DIGITAL SYSTEM DESIGN LABORATORY

LAB 6 REPORT

SINGLE CYCLE MICROPROCESSOR DESIGN

NAME: BÙI GIA BẢO
ID: ITITIU22019

I. LAB OBJECTIVES

This Lab experiments are intended to design and test a Single Cycle Microprocessor

II. DESCRIPTION

Single Cycle Microprocessor datapath to be implemented is in figure 2.1.

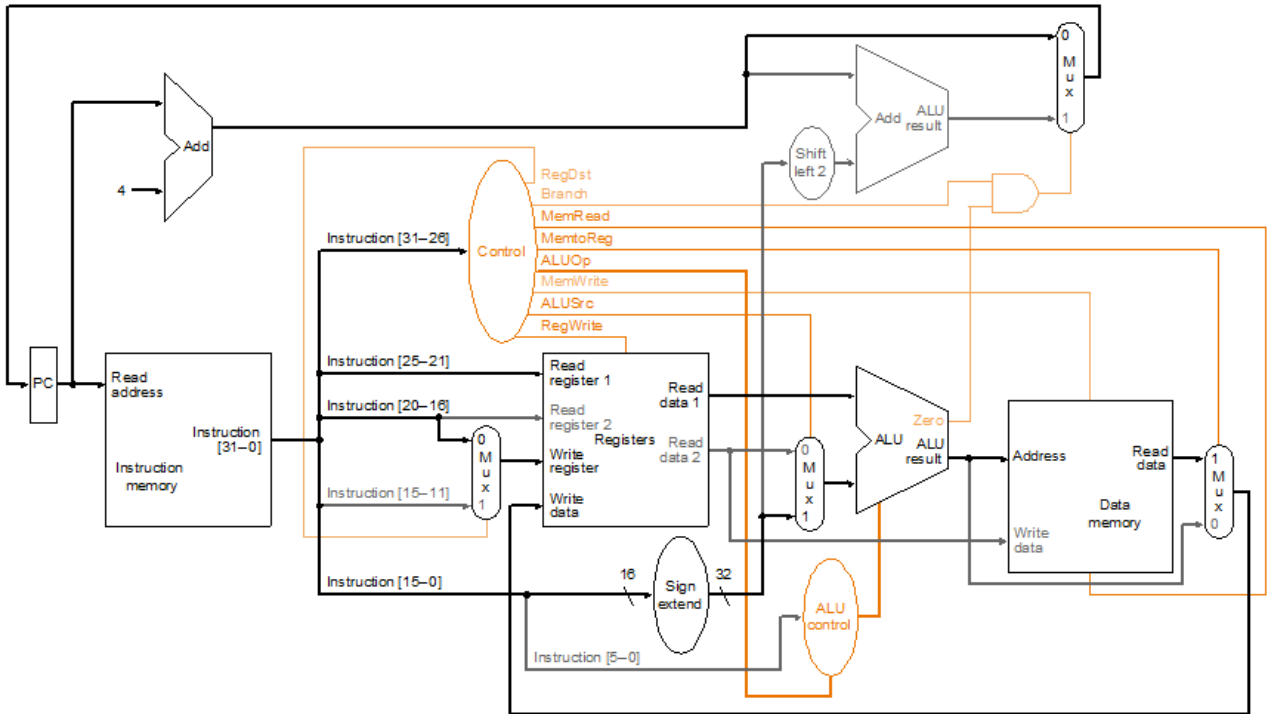


Figure 2.1: Single Cycle Microprocessor DataPath

III. LAB PROCEDURE

III.1 EXPERIMENT NO. 1

III.1.1 AIM: To understand and write the assembly code using MIPS Instruction set

register number of MIPS compiler conventions

Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved (by callee)
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

MIPS Assembly Language Sumarize:

MIPS assembly language				
Category	Instruction	Example	Meaning	Comments
	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
Arithmetic	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
	load word	`lw $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store word	`sw $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory
Data transfer	load byte	`lb $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
Conditional branch	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
	jump	`j 2500`	go to 10000	Jump to target address
Uncondi-	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
tional jump	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Operation code (Op code) sumarize:

