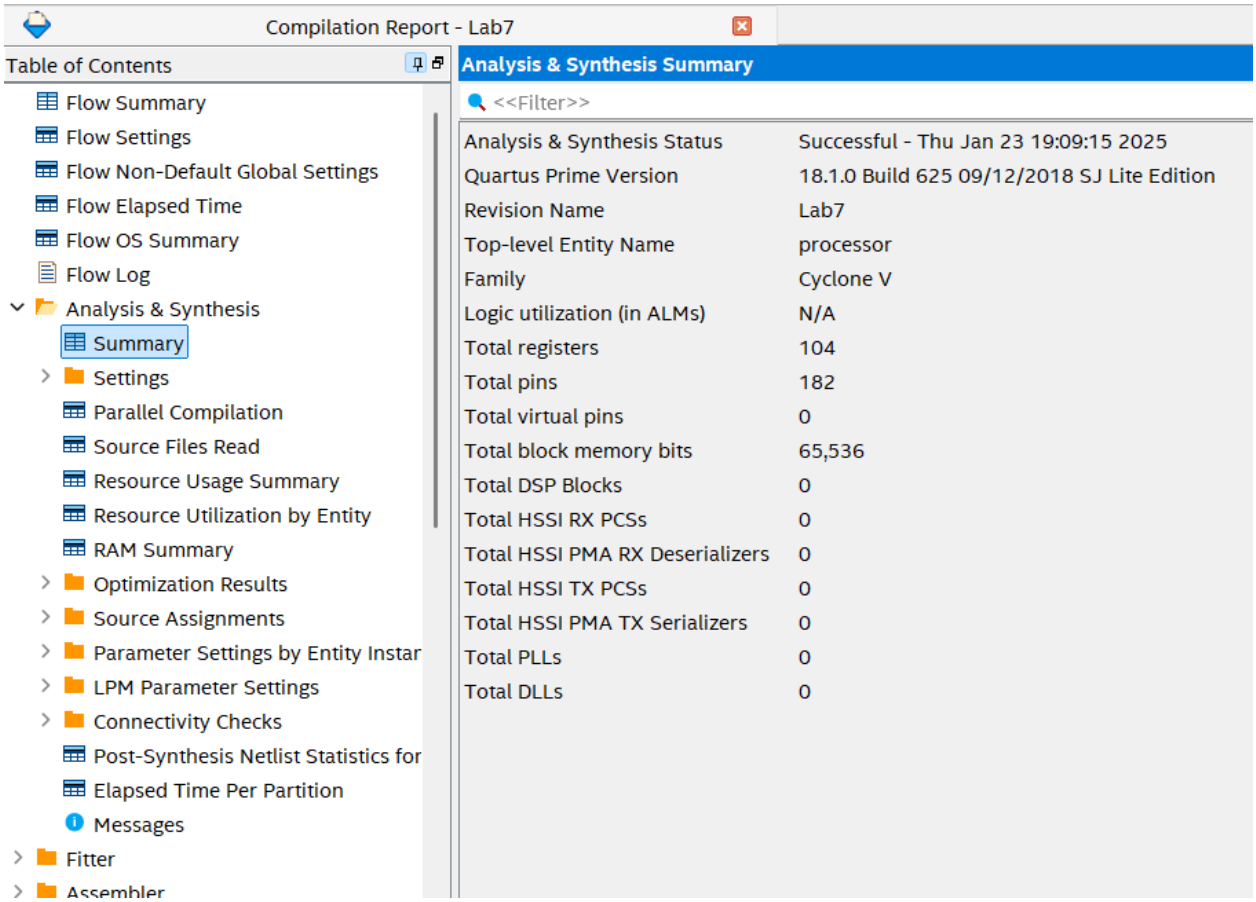


Figure 37: Total registers of the Processor Module

FOR FPGA:

