|  |
| --- |
| **DIGITAL SYSTEM DESIGN LABORATORY** |
| **LAB 6** |

**SINGLE CYCLE MICROPROCESSOR DESIGN**

### 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

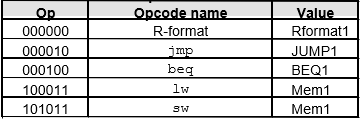
##### III.1.1 AIM: To understand and write the assembly code using MIPS Instruction set

##### register number of MIPS compiler conventions

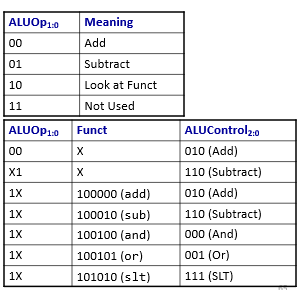
##### 

##### MIPS Assembly Language Sumarize:

##### Operation code (Op code) sumarize:

****

**ALU opcode and Function**

****

##### Instruction Formats

**Timeline

Description automatically generated**

**III.1.2 CODE**

**a) Assembly Code sample 1:**

Instruction Meaning

addi $s0, $zero, 33 load immediate value 33 to register $s0

addi $s1, $zero, 66 load immediate value 66 to register $s1

add $s2, $s0, $s1 $s2 = $s0 + $s1  
sub $s3, $s1, $s0 $s1 = $s1 – $s0  
sw $s3, 10($s2) Memory[$s2+10] = $s3

lw $s1, 10($s2) $s1 = Memory[$s2+10]

**Ans:**

00100000000100000000000000100001 (537.919.521)

00100000000100010000000001000010 (537.985.090)

00000010000100011001000000100000 (34.705.440)

00000010001100001001100000100010 (36.739.106)

10101110010100110000000000001010 (2.924.675.082)

10001110010100010000000000001010 (2.387.673.098)

This program demonstrates basic arithmetic and memory operations:

* Values are added, subtracted, and stored in memory.
* The memory is then accessed, and the stored value is retrieved into a register.

**b) Assembly Code sample 2:**

Assume the code start from address PC=0x00000000, one instruction is store in one memory location.

Instruction Meaning

addi $s2, $zero, 55 load immediate value 55 to register $S2

addi $s3, $zero, 22 load immediate value 22 to register $S3

addi $s5, $zero, 33 load immediate value 55 to register $S3

add $s4,$s2,$s3 $s4 = $s2 + $s3  
sub $s1,$s2,$s3 $s1 = $s2 – $s3  
sw $s1,100($s2) Memory[$s2+100] = $s1

lw $s1,100($s2) $s1 = Memory[$s2+100]   
bne $s1,$s5,End Next instr. is at End if $s4 !=$s5

addi $s6, $zero, 10 load immediate value 10 to register $s6  
beq $s4,$s5, End Next instr. is at End if $s4 = $s5

addi $s6, $zero, 20 load immediate value 20 to register $s6

End: j End jump Here

**Ans:**

00100000000100100000000000110111 (538.050.615)

00100000000100110000000000010110 (538.116.118)

00100000000101010000000000100001 (538.247.201)

00000010010100111010000000100000 (39.034.912)

00000010010100111000100000100010 (39.028.770)

10101110010100010000000001100100 (2.924.544.100)

10001110010100010000000001100100 (2.387.673.188)

00010110001101010000 0000 00000101 (372.572.165)

00100000000101100000000000001010 (538.312.714)

00010010100101010000000000000011 (311.754.755)

00100000000101100000000000010100 (538.312.724)

00001000000000000000000000000001 (134.217.729)

* + - * The program performs arithmetic operations on registers $s2, $s3, $s4, $s1, and $s5.
* It stores and loads values from memory.
* Conditional branching (bne and beq) checks if certain register values are equal or not and branches accordingly.
* The program contains an infinite loop at the End label (j End), which halts the execution flow.

**III.1.3 LAB ASSIGNMENT**a) Compile the Assembly **Assembly Code sample 1** into machine code (decimal code and binary code)

b) Explain briefly the meaning of **Assembly Code sample 1**

c) Compile the Assembly **Assembly Code sample 2** into machine code (decimal code and binary code)

d) Explain briefly the meaning of **Assembly Code sample 2**

### III.2 EXPERIMENT NO. 2

##### III.2.1 AIM: To implement R-Type Datapath

##### 

##### Given the assembly code

addi $s1, $zero, 33 load immediate value 33 to register $s0

addi $s2, $zero, 66 load immediate value 66 to register $s1

##### add $s0, $s1, $s2 $s0 = $s1 + $s2

##### Translate into Macchine code:

##### 

##### 

##### Modify the code register file:

##### Assign initial value of the register 17=33 ( $s1=33)

##### Assign initial value of the register 18=33 ( $s2=66)

##### 

##### 

##### 

**III.2.2 CODE**

**III.2.3 LAB ASSIGNMENT**

|  |
| --- |
| **Top module** |
| module lab6\_ex1(SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7,KEY);  input [17:0]SW;  input [3:0] KEY;  output [17:0]LEDR;  output [7:0]LEDG;  output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;  wire [31:0] W\_LED\_SEG;    assign LEDR=SW;  assign LEDG[3:0]=KEY;  hex\_ssd (W\_LED\_SEG[3:0], HEX0);  hex\_ssd (W\_LED\_SEG[7:4], HEX1);  hex\_ssd (W\_LED\_SEG[11:8],HEX2);  hex\_ssd (W\_LED\_SEG[15:12],HEX3);  hex\_ssd (W\_LED\_SEG[19:16],HEX4);  hex\_ssd (W\_LED\_SEG[23:20],HEX5);  hex\_ssd (W\_LED\_SEG[27:24],HEX6);  hex\_ssd (W\_LED\_SEG[31:28],HEX7);  Datapath\_R\_Type(.clk(KEY[0]),.rst(KEY[1]),.RegWrite(0),.A(SW[4:0]),.B(SW[9:5]),.C(SW[14:10]),.ALU\_Sel(SW[17:15]),.Zero(LEDG[7]),.ALU\_Out(W\_LED\_SEG));  endmodule  module hex\_ssd (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 |
| **Main module** |
| module Datapath\_R\_Type(clk,rst,RegWrite, A, B, C,ALU\_Sel,Zero,ALU\_Out);  input clk,rst,RegWrite;  input [4:0] A, B, C;  input [2:0] ALU\_Sel;  output Zero;  output [31:0]ALU\_Out;  wire [31:0] W\_RD1,W\_RD2;  Register\_File\_16bit C1(.clk(clk), .rst(rst),.RegWrite(RegWrite),  .write\_data(ALU\_Out),.read\_addr\_1(A), .read\_addr\_2(B), .write\_addr(C),  .read\_data\_1(W\_RD1), .read\_data\_2(W\_RD2));  ALU\_16bit C2(.ALU\_Sel(ALU\_Sel),.A(W\_RD1),.B(W\_RD2),.ALU\_Out(ALU\_Out),.zero(Zero));  endmodule  module Register\_File\_16bit (clk,rst, 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 [15:0] write\_data;  input clk,rst ,RegWrite;  output reg [31:0] read\_data\_1, read\_data\_2;  reg [15:0] Regfile [31:0];  integer k;  // Read Data 1  always @(\*) begin  if (read\_addr\_1 == 5'b00000)  read\_data\_1 = 31'd0;  else  read\_data\_1 = Regfile[read\_addr\_1];  end  // Read Data 2  always @(\*) begin  if (read\_addr\_2 == 5'b00000)  read\_data\_2 = 31'd0;  else  read\_data\_2 = Regfile[read\_addr\_2];  end  // Write Data and Reset Logic  always @(negedge clk or negedge rst) begin  if (!rst) begin  // Reset all registers  for (k = 0; k < 32; k = k + 1)  Regfile[k] = k;  end else if (!RegWrite) begin  // Write data to register file  Regfile[write\_addr] = write\_data;  end  end  endmodule  module ALU\_16bit(ALU\_Sel, A, B, ALU\_Out, zero);  input [2:0] ALU\_Sel; // ALU selection  input [15:0] A; // 16-bit input 1  input [15:0] B; // 16-bit input 2  output reg [15:0] ALU\_Out; // ALU 16-bit output  output reg zero;  parameter ALU\_OP\_ADD = 3'b000,  ALU\_OP\_SUB = 3'b001,  ALU\_OP\_MUL = 3'b010,  ALU\_OP\_DIV = 3'b011,  ALU\_OP\_AND = 3'b100,  ALU\_OP\_OR = 3'b101,  ALU\_OP\_BEQ = 3'b110,  ALU\_OP\_BNE = 3'b111;  always @(\*)  begin  case(ALU\_Sel)  ALU\_OP\_ADD : ALU\_Out = A + B;  ALU\_OP\_SUB : ALU\_Out = A - B;  ALU\_OP\_MUL : ALU\_Out = A \* B;  ALU\_OP\_DIV : ALU\_Out = A / B;  ALU\_OP\_AND : ALU\_Out = A & B;  ALU\_OP\_OR : ALU\_Out = A | B;  ALU\_OP\_BEQ : begin  zero = (A==B)?1'b1:1'b0;  ALU\_Out = (A==B)?31'd1:31'd0;  end  ALU\_OP\_BNE : begin  zero = (A!=B)?1'b1:1'b0;  // zero here is different from the above zero  // zero here is equal to 1, when A is different from B  // zero here is equal to 0, when A is the same as B  ALU\_Out = (A!=B)?16'd1:16'd0;  end  endcase  end  endmodule |
| **Testbench** |
|  |
|  |
|  |

### III.3 EXPERIMENT NO. 3

##### III.3.1 AIM: To implement SW I-Type Instruction Datapath

##### 

**III.3.2 CODE**

**III.3.3 LAB ASSIGNMENT**

|  |
| --- |
| **Top module** |
| module lab6\_ex2(SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3);  input [17:0]SW;  output [17:0]LEDR;  output [7:0]LEDG;  output [0:6] HEX3, HEX2, HEX1, HEX0;  wire [15:0] W\_LED\_SEG;    assign LEDR=SW;  hex\_ssd (W\_LED\_SEG[3:0], HEX0);  hex\_ssd (W\_LED\_SEG[7:4], HEX1);  hex\_ssd (W\_LED\_SEG[11:8],HEX2);  hex\_ssd (W\_LED\_SEG[15:12],HEX3);    SW\_datapath(.rs(SW[17:13]), .rt(SW[12:8]), .offset(SW[7:1]),.clk(SW[0]),.ALU\_Sel(4'b0000),.MemWrite(1'b1),.MemRead(1'b1),.Mem\_Out(W\_LED\_SEG));  endmodule  module hex\_ssd (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 |
| **Main module** |
| module SW\_datapath(rs, rt, offset, clk, ALU\_Sel, MemWrite, MemRead, Mem\_Out);  input [4:0] rs, rt;  input [7:0] offset;  input MemWrite, MemRead, clk;  input [3:0] ALU\_Sel;  output [15:0] Mem\_Out;  wire [15:0] W\_RD1, W\_RD2, W\_RD3, ALUout;  wire zero;  Register\_File\_16bit C1(  .clk(clk),  .RegWrite(1'b0),  .write\_data(Mem\_Out),  .read\_addr\_1(rs),  .read\_addr\_2(rt),  .write\_addr(rt),  .read\_data\_1(W\_RD1),  .read\_data\_2(W\_RD2)  );  Sign\_Extension C3(  .sign\_in(offset),  .sign\_out(W\_RD3)  );  ALU\_16bit C2(  .ALU\_Sel(ALU\_Sel),  .A(W\_RD1),  .B(W\_RD3),  .ALU\_Out(ALUout),  .zero(zero)  );  Data\_Memory\_16bit C4(  .clk(clk),  .addr(ALUout),  .write\_data(W\_RD2),  .read\_data(Mem\_Out),  .MemRead(MemRead),  .MemWrite(MemWrite)  );  endmodule  module Register\_File\_16bit (clk,read\_addr\_1, read\_addr\_2, write\_addr, write\_data, RegWrite,read\_data\_1, read\_data\_2);  input clk,RegWrite;  input [4:0] read\_addr\_1, read\_addr\_2, write\_addr;  input [15:0] write\_data;  output [15:0] read\_data\_1, read\_data\_2;  reg [15:0] Regfile [31:0];  integer i;  // Initialize registers  initial begin  for (i = 0; i < 32; i = i + 1) begin  Regfile[i] = 16'b0;  end  Regfile[16] = 16'h33; // Register 16 (contains 0x33)  Regfile[17] = 16'h66; // Register 17 (contains 0x66)  Regfile[18] = 16'h2; // Register 16 (contains 0x33)  Regfile[19] = 16'h5; // Register 17 (contains 0x66)  end  // Read Data Logic  assign read\_data\_1 = Regfile[read\_addr\_1];  assign read\_data\_2 = Regfile[read\_addr\_2];  // Write Data Logic  always @(posedge clk) begin  if (RegWrite) begin  Regfile[write\_addr] <= write\_data;  $display("write\_addr=%d write\_data=%h",write\_addr,write\_data);  end  end  endmodule  module ALU\_16bit(ALU\_Sel, A, B, ALU\_Out, zero);  input [3:0] ALU\_Sel; // ALU selection  input [15:0] A; // 16-bit input 1  input [15:0] B; // 16-bit input 2  output reg [15:0] ALU\_Out; // ALU 16-bit output  output reg zero;  parameter ALU\_OP\_ADD = 4'b0000, // Addition  ALU\_OP\_SUB = 4'b0001, // Subtraction  ALU\_OP\_MUL = 4'b0010, // Multiplication  ALU\_OP\_DIV = 4'b0011, // Division  ALU\_OP\_AND = 4'b0100, // Logical and  ALU\_OP\_OR = 4'b0101, // Logical or  ALU\_OP\_XOR = 4'b0110, // Logical xor  ALU\_OP\_NOR = 4'b0111, // Logical nor  ALU\_OP\_NAND = 4'b1000, // Logical nand  ALU\_OP\_XNOR = 4'b1001, // Logical xnor  ALU\_OP\_SHL = 4'b1010, // Logical shift left  ALU\_OP\_SHR = 4'b1011, // Logical shift right  ALU\_OP\_ROL = 4'b1100, // Rotate left  ALU\_OP\_ROR = 4'b1101, // Rotate right  ALU\_OP\_BEQ = 4'b1110, // Equal comparison  ALU\_OP\_BNE = 4'b1111; // Not Equal comparison  always @(\*)  begin  case(ALU\_Sel)  ALU\_OP\_ADD : ALU\_Out = A + B;  ALU\_OP\_SUB : ALU\_Out = A - B;  ALU\_OP\_MUL : ALU\_Out = A \* B;  ALU\_OP\_DIV : ALU\_Out = A / B;  ALU\_OP\_AND : ALU\_Out = A & B;  ALU\_OP\_OR : ALU\_Out = A | B;  ALU\_OP\_XOR : ALU\_Out = A ^ B;  ALU\_OP\_NOR : ALU\_Out = ~(A | B);  ALU\_OP\_NAND : ALU\_Out = ~(A & B);  ALU\_OP\_XNOR : ALU\_Out = ~(A ^ B);  ALU\_OP\_SHL : ALU\_Out = A<<1;  ALU\_OP\_SHR : ALU\_Out = A>>1;  ALU\_OP\_ROL : ALU\_Out = {A[6:0],A[7]};  ALU\_OP\_ROR : ALU\_Out = {A[0],A[7:1]};  ALU\_OP\_BEQ : begin  zero = (A==B)?1'b1:1'b0;  ALU\_Out = (A==B)?16'd1:16'd0;  end  ALU\_OP\_BNE : begin  zero = (A!=B)?1'b1:1'b0;  // zero here is different from the above zero  // zero here is equal to 1, when A is different from B  // zero here is equal to 0, when A is the same as B  ALU\_Out = (A!=B)?16'd1:16'd0;  end  endcase  end  endmodule  // Sign Extension  module Sign\_Extension (sign\_in, sign\_out);  input [7:0] sign\_in;  output [15:0] sign\_out;  assign sign\_out[7:0]=sign\_in[7:0];  assign sign\_out[15:8]=sign\_in[7]?8'b1111\_1111:8'b0;  endmodule  module Data\_Memory\_16bit (clk,addr,write\_data,read\_data,MemRead,MemWrite);  input clk;  input [15:0] addr;  input [15:0] write\_data;  output reg [15:0] read\_data;  input MemRead;  input MemWrite;  integer i;  reg [15:0] DMemory [0:255];  // Initialize memory for simulation  initial begin    for (i = 0; i < 256; i = i + 1) begin  DMemory[i] = 16'h0000;  end  end  // Synchronous write  always @(posedge clk) begin  if (MemWrite) begin  DMemory[addr] <= write\_data;  $display("Memory Write: Address=%h Data=%h", addr, write\_data);  end  end  // Asynchronous read  always @(\*) begin  if (MemRead)  read\_data = DMemory[addr];  else  read\_data = 16'h0000;  end  endmodule |
| **Testbench** |
| module test\_SW\_datapath;  reg [4:0] rs, rt;  reg [7:0] offset;  reg MemWrite, MemRead, clk;  reg [3:0] ALU\_Sel;  wire [15:0] Mem\_Out;    SW\_datapath uut (  .rs(rs),  .rt(rt),  .offset(offset),  .clk(clk),  .ALU\_Sel(ALU\_Sel),  .MemWrite(MemWrite),  .MemRead(MemRead),  .Mem\_Out(Mem\_Out)  );  // Clock generation  initial begin  clk = 0;  forever #5 clk = ~clk;  end  // Enhanced monitoring  initial begin  $monitor("Time=%0t rs=%h rt=%h offset=%h RD1=%h RD2=%h ALUout=%h Mem\_Out=%h MW=%b MR=%b",  $time, rs, rt, offset,  uut.W\_RD1, uut.W\_RD2, uut.ALUout,  Mem\_Out, MemWrite, MemRead);  end  // Test stimulus  initial begin  // Initialize  rs = 5'h10; // Register 16 (contains 0x33)  rt = 5'h11; // Register 17 (contains 0x66)  offset = 8'h0;  ALU\_Sel = 4'b0000;  MemWrite = 0;  MemRead = 0;    // Wait for stable state  #10;    #2; // Small delay after clock edge  rs = 5'h10;  rt = 5'h11;  offset = 8'h4;  MemWrite = 1;  MemRead = 0;  #10;    #2; // Small delay after clock edge  offset = 8'h8;  MemWrite = 1;  MemRead = 0;  #10;  // End simulation  #10 $finish;  end  endmodule |
|  |
|  |