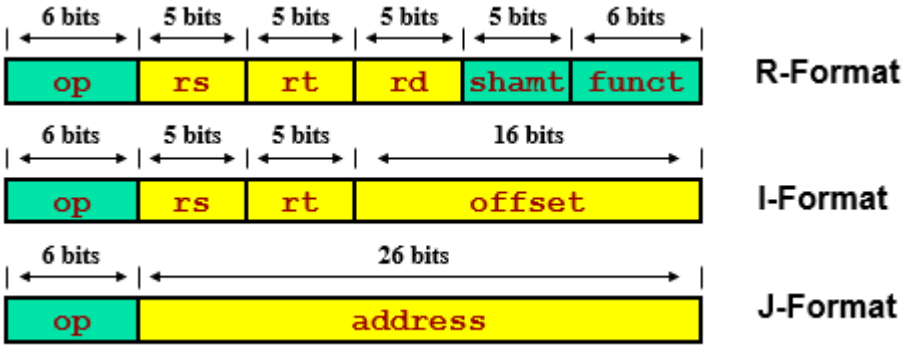
Op	Opcode name	Value
000000	R-format	Rformat1
000010	jmp	JUMP1
000100	beq	BEQ1
100011	lw	Mem1
101011	sw	Mem1

ALU opcode and Function

$ALUOp_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used

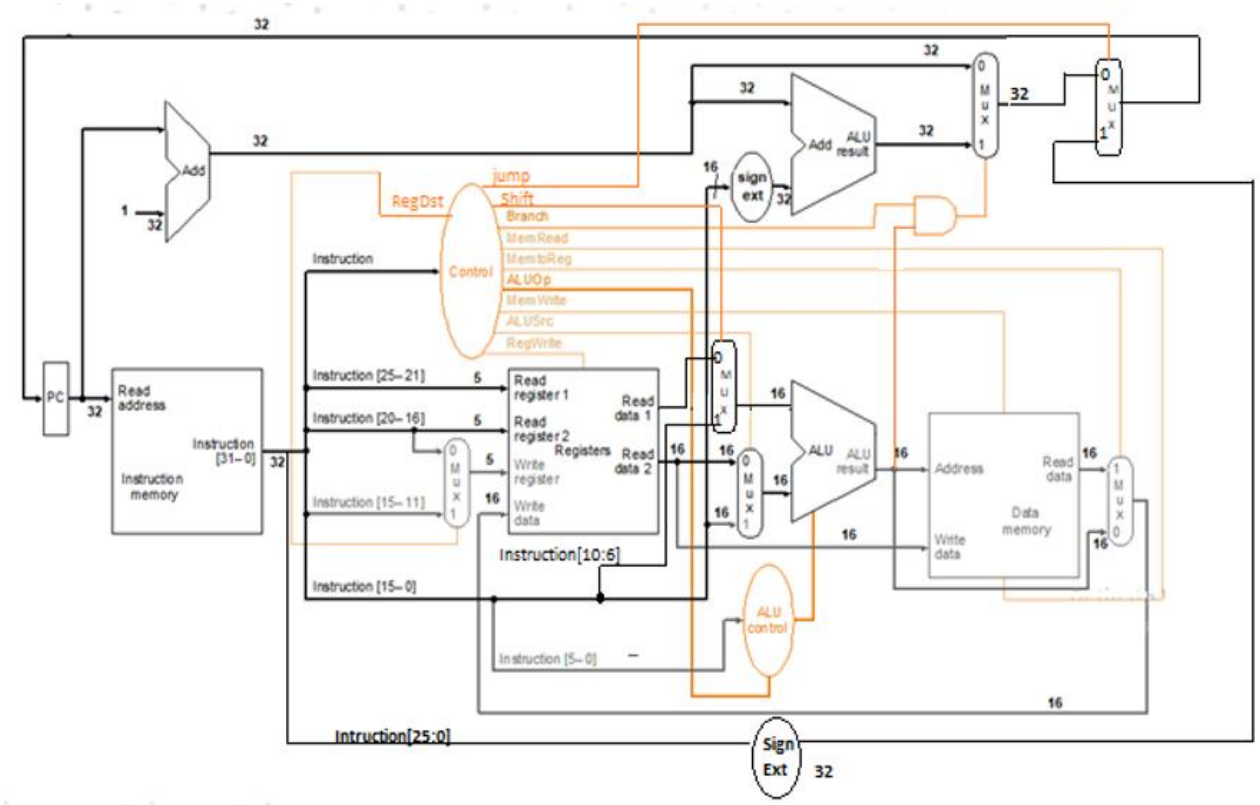
$ALUOp_{1:0}$	Funct	$ALUControl_{2:0}$
00	X	010 (Add)
X1	X	110 (Subtract)
1X	100000 (add)	010 (Add)
1X	100010 (sub)	110 (Subtract)
1X	100100 (and)	000 (And)
1X	100101 (or)	001 (Or)
1X	101010 (slt)	111 (SLT)