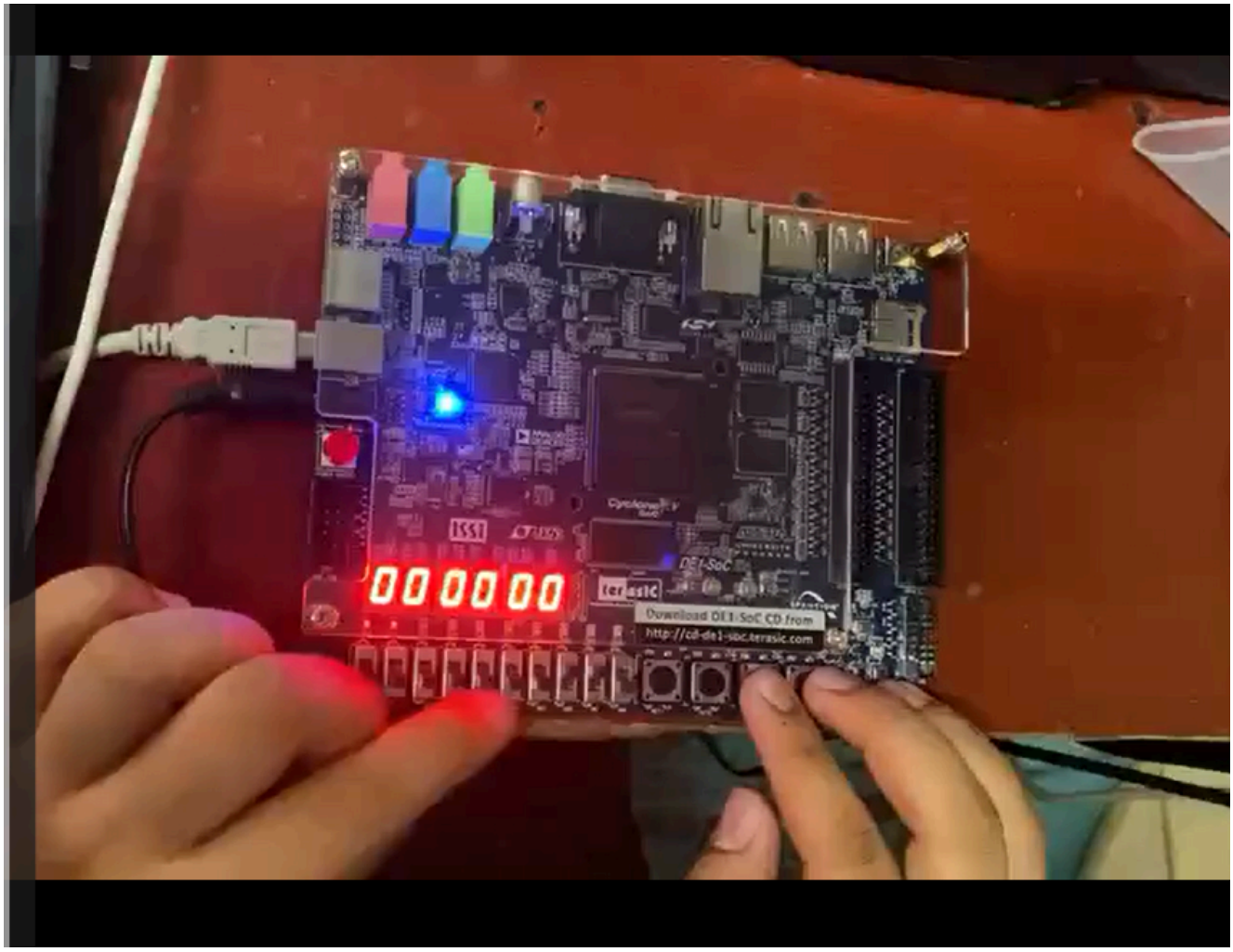
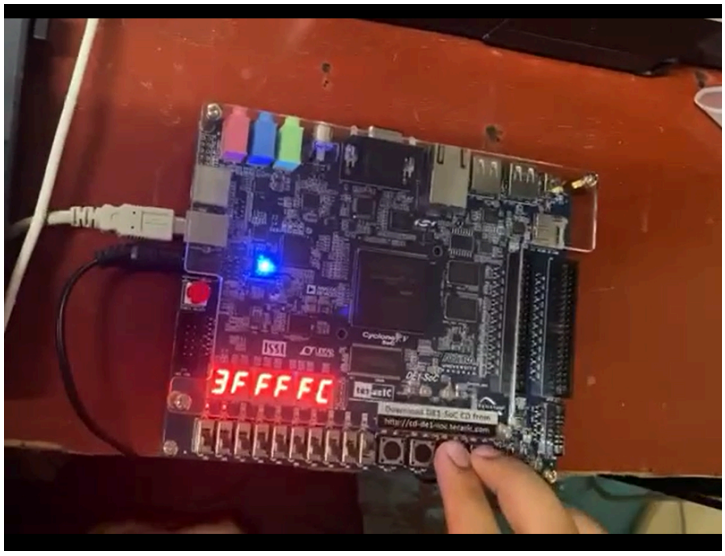