### III.4 EXPERIMENT NO. 4

##### III.4.1 AIM: To implement LW I-Type Instruction Datapath

##### 

**III.4.2 CODE**

**III.4.3 LAB ASSIGNMENT**

|  |
| --- |
| **Top module** |
| module lab6\_ex3(SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3);  input [17:0]SW;  output [17:0]LEDR;  output [7:0]LEDG;  output [0:6] HEX3, HEX2, HEX1, HEX0;  wire [15:0] W\_LED\_SEG;    assign LEDR=SW;  hex\_ssd (W\_LED\_SEG[3:0], HEX0);  hex\_ssd (W\_LED\_SEG[7:4], HEX1);  hex\_ssd (W\_LED\_SEG[11:8],HEX2);  hex\_ssd (W\_LED\_SEG[15:12],HEX3);    LW\_datapath(.rs(SW[17:13]), .rt(SW[12:8]), .offset(SW[7:1]),.clk(SW[0]),.ALU\_Sel(4'b0000),.MemWrite(1'b0),.MemRead(1'b1),.Mem\_Out(W\_LED\_SEG));  Endmodule  module hex\_ssd (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 |
| **Main module** |
| module LW\_datapath(rs, rt, offset, clk, ALU\_Sel, MemWrite, MemRead, Mem\_Out);  input [4:0] rs, rt;  input [7:0] offset;  input MemWrite, MemRead, clk;  input [3:0] ALU\_Sel;  output [15:0] Mem\_Out;  wire [15:0] W\_RD1, W\_RD2, W\_RD3, ALUout;  wire zero;  Register\_File\_16bit C1(  .clk(clk),  .RegWrite(1'b1),  .write\_data(Mem\_Out),  .read\_addr\_1(rs),  .read\_addr\_2(rt),  .write\_addr(rt),  .read\_data\_1(W\_RD1),  .read\_data\_2(W\_RD2)  );  Sign\_Extension C3(  .sign\_in(offset),  .sign\_out(W\_RD3)  );  ALU\_16bit C2(  .ALU\_Sel(ALU\_Sel),  .A(W\_RD1),  .B(W\_RD3),  .ALU\_Out(ALUout),  .zero(zero)  );  Data\_Memory\_16bit C4(  .clk(clk),  .addr(ALUout),  .write\_data(W\_RD2),  .read\_data(Mem\_Out),  .MemRead(MemRead),  .MemWrite(MemWrite)  );  endmodule  module Register\_File\_16bit (clk,read\_addr\_1, read\_addr\_2, write\_addr, write\_data, RegWrite,read\_data\_1, read\_data\_2);  input clk,RegWrite;  input [4:0] read\_addr\_1, read\_addr\_2, write\_addr;  input [15:0] write\_data;  output [15:0] read\_data\_1, read\_data\_2;  reg [15:0] Regfile [31:0];  integer i;  // Initialize registers  initial begin  for (i = 0; i < 32; i = i + 1) begin  Regfile[i] = 16'b0;  end  Regfile[16] = 16'h33; // Register 16 (contains 0x33)  Regfile[17] = 16'h66; // Register 17 (contains 0x66)  Regfile[18] = 16'h2; // Register 16 (contains 0x33)  Regfile[19] = 16'h5; // Register 17 (contains 0x66)  end  // Read Data Logic  assign read\_data\_1 = Regfile[read\_addr\_1];  assign read\_data\_2 = Regfile[read\_addr\_2];  // Write Data Logic  always @(posedge clk) begin  if (RegWrite) begin  Regfile[write\_addr] <= write\_data;  $display("write\_addr=%h write\_data=%h",write\_addr,write\_data);  end  end  endmodule  module ALU\_16bit(ALU\_Sel, A, B, ALU\_Out, zero);  input [3:0] ALU\_Sel; // ALU selection  input [15:0] A; // 16-bit input 1  input [15:0] B; // 16-bit input 2  output reg [15:0] ALU\_Out; // ALU 16-bit output  output reg zero;  parameter ALU\_OP\_ADD = 4'b0000, // Addition  ALU\_OP\_SUB = 4'b0001, // Subtraction  ALU\_OP\_MUL = 4'b0010, // Multiplication  ALU\_OP\_DIV = 4'b0011, // Division  ALU\_OP\_AND = 4'b0100, // Logical and  ALU\_OP\_OR = 4'b0101, // Logical or  ALU\_OP\_XOR = 4'b0110, // Logical xor  ALU\_OP\_NOR = 4'b0111, // Logical nor  ALU\_OP\_NAND = 4'b1000, // Logical nand  ALU\_OP\_XNOR = 4'b1001, // Logical xnor  ALU\_OP\_SHL = 4'b1010, // Logical shift left  ALU\_OP\_SHR = 4'b1011, // Logical shift right  ALU\_OP\_ROL = 4'b1100, // Rotate left  ALU\_OP\_ROR = 4'b1101, // Rotate right  ALU\_OP\_BEQ = 4'b1110, // Equal comparison  ALU\_OP\_BNE = 4'b1111; // Not Equal comparison  always @(\*)  begin  case(ALU\_Sel)  ALU\_OP\_ADD : ALU\_Out = A + B;  ALU\_OP\_SUB : ALU\_Out = A - B;  ALU\_OP\_MUL : ALU\_Out = A \* B;  ALU\_OP\_DIV : ALU\_Out = A / B;  ALU\_OP\_AND : ALU\_Out = A & B;  ALU\_OP\_OR : ALU\_Out = A | B;  ALU\_OP\_XOR : ALU\_Out = A ^ B;  ALU\_OP\_NOR : ALU\_Out = ~(A | B);  ALU\_OP\_NAND : ALU\_Out = ~(A & B);  ALU\_OP\_XNOR : ALU\_Out = ~(A ^ B);  ALU\_OP\_SHL : ALU\_Out = A<<1;  ALU\_OP\_SHR : ALU\_Out = A>>1;  ALU\_OP\_ROL : ALU\_Out = {A[6:0],A[7]};  ALU\_OP\_ROR : ALU\_Out = {A[0],A[7:1]};  ALU\_OP\_BEQ : begin  zero = (A==B)?1'b1:1'b0;  ALU\_Out = (A==B)?16'd1:16'd0;  end  ALU\_OP\_BNE : begin  zero = (A!=B)?1'b1:1'b0;  // zero here is different from the above zero  // zero here is equal to 1, when A is different from B  // zero here is equal to 0, when A is the same as B  ALU\_Out = (A!=B)?16'd1:16'd0;  end  endcase  end  endmodule  // Sign Extension  module Sign\_Extension (sign\_in, sign\_out);  input [7:0] sign\_in;  output [15:0] sign\_out;  assign sign\_out[7:0]=sign\_in[7:0];  assign sign\_out[15:8]=sign\_in[7]?8'b1111\_1111:8'b0;  endmodule  module Data\_Memory\_16bit (clk,addr,write\_data,read\_data,MemRead,MemWrite);  input clk;  input [15:0] addr;  input [15:0] write\_data;  output reg [15:0] read\_data;  input MemRead;  input MemWrite;  integer i;  reg [15:0] DMemory [0:255];  // Initialize memory for simulation  initial begin    for (i = 0; i < 256; i = i + 1) begin  DMemory[i] = 16'h0000;  end  DMemory[16'h0037] = 16'hAAAA;  DMemory[16'h003B] = 16'hBBBB;  end  // Synchronous write  always @(posedge clk) begin  if (MemWrite) begin  DMemory[addr] <= write\_data;  $display("Memory Write: Address=%h Data=%h", addr, write\_data);  end  end  // Asynchronous read  always @(\*) begin  if (MemRead)  read\_data = DMemory[addr];  else  read\_data = 16'h0000;  end  endmodule |
| **Testbench** |
| module test\_LW\_datapath;  reg [4:0] rs, rt;  reg [7:0] offset;  reg MemWrite, MemRead, clk;  reg [3:0] ALU\_Sel;  wire [15:0] Mem\_Out;  LW\_datapath uut (  .rs(rs),  .rt(rt),  .offset(offset),  .clk(clk),  .ALU\_Sel(ALU\_Sel),  .MemWrite(MemWrite),  .MemRead(MemRead),  .Mem\_Out(Mem\_Out)  );  // Clock generation  initial begin  clk = 0;  forever #5 clk = ~clk;  end  // Monitor  initial begin  $monitor("Time=%0t rs=%h rt=%h offset=%h RD1=%h RD2=%h ALUout=%h Mem\_Out=%h MW=%b MR=%b",  $time, rs, rt, offset,  uut.W\_RD1, uut.W\_RD2, uut.ALUout,  Mem\_Out, MemWrite, MemRead);  end  initial begin    // Initialize signals  rs = 5'h10; // R16 contains 0x33  rt = 5'h11; // Target register  offset = 8'h4;  ALU\_Sel = 4'b0000;  MemWrite = 0;  MemRead = 1; // LW needs read enabled  // Test Case 1: Load from first location  #20;  // Test Case 2: Load from second location  offset = 8'h8;  #20;  $finish;  end  endmodule |
|  |
|  |