Instruction Formats



III.7.1 AIM: Complet Single Cycle processor 16 bit

Op	Opcode name	Value
000000	R-format	R-Type
000010	jmp	J-Tpye
000100	beq	I-Type
100011	lw	
101011	sw	
010000	addi	



III.7.2 CODE

```
module
lab6_ex1(SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HE
X7);
        input  [17:0] SW;
        output[17:0] LEDR;
        output[7:0] LEDG;

        output [0:6] HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7;
        wire[31:0] pc_out,alu_out, instr;
        reg[31:0] w_hex;
```

```
	R_Type_Proc(.reset(SW[16]), .clk(SW[17]), .zero(LEDG[7]),
.Instruction(instr), .w_ALUout(alu_out), .w_pc_out(pc_out));


	always @(*) begin
		case (SW[2:0])
			3'b000: w_hex = instr;    // Instruction
			3'b001: w_hex = alu_out;  // ALU result
			3'b010: w_hex = pc_out;   // PC address
			default: w_hex = 32'b0;
		endcase
	    end
	HEX_7SEG_DECODE H0(.BIN(w_hex[3:0]), .SSD(HEX0));
	HEX_7SEG_DECODE H1(.BIN(w_hex[7:4]), .SSD(HEX1));
	HEX_7SEG_DECODE H2(.BIN(w_hex[11:8]), .SSD(HEX2));
	HEX_7SEG_DECODE H3(.BIN(w_hex[15:12]), .SSD(HEX3));
	HEX_7SEG_DECODE H4(.BIN(w_hex[19:16]), .SSD(HEX4));
	HEX_7SEG_DECODE H5(.BIN(w_hex[23:20]), .SSD(HEX5));
	HEX_7SEG_DECODE H6(.BIN(w_hex[27:24]), .SSD(HEX6));
	HEX_7SEG_DECODE H7(.BIN(w_hex[31:28]), .SSD(HEX7));

endmodule

module R_Type_Proc(reset, clk,zero, Instruction, w_ALUout, w_pc_out);

	input clk;
	input reset;
	output zero;

	output[31:0] Instruction;
	wire[31:0] w_pc_in;
	wire[31:0] w_mux1_out;
	output[31:0] w_pc_out;
	wire[31:0] w_pc_plus_1;
	wire[31:0] w_m2;
	wire[31:0] w_branch_add;

	wire[4:0] w_m3;
	wire[31:0] W_RD1, W_RD2;
	wire[31:0] W_MemtoReg;
```

```
        wire[31:0] w_m1;
        output[31:0] w_ALUout;
        wire[31:0] w_RDm;

        wire RegWrite;
        wire ALUsrc;
        wire ALUop;
        wire PCsrc;
        wire MemRead;
        wire MemWrite;
        wire MemtoReg;
        wire RegDst;

        Program_Counter c1(.clk(clk), .reset(reset), .PC_in(w_pc_in),
.PC_out(w_pc_out));

        Adder32Bit c2(.input1(w_pc_out), .input2(32'b1), .out(w_pc_plus_1));

        Mux_32_bit c3(.in0(w_pc_plus_1), .in1(w_m2), .mux_out(w_pc_in),
.select(PCsrc));

        Adder32Bit c4(.input1(w_pc_plus_1), .input2(w_branch_add),
.out(w_m2));

        Instruction_Memory c5(.read_address(w_pc_out), .instruction(Instruction),
.reset(reset));

        Register_File c6(    .clk(clk),
                                        .read_addr_1(Instruction[25:21]),
                                        .read_addr_2(Instruction[20:16]),
                                        .write_addr(w_m3),
                                        .read_data_1(W_RD1),
                                        .read_data_2(W_RD2),
                                        .write_data(W_MemtoReg),
                                        .RegWrite(RegWrite));

        Sign_Extension c7(.sign_in(Instruction[15:0]), .sign_out(w_branch_add));

        Mux_32_bit c8(.in0(W_RD2), .in1(w_branch_add), .mux_out(w_m1),
.select(ALUsrc));
```

```
        alu     c9(     .alufn(ALUop),
                        .ra(W_RD1),
                        .rb_or_imm(w_m1),
                        .aluout(w_ALUout),
                        .zero(zero));

        Data_Memory c10(.addr(w_ALUout),
                                    .write_data(W_RD2),
                                    .read_data(w_RDm),
                                    .MemRead(MemRead),
                                    .MemWrite(MemWrite));

        Mux_32_bit c11(.in0(w_ALUout), .in1(w_RDm),
.mux_out(W_MemtoReg), .select(MemtoReg));

        control c12(
                                .op_code(Instruction[31:26]),
                                .RegDst(RegDst),
                                .PCsrc(PCsrc),
                                .MemRead(MemRead),
                                .MemtoReg(MemtoReg),
                                .ALUop(ALUop),