Figure 38: FPGA Setup Pins

Figure 38 shows the output of 0000000 as it starts the program as successful flashing the compilation of Quartus to FPGA, since button turn 0 when its being pushed thats why I hold the Key[1] since thai is assigned as the Reset and the Key[0] as the clock , the values of SW[3:0] serve as my control to what to show on the 7 segments and led so either the one I assigned as the wires on the processor




```
VSIM 8> run -all
# Time: 0 | PC: xxxxxxxx | Instr: 00000000 | ALU A: 00(
# Time: 5000 | PC: 003ffffc | Instr: 00000000 | ALU A:
# Time: 20000 | PC: 003ffffc | Instr: xxxxxxxx | ALU A:
# Time: 25000 | PC: 00400000 | Instr: 200a4000 | ALU A:
# Time: 35000 | PC: 00400004 | Instr: 000aa022 | ALU A:
# Time: 45000 | PC: 00400008 | Instr: 028aa022 | ALU A:
# Time: 55000 | PC: 0040000c | Instr: 028aa022 | ALU A:
```

Figure 39: Program Counter on FGPA Output at RESET

Based on the figure above and the reference of these outputs this shows as it makes positive edge on the clock as I release the KEY[0] and KEY[1] this shows the program counter is 003FFFFC including the 8 led lights that serve as the 00 on the output , This is the value of t he reset value as I put on the program counter module which is shown on figure 6.

```
# Time: 85000 | PC: 00400018 | Instr: ae8a000c | ALU A: ffff0000 | ALU B: 0000000c | ALU Out: ffff000c | Serial Out:  | Serial WREN: 0
# Time: 95000 | PC: 0040001c | Instr: 200a0065 | ALU A: 00000000 | ALU B: 00000065 | ALU Out: 00000065 | Serial Out: H | Serial WREN: 1
# Time: 105000 | Serial Write: H (ASCII: 48)
```

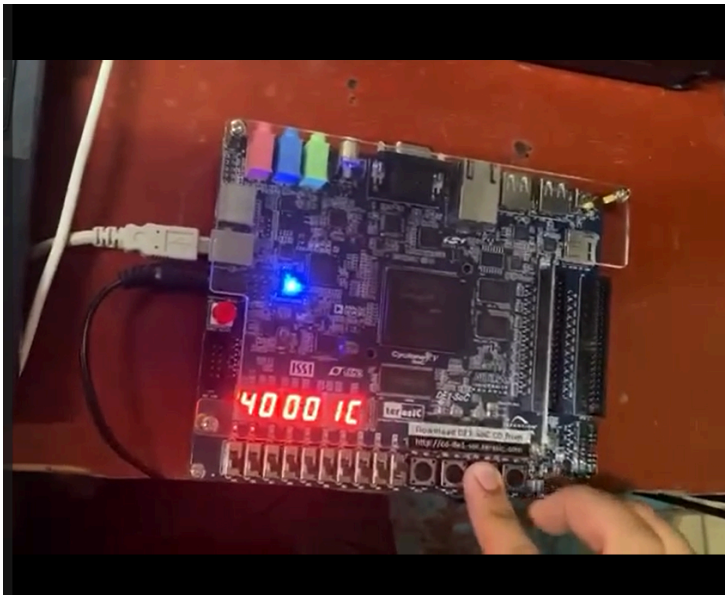
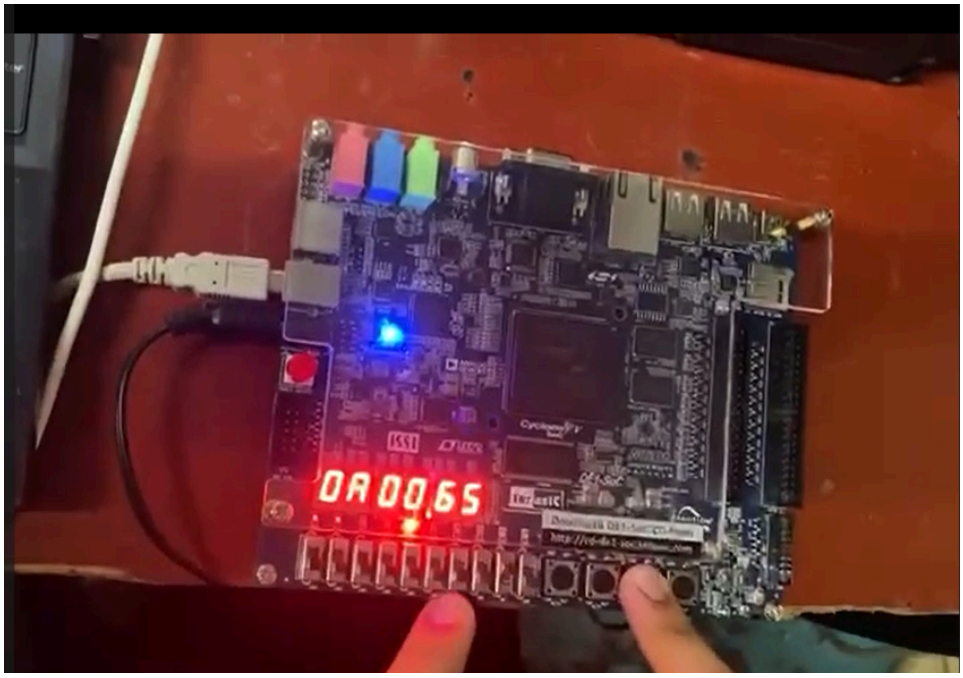