### III.5 EXPERIMENT NO. 5

##### III.5.1 AIM: To implement I-Type Instruction Beq datapath

##### 

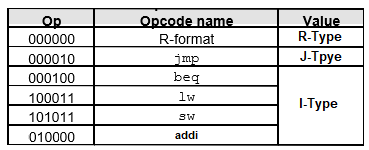
**III.5.2 CODE**

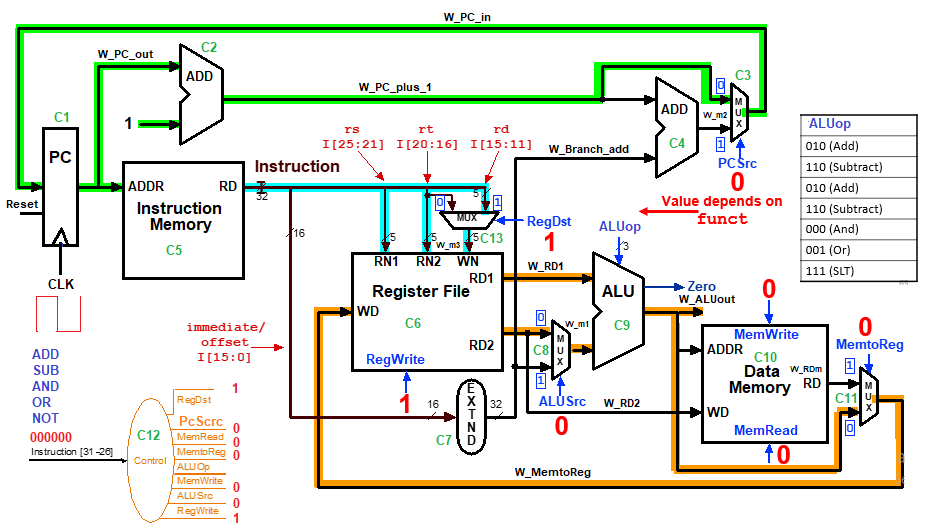
**III.5.3 LAB ASSIGNMENT**

|  |
| --- |
| **Top module** |
| module lab6\_ex4(SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3);  input [17:0]SW;  output [17:0]LEDR;  output [7:0]LEDG;  output [0:6] HEX3, HEX2, HEX1, HEX0;  wire [15:0] W\_LED\_SEG;    assign LEDR=SW;  hex\_ssd (W\_LED\_SEG[3:0], HEX0);  hex\_ssd (W\_LED\_SEG[7:4], HEX1);  hex\_ssd (W\_LED\_SEG[11:8],HEX2);  hex\_ssd (W\_LED\_SEG[15:12],HEX3);    beq\_Datapath (.clk(SW[0]),.rs(SW[17:13]),.rt(SW[12:8]),.offset(SW[7:5]),.ALU\_Sel(4'b1110),.PC\_plus\_4(SW[4:1]),.zero(LEDG[7]),.beq\_add(W\_LED\_SEG));  endmodule  module hex\_ssd (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 |
| **Main module** |
| module beq\_Datapath (clk,rs,rt,offset,ALU\_Sel,PC\_plus\_4,zero,beq\_add);  input [4:0] rs, rt;  input [7:0] offset;  input [15:0] PC\_plus\_4;  input clk;  input [3:0] ALU\_Sel;  output [15:0] beq\_add;  output zero;  wire [15:0] W\_RD1, W\_RD2, W\_RD3, ALUout;    Register\_File\_16bit C1(  .clk(clk),  .RegWrite(1'b0),  .write\_data(1'b0),  .read\_addr\_1(rs),  .read\_addr\_2(rt),  .write\_addr(1'b0),  .read\_data\_1(W\_RD1),  .read\_data\_2(W\_RD2)  );  Sign\_Extension C3(  .sign\_in(offset),  .sign\_out(W\_RD3)  );  ALU\_16bit C2(  .ALU\_Sel(ALU\_Sel),  .A(W\_RD1),  .B(W\_RD2),  .ALU\_Out(ALUout),  .zero(zero)  );  Adder32Bit C4(.input1(PC\_plus\_4),.input2(W\_RD3<<2),.out(beq\_add));  endmodule  module Register\_File\_16bit (clk,read\_addr\_1, read\_addr\_2, write\_addr, write\_data, RegWrite,read\_data\_1, read\_data\_2);  input clk,RegWrite;  input [4:0] read\_addr\_1, read\_addr\_2, write\_addr;  input [15:0] write\_data;  output [15:0] read\_data\_1, read\_data\_2;  reg [15:0] Regfile [31:0];  integer i;  // Initialize registers  initial begin  for (i = 0; i < 32; i = i + 1) begin  Regfile[i] = 16'b0;  end  Regfile[16] = 16'h33; // Register 16 (contains 0x33)  Regfile[17] = 16'h66; // Register 17 (contains 0x66)  Regfile[18] = 16'h2; // Register 16 (contains 0x33)  Regfile[19] = 16'h5; // Register 17 (contains 0x66)  end  // Read Data Logic  assign read\_data\_1 = Regfile[read\_addr\_1];  assign read\_data\_2 = Regfile[read\_addr\_2];  // Write Data Logic  always @(posedge clk) begin  if (RegWrite) begin  Regfile[write\_addr] <= write\_data;  $display("write\_addr=%h write\_data=%h",write\_addr,write\_data);  end  end  endmodule  module ALU\_16bit(ALU\_Sel, A, B, ALU\_Out, zero);  input [3:0] ALU\_Sel; // ALU selection  input [15:0] A; // 16-bit input 1  input [15:0] B; // 16-bit input 2  output reg [15:0] ALU\_Out; // ALU 16-bit output  output reg zero;  parameter ALU\_OP\_ADD = 4'b0000, // Addition  ALU\_OP\_SUB = 4'b0001, // Subtraction  ALU\_OP\_MUL = 4'b0010, // Multiplication  ALU\_OP\_DIV = 4'b0011, // Division  ALU\_OP\_AND = 4'b0100, // Logical and  ALU\_OP\_OR = 4'b0101, // Logical or  ALU\_OP\_XOR = 4'b0110, // Logical xor  ALU\_OP\_NOR = 4'b0111, // Logical nor  ALU\_OP\_NAND = 4'b1000, // Logical nand  ALU\_OP\_XNOR = 4'b1001, // Logical xnor  ALU\_OP\_SHL = 4'b1010, // Logical shift left  ALU\_OP\_SHR = 4'b1011, // Logical shift right  ALU\_OP\_ROL = 4'b1100, // Rotate left  ALU\_OP\_ROR = 4'b1101, // Rotate right  ALU\_OP\_BEQ = 4'b1110, // Equal comparison  ALU\_OP\_BNE = 4'b1111; // Not Equal comparison  always @(\*)  begin  case(ALU\_Sel)  ALU\_OP\_ADD : ALU\_Out = A + B;  ALU\_OP\_SUB : ALU\_Out = A - B;  ALU\_OP\_MUL : ALU\_Out = A \* B;  ALU\_OP\_DIV : ALU\_Out = A / B;  ALU\_OP\_AND : ALU\_Out = A & B;  ALU\_OP\_OR : ALU\_Out = A | B;  ALU\_OP\_XOR : ALU\_Out = A ^ B;  ALU\_OP\_NOR : ALU\_Out = ~(A | B);  ALU\_OP\_NAND : ALU\_Out = ~(A & B);  ALU\_OP\_XNOR : ALU\_Out = ~(A ^ B);  ALU\_OP\_SHL : ALU\_Out = A<<1;  ALU\_OP\_SHR : ALU\_Out = A>>1;  ALU\_OP\_ROL : ALU\_Out = {A[6:0],A[7]};  ALU\_OP\_ROR : ALU\_Out = {A[0],A[7:1]};  ALU\_OP\_BEQ : begin  zero = (A==B)?1'b1:1'b0;  ALU\_Out = (A==B)?16'd1:16'd0;  end  ALU\_OP\_BNE : begin  zero = (A!=B)?1'b1:1'b0;  // zero here is different from the above zero  // zero here is equal to 1, when A is different from B  // zero here is equal to 0, when A is the same as B  ALU\_Out = (A!=B)?16'd1:16'd0;  end  endcase  end  endmodule  // Sign Extension  module Sign\_Extension (sign\_in, sign\_out);  input [7:0] sign\_in;  output [15:0] sign\_out;  assign sign\_out[7:0]=sign\_in[7:0];  assign sign\_out[15:8]=sign\_in[7]?8'b1111\_1111:8'b0;  endmodule  module Adder32Bit(input1, input2, out);  input [15:0] input1, input2;  output [15:0] out;  reg [15:0]out;  always@( input1 or input2)  begin  out <= input1 + input2;  end  endmodule |
| **Testbench** |
| module tb\_beq\_Datapath;  // Inputs  reg clk;  reg [4:0] rs, rt;  reg [7:0] offset;  reg [15:0] PC\_plus\_4;  reg [3:0] ALU\_Sel;  // Outputs  wire [15:0] beq\_add;  wire zero;  // Instantiate the Unit Under Test (UUT)  beq\_Datapath uut (  .clk(clk),  .rs(rs),  .rt(rt),  .offset(offset),  .PC\_plus\_4(PC\_plus\_4),  .ALU\_Sel(ALU\_Sel),  .beq\_add(beq\_add),  .zero(zero)  );  // Clock generation  always #5 clk = ~clk;  initial begin  // Initialize Inputs  clk = 0;  rs = 5'h11; // Register 17  rt = 5'h10; // Register 16  offset = 8'h04; // Offset  PC\_plus\_4 = 16'h0010; // Initial PC + 4 value  ALU\_Sel = 4'b1110; // BEQ operation  // Display header  $display("Time\tPC\_plus\_4\tBEQ\_Address\tZero");  // Monitor signals  $monitor("%0d\t%h\t%h\t%b", $time, PC\_plus\_4, beq\_add, zero);  // Test Case 1: rs != rt  #10;  rs = 5'h18; // Register 24  rt = 5'h19; // Register 25  // Test Case 2: rs == rt  #10;  rs = 5'h11; // Register 17  rt = 5'h11; // Register 17  // Test Case 3: Change offset  #10;  offset = 8'hFF; // Negative offset  #10;  $finish;  end  endmodule |
|  |
|  |

**III.6 EXPERIMENT NO. 6**

**III.6.1 AIM: To implement R-Type Processor**

****

****

**III.6.2 CODE**

module R\_Type\_Proc(reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout);

// Your code here

Endmodule

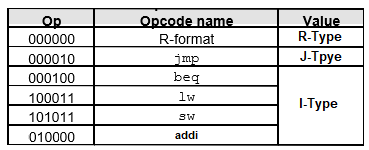
**III.6.3 LAB ASSIGNMENT**

|  |
| --- |
| **Top module** |
| module lab6\_ex5(SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7);  input [17:0]SW;  output [17:0]LEDR;  output [7:0]LEDG;  output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;  wire [31:0] W\_PC\_out,W\_LED\_SEG, W\_m2;  wire [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;    assign LEDR=SW;  hex\_ssd (W\_RD1 [3:0], HEX0);  hex\_ssd (W\_RD1[7:4], HEX1);  hex\_ssd ( W\_RD2[3:0],HEX2);  hex\_ssd ( W\_RD2[7:4],HEX3);  hex\_ssd (W\_PC\_out[3:0],HEX4);  hex\_ssd (W\_PC\_out[7:4],HEX5);  hex\_ssd (W\_ALUout[3:0],HEX6);  hex\_ssd (W\_ALUout[7:4],HEX7);  R\_Type\_Proc(.reset(SW[0]),.clk(SW[1]),.W\_PC\_out(W\_PC\_out),.instruction(W\_LED\_SEG),.W\_RD1(W\_RD1),.W\_RD2(W\_RD2),.W\_m1(W\_m1),.W\_m2(W\_m2),.W\_ALUout(W\_ALUout));  endmodule  module hex\_ssd (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 |
| **Main module** |
| module R\_Type\_Proc(reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout);  input reset, clk;  output [31:0] W\_PC\_out, instruction, W\_m2;  output [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;  wire [31:0] W\_PC\_in,W\_PC\_plus\_1,W\_Branch\_add;  wire PCsrc,RegWrite,zero,ALUsrc,MemRead,MemWrite,MemtoReg,RegDst,Branch,jump;  wire [15:0] W\_MemtoReg,W\_RD1,W\_RD2,W\_ALUout,W\_m1,W\_RDm;  wire [3:0] ALUop;  wire [4:0] W\_m3;      Program\_Counter C1(.clk(clk),.reset(reset),.PC\_in( W\_PC\_plus\_1),.PC\_out(W\_PC\_out));  Adder32Bit C2(.input1(W\_PC\_out),.input2(32'd1),.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\_16bit C6 (.clk(clk),.read\_addr\_1(instruction[25:21]),.read\_addr\_2(instruction[20:16]),.write\_addr(W\_m3),.write\_data(W\_MemtoReg),.RegWrite(RegWrite),.read\_data\_1(W\_RD1),.read\_data\_2(W\_RD2));  Sign\_Extension C7(.sign\_in(instruction[15:0]),.sign\_out(W\_Branch\_add));  Mux\_16\_bit C8(.in0(W\_RD2),.in1(instruction[15:0]),.mux\_out(W\_m1),.select(ALUsrc));  ALU\_16bit C9(.ALU\_Sel(ALUop),.A(W\_RD1),.B(W\_m1),.ALU\_Out(W\_ALUout),.zero(zero));  Data\_Memory\_16bit C10(.clk(clk),.addr(W\_ALUout),.write\_data(W\_RD2),.read\_data(W\_RDm),.MemRead(MemRead),.MemWrite(MemWrite));  Mux\_16\_bit C11(.in0(W\_ALUout),.in1(W\_RDm),.mux\_out(W\_MemtoReg),.select(MemtoReg));  Control C12(.clk(clk),.Op\_intstruct(instruction[31:26]),.ints\_function(instruction[5:0]), .RegDst(RegDst),.Branch(Branch),.MemRead(MemRead),.MemtoReg(MemtoReg),.ALUOp(ALUop),.MemWrite(MemWrite),.ALUSrc(ALUsrc),.RegWrite(RegWrite),.Zero(zero),.Jump(jump));  Mux\_5\_bit C13(.in0(instruction[20:16]),.in1(instruction[15:11]),.mux\_out(W\_m3),.select(RegDst));  endmodule  module Control(clk, Op\_intstruct, ints\_function, RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite, Zero, Jump);  input clk,Zero;  input [5:0] ints\_function;  input [5:0] Op\_intstruct;  output reg RegDst,Branch,MemRead,MemtoReg,Jump;  output reg [3:0] ALUOp;  output reg MemWrite,ALUSrc,RegWrite;  always @(\*)  begin  case (Op\_intstruct)  6'b000000: begin // R-Type Instruction  RegDst = 1;  Jump = 0;  Branch = 0;  MemRead = 0;  MemtoReg = 0;  MemWrite = 0;  ALUSrc = 0;  RegWrite = 1;  case (ints\_function)  6'b100000: ALUOp = 4'b0000; // Addition  6'b100010: ALUOp = 4'b0001; // Subtraction  6'b011000: ALUOp = 4'b0010; // Multiplication  6'b011010: ALUOp = 4'b0011; // Division  6'b100100: ALUOp = 4'b0100; // Logical AND  6'b100101: ALUOp = 4'b0101; // Logical OR  6'b100110: ALUOp = 4'b0110; // Logical XOR  6'b100111: ALUOp = 4'b0111; // Logical NOR  6'b101000: ALUOp = 4'b1000; // Logical NAND  6'b101010: ALUOp = 4'b1001; // Logical XNOR  6'b000000: ALUOp = 4'b1010; // Logical Shift Left  6'b000010: ALUOp = 4'b1011; // Logical Shift Right  6'b111000: ALUOp = 4'b1100; // Rotate Left  6'b110000: ALUOp = 4'b1101; // Rotate Right  default: ALUOp = 4'b0000;  endcase  end  endcase  end  endmodule    module Register\_File\_16bit (clk,read\_addr\_1, read\_addr\_2, write\_addr, write\_data, RegWrite,read\_data\_1, read\_data\_2);  input clk,RegWrite;  input [4:0] read\_addr\_1, read\_addr\_2, write\_addr;  input [15:0] write\_data;  output [15:0] read\_data\_1, read\_data\_2;  reg [15:0] Regfile [31:0];  integer i;  // Initialize registers  initial begin  for (i = 0; i < 32; i = i + 1) begin  Regfile[i] = 16'b0;  end  Regfile[19]=1; // $s3 = 1  Regfile[17]=3; // $s1 = 3  Regfile[20]=2; // $s4 = 2  Regfile[21]=5; // $s5 = 5  end  // Read Data Logic  assign read\_data\_1 = Regfile[read\_addr\_1];  assign read\_data\_2 = Regfile[read\_addr\_2];  // Write Data Logic  always @(posedge clk) begin  if (RegWrite) begin  Regfile[write\_addr] <= write\_data;  //$display("write\_addr=%h write\_data=%h",write\_addr,write\_data);  end  end  endmodule  module ALU\_16bit(ALU\_Sel, A, B, ALU\_Out, zero);  input [3:0] ALU\_Sel; // ALU selection  input [15:0] A; // 16-bit input 1  input [15:0] B; // 16-bit input 2  output reg [15:0] ALU\_Out; // ALU 16-bit output  output reg zero;  parameter ALU\_OP\_ADD = 4'b0000, // Addition  ALU\_OP\_SUB = 4'b0001, // Subtraction  ALU\_OP\_MUL = 4'b0010, // Multiplication  ALU\_OP\_DIV = 4'b0011, // Division  ALU\_OP\_AND = 4'b0100, // Logical and  ALU\_OP\_OR = 4'b0101, // Logical or  ALU\_OP\_XOR = 4'b0110, // Logical xor  ALU\_OP\_NOR = 4'b0111, // Logical nor  ALU\_OP\_NAND = 4'b1000, // Logical nand  ALU\_OP\_XNOR = 4'b1001, // Logical xnor  ALU\_OP\_SHL = 4'b1010, // Logical shift left  ALU\_OP\_SHR = 4'b1011, // Logical shift right  ALU\_OP\_ROL = 4'b1100, // Rotate left  ALU\_OP\_ROR = 4'b1101, // Rotate right  ALU\_OP\_BEQ = 4'b1110, // Equal comparison  ALU\_OP\_BNE = 4'b1111; // Not Equal comparison  always @(\*)  begin  case(ALU\_Sel)  ALU\_OP\_ADD : ALU\_Out = A + B;  ALU\_OP\_SUB : ALU\_Out = A - B;  ALU\_OP\_MUL : ALU\_Out = A \* B;  ALU\_OP\_DIV : ALU\_Out = A / B;  ALU\_OP\_AND : ALU\_Out = A & B;  ALU\_OP\_OR : ALU\_Out = A | B;  ALU\_OP\_XOR : ALU\_Out = A ^ B;  ALU\_OP\_NOR : ALU\_Out = ~(A | B);  ALU\_OP\_NAND : ALU\_Out = ~(A & B);  ALU\_OP\_XNOR : ALU\_Out = ~(A ^ B);  ALU\_OP\_SHL : ALU\_Out = A<<1;  ALU\_OP\_SHR : ALU\_Out = A>>1;  ALU\_OP\_ROL : ALU\_Out = {A[6:0],A[7]};  ALU\_OP\_ROR : ALU\_Out = {A[0],A[7:1]};  ALU\_OP\_BEQ : begin  zero = (A==B)?1'b1:1'b0;  ALU\_Out = (A==B)?16'd1:16'd0;  end  ALU\_OP\_BNE : begin  zero = (A!=B)?1'b1:1'b0;  // zero here is different from the above zero  // zero here is equal to 1, when A is different from B  // zero here is equal to 0, when A is the same as B  ALU\_Out = (A!=B)?16'd1:16'd0;  end  endcase  end  endmodule  // Sign Extension  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 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 Program\_Counter(clk, reset, PC\_in, PC\_out);  input clk, reset;  input [31:0] PC\_in;  output reg [31:0] PC\_out;  initial begin  PC\_out = 32'b0;  end    always @(posedge clk) begin  if(reset)  PC\_out <= 32'b0;  else  PC\_out <= PC\_in;  end  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\_16\_bit (in0, in1, mux\_out, select);  parameter N=16;  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 Data\_Memory\_16bit (clk,addr,write\_data,read\_data,MemRead,MemWrite);  input clk;  input [15:0] addr;  input [15:0] write\_data;  output reg [15:0] read\_data;  input MemRead;  input MemWrite;  integer i;  reg [15:0] DMemory [0:255];  // Initialize memory for simulation  initial begin    for (i = 0; i < 256; i = i + 1) begin  DMemory[i] = 16'h0000;  end  end  // Synchronous write  always @(posedge clk) begin  if (MemWrite) begin  DMemory[addr] <= write\_data;  $display("Memory Write: Address=%h Data=%h", addr, write\_data);  end  end  // Asynchronous read  always @(\*) begin  if (MemRead)  read\_data = DMemory[addr];  else  read\_data = 16'h0000;  end  endmodule  module Instruction\_Memory (read\_address, instruction, reset);  input reset;  input [31:0] read\_address;  output [31:0] instruction;  reg [31:0] Imemory [256:0];  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=0; k<32; k=k+1)  begin  // here Out changes k=0 to k=16  Imemory[k] = 32'b0;  end  /\*Test R-type\*/  // add $s6, $s5, $s4 R22 = R21 + R20 = 0x7 (0x5 + 0x2)  Imemory[0] = 32'b000000\_10101\_10100\_10110\_00000\_100000;  // sub $s6, $s5, $s4 R22 = R21 - R20 = 0x3 (0x5 - 0x2)  Imemory[1] = 32'b000000\_10101\_10100\_10110\_00000\_100010;  // mult $s5, $s4 R21 \* R20 = 0xA (0x5 \* 0x2)  Imemory[2] = 32'b000000\_10101\_10100\_00000\_00000\_011000;  // div $s5, $s4 R21 / R20 = 0x2 (0x5 / 0x2)  Imemory[3] = 32'b000000\_10101\_10100\_00000\_00000\_011010;  // and $s6, $s5, $s4 R22 = AND(R21, R20) = 0x0 (0x5 AND 0x2)  Imemory[4] = 32'b000000\_10101\_10100\_10110\_00000\_100100;  // or $s6, $s5, $s4 R22 = OR(R21, R20) = 0x7 (0x5 OR 0x2)  Imemory[5] = 32'b000000\_10101\_10100\_10110\_00000\_100101;  // xor $s6, $s5, $s4 R22 = XOR(R21, R20) = 0x7 (0x5 XOR 0x2)  Imemory[6] = 32'b000000\_10101\_10100\_10110\_00000\_100110;  // nor $s6, $s5, $s4 R22 = NOR(R21, R20) = 0xFFFFFFF8 (NOR of 0x5 and 0x2)  Imemory[7] = 32'b000000\_10101\_10100\_10110\_00000\_100111;  // nand $s6, $s5, $s4 R22 = NAND(R21, R20) = 0xFFFFFFFD (NAND of 0x5 and 0x2)  Imemory[8] = 32'b000000\_10101\_10100\_10110\_00000\_101000;  // xnor $s6, $s5, $s4 R22 = XNOR(R21, R20) = 0xFFFFFFFE (XNOR of 0x5 and 0x2)  Imemory[9] = 32'b000000\_10101\_10100\_10110\_00000\_101010;  // sll $s6, $s5, $s4 R22 = R21 << R20 = 0x14 (0x5 << 0x2)  Imemory[10] = 32'b000000\_10101\_10100\_10110\_00000\_000000;  // srl $s6, $s5, $s4 R22 = R21 >> R20 = 0x1 (0x5 >> 0x2)  Imemory[11] = 32'b000000\_10101\_10100\_10110\_00000\_000010;  // rol $s6, $s5 R22 = R21 rotated 1 bit left = 0xA (rotate 0x5 left by 1)  Imemory[12] = 32'b000000\_10101\_00000\_10110\_00000\_111000;  // ror $s6, $s5 R22 = R21 rotated 1 bit right = 0x2 (rotate 0x5 right by 1)  Imemory[13] = 32'b000000\_10101\_00000\_10110\_00000\_110000;  end  endmodule |
| **Testbench** |
| module tb\_R\_Type\_Proc;  reg reset, clk;  wire [31:0] W\_PC\_out, instruction, W\_m2;  wire [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;  reg [127:0] instr\_name;  R\_Type\_Proc dut(reset, clk, W\_PC\_out, instruction, W\_RD1, W\_RD2, W\_m1, W\_m2, W\_ALUout);  // Clock generation  always #5 clk = ~clk;  // Task to decode instruction name  task decode\_instruction;  input [31:0] instr;  output reg [127:0] name; // String for instruction name  begin  if (instr[31:26] == 6'b000000) begin // R-Type instructions  case (instr[5:0]) // funct field  6'b100000: name = "Addition";  6'b100010: name = "Subtraction";  6'b011000: name = "Multiplication";  6'b011010: name = "Division";  6'b100100: name = "Logical AND";  6'b100101: name = "Logical OR";  6'b100110: name = "Logical XOR";  6'b100111: name = "Logical NOR";  6'b101000: name = "Logical NAND";  6'b101010: name = "Logical XNOR";  6'b000000: name = "Shift Left";  6'b000010: name = "Shift Right";  6'b111000: name = "Rotate Left";  6'b110000: name = "Rotate Right";  default: name = "Unknown Instruction";  endcase  end else begin  name = "Not R-Type Instruction";  end  end  endtask  initial begin    clk = 0;  reset = 1;  $display(" ----------TEST R-TYPE----------");  #10 reset = 0;$monitor($time,  " reset=%b | clk=%b | W\_PC\_out=%4d | Instruction=%s | Opcode=%6b | rs=%d | rt=%d | rd=%d | Imm=%h | W\_ALUout=%h | W\_RD1=%h | W\_RD2=%h",  reset, clk, W\_PC\_out, instr\_name, instruction[31:26], instruction[25:21], instruction[20:16], instruction[15:11],  instruction[15:0], W\_ALUout, W\_RD1, W\_RD2);  // Simulate decoding each instruction during runtime for a limited duration  repeat (13) begin  decode\_instruction(instruction, instr\_name);  #10; // Wait for the next clock cycle  end  $finish;  end  endmodule |
|  |
|  |