                                .MemWrite(MemWrite),
                                .ALUsrc(ALUsrc),
                                .RegWrite(RegWrite)
);

        Mux_5_bit c13(.in0(Instruction[20:16]), .in1(Instruction[15:11]),
.mux_out(w_m3), .select(RegDst));

endmodule



module HEX_7SEG_DECODE(BIN, SSD);
  input [3:0] BIN;
  output reg [0:6] SSD;
  always begin
    case(BIN)
      0:SSD=7'b0000001;
      1:SSD=7'b1001111;
```

```
      2:SSD=7'b0010010;
      3:SSD=7'b0000110;
      4:SSD=7'b1001100;
      5:SSD=7'b0100100;
      6:SSD=7'b0100000;
      7:SSD=7'b0001111;
      8:SSD=7'b0000000;
      9:SSD=7'b0001100;
      10:SSD=7'b0001000;
      11:SSD=7'b1100000;
      12:SSD=7'b0110001;
      13:SSD=7'b1000010;
      14:SSD=7'b0110000;
      15:SSD=7'b0111000;
    endcase
  end
endmodule

module Program_Counter (clk, reset, PC_in, PC_out);
        input clk, reset;
        input [31:0] PC_in;
        output reg [31:0] PC_out;
        always @ (posedge clk or posedge reset)
        begin
                if(reset==1'b1)
                        PC_out<=0;
                else
                        PC_out<=PC_in;
        end
endmodule

module Adder32Bit(input1, input2, out);
    input [31:0] input1, input2;
    output [31:0] out;
    reg [31:0]out;
    always@( input1 or input2)
       begin
          out <= input1 + input2;
       end
endmodule
```

```
module alu(
        input [2:0] alufn,
        input [31:0] ra,
        input [31:0] rb_or_imm,
        output reg [31:0] aluout,
        output reg zero);
        parameter    ALU_OP_ADD      = 3'b000,
                     ALU_OP_SUB      = 3'b001,
                     ALU_OP_AND      = 3'b010,
                     ALU_OP_OR       = 3'b011,
                     ALU_OP_XOR      = 3'b100,
                     ALU_OP_LW       = 3'b101,
                     ALU_OP_SW       = 3'b110,
                     ALU_OP_BEQ      = 3'b111;

always @(*)
begin
                case(alufn)
                        ALU_OP_ADD      : aluout = ra + rb_or_imm;
                        ALU_OP_SUB      : aluout = ra - rb_or_imm;
                        ALU_OP_AND      : aluout = ra & rb_or_imm;
                        ALU_OP_OR       : aluout = ra | rb_or_imm;
                        ALU_OP_XOR      : aluout = ra ^ rb_or_imm;
                        ALU_OP_LW       : aluout = ra + rb_or_imm;
                        ALU_OP_SW       : aluout = ra + rb_or_imm;
                        ALU_OP_BEQ      : begin
                                          zero = (ra==rb_or_imm)?1'b1:1'b0;
                                          aluout = ra - rb_or_imm;
                                        end
                endcase
end
endmodule
module Register_File (clk,read_addr_1, read_addr_2, write_addr, read_data_1,
read_data_2, write_data, RegWrite);
        input [4:0] read_addr_1, read_addr_2, write_addr;
        input [31:0] write_data;
        input  clk,RegWrite;
        reg checkRegWrite;
        output reg [31:0] read_data_1, read_data_2;
        reg [31:0] Regfile [31:0];
        integer k;
```

```
        initial begin
        for (k=0; k<32; k=k+1)
                        begin
                                Regfile[k] = 32'd0;
                        end



        end

        //assign read_data_1 = Regfile[read_addr_1];
      always @(read_data_1 or Regfile[read_addr_1])
              begin
                if (read_addr_1 == 0) read_data_1 = 0;
                else
                begin
                read_data_1 = Regfile[read_addr_1];

//$display("read_addr_1=%d,read_data_1=%h",read_addr_1,read_data_1);
                end
              end
        //assign read_data_2 = Regfile[read_addr_2];
      always @(read_data_2 or Regfile[read_addr_2])
              begin
                if (read_addr_2 == 0) read_data_2 = 0;
                else
                begin
                read_data_2 = Regfile[read_addr_2];