Figure 40: Program Counter on at the Serial Output H

In this figure above, it shows the program counter of the Serial output H of the Hello World, this is the value of 1 of the SW[0:2] as I only toggle up the SW[0] in the FPGA Switches




```
# Time: 85000 | PC: 00400018 | Instr: ae8a000c | ALU A: ffff0000 | ALU B: 0000000c | ALU Out: ffff000c | Serial Out:  | Serial WREN: 0
# Time: 95000 | PC: 0040001c | Instr: 200a0065 | ALU A: 00000000 | ALU B: 00000065 | ALU Out: 00000065 | Serial Out: H | Serial WREN: 1
# Time: 105000 | Serial Write: H (ASCII: 48)
```

Figure 41: Instruction on at the Serial Output H

This figure shows the instruction of the program counter 0040001C which is the instruction of the output H, this shows 200A0065, as the led lights up on then 5th index which represents 0010 0000 or 02 and then followed by 0A0065 which shows the instruction is at 200A0065 as it shows the same instruction shown at the 0040001C on the testbench of processor

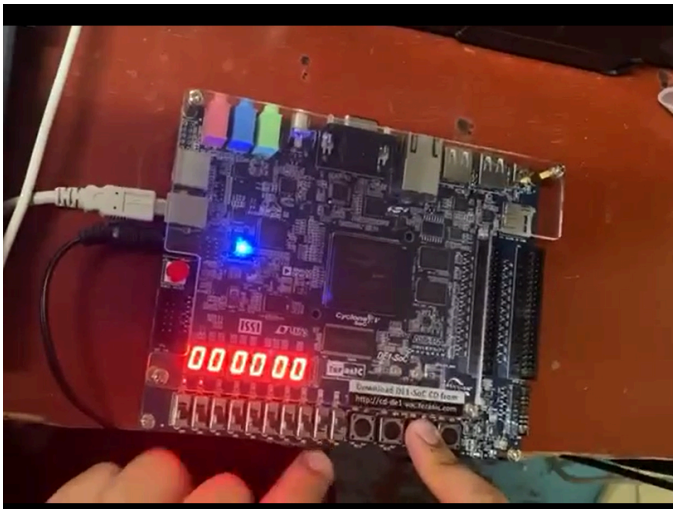


Figure 42:Showing of ALU A in the FPGA from the instruction

Figure 42 shows the value of ALU A in the FPGA From the instruction that is hardcoded in the module, from the original instruction of h65000a20 ,this breakdown to having endian format to process therefore:

from h6500a20 this is converted as endian format

into 20 0A 00 65 -this is the the instruction shown on Figure 41

And on this instruction we make these as decimal values 1 by 1

20 → 0010 0000

0A → 0000 1010

00 → 0000 0000

65 → 0110 0101

Result: 0010 0000 0000 1010 0000 0000 0110 0101

Field	Bit Range	Binary Value	Decimal/Hex Value
Opcode	31-26	001000	8
rs	25-21	00000	0
rt	20-16	01010	10
Immediate	15-0	0000 0110 0101	65