3) Write the assembly code with Addittion, Substraction instructions, compile into binary macchinecode and test the operation of the designed R\_Type\_Processor processor

**III.7 EXPERIMENT NO. 7**

**III.7.1 AIM: To implement LW I-Type Processor**

****

**bebeq**

**III.7.2 CODE**

module LW\_processor (reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout );

//Your Verilog code here

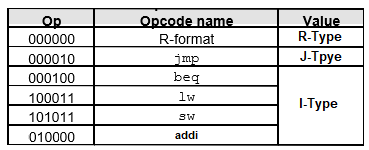
endmodule

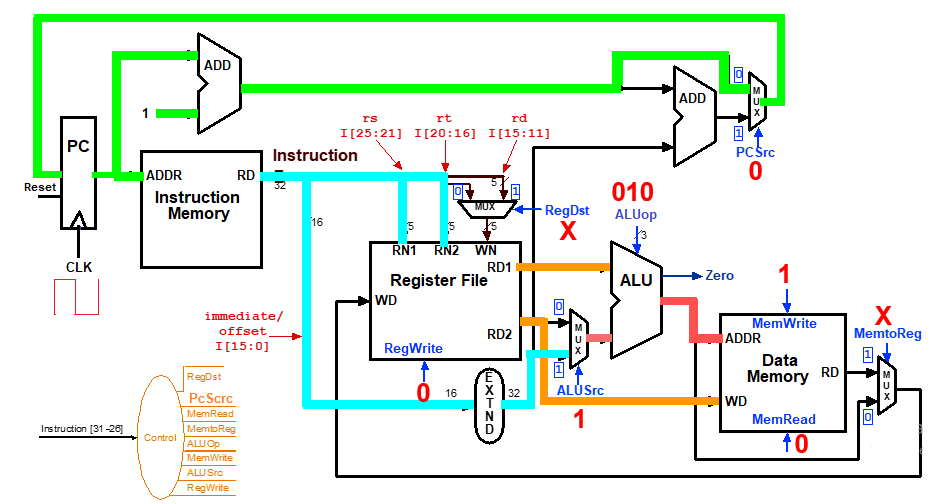
**III.7.3 LAB ASSIGNMENT**

|  |
| --- |
| **Top module** |
| module lab6\_ex6(SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7);  input [17:0]SW;  output [17:0]LEDR;  output [7:0]LEDG;  output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;  wire [31:0] W\_PC\_out,W\_LED\_SEG, W\_m2;  wire [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;    assign LEDR=SW;  hex\_ssd (W\_RD1 [3:0], HEX0);  hex\_ssd (W\_RD1[7:4], HEX1);  hex\_ssd ( W\_RD2[3:0],HEX2);  hex\_ssd ( W\_RD2[7:4],HEX3);  hex\_ssd (W\_PC\_out[3:0],HEX4);  hex\_ssd (W\_PC\_out[7:4],HEX5);  hex\_ssd (W\_ALUout[3:0],HEX6);  hex\_ssd (W\_ALUout[7:4],HEX7);  LW\_processor (.reset(SW[0]),.clk(SW[1]),.W\_PC\_out(W\_PC\_out),.instruction(W\_LED\_SEG),.W\_RD1(W\_RD1),.W\_RD2(W\_RD2),.W\_m1(W\_m1),.W\_m2(W\_m2),.W\_ALUout(W\_ALUout));  endmodule  module hex\_ssd (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 |
| **Main module** |
| module LW\_processor(reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout);  input reset, clk;  output [31:0] W\_PC\_out, instruction, W\_m2;  output [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;  wire [31:0] W\_PC\_in,W\_PC\_plus\_1,W\_Branch\_add;  wire PCsrc,RegWrite,zero,ALUsrc,MemRead,MemWrite,MemtoReg,RegDst,Branch,jump;  wire [15:0] W\_MemtoReg,W\_RD1,W\_RD2,W\_ALUout,W\_m1,W\_RDm;  wire [3:0] ALUop;  wire [4:0] W\_m3;      Program\_Counter C1(.clk(clk),.reset(reset),.PC\_in( W\_PC\_plus\_1),.PC\_out(W\_PC\_out));  Adder32Bit C2(.input1(W\_PC\_out),.input2(32'd1),.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\_16bit C6 (.clk(clk),.read\_addr\_1(instruction[25:21]),.read\_addr\_2(instruction[20:16]),.write\_addr(W\_m3),.write\_data(W\_MemtoReg),.RegWrite(RegWrite),.read\_data\_1(W\_RD1),.read\_data\_2(W\_RD2));  Sign\_Extension C7(.sign\_in(instruction[15:0]),.sign\_out(W\_Branch\_add));  Mux\_16\_bit C8(.in0(W\_RD2),.in1(instruction[15:0]),.mux\_out(W\_m1),.select(ALUsrc));  ALU\_16bit C9(.ALU\_Sel(ALUop),.A(W\_RD1),.B(W\_m1),.ALU\_Out(W\_ALUout),.zero(zero));  Data\_Memory\_16bit C10(.clk(clk),.addr(W\_ALUout),.write\_data(W\_RD2),.read\_data(W\_RDm),.MemRead(MemRead),.MemWrite(MemWrite));  Mux\_16\_bit C11(.in0(W\_ALUout),.in1(W\_RDm),.mux\_out(W\_MemtoReg),.select(MemtoReg));  Control C12(.clk(clk),.Op\_intstruct(instruction[31:26]),.ints\_function(instruction[5:0]), .RegDst(RegDst),.Branch(Branch),.MemRead(MemRead),.MemtoReg(MemtoReg),.ALUOp(ALUop),.MemWrite(MemWrite),.ALUSrc(ALUsrc),.RegWrite(RegWrite),.Zero(zero),.Jump(jump));  Mux\_5\_bit C13(.in0(instruction[20:16]),.in1(instruction[15:11]),.mux\_out(W\_m3),.select(RegDst));  endmodule  module Control(clk, Op\_intstruct, ints\_function, RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite, Zero, Jump);  input clk,Zero;  input [5:0] ints\_function;  input [5:0] Op\_intstruct;  output reg RegDst,Branch,MemRead,MemtoReg,Jump;  output reg [3:0] ALUOp;  output reg MemWrite,ALUSrc,RegWrite;  always @(\*)  begin  case (Op\_intstruct)  6'b100011: begin // LW  RegDst = 0;  Jump = 0;  Branch = 0;  MemRead = 1;  MemtoReg = 1;  ALUOp = 4'b0000;  MemWrite = 0;  ALUSrc = 1;  RegWrite = 1;  end  endcase  end  endmodule    module Register\_File\_16bit (clk,read\_addr\_1, read\_addr\_2, write\_addr, write\_data, RegWrite,read\_data\_1, read\_data\_2);  input clk,RegWrite;  input [4:0] read\_addr\_1, read\_addr\_2, write\_addr;  input [15:0] write\_data;  output [15:0] read\_data\_1, read\_data\_2;  reg [15:0] Regfile [31:0];  integer i;  // Initialize registers  initial begin  for (i = 0; i < 32; i = i + 1) begin  Regfile[i] = 16'b0;  end  Regfile[19]=1; // $s3 = 1  Regfile[17]=3; // $s1 = 3  Regfile[20]=2; // $s4 = 2  Regfile[21]=5; // $s5 = 5  end  // Read Data Logic  assign read\_data\_1 = Regfile[read\_addr\_1];  assign read\_data\_2 = Regfile[read\_addr\_2];  // Write Data Logic  always @(posedge clk) begin  if (RegWrite) begin  Regfile[write\_addr] <= write\_data;  //$display("write\_addr=%h write\_data=%h",write\_addr,write\_data);  end  end  endmodule  module ALU\_16bit(ALU\_Sel, A, B, ALU\_Out, zero);  input [3:0] ALU\_Sel; // ALU selection  input [15:0] A; // 16-bit input 1  input [15:0] B; // 16-bit input 2  output reg [15:0] ALU\_Out; // ALU 16-bit output  output reg zero;  parameter ALU\_OP\_ADD = 4'b0000, // Addition  ALU\_OP\_SUB = 4'b0001, // Subtraction  ALU\_OP\_MUL = 4'b0010, // Multiplication  ALU\_OP\_DIV = 4'b0011, // Division  ALU\_OP\_AND = 4'b0100, // Logical and  ALU\_OP\_OR = 4'b0101, // Logical or  ALU\_OP\_XOR = 4'b0110, // Logical xor  ALU\_OP\_NOR = 4'b0111, // Logical nor  ALU\_OP\_NAND = 4'b1000, // Logical nand  ALU\_OP\_XNOR = 4'b1001, // Logical xnor  ALU\_OP\_SHL = 4'b1010, // Logical shift left  ALU\_OP\_SHR = 4'b1011, // Logical shift right  ALU\_OP\_ROL = 4'b1100, // Rotate left  ALU\_OP\_ROR = 4'b1101, // Rotate right  ALU\_OP\_BEQ = 4'b1110, // Equal comparison  ALU\_OP\_BNE = 4'b1111; // Not Equal comparison  always @(\*)  begin  case(ALU\_Sel)  ALU\_OP\_ADD : ALU\_Out = A + B;  ALU\_OP\_SUB : ALU\_Out = A - B;  ALU\_OP\_MUL : ALU\_Out = A \* B;  ALU\_OP\_DIV : ALU\_Out = A / B;  ALU\_OP\_AND : ALU\_Out = A & B;  ALU\_OP\_OR : ALU\_Out = A | B;  ALU\_OP\_XOR : ALU\_Out = A ^ B;  ALU\_OP\_NOR : ALU\_Out = ~(A | B);  ALU\_OP\_NAND : ALU\_Out = ~(A & B);  ALU\_OP\_XNOR : ALU\_Out = ~(A ^ B);  ALU\_OP\_SHL : ALU\_Out = A<<1;  ALU\_OP\_SHR : ALU\_Out = A>>1;  ALU\_OP\_ROL : ALU\_Out = {A[6:0],A[7]};  ALU\_OP\_ROR : ALU\_Out = {A[0],A[7:1]};  ALU\_OP\_BEQ : begin  zero = (A==B)?1'b1:1'b0;  ALU\_Out = (A==B)?16'd1:16'd0;  end  ALU\_OP\_BNE : begin  zero = (A!=B)?1'b1:1'b0;  // zero here is different from the above zero  // zero here is equal to 1, when A is different from B  // zero here is equal to 0, when A is the same as B  ALU\_Out = (A!=B)?16'd1:16'd0;  end  endcase  end  endmodule  // Sign Extension  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 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 Program\_Counter(clk, reset, PC\_in, PC\_out);  input clk, reset;  input [31:0] PC\_in;  output reg [31:0] PC\_out;  initial begin  PC\_out = 32'b0;  end    always @(posedge clk) begin  if(reset)  PC\_out <= 32'b0;  else  PC\_out <= PC\_in;  end  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\_16\_bit (in0, in1, mux\_out, select);  parameter N=16;  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 Data\_Memory\_16bit (clk,addr,write\_data,read\_data,MemRead,MemWrite);  input clk;  input [15:0] addr;  input [15:0] write\_data;  output reg [15:0] read\_data;  input MemRead;  input MemWrite;  integer i;  reg [15:0] DMemory [0:255];  // Initialize memory for simulation  initial begin    for (i = 0; i < 256; i = i + 1) begin  DMemory[i] = 16'h0000;  end  end  // Synchronous write  always @(posedge clk) begin  if (MemWrite) begin  DMemory[addr] <= write\_data;  $display("Memory Write: Address=%h Data=%h", addr, write\_data);  end  end  // Asynchronous read  always @(\*) begin  if (MemRead)  read\_data = DMemory[addr];  else  read\_data = 16'h0000;  end  endmodule  module Instruction\_Memory (read\_address, instruction, reset);  input reset;  input [31:0] read\_address;  output [31:0] instruction;  reg [31:0] Imemory [256:0];  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=0; k<32; k=k+1)  begin  // here Out changes k=0 to k=16  Imemory[k] = 32'b0;  end  /\*Test LW-type\*/  Imemory[0] = 32'b100011\_00000\_10011\_0000000000000000; // lw $s3, 0($zero) -> 0x10030000  Imemory[1] = 32'b100011\_00000\_10001\_0000000000000100; // lw $s1, 4($zero) -> 0x10010004  Imemory[2] = 32'b100011\_00000\_10100\_0000000000001000; // lw $s4, 8($zero) -> 0x10020008  Imemory[3] = 32'b100011\_00000\_10101\_0000000000001100; // lw $s5, 12($zero) -> 0x1002500C  end  endmodule |
| **Testbench** |
| module tb\_LW\_Proc;  reg reset, clk;  wire [31:0] W\_PC\_out, instruction, W\_m2;  wire [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;    LW\_processor dut(reset, clk, W\_PC\_out, instruction, W\_RD1, W\_RD2, W\_m1, W\_m2, W\_ALUout);  // Clock generation  always #5 clk = ~clk;  initial begin  $monitor($time,  " reset=%b | clk=%b | W\_PC\_out=%4d | Opcode=%6b | rs=%d | rt=%d | Imm=%h | W\_ALUout=%h | W\_RD1=%h | W\_RD2=%h",  reset, clk, W\_PC\_out, instruction[31:26], instruction[25:21], instruction[20:16],  instruction[15:0], W\_ALUout, W\_RD1, W\_RD2);  clk = 0;  reset = 1;  $display(" ----------TEST LW-PROCESSOR----------");  #10 reset = 0;  #30 $finish;  end  endmodule |
|  |
|  |