//$display("read_addr_2=%d,read_data_2=%h",read_addr_2,read_data_2);
                end
              end
        always @(posedge clk)
              begin
                    if (RegWrite == 1'b1)
                      begin
                         Regfile[write_addr] = write_data;

                      end
              end
endmodule
```

```
module Data_Memory (addr, write_data, read_data, MemRead, MemWrite);
	input [31:0] addr;
	input [31:0] write_data;
	output [31:0] read_data;
	input MemRead, MemWrite;
	reg [31:0] DMemory [255:0];
	integer k;
	assign read_data = (MemRead) ? DMemory[addr] : 32'bx;
	initial begin
		for (k=0; k<64; k=k+1)
	 begin
			DMemory[k] = 32'b0;
			end
		DMemory[11] = 99;
	end
	always @(*)
	begin
		if (MemWrite) DMemory[addr] = write_data;
	end
endmodule

module Sign_Extension (sign_in, sign_out);
	input [15:0] sign_in;
	output [31:0] sign_out;
	assign sign_out[15:0]=sign_in[15:0];
	assign sign_out[31:16]=sign_in[15]?16'b1111_1111_1111_1111:16'b0;
endmodule

module Mux_32_bit (in0, in1, mux_out, select);
	parameter N = 32;
	input [N-1:0] in0, in1;
	output [N-1:0] mux_out;
	input select;
	assign mux_out = select? in1: in0 ;
endmodule

module Mux_5_bit (in0, in1, mux_out, select);
	parameter N = 5;
	input [N-1:0] in0, in1;
	output [N-1:0] mux_out;
```

```
        input select;
        assign mux_out = select? in1: in0 ;
endmodule

module Instruction_Memory (read_address, instruction, reset);
        input reset;
        input [31:0] read_address;
        output [31:0] instruction;
        reg [31:0] Imemory [63:0];
        integer k;
        // I-MEM in this case is addressed by word, not by byte
        assign instruction = Imemory[read_address];
        always @(posedge reset)
        begin
        for (k=16; k<32; k=k+1)
                begin
                  // here Out changes k=0 to k=16
                  Imemory[k] = 32'b0;
                end

   // addi $s2, $zero, 1
   Imemory[0] = 32'b001000_00000_10010_0000000000000001;

   // addi $s3, $zero, 10
   Imemory[1] = 32'b001000_00000_10011_0000000000001010;

   // addi $s4, $zero, 0
   Imemory[2] = 32'b001000_00000_10100_0000000000000000;

   // add $s4, $s2, $s4
   Imemory[3] = 32'b000000_10010_10100_10100_00000_100000;

   // add $s4, $s3, $s4
   Imemory[4] = 32'b000000_10011_10100_10100_00000_100000;

   // add $s5, $s4, $s2
   Imemory[5] = 32'b000000_10100_10010_10101_00000_100000;

   // rest = NOP
   Imemory[6] = 32'b0;
   Imemory[7] = 32'b0;
```

```
        end
endmodule

module Sign_Extension_26 (sign_in, sign_out);
        input [25:0] sign_in;
        output [31:0] sign_out;
        assign sign_out[25:0]=sign_in[25:0];
        assign sign_out[31:26]=sign_in[25]?6'b111111:6'b0;
endmodule

module control(
    input  [5:0] op_code,
    output reg RegDst,
    output reg PCsrc,
    output reg MemRead,
    output reg MemtoReg,
    output reg [2:0] ALUop,
    output reg MemWrite,
    output reg ALUsrc,
    output reg RegWrite
);

always @(*) begin
    // default values
    RegDst   = 0;
    PCsrc    = 0;
    MemRead  = 0;
    MemtoReg = 0;
    ALUop    = 3'b000;
    MemWrite = 0;
    ALUsrc   = 0;
    RegWrite = 0;

    case(op_code)

    // R-type
    6'b000000: begin
        RegDst   = 1;
        ALUop    = 3'b000; // ADD
        RegWrite = 1;
    end
```

```
    // addi
    6'b001000: begin
        ALUsrc   = 1;
        ALUop    = 3'b000; // ADD
        RegWrite = 1;
    end

    // lw
    6'b100011: begin
        ALUsrc   = 1;
        MemRead  = 1;
        MemtoReg = 1;
        ALUop    = 3'b000; // ADD
        RegWrite = 1;
    end

    // sw
    6'b101011: begin
        ALUsrc   = 1;
        MemWrite = 1;
        ALUop    = 3'b000; // ADD
    end

    // beq
    6'b000100: begin
        ALUop = 3'b001;  // SUB
        PCsrc = 1;       // (should be AND with zero later)
    end

    // j (basic support)
    6'b000010: begin
        PCsrc = 1;
    end

    endcase
end
endmodule
```