And these interpret as

ADDI $10, $0, 165

or Register $10 = Register $0 + Immediate(65)

And this shows that Figure 42 shows the 0 value of ALU A

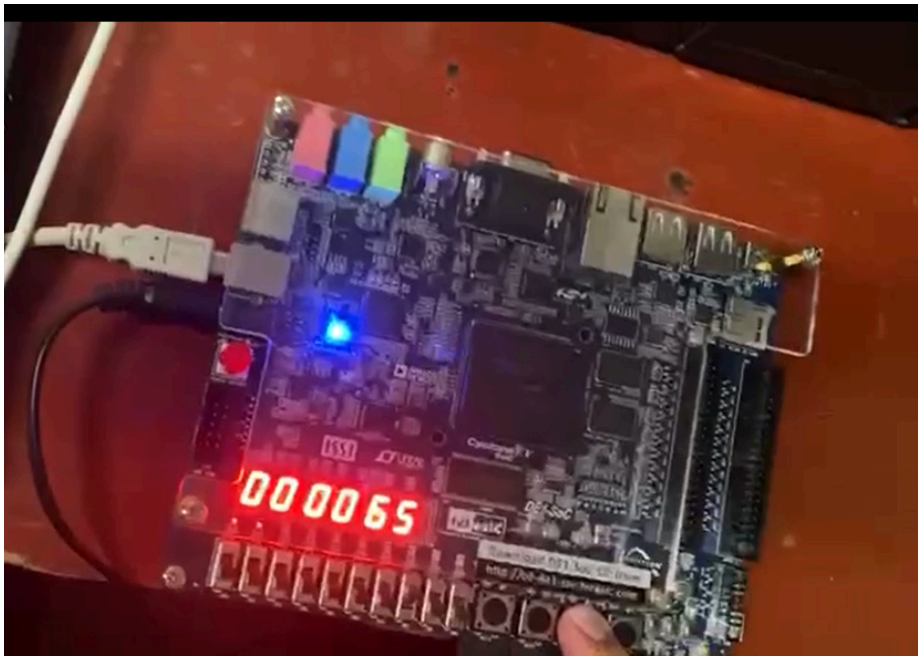


Figure 43:ALU B Value of the H instruction

As breakdown the values mentioned above, Figure 43 shows the ALU B which is a value of 65 or 0000 0110 0101 in the instruction values, the value of SW here is 4 which is the SW[2] is toggled up, and shows the ALU B value