**III.8 EXPERIMENT NO. 8**

**III.8.1 AIM: To implement SW I-Type Processor**

****

****

**III.8.2 CODE**

module SW\_Processor ( reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout );

//Your Verilog code here

endmodule

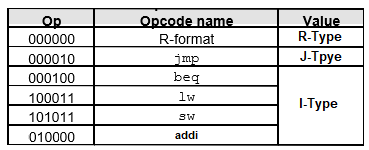
**III.8.3 LAB ASSIGNMENT**

|  |
| --- |
| **Top module** |
| module lab6\_ex7(SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7);  input [17:0]SW;  output [17:0]LEDR;  output [7:0]LEDG;  output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;  wire [31:0] W\_PC\_out,W\_LED\_SEG, W\_m2;  wire [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;    assign LEDR=SW;  hex\_ssd (W\_RD1 [3:0], HEX0);  hex\_ssd (W\_RD1[7:4], HEX1);  hex\_ssd ( W\_RD2[3:0],HEX2);  hex\_ssd ( W\_RD2[7:4],HEX3);  hex\_ssd (W\_PC\_out[3:0],HEX4);  hex\_ssd (W\_PC\_out[7:4],HEX5);  hex\_ssd (W\_ALUout[3:0],HEX6);  hex\_ssd (W\_ALUout[7:4],HEX7);  SW\_processor (.reset(SW[0]),.clk(SW[1]),.W\_PC\_out(W\_PC\_out),.instruction(W\_LED\_SEG),.W\_RD1(W\_RD1),.W\_RD2(W\_RD2),.W\_m1(W\_m1),.W\_m2(W\_m2),.W\_ALUout(W\_ALUout));  endmodule  module hex\_ssd (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 |
| **Main module** |
| module SW\_processor(reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout);  input reset, clk;  output [31:0] W\_PC\_out, instruction, W\_m2;  output [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;  wire [31:0] W\_PC\_in,W\_PC\_plus\_1,W\_Branch\_add;  wire PCsrc,RegWrite,zero,ALUsrc,MemRead,MemWrite,MemtoReg,RegDst,Branch,jump;  wire [15:0] W\_MemtoReg,W\_RD1,W\_RD2,W\_ALUout,W\_m1,W\_RDm;  wire [3:0] ALUop;  wire [4:0] W\_m3;      Program\_Counter C1(.clk(clk),.reset(reset),.PC\_in( W\_PC\_plus\_1),.PC\_out(W\_PC\_out));  Adder32Bit C2(.input1(W\_PC\_out),.input2(32'd1),.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\_16bit C6 (.clk(clk),.read\_addr\_1(instruction[25:21]),.read\_addr\_2(instruction[20:16]),.write\_addr(W\_m3),.write\_data(W\_MemtoReg),.RegWrite(RegWrite),.read\_data\_1(W\_RD1),.read\_data\_2(W\_RD2));  Sign\_Extension C7(.sign\_in(instruction[15:0]),.sign\_out(W\_Branch\_add));  Mux\_16\_bit C8(.in0(W\_RD2),.in1(instruction[15:0]),.mux\_out(W\_m1),.select(ALUsrc));  ALU\_16bit C9(.ALU\_Sel(ALUop),.A(W\_RD1),.B(W\_m1),.ALU\_Out(W\_ALUout),.zero(zero));  Data\_Memory\_16bit C10(.clk(clk),.addr(W\_ALUout),.write\_data(W\_RD2),.read\_data(W\_RDm),.MemRead(MemRead),.MemWrite(MemWrite));  Mux\_16\_bit C11(.in0(W\_ALUout),.in1(W\_RDm),.mux\_out(W\_MemtoReg),.select(MemtoReg));  Control C12(.clk(clk),.Op\_intstruct(instruction[31:26]),.ints\_function(instruction[5:0]), .RegDst(RegDst),.Branch(Branch),.MemRead(MemRead),.MemtoReg(MemtoReg),.ALUOp(ALUop),.MemWrite(MemWrite),.ALUSrc(ALUsrc),.RegWrite(RegWrite),.Zero(zero),.Jump(jump));  Mux\_5\_bit C13(.in0(instruction[20:16]),.in1(instruction[15:11]),.mux\_out(W\_m3),.select(RegDst));  endmodule  module Control(clk, Op\_intstruct, ints\_function, RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite, Zero, Jump);  input clk,Zero;  input [5:0] ints\_function;  input [5:0] Op\_intstruct;  output reg RegDst,Branch,MemRead,MemtoReg,Jump;  output reg [3:0] ALUOp;  output reg MemWrite,ALUSrc,RegWrite;  always @(\*)  begin  case (Op\_intstruct)  6'b101011: begin // SW  RegDst = 0;  Jump = 0;  Branch = 0;  MemRead = 0;  MemtoReg = 0;  ALUOp = 4'b0000;  MemWrite = 1;  ALUSrc = 1;  RegWrite = 0;  end  endcase  end  endmodule    module Register\_File\_16bit (clk,read\_addr\_1, read\_addr\_2, write\_addr, write\_data, RegWrite,read\_data\_1, read\_data\_2);  input clk,RegWrite;  input [4:0] read\_addr\_1, read\_addr\_2, write\_addr;  input [15:0] write\_data;  output [15:0] read\_data\_1, read\_data\_2;  reg [15:0] Regfile [31:0];  integer i;  // Initialize registers  initial begin  for (i = 0; i < 32; i = i + 1) begin  Regfile[i] = 16'b0;  end  Regfile[19]=1; // $s3 = 1  Regfile[17]=3; // $s1 = 3  Regfile[20]=2; // $s4 = 2  Regfile[21]=5; // $s5 = 5  end  // Read Data Logic  assign read\_data\_1 = Regfile[read\_addr\_1];  assign read\_data\_2 = Regfile[read\_addr\_2];  // Write Data Logic  always @(posedge clk) begin  if (RegWrite) begin  Regfile[write\_addr] <= write\_data;  //$display("write\_addr=%h write\_data=%h",write\_addr,write\_data);  end  end  endmodule  module ALU\_16bit(ALU\_Sel, A, B, ALU\_Out, zero);  input [3:0] ALU\_Sel; // ALU selection  input [15:0] A; // 16-bit input 1  input [15:0] B; // 16-bit input 2  output reg [15:0] ALU\_Out; // ALU 16-bit output  output reg zero;  parameter ALU\_OP\_ADD = 4'b0000, // Addition  ALU\_OP\_SUB = 4'b0001, // Subtraction  ALU\_OP\_MUL = 4'b0010, // Multiplication  ALU\_OP\_DIV = 4'b0011, // Division  ALU\_OP\_AND = 4'b0100, // Logical and  ALU\_OP\_OR = 4'b0101, // Logical or  ALU\_OP\_XOR = 4'b0110, // Logical xor  ALU\_OP\_NOR = 4'b0111, // Logical nor  ALU\_OP\_NAND = 4'b1000, // Logical nand  ALU\_OP\_XNOR = 4'b1001, // Logical xnor  ALU\_OP\_SHL = 4'b1010, // Logical shift left  ALU\_OP\_SHR = 4'b1011, // Logical shift right  ALU\_OP\_ROL = 4'b1100, // Rotate left  ALU\_OP\_ROR = 4'b1101, // Rotate right  ALU\_OP\_BEQ = 4'b1110, // Equal comparison  ALU\_OP\_BNE = 4'b1111; // Not Equal comparison  always @(\*)  begin  case(ALU\_Sel)  ALU\_OP\_ADD : ALU\_Out = A + B;  ALU\_OP\_SUB : ALU\_Out = A - B;  ALU\_OP\_MUL : ALU\_Out = A \* B;  ALU\_OP\_DIV : ALU\_Out = A / B;  ALU\_OP\_AND : ALU\_Out = A & B;  ALU\_OP\_OR : ALU\_Out = A | B;  ALU\_OP\_XOR : ALU\_Out = A ^ B;  ALU\_OP\_NOR : ALU\_Out = ~(A | B);  ALU\_OP\_NAND : ALU\_Out = ~(A & B);  ALU\_OP\_XNOR : ALU\_Out = ~(A ^ B);  ALU\_OP\_SHL : ALU\_Out = A<<1;  ALU\_OP\_SHR : ALU\_Out = A>>1;  ALU\_OP\_ROL : ALU\_Out = {A[6:0],A[7]};  ALU\_OP\_ROR : ALU\_Out = {A[0],A[7:1]};  ALU\_OP\_BEQ : begin  zero = (A==B)?1'b1:1'b0;  ALU\_Out = (A==B)?16'd1:16'd0;  end  ALU\_OP\_BNE : begin  zero = (A!=B)?1'b1:1'b0;  // zero here is different from the above zero  // zero here is equal to 1, when A is different from B  // zero here is equal to 0, when A is the same as B  ALU\_Out = (A!=B)?16'd1:16'd0;  end  endcase  end  endmodule  // Sign Extension  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 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 Program\_Counter(clk, reset, PC\_in, PC\_out);  input clk, reset;  input [31:0] PC\_in;  output reg [31:0] PC\_out;  initial begin  PC\_out = 32'b0;  end    always @(posedge clk) begin  if(reset)  PC\_out <= 32'b0;  else  PC\_out <= PC\_in;  end  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\_16\_bit (in0, in1, mux\_out, select);  parameter N=16;  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 Data\_Memory\_16bit (clk,addr,write\_data,read\_data,MemRead,MemWrite);  input clk;  input [15:0] addr;  input [15:0] write\_data;  output reg [15:0] read\_data;  input MemRead;  input MemWrite;  integer i;  reg [15:0] DMemory [0:255];  // Initialize memory for simulation  initial begin    for (i = 0; i < 256; i = i + 1) begin  DMemory[i] = 16'h0000;  end  end  // Synchronous write  always @(posedge clk) begin  if (MemWrite) begin  DMemory[addr] <= write\_data;  $display("Memory Write: Address=%h Data=%h", addr, write\_data);  end  end  // Asynchronous read  always @(\*) begin  if (MemRead)  read\_data = DMemory[addr];  else  read\_data = 16'h0000;  end  endmodule  module Instruction\_Memory (read\_address, instruction, reset);  input reset;  input [31:0] read\_address;  output [31:0] instruction;  reg [31:0] Imemory [256:0];  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=0; k<32; k=k+1)  begin  // here Out changes k=0 to k=16  Imemory[k] = 32'b0;  end  /\*Test SW-type\*/  Imemory[0] = 32'b101011\_00000\_10011\_0000000000000000; // sw $s3, 0($zero) -> 0x10130000  Imemory[1] = 32'b101011\_00000\_10001\_0000000000000100; // sw $s1, 4($zero) -> 0x10110004  Imemory[2] = 32'b101011\_00000\_10100\_0000000000001000; // sw $s4, 8($zero) -> 0x10120008  Imemory[3] = 32'b101011\_00000\_10101\_0000000000001100; // sw $s5, 12($zero) -> 0x1012500C  end  endmodule |
| **Testbench** |
| module tb\_SW\_Proc;  reg reset, clk;  wire [31:0] W\_PC\_out, instruction, W\_m2;  wire [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;    SW\_processor dut(reset, clk, W\_PC\_out, instruction, W\_RD1, W\_RD2, W\_m1, W\_m2, W\_ALUout);  // Clock generation  always #5 clk = ~clk;  initial begin  $monitor($time,  " reset=%b | clk=%b | W\_PC\_out=%4d | Opcode=%6b | rs=%d | rt=%d | Imm=%h | W\_ALUout=%h | W\_RD1=%h | W\_RD2=%h",  reset, clk, W\_PC\_out, instruction[31:26], instruction[25:21], instruction[20:16],  instruction[15:0], W\_ALUout, W\_RD1, W\_RD2);  clk = 0;  reset = 1;  $display(" ----------TEST SW-PROCESSOR----------");  #10 reset = 0;  #30 $finish;  end  endmodule |
|  |
|  |