III.7.3 LAB ASSIGNMENT

1) Write Verilog code to implement Complete Processor module with all following instructions:

Op	Opcode name	Value
000000	R-format	R-Type
000010	jmp	J-Tpye
000100	beq	I-Type
100011	lw	
101011	sw	
010000	addi	

2) Write testbenches to verify Complete Processor, simulate and check the simulation output data.

3) Compile the following code into binary machine code and store in Instruction memory to test the Complete Processor.

Testing Assembly Program 1:

```
        Instruction                 Meaning
Begin:
        addi $s2, $zero, 0x55 //  load immediate value 0x55 to register $s2
        addi $s3, $zero, 0x22 //        load immediate value 0x22 to register $s3
        addi $s5, $zero, 0x77 //        load immediate value 0x77 to register $s5
        add $s4, $s2, $s3       //   $s4 = $s2 + $s3  => R20=0x77
        sub $s1, $s2, $s3       //   $s1 = $s2 – $s3  => R17=0x22
        sw $s1, 0x02($s2)       //   Memory[$s2+0x02] = $s1
        lw $s6, 0x02($s2)      //   $s6 = Memory[$s2+0x02]
        bne $s5, $s4, End       //   Next instr. is at End if $s5 != $s4
        addi $s8, $zero, 0x10  //        load immediate value 10 to register $s8
        beq $s5,$s4, End        //   Next instr. is at End if $s7 == $s4
        addi $s8, $zero, 0x20 //  load immediate value 20 to register $s8
        End:  j End                 //  jump End
```

3) Compile the following code into binary machine code and store in Instruction memory to test the Complete Processor.

Testing Assembly Program 2:

```
        Instruction                 Meaning
Begin:  addi $s2, $zero, 0x55 //  load immediate value 0x55 to register $s2
        addi $s3, $zero, 0x22 //   load immediate value 0x22 to register $s3
        addi $s5, $zero, 0x77 //   load immediate value 0x77 to register $s5
        add $s4, $s2, $s3       //   $s4 = $s2 + $s3  => R20=0x77
        sub $s1, $s2, $s3       //   $s1 = $s2 – $s3  => R17=0x22
        sw $s1, 0x02($s2)       //   Memory[$s2+0x02] = $s1
        lw $s6, 0x02($s2)      //   $s6 = Memory[$s2+0x02]
        beq $s5,$s4, End        //   Next instr. is at End if $s7 == $s4
```

```
addi $s8, $zero, 0x10 //  load immediate value 10 to register $s8
bne $s5, $s4, End    //     Next instr. is at End if $s5 != $s4
addi $s8, $zero, 0x20 //  load immediate value 20 to register $s8
End:          j End                    //     jump End
```

III. LAB REPORT GUIDELINES

Students write up a report on the Verilog HDL implementation experiment projects created in this lab. The lab report should include Verilog code for the module under test, Verilog test bench code and a truth table results, and example data input and output to validate the experiment. Simulation Result in form of Simulation Capture Screen. The implementation results in FPGA Kit, compare the simulation results and implementation results.