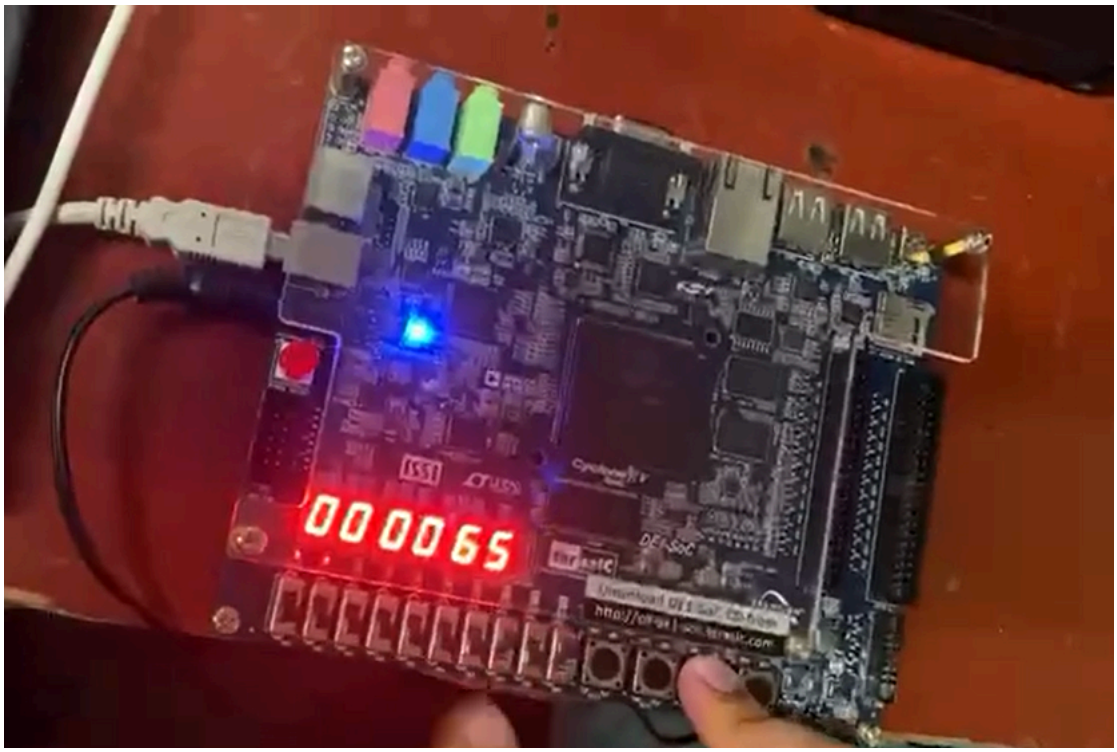


Figure 44:ALU_OUT value shown on FPGA in instruction H

Figure 44 shows the value of ALU OUT, which is in the SW the value is 5 the SW[2] and SW[0] is toggled up, since the instruction is ADDI operation this adds the instruction of

ALU A+ ALU B =ALU OUT

0000 0000 + =0110 0101 = 00000065

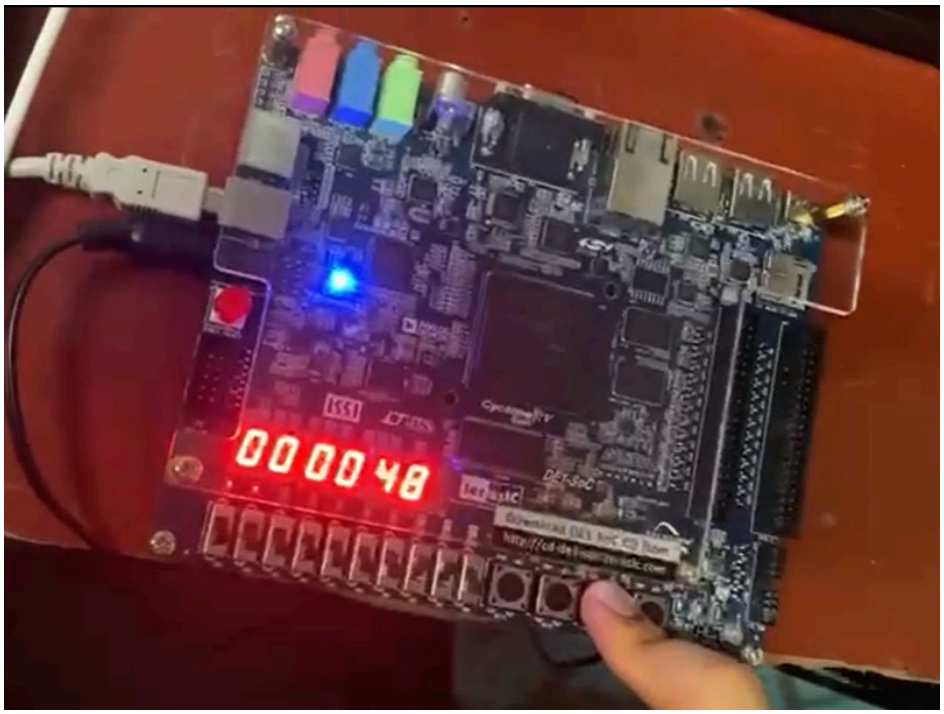


Figure 45:ASCII CODE for instruction code H Serial output

In the figure above this shows the h ascii code of ‘h’ character, on the modelsim this shows as h because of the format of %c that is used on the testbench which is a character representation of the ascii code, but since fpga doesnt have or cannot somehow show some characters , so I decided to just show the ascii code instead which is in this case 48 or the %h representation of the serial out, to breakdown how it got 48, this is a sequence of the instructions from the start