3) Write the assembly code with store instructions, compile into binary macchinecode and test the operation of the designed SW I\_Type\_Processor processor

**III.9 EXPERIMENT NO. 9**

**III.9.1 AIM: To implement I-Type Instruction Beq processor**

****

**Diagram

Description automatically generated**

**III.9.2 CODE**

module beq\_Processor (reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout);

endmodule

**III.9.3 LAB ASSIGNMENT**

|  |
| --- |
| **Top module** |
| module lab6\_ex8(SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7);  input [17:0]SW;  output [17:0]LEDR;  output [7:0]LEDG;  output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;  wire [31:0] W\_PC\_out,W\_LED\_SEG, W\_m2;  wire [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;    assign LEDR=SW;  hex\_ssd (W\_RD1 [3:0], HEX0);  hex\_ssd (W\_RD1[7:4], HEX1);  hex\_ssd ( W\_RD2[3:0],HEX2);  hex\_ssd ( W\_RD2[7:4],HEX3);  hex\_ssd (W\_PC\_out[3:0],HEX4);  hex\_ssd (W\_PC\_out[7:4],HEX5);  hex\_ssd (W\_ALUout[3:0],HEX6);  hex\_ssd (W\_ALUout[7:4],HEX7);  beq\_Processor (.reset(SW[0]),.clk(SW[1]),.W\_PC\_out(W\_PC\_out),.instruction(W\_LED\_SEG),.W\_RD1(W\_RD1),.W\_RD2(W\_RD2),.W\_m1(W\_m1),.W\_m2(W\_m2),.W\_ALUout(W\_ALUout));  endmodule  module hex\_ssd (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 |
| **Main module** |
| module beq\_Processor (reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout);  input reset, clk;  output [31:0] W\_PC\_out, instruction, W\_m2;  output [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;  wire [31:0] W\_PC\_in,W\_PC\_plus\_1,W\_Branch\_add;  wire PCsrc,RegWrite,zero,ALUsrc,MemRead,MemWrite,MemtoReg,RegDst,Branch,jump;  wire [15:0] W\_MemtoReg,W\_RD1,W\_RD2,W\_ALUout,W\_m1,W\_RDm;  wire [3:0] ALUop;  wire [4:0] W\_m3;      Program\_Counter C1(.clk(clk),.reset(reset),.PC\_in( W\_PC\_plus\_1),.PC\_out(W\_PC\_out));  Adder32Bit C2(.input1(W\_PC\_out),.input2(32'd1),.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\_16bit C6 (.clk(clk),.read\_addr\_1(instruction[25:21]),.read\_addr\_2(instruction[20:16]),.write\_addr(W\_m3),.write\_data(W\_MemtoReg),.RegWrite(RegWrite),.read\_data\_1(W\_RD1),.read\_data\_2(W\_RD2));  Sign\_Extension C7(.sign\_in(instruction[15:0]),.sign\_out(W\_Branch\_add));  Mux\_16\_bit C8(.in0(W\_RD2),.in1(instruction[15:0]),.mux\_out(W\_m1),.select(ALUsrc));  ALU\_16bit C9(.ALU\_Sel(ALUop),.A(W\_RD1),.B(W\_m1),.ALU\_Out(W\_ALUout),.zero(zero));  Data\_Memory\_16bit C10(.clk(clk),.addr(W\_ALUout),.write\_data(W\_RD2),.read\_data(W\_RDm),.MemRead(MemRead),.MemWrite(MemWrite));  Mux\_16\_bit C11(.in0(W\_ALUout),.in1(W\_RDm),.mux\_out(W\_MemtoReg),.select(MemtoReg));  Control C12(.clk(clk),.Op\_intstruct(instruction[31:26]),.ints\_function(instruction[5:0]), .RegDst(RegDst),.Branch(Branch),.MemRead(MemRead),.MemtoReg(MemtoReg),.ALUOp(ALUop),.MemWrite(MemWrite),.ALUSrc(ALUsrc),.RegWrite(RegWrite),.Zero(zero),.Jump(jump));  Mux\_5\_bit C13(.in0(instruction[20:16]),.in1(instruction[15:11]),.mux\_out(W\_m3),.select(RegDst));  endmodule  module Control(clk, Op\_intstruct, ints\_function, RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite, Zero, Jump);  input clk,Zero;  input [5:0] ints\_function;  input [5:0] Op\_intstruct;  output reg RegDst,Branch,MemRead,MemtoReg,Jump;  output reg [3:0] ALUOp;  output reg MemWrite,ALUSrc,RegWrite;  always @(\*)  begin  case (Op\_intstruct)  6'b000100: begin // BEQ  RegDst = 0;  Jump = 0;  Branch = (Zero == 1) ? 1 : 0;  MemRead = 0;  MemtoReg = 0;  ALUOp = 4'b1110;  MemWrite = 0;  ALUSrc = 0;  RegWrite = 0;  end  endcase  end  endmodule    module Register\_File\_16bit (clk,read\_addr\_1, read\_addr\_2, write\_addr, write\_data, RegWrite,read\_data\_1, read\_data\_2);  input clk,RegWrite;  input [4:0] read\_addr\_1, read\_addr\_2, write\_addr;  input [15:0] write\_data;  output [15:0] read\_data\_1, read\_data\_2;  reg [15:0] Regfile [31:0];  integer i;  // Initialize registers  initial begin  for (i = 0; i < 32; i = i + 1) begin  Regfile[i] = 16'b0;  end  Regfile[19]=1; // $s3 = 1  Regfile[17]=3; // $s1 = 3  Regfile[20]=2; // $s4 = 2  Regfile[21]=5; // $s5 = 5  end  // Read Data Logic  assign read\_data\_1 = Regfile[read\_addr\_1];  assign read\_data\_2 = Regfile[read\_addr\_2];  // Write Data Logic  always @(posedge clk) begin  if (RegWrite) begin  Regfile[write\_addr] <= write\_data;  //$display("write\_addr=%h write\_data=%h",write\_addr,write\_data);  end  end  endmodule  module ALU\_16bit(ALU\_Sel, A, B, ALU\_Out, zero);  input [3:0] ALU\_Sel; // ALU selection  input [15:0] A; // 16-bit input 1  input [15:0] B; // 16-bit input 2  output reg [15:0] ALU\_Out; // ALU 16-bit output  output reg zero;  parameter ALU\_OP\_ADD = 4'b0000, // Addition  ALU\_OP\_SUB = 4'b0001, // Subtraction  ALU\_OP\_MUL = 4'b0010, // Multiplication  ALU\_OP\_DIV = 4'b0011, // Division  ALU\_OP\_AND = 4'b0100, // Logical and  ALU\_OP\_OR = 4'b0101, // Logical or  ALU\_OP\_XOR = 4'b0110, // Logical xor  ALU\_OP\_NOR = 4'b0111, // Logical nor  ALU\_OP\_NAND = 4'b1000, // Logical nand  ALU\_OP\_XNOR = 4'b1001, // Logical xnor  ALU\_OP\_SHL = 4'b1010, // Logical shift left  ALU\_OP\_SHR = 4'b1011, // Logical shift right  ALU\_OP\_ROL = 4'b1100, // Rotate left  ALU\_OP\_ROR = 4'b1101, // Rotate right  ALU\_OP\_BEQ = 4'b1110, // Equal comparison  ALU\_OP\_BNE = 4'b1111; // Not Equal comparison  always @(\*)  begin  case(ALU\_Sel)  ALU\_OP\_ADD : ALU\_Out = A + B;  ALU\_OP\_SUB : ALU\_Out = A - B;  ALU\_OP\_MUL : ALU\_Out = A \* B;  ALU\_OP\_DIV : ALU\_Out = A / B;  ALU\_OP\_AND : ALU\_Out = A & B;  ALU\_OP\_OR : ALU\_Out = A | B;  ALU\_OP\_XOR : ALU\_Out = A ^ B;  ALU\_OP\_NOR : ALU\_Out = ~(A | B);  ALU\_OP\_NAND : ALU\_Out = ~(A & B);  ALU\_OP\_XNOR : ALU\_Out = ~(A ^ B);  ALU\_OP\_SHL : ALU\_Out = A<<1;  ALU\_OP\_SHR : ALU\_Out = A>>1;  ALU\_OP\_ROL : ALU\_Out = {A[6:0],A[7]};  ALU\_OP\_ROR : ALU\_Out = {A[0],A[7:1]};  ALU\_OP\_BEQ : begin  zero = (A==B)?1'b1:1'b0;  ALU\_Out = (A==B)?16'd1:16'd0;  end  ALU\_OP\_BNE : begin  zero = (A!=B)?1'b1:1'b0;  // zero here is different from the above zero  // zero here is equal to 1, when A is different from B  // zero here is equal to 0, when A is the same as B  ALU\_Out = (A!=B)?16'd1:16'd0;  end  endcase  end  endmodule  // Sign Extension  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 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 Program\_Counter(clk, reset, PC\_in, PC\_out);  input clk, reset;  input [31:0] PC\_in;  output reg [31:0] PC\_out;  initial begin  PC\_out = 32'b0;  end    always @(posedge clk) begin  if(reset)  PC\_out <= 32'b0;  else  PC\_out <= PC\_in;  end  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\_16\_bit (in0, in1, mux\_out, select);  parameter N=16;  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 Data\_Memory\_16bit (clk,addr,write\_data,read\_data,MemRead,MemWrite);  input clk;  input [15:0] addr;  input [15:0] write\_data;  output reg [15:0] read\_data;  input MemRead;  input MemWrite;  integer i;  reg [15:0] DMemory [0:255];  // Initialize memory for simulation  initial begin    for (i = 0; i < 256; i = i + 1) begin  DMemory[i] = 16'h0000;  end  end  // Synchronous write  always @(posedge clk) begin  if (MemWrite) begin  DMemory[addr] <= write\_data;  $display("Memory Write: Address=%h Data=%h", addr, write\_data);  end  end  // Asynchronous read  always @(\*) begin  if (MemRead)  read\_data = DMemory[addr];  else  read\_data = 16'h0000;  end  endmodule  module Instruction\_Memory (read\_address, instruction, reset);  input reset;  input [31:0] read\_address;  output [31:0] instruction;  reg [31:0] Imemory [256:0];  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=0; k<32; k=k+1)  begin  // here Out changes k=0 to k=16  Imemory[k] = 32'b0;  end  /\*Test BEQ-type\*/  Imemory[0] = 32'b000100\_10010\_10011\_0000000000000011; // beq $s2, $s3, label3 -> offset = 3  Imemory[1] = 32'b000100\_10001\_10101\_0000000000000100; // beq $s1, $s5, label4 -> offset = 4  Imemory[2] = 32'b000100\_10000\_10100\_0000000000000110; // beq $s0, $s4, label5 -> offset = 6  Imemory[3] = 32'b000100\_10010\_10000\_0000000000000111; // beq $s2, $s0, label6 -> offset = 7  end  endmodule |
| **Testbench** |
| module tb\_BEQ\_Proc;  reg reset, clk;  wire [31:0] W\_PC\_out, instruction, W\_m2;  wire [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;    beq\_Processor dut(reset, clk, W\_PC\_out, instruction, W\_RD1, W\_RD2, W\_m1, W\_m2, W\_ALUout);  // Clock generation  always #5 clk = ~clk;  initial begin  $monitor($time,  " reset=%b | clk=%b | W\_PC\_out=%4d | Opcode=%6b | rs=%d | rt=%d | zero=%d | Imm=%h | W\_ALUout=%h | W\_RD1=%h | W\_RD2=%h",  reset, clk, W\_PC\_out, instruction[31:26], instruction[25:21], instruction[20:16], dut.C9.zero,  instruction[15:0], W\_ALUout, W\_RD1, W\_RD2);  clk = 0;  reset = 1;  $display(" ----------TEST BEQ-PROCESSOR----------");  #10 reset = 0;  #30 $finish;  end  endmodule |
|  |
|  |

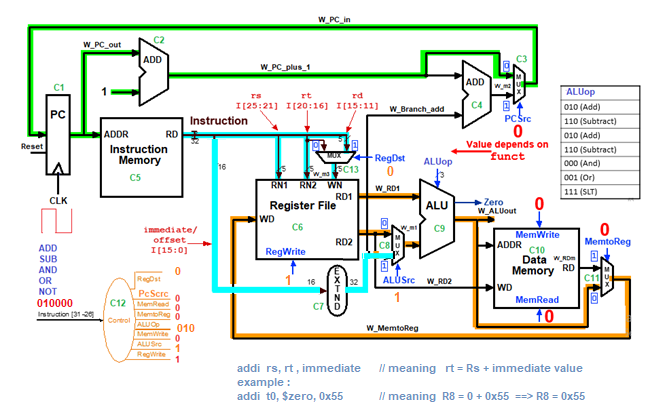
3) Write the assembly code with branch equal instructions, compile into binary macchinecode and test the operation of the designed Beq I\_Type\_Processor processor

**III.10 EXPERIMENT NO. 10**

**III.10.1 AIM: To implement I-Type Instruction addi processor**

**Table

Description automatically generated**

****

**III.10.2 CODE**

module I\_Type\_Processor (reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout);

endmodule

**III.10.3 LAB ASSIGNMENT**

|  |
| --- |
| **Top module** |
| module lab6\_ex9(SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7);  input [17:0]SW;  output [17:0]LEDR;  output [7:0]LEDG;  output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;  wire [31:0] W\_PC\_out,W\_LED\_SEG, W\_m2;  wire [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;    assign LEDR=SW;  hex\_ssd (W\_RD1 [3:0], HEX0);  hex\_ssd (W\_RD1[7:4], HEX1);  hex\_ssd ( W\_RD2[3:0],HEX2);  hex\_ssd ( W\_RD2[7:4],HEX3);  hex\_ssd (W\_PC\_out[3:0],HEX4);  hex\_ssd (W\_PC\_out[7:4],HEX5);  hex\_ssd (W\_ALUout[3:0],HEX6);  hex\_ssd (W\_ALUout[7:4],HEX7);  I\_Type\_Processor (.reset(SW[0]),.clk(SW[1]),.W\_PC\_out(W\_PC\_out),.instruction(W\_LED\_SEG),.W\_RD1(W\_RD1),.W\_RD2(W\_RD2),.W\_m1(W\_m1),.W\_m2(W\_m2),.W\_ALUout(W\_ALUout));  endmodule  module hex\_ssd (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 |
| **Main module** |
| module I\_Type\_Processor (reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout);  input reset, clk;  output [31:0] W\_PC\_out, instruction, W\_m2;  output [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;  wire [31:0] W\_PC\_in,W\_PC\_plus\_1,W\_Branch\_add;  wire PCsrc,RegWrite,zero,ALUsrc,MemRead,MemWrite,MemtoReg,RegDst,Branch,jump;  wire [15:0] W\_MemtoReg,W\_RD1,W\_RD2,W\_ALUout,W\_m1,W\_RDm;  wire [3:0] ALUop;  wire [4:0] W\_m3;      Program\_Counter C1(.clk(clk),.reset(reset),.PC\_in( W\_PC\_plus\_1),.PC\_out(W\_PC\_out));  Adder32Bit C2(.input1(W\_PC\_out),.input2(32'd1),.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\_16bit C6 (.clk(clk),.read\_addr\_1(instruction[25:21]),.read\_addr\_2(instruction[20:16]),.write\_addr(W\_m3),.write\_data(W\_MemtoReg),.RegWrite(RegWrite),.read\_data\_1(W\_RD1),.read\_data\_2(W\_RD2));  Sign\_Extension C7(.sign\_in(instruction[15:0]),.sign\_out(W\_Branch\_add));  Mux\_16\_bit C8(.in0(W\_RD2),.in1(instruction[15:0]),.mux\_out(W\_m1),.select(ALUsrc));  ALU\_16bit C9(.ALU\_Sel(ALUop),.A(W\_RD1),.B(W\_m1),.ALU\_Out(W\_ALUout),.zero(zero));  Data\_Memory\_16bit C10(.clk(clk),.addr(W\_ALUout),.write\_data(W\_RD2),.read\_data(W\_RDm),.MemRead(MemRead),.MemWrite(MemWrite));  Mux\_16\_bit C11(.in0(W\_ALUout),.in1(W\_RDm),.mux\_out(W\_MemtoReg),.select(MemtoReg));  Control C12(.clk(clk),.Op\_intstruct(instruction[31:26]),.ints\_function(instruction[5:0]), .RegDst(RegDst),.Branch(Branch),.MemRead(MemRead),.MemtoReg(MemtoReg),.ALUOp(ALUop),.MemWrite(MemWrite),.ALUSrc(ALUsrc),.RegWrite(RegWrite),.Zero(zero),.Jump(jump));  Mux\_5\_bit C13(.in0(instruction[20:16]),.in1(instruction[15:11]),.mux\_out(W\_m3),.select(RegDst));  endmodule  module Control(clk, Op\_intstruct, ints\_function, RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite, Zero, Jump);  input clk,Zero;  input [5:0] ints\_function;  input [5:0] Op\_intstruct;  output reg RegDst,Branch,MemRead,MemtoReg,Jump;  output reg [3:0] ALUOp;  output reg MemWrite,ALUSrc,RegWrite;  always @(\*)  begin  case (Op\_intstruct)  6'b000100: begin // BEQ  RegDst = 0;  Jump = 0;  Branch = (Zero == 1) ? 1 : 0;  MemRead = 0;  MemtoReg = 0;  ALUOp = 4'b1110;  MemWrite = 0;  ALUSrc = 0;  RegWrite = 0;  end  6'b100011: begin // LW  RegDst = 0;  Jump = 0;  Branch = 0;  MemRead = 1;  MemtoReg = 1;  ALUOp = 4'b0000;  MemWrite = 0;  ALUSrc = 1;  RegWrite = 1;  end  6'b101011: begin // SW  RegDst = 0;  Jump = 0;  Branch = 0;  MemRead = 0;  MemtoReg = 0;  ALUOp = 4'b0000;  MemWrite = 1;  ALUSrc = 1;  RegWrite = 0;  end  6'b001000: begin // ADDI  RegDst = 0;  Jump = 0;  Branch = 0;  MemRead = 0;  MemtoReg = 0;  ALUOp = 4'b0000;  MemWrite = 0;  ALUSrc = 1;  RegWrite = 1;  end  6'b000101: begin // BNE  RegDst = 0;  Jump = 0;  Branch = (Zero == 0) ? 1 : 0;  MemRead = 0;  MemtoReg = 0;  ALUOp = 4'b1111;  MemWrite = 0;  ALUSrc = 0;  RegWrite = 0;  end  endcase  end  endmodule    module Register\_File\_16bit (clk,read\_addr\_1, read\_addr\_2, write\_addr, write\_data, RegWrite,read\_data\_1, read\_data\_2);  input clk,RegWrite;  input [4:0] read\_addr\_1, read\_addr\_2, write\_addr;  input [15:0] write\_data;  output [15:0] read\_data\_1, read\_data\_2;  reg [15:0] Regfile [31:0];  integer i;  // Initialize registers  initial begin  for (i = 0; i < 32; i = i + 1) begin  Regfile[i] = 16'b0;  end  Regfile[19]=1; // $s3 = 1  Regfile[17]=3; // $s1 = 3  Regfile[20]=2; // $s4 = 2  Regfile[21]=5; // $s5 = 5  end  // Read Data Logic  assign read\_data\_1 = Regfile[read\_addr\_1];  assign read\_data\_2 = Regfile[read\_addr\_2];  // Write Data Logic  always @(posedge clk) begin  if (RegWrite) begin  Regfile[write\_addr] <= write\_data;  //$display("write\_addr=%h write\_data=%h",write\_addr,write\_data);  end  end  endmodule  module ALU\_16bit(ALU\_Sel, A, B, ALU\_Out, zero);  input [3:0] ALU\_Sel; // ALU selection  input [15:0] A; // 16-bit input 1  input [15:0] B; // 16-bit input 2  output reg [15:0] ALU\_Out; // ALU 16-bit output  output reg zero;  parameter ALU\_OP\_ADD = 4'b0000, // Addition  ALU\_OP\_SUB = 4'b0001, // Subtraction  ALU\_OP\_MUL = 4'b0010, // Multiplication  ALU\_OP\_DIV = 4'b0011, // Division  ALU\_OP\_AND = 4'b0100, // Logical and  ALU\_OP\_OR = 4'b0101, // Logical or  ALU\_OP\_XOR = 4'b0110, // Logical xor  ALU\_OP\_NOR = 4'b0111, // Logical nor  ALU\_OP\_NAND = 4'b1000, // Logical nand  ALU\_OP\_XNOR = 4'b1001, // Logical xnor  ALU\_OP\_SHL = 4'b1010, // Logical shift left  ALU\_OP\_SHR = 4'b1011, // Logical shift right  ALU\_OP\_ROL = 4'b1100, // Rotate left  ALU\_OP\_ROR = 4'b1101, // Rotate right  ALU\_OP\_BEQ = 4'b1110, // Equal comparison  ALU\_OP\_BNE = 4'b1111; // Not Equal comparison  always @(\*)  begin  case(ALU\_Sel)  ALU\_OP\_ADD : ALU\_Out = A + B;  ALU\_OP\_SUB : ALU\_Out = A - B;  ALU\_OP\_MUL : ALU\_Out = A \* B;  ALU\_OP\_DIV : ALU\_Out = A / B;  ALU\_OP\_AND : ALU\_Out = A & B;  ALU\_OP\_OR : ALU\_Out = A | B;  ALU\_OP\_XOR : ALU\_Out = A ^ B;  ALU\_OP\_NOR : ALU\_Out = ~(A | B);  ALU\_OP\_NAND : ALU\_Out = ~(A & B);  ALU\_OP\_XNOR : ALU\_Out = ~(A ^ B);  ALU\_OP\_SHL : ALU\_Out = A<<1;  ALU\_OP\_SHR : ALU\_Out = A>>1;  ALU\_OP\_ROL : ALU\_Out = {A[6:0],A[7]};  ALU\_OP\_ROR : ALU\_Out = {A[0],A[7:1]};  ALU\_OP\_BEQ : begin  zero = (A==B)?1'b1:1'b0;  ALU\_Out = (A==B)?16'd1:16'd0;  end  ALU\_OP\_BNE : begin  zero = (A!=B)?1'b1:1'b0;  // zero here is different from the above zero  // zero here is equal to 1, when A is different from B  // zero here is equal to 0, when A is the same as B  ALU\_Out = (A!=B)?16'd1:16'd0;  end  endcase  end  endmodule  // Sign Extension  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 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 Program\_Counter(clk, reset, PC\_in, PC\_out);  input clk, reset;  input [31:0] PC\_in;  output reg [31:0] PC\_out;  initial begin  PC\_out = 32'b0;  end    always @(posedge clk) begin  if(reset)  PC\_out <= 32'b0;  else  PC\_out <= PC\_in;  end  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\_16\_bit (in0, in1, mux\_out, select);  parameter N=16;  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 Data\_Memory\_16bit (clk,addr,write\_data,read\_data,MemRead,MemWrite);  input clk;  input [15:0] addr;  input [15:0] write\_data;  output reg [15:0] read\_data;  input MemRead;  input MemWrite;  integer i;  reg [15:0] DMemory [0:255];  // Initialize memory for simulation  initial begin    for (i = 0; i < 256; i = i + 1) begin  DMemory[i] = 16'h0000;  end  end  // Synchronous write  always @(posedge clk) begin  if (MemWrite) begin  DMemory[addr] <= write\_data;  $display("Memory Write: Address=%h Data=%h", addr, write\_data);  end  end  // Asynchronous read  always @(\*) begin  if (MemRead)  read\_data = DMemory[addr];  else  read\_data = 16'h0000;  end  endmodule  module Instruction\_Memory (read\_address, instruction, reset);  input reset;  input [31:0] read\_address;  output [31:0] instruction;  reg [31:0] Imemory [256:0];  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=0; k<32; k=k+1)  begin  // here Out changes k=0 to k=16  Imemory[k] = 32'b0;  end  /\*Test I-type\*/  // addi $s1, $s1, 0x05 R17 = 0x4E => R17 = R17 + 0x5 = 0x53  Imemory[1] = 32'b001000\_10001\_10001\_00000\_00000\_000101;  // addi $s1, $s1, 0x05 R17 = 0x53 => R17 = R17 + 0x5 = 0x58  Imemory[2] = 32'b001000\_10001\_10001\_00000\_00000\_000101;  // addi $s3, $s1, 0x10 R17 = 0x58 => R19 = R17 + 0x10 = 0x68  Imemory[3] = 32'b001000\_10001\_10011\_00000\_00000\_010000;  // sw $s3, 0x04($s1) Memory[$s1 + 0x04] = $s3 => Memory[0x58 + 0x04] = 0x68  Imemory[4] = 32'b101011\_10001\_10011\_00000\_00000\_000100;  // lw $s7, 0x04($s1) $s7 = Memory[$s1 + 0x04] = 0x68  Imemory[5] = 32'b100011\_10001\_10111\_00000\_00000\_000100;  // beq $s7, $s3, 0x08 R23 (0x68) == R19 (0x68) move to Imemory[22 + 8 + 1]  Imemory[6] = 32'b000100\_10111\_10011\_00000\_00000\_001000;  // addi $s1, $zero, 0x11 R17 = 0x11  Imemory[7] = 32'b001000\_00000\_10001\_00000\_00000\_010001;  // addi $s1, $zero, 0x22 R17 = 0x22  Imemory[8] = 32'b001000\_00000\_10001\_00000\_00000\_100010;  // addi $s1, $zero, 0x33 R17 = 0x33  Imemory[9] = 32'b001000\_00000\_10001\_00000\_00000\_110011;  // bne $s1, $s7, 0x04 (R17 (0x33) != R23 (0x68)) move to Imemory[26 + 4 + 1]  Imemory[10] = 32'b000101\_10001\_10111\_00000\_00000\_000100;  // addi $s2, $zero, 0x66 R18 = 0x66  Imemory[11] = 32'b001000\_00000\_10010\_00000\_00001\_100110;  // addi $s2, $zero, 0x77 R18 = 0x77  Imemory[12] = 32'b001000\_00000\_10010\_00000\_00001\_110111;  end  endmodule |
| **Testbench** |
| module tb\_I\_Type\_Proc;  reg reset, clk;  wire [31:0] W\_PC\_out, instruction, W\_m2;  wire [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;  reg [127:0] instr\_name;  I\_Type\_Processor dut(reset, clk, W\_PC\_out, instruction, W\_RD1, W\_RD2, W\_m1, W\_m2, W\_ALUout);  // Clock generation  always #5 clk = ~clk;  // Task to decode instruction name  task decode\_instruction;  input [31:0] instr;  output reg [127:0] name; // String for instruction name  begin  case (instr[31:26]) // Use opcode field for I-Type instructions  6'b000100: name = "BEQ";  6'b100011: name = "LW";  6'b101011: name = "SW";  6'b001000: name = "ADDI";  6'b000101: name = "BNE";  default: name = "Unknown Instruction";  endcase  end  endtask  initial begin  clk = 0;  reset = 1;  $display(" ----------TEST I-TYPE----------");  #10 reset = 0;  $monitor($time,  " reset=%b | clk=%b | W\_PC\_out=%4d | Instruction=%s | Opcode=%6b | rs=%d | rt=%d | rd=%d | zero=%b | Imm=%h | W\_ALUout=%h | W\_RD1=%h | W\_RD2=%h",  reset, clk, W\_PC\_out, instr\_name, instruction[31:26], instruction[25:21], instruction[20:16], instruction[15:11], dut.C9.zero,  instruction[15:0], W\_ALUout, W\_RD1, W\_RD2);  // Simulate decoding each instruction during runtime for a limited duration  repeat (12) begin  decode\_instruction(instruction, instr\_name);  #10; // Wait for the next clock cycle  end  $finish;  end  endmodule |
|  |
|  |

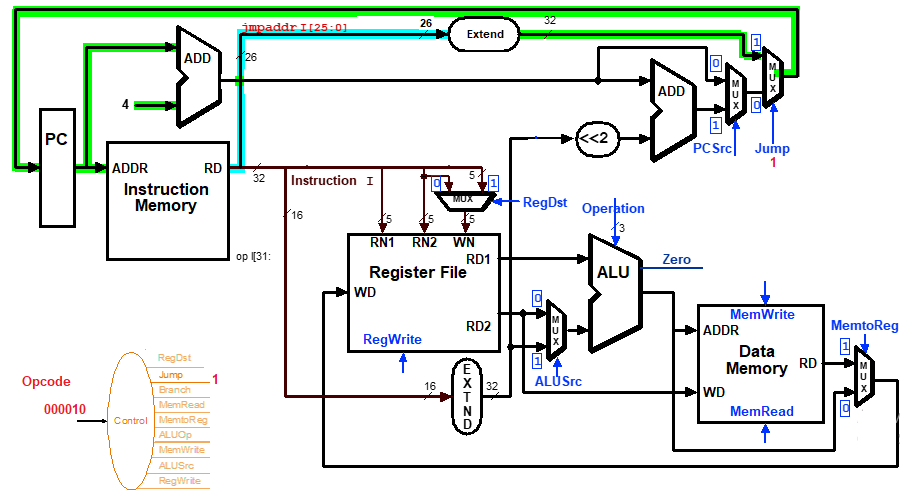
3) Write the assembly code with addi instructions, compile into binary macchinecode and test the operation of the designed addi I\_Type\_Processor processor