In the series of code

```
[0x00400000]    0x200a4000  addi $10, $0, 16384          ; 18: addi          $10,$0,0x4000

[0x00400004]    0x000aa022  sub $20, $0, $10             ; 19: sub  $20,$0,$10
```

```
[0x00400008]  0x028aa022  sub $20, $20, $10    ; 20: sub $20,$20,$10    # $20 should now contain 0xffffc000

[0x0040000c]  0x028aa022  sub $20, $20, $10    ; 21: sub $20,$20,$10    # $20 should now contain 0xffff8000

[0x00400010]  0x028aa022  sub $20, $20, $10    ; 22: sub $20,$20,$10    # $20 should now contain 0xffff0000

[0x00400014]  0x200a0048  addi $10, $0, 72     ; 24: addi $10,$0,0x48   # $10 is now 'H'

[0x00400018]  0xae8a000c  sw $10, 12($20)      ; 25: sw   $10,12($20)   # write word

[0x0040001c]  0x200a0065  addi $10, $0, 101    ; 27: addi$10,$0,0x65    # 'e'
```

In the part of 0x00400014 , this is the time it writes the H is being stored as 72 and the equivalent of 72 as hexadecimal is 48, so this stores at the address $10, but on thai part this is not being stored yet , this is just the instruction to add,. but with the instruction in 0x0040018 this time it stores and write the word which is it is now being display in the serial out, thats why 48 is shown on the FPGA, and thai is the value of SW of 6 which assigned and the output for ascii codes for H.

With the series of instructions , this is done the same with the remaining letters of the hello world and we can reset also if we just release the reset button to start again from the top of the instructions

Conclusion:

Overall, this activity is a bit hard than imagine as the processor works, aside from the each wires thoroughly that we have to make sure it works every component cause in later part this would be a problem to do in quartus project, We have to make it sure that it is output properly before jumping to the next one. But with the helo of control unit , it somehow also straightforward and this converts all the instructions easily adaptive and can check which is error throughout the codes and will be challenging to debug.

The control unit plays a role on managing the flow of instructions and coordinating the components of the processor. This effectively decodes and will convert the instructions into control signal

What stood out in this activity was the importance of careful debugging and testing. By breaking down the implementation into smaller, manageable steps, we can ensure that each part works correctly before integrating it into the larger system. This iterative approach not only saves time but also improves the reliability of the final design.Furthermore, the hardcoded instructions in

the ROM provided a structured way to understand how memory interacts with the processor since somehow the memh doesnt work in the FPGA part. It allows us to simulate real-world scenarios where programs are stored in memory and accessed sequentially or conditionally based on the program flow.I n this also we can represent in the FPGA different ways to show the output, so its not limited to what I do on the activity , it acn be anything to be shown as long as its possible to the documentation of FPGA de1-soc.

In conclusion, while the task was challenging, it provided a hands-on understanding of processor design and instruction handling in Single Processing MIPS Control. The lessons learned here will undoubtedly be invaluable in future projects and also for my career as Computer Engineering student.

Additional Notes/Observations

Screenshots: (if applicable)

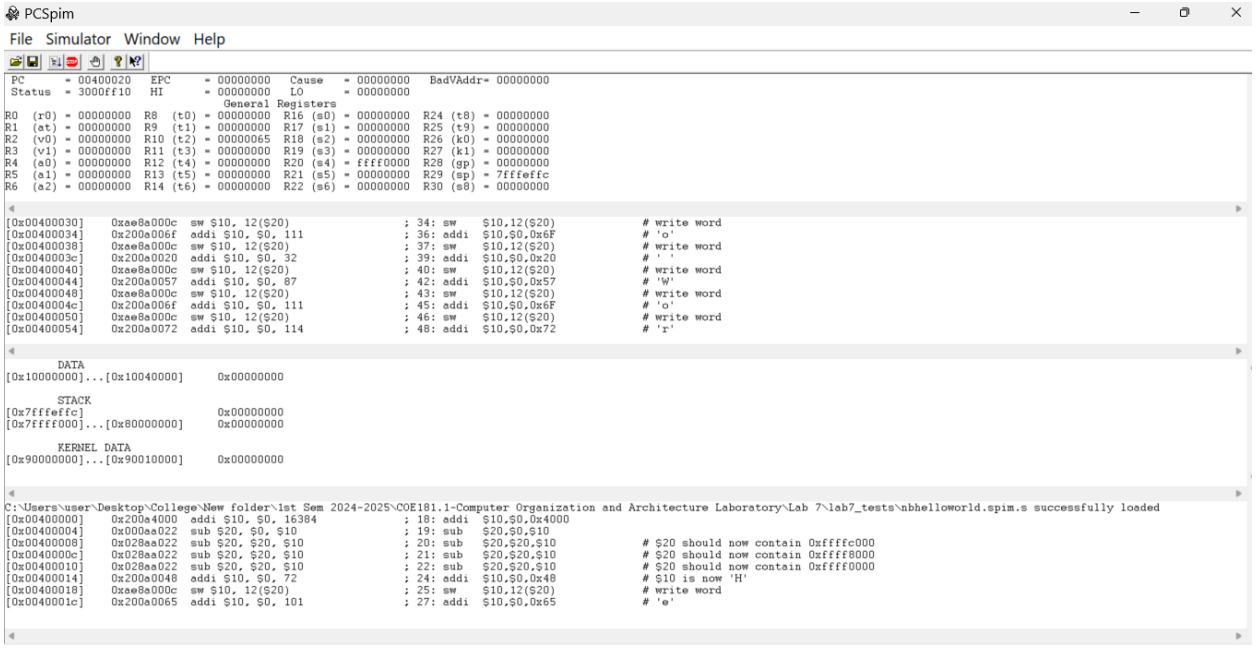


Figure 46: Screenshot of PC Spim Hello world.spim

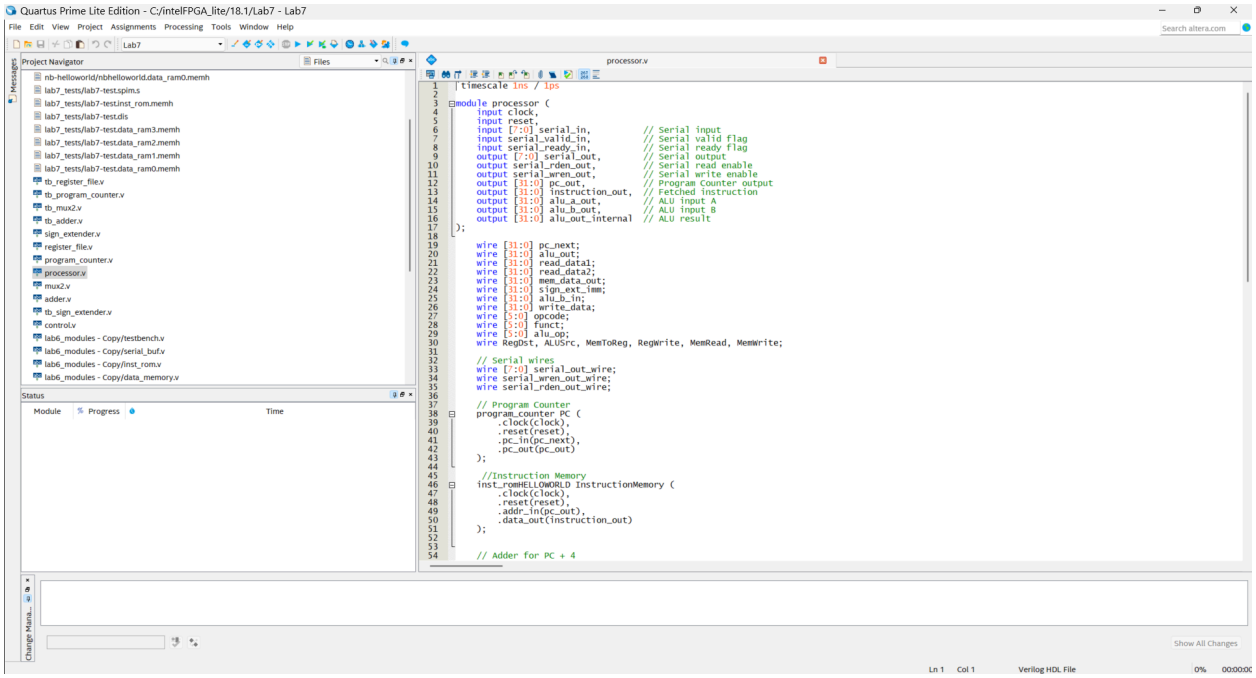


Figure 47: Screenshot of Quartus Project