**III.11 EXPERIMENT NO. 11**

**III.11.1 AIM: To implement J-Type Instruction processor**

**Table

Description automatically generated**

****

**III.11.2 CODE**

module J\_Type\_Processor (reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout);

endmodule

**III.11.3 LAB ASSIGNMENT**

|  |
| --- |
| **Top module** |
| module lab6\_ex10(SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7);  input [17:0]SW;  output [17:0]LEDR;  output [7:0]LEDG;  output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;  wire [31:0] W\_PC\_out,W\_LED\_SEG, W\_m2;  wire [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;    assign LEDR=SW;  hex\_ssd (W\_RD1 [3:0], HEX0);  hex\_ssd (W\_RD1[7:4], HEX1);  hex\_ssd ( W\_RD2[3:0],HEX2);  hex\_ssd ( W\_RD2[7:4],HEX3);  hex\_ssd (W\_PC\_out[3:0],HEX4);  hex\_ssd (W\_PC\_out[7:4],HEX5);  hex\_ssd (W\_ALUout[3:0],HEX6);  hex\_ssd (W\_ALUout[7:4],HEX7);  J\_Type\_Processor (.reset(SW[0]),.clk(SW[1]),.W\_PC\_out(W\_PC\_out),.instruction(W\_LED\_SEG),.W\_RD1(W\_RD1),.W\_RD2(W\_RD2),.W\_m1(W\_m1),.W\_m2(W\_m2),.W\_ALUout(W\_ALUout));  endmodule  module hex\_ssd (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 |
| **Main module** |
| module J\_Type\_Processor (reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout);  input reset, clk;  output [31:0] W\_PC\_out, instruction, W\_m2;  output [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;  wire [31:0] W\_PC\_in,W\_PC\_plus\_1,W\_Branch\_add;  wire PCsrc,RegWrite,zero,ALUsrc,MemRead,MemWrite,MemtoReg,RegDst,Branch,jump;  wire [15:0] W\_MemtoReg,W\_RD1,W\_RD2,W\_ALUout,W\_m1,W\_RDm;  wire [3:0] ALUop;  wire [4:0] W\_m3;      Program\_Counter C1(.clk(clk),.reset(reset),.PC\_in( W\_PC\_plus\_1),.PC\_out(W\_PC\_out));  Adder32Bit C2(.input1(W\_PC\_out),.input2(32'd1),.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\_16bit C6 (.clk(clk),.read\_addr\_1(instruction[25:21]),.read\_addr\_2(instruction[20:16]),.write\_addr(W\_m3),.write\_data(W\_MemtoReg),.RegWrite(RegWrite),.read\_data\_1(W\_RD1),.read\_data\_2(W\_RD2));  Sign\_Extension C7(.sign\_in(instruction[15:0]),.sign\_out(W\_Branch\_add));  Mux\_16\_bit C8(.in0(W\_RD2),.in1(instruction[15:0]),.mux\_out(W\_m1),.select(ALUsrc));  ALU\_16bit C9(.ALU\_Sel(ALUop),.A(W\_RD1),.B(W\_m1),.ALU\_Out(W\_ALUout),.zero(zero));  Data\_Memory\_16bit C10(.clk(clk),.addr(W\_ALUout),.write\_data(W\_RD2),.read\_data(W\_RDm),.MemRead(MemRead),.MemWrite(MemWrite));  Mux\_16\_bit C11(.in0(W\_ALUout),.in1(W\_RDm),.mux\_out(W\_MemtoReg),.select(MemtoReg));  Control C12(.clk(clk),.Op\_intstruct(instruction[31:26]),.ints\_function(instruction[5:0]), .RegDst(RegDst),.Branch(Branch),.MemRead(MemRead),.MemtoReg(MemtoReg),.ALUOp(ALUop),.MemWrite(MemWrite),.ALUSrc(ALUsrc),.RegWrite(RegWrite),.Zero(zero),.Jump(jump));  Mux\_5\_bit C13(.in0(instruction[20:16]),.in1(instruction[15:11]),.mux\_out(W\_m3),.select(RegDst));  endmodule  module Control(clk, Op\_intstruct, ints\_function, RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite, Zero, Jump);  input clk,Zero;  input [5:0] ints\_function;  input [5:0] Op\_intstruct;  output reg RegDst,Branch,MemRead,MemtoReg,Jump;  output reg [3:0] ALUOp;  output reg MemWrite,ALUSrc,RegWrite;  always @(\*)  begin  case (Op\_intstruct)  6'b000010: begin // JUMP  RegDst = 0;  Jump = 1;  Branch = 0;  MemRead = 0;  MemtoReg = 0;  ALUOp = 4'b0000;  MemWrite = 0;  ALUSrc = 1;  RegWrite = 0;  end  6'b000011: begin // JUMP and LINK  RegDst= 1;  Jump = 1;  Branch = 0;  MemRead = 0;  MemtoReg = 1;  ALUOp = 4'b0000;  MemWrite= 0;  ALUSrc = 0;  RegWrite = 1;  end  endcase  end  endmodule    module Register\_File\_16bit (clk,read\_addr\_1, read\_addr\_2, write\_addr, write\_data, RegWrite,read\_data\_1, read\_data\_2);  input clk,RegWrite;  input [4:0] read\_addr\_1, read\_addr\_2, write\_addr;  input [15:0] write\_data;  output [15:0] read\_data\_1, read\_data\_2;  reg [15:0] Regfile [31:0];  integer i;  // Initialize registers  initial begin  for (i = 0; i < 32; i = i + 1) begin  Regfile[i] = 16'b0;  end  Regfile[19]=1; // $s3 = 1  Regfile[17]=3; // $s1 = 3  Regfile[20]=2; // $s4 = 2  Regfile[21]=5; // $s5 = 5  end  // Read Data Logic  assign read\_data\_1 = Regfile[read\_addr\_1];  assign read\_data\_2 = Regfile[read\_addr\_2];  // Write Data Logic  always @(posedge clk) begin  if (RegWrite) begin  Regfile[write\_addr] <= write\_data;  //$display("write\_addr=%h write\_data=%h",write\_addr,write\_data);  end  end  endmodule  module ALU\_16bit(ALU\_Sel, A, B, ALU\_Out, zero);  input [3:0] ALU\_Sel; // ALU selection  input [15:0] A; // 16-bit input 1  input [15:0] B; // 16-bit input 2  output reg [15:0] ALU\_Out; // ALU 16-bit output  output reg zero;  parameter ALU\_OP\_ADD = 4'b0000, // Addition  ALU\_OP\_SUB = 4'b0001, // Subtraction  ALU\_OP\_MUL = 4'b0010, // Multiplication  ALU\_OP\_DIV = 4'b0011, // Division  ALU\_OP\_AND = 4'b0100, // Logical and  ALU\_OP\_OR = 4'b0101, // Logical or  ALU\_OP\_XOR = 4'b0110, // Logical xor  ALU\_OP\_NOR = 4'b0111, // Logical nor  ALU\_OP\_NAND = 4'b1000, // Logical nand  ALU\_OP\_XNOR = 4'b1001, // Logical xnor  ALU\_OP\_SHL = 4'b1010, // Logical shift left  ALU\_OP\_SHR = 4'b1011, // Logical shift right  ALU\_OP\_ROL = 4'b1100, // Rotate left  ALU\_OP\_ROR = 4'b1101, // Rotate right  ALU\_OP\_BEQ = 4'b1110, // Equal comparison  ALU\_OP\_BNE = 4'b1111; // Not Equal comparison  always @(\*)  begin  case(ALU\_Sel)  ALU\_OP\_ADD : ALU\_Out = A + B;  ALU\_OP\_SUB : ALU\_Out = A - B;  ALU\_OP\_MUL : ALU\_Out = A \* B;  ALU\_OP\_DIV : ALU\_Out = A / B;  ALU\_OP\_AND : ALU\_Out = A & B;  ALU\_OP\_OR : ALU\_Out = A | B;  ALU\_OP\_XOR : ALU\_Out = A ^ B;  ALU\_OP\_NOR : ALU\_Out = ~(A | B);  ALU\_OP\_NAND : ALU\_Out = ~(A & B);  ALU\_OP\_XNOR : ALU\_Out = ~(A ^ B);  ALU\_OP\_SHL : ALU\_Out = A<<1;  ALU\_OP\_SHR : ALU\_Out = A>>1;  ALU\_OP\_ROL : ALU\_Out = {A[6:0],A[7]};  ALU\_OP\_ROR : ALU\_Out = {A[0],A[7:1]};  ALU\_OP\_BEQ : begin  zero = (A==B)?1'b1:1'b0;  ALU\_Out = (A==B)?16'd1:16'd0;  end  ALU\_OP\_BNE : begin  zero = (A!=B)?1'b1:1'b0;  // zero here is different from the above zero  // zero here is equal to 1, when A is different from B  // zero here is equal to 0, when A is the same as B  ALU\_Out = (A!=B)?16'd1:16'd0;  end  endcase  end  endmodule  // Sign Extension  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 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 Program\_Counter(clk, reset, PC\_in, PC\_out);  input clk, reset;  input [31:0] PC\_in;  output reg [31:0] PC\_out;  initial begin  PC\_out = 32'b0;  end    always @(posedge clk) begin  if(reset)  PC\_out <= 32'b0;  else  PC\_out <= PC\_in;  end  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\_16\_bit (in0, in1, mux\_out, select);  parameter N=16;  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 Data\_Memory\_16bit (clk,addr,write\_data,read\_data,MemRead,MemWrite);  input clk;  input [15:0] addr;  input [15:0] write\_data;  output reg [15:0] read\_data;  input MemRead;  input MemWrite;  integer i;  reg [15:0] DMemory [0:255];  // Initialize memory for simulation  initial begin    for (i = 0; i < 256; i = i + 1) begin  DMemory[i] = 16'h0000;  end  end  // Synchronous write  always @(posedge clk) begin  if (MemWrite) begin  DMemory[addr] <= write\_data;  $display("Memory Write: Address=%h Data=%h", addr, write\_data);  end  end  // Asynchronous read  always @(\*) begin  if (MemRead)  read\_data = DMemory[addr];  else  read\_data = 16'h0000;  end  endmodule  module Instruction\_Memory (read\_address, instruction, reset);  input reset;  input [31:0] read\_address;  output [31:0] instruction;  reg [31:0] Imemory [256:0];  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=0; k<32; k=k+1)  begin  // here Out changes k=0 to k=16  Imemory[k] = 32'b0;  end  /\* Test J-type Instructions \*/  // Jump to Imemory[116] using j (JUMP)  Imemory[0] = 32'b000010\_00000\_00000\_00000\_00001\_110100; // j 0x74 (JUMP to Imemory[116])    // JAL instruction to target 0x74 (Jump and Link)  Imemory[1] = 32'b000011\_00000\_00000\_00000\_00001\_110100; // jal 0x74 (Jump and Link to Imemory[116])    // Another JUMP to Imemory[120] using j  Imemory[2] = 32'b000010\_00000\_00000\_00000\_00001\_111000; // j 0x78 (JUMP to Imemory[120])    // Another JAL instruction to target 0x78  Imemory[3] = 32'b000011\_00000\_00000\_00000\_00001\_111000; // jal 0x78 (Jump and Link to Imemory[120])    // JUMP back to Imemory[100] (looping behavior)  Imemory[4] = 32'b000010\_00000\_00000\_00000\_00001\_100100; // j 0x64 (JUMP to Imemory[100])    // JAL to Imemory[104] (jump and link behavior)  Imemory[5] = 32'b000011\_00000\_00000\_00000\_00001\_101000; // jal 0x68 (Jump and Link to Imemory[104])    // JUMP again to Imemory[108]  Imemory[6] = 32'b000010\_00000\_00000\_00000\_00001\_101100; // j 0x6C (JUMP to Imemory[108])    // JAL back to Imemory[112]  Imemory[7] = 32'b000011\_00000\_00000\_00000\_00001\_110000; // jal 0x70 (Jump and Link to Imemory[112])    // Final JUMP instruction to Imemory[116]  Imemory[8] = 32'b000010\_00000\_00000\_00000\_00001\_110100; // j 0x74 (JUMP to Imemory[116])    // END of the test program  Imemory[9] = 32'b000010\_00000\_00000\_00000\_00001\_111100; // j 0x7C (JUMP to Imemory[124])  end  endmodule |
| **Testbench** |
| module tb\_J\_Type\_Proc;  reg reset, clk;  wire [31:0] W\_PC\_out, instruction, W\_m2;  wire [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;  reg [127:0] instr\_name;  J\_Type\_Processor dut(reset, clk, W\_PC\_out, instruction, W\_RD1, W\_RD2, W\_m1, W\_m2, W\_ALUout);  // Clock generation  always #5 clk = ~clk;  // Task to decode instruction name  task decode\_instruction;  input [31:0] instr;  output reg [127:0] name; // String for instruction name  begin  case (instr[31:26]) // Use opcode field for J-Type instructions  6'b000010: name = "JUMP (J)";  6'b000011: name = "JUMP AND LINK (JAL)";  default: name = "Unknown Instruction";  endcase  end  endtask  initial begin  clk = 0;  reset = 1;  $display(" ----------TEST J-TYPE----------");  #10 reset = 0;  $monitor($time,  " reset=%b | clk=%b | W\_PC\_out=%4d | Instruction=%s | Opcode=%6b | Imm=%h | W\_ALUout=%h ",  reset, clk, W\_PC\_out, instr\_name, instruction[31:26],  instruction[15:0], W\_ALUout);  // Simulate decoding each instruction during runtime for a limited duration  repeat (9) begin  decode\_instruction(instruction, instr\_name);  #10; // Wait for the next clock cycle  end  $finish;  end  endmodule |
|  |
|  |

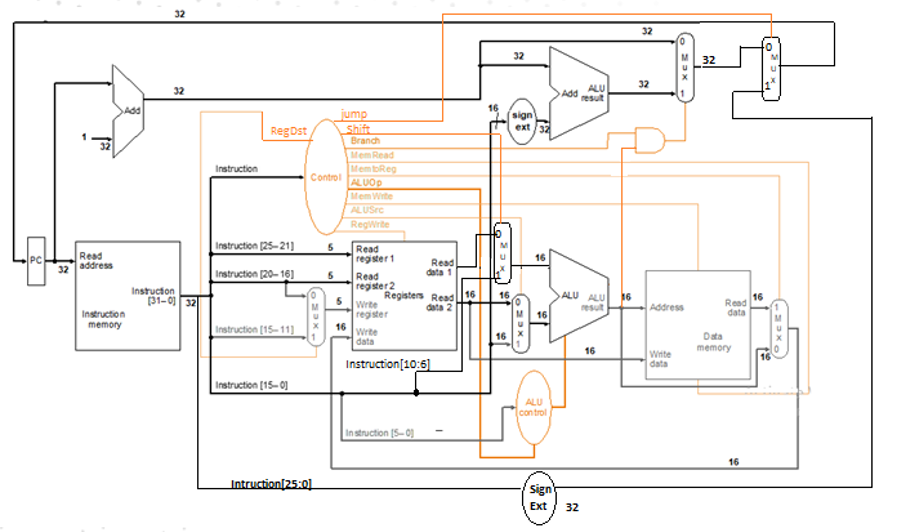
3) Write the assembly code with Jump instructions, compile into binary macchinecode and test the operation of the designed J\_Type\_Processor processor

**III.12 EXPERIMENT NO. 12**

**III.12.1 AIM: Complet Single Cycle processor 16 bit**

**Table

Description automatically generated**

****

**III.12.2 CODE**

**III.12.3 LAB ASSIGNMENT**

|  |
| --- |
| **Top module** |
| module lab6\_ex11(SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7);  input [17:0]SW;  output [17:0]LEDR;  output [7:0]LEDG;  output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;  wire [31:0] W\_PC\_out,W\_LED\_SEG, W\_m2;  wire [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;    assign LEDR=SW;  hex\_ssd (W\_RD1 [3:0], HEX0);  hex\_ssd (W\_RD1[7:4], HEX1);  hex\_ssd ( W\_RD2[3:0],HEX2);  hex\_ssd ( W\_RD2[7:4],HEX3);  hex\_ssd (W\_PC\_out[3:0],HEX4);  hex\_ssd (W\_PC\_out[7:4],HEX5);  hex\_ssd (W\_ALUout[3:0],HEX6);  hex\_ssd (W\_ALUout[7:4],HEX7);  Single\_Cycle\_Processor\_16bit(.reset(SW[0]),.clk(SW[1]),.W\_PC\_out(W\_PC\_out),.instruction(W\_LED\_SEG),.W\_RD1(W\_RD1),.W\_RD2(W\_RD2),.W\_m1(W\_m1),.W\_m2(W\_m2),.W\_ALUout(W\_ALUout));  endmodule  module hex\_ssd (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 |
| **Main module** |
| //Single Cycle Processor 16bit  module Single\_Cycle\_Processor\_16bit(reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout);  input reset, clk;  output [31:0] W\_PC\_out, instruction, W\_m2;  output [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;  wire [31:0] W\_PC\_in,W\_PC\_plus\_1,W\_Branch\_add,W\_m4,W\_jump\_PC;  wire PCsrc,RegWrite,zero,ALUsrc,MemRead,MemWrite,MemtoReg,RegDst,Branch,jump,shift;  wire [15:0] W\_MemtoReg,W\_RD1,W\_RD2,W\_ALUout,W\_m1,W\_RDm,W\_m5;  wire [3:0] ALUop;  wire [4:0] W\_m3;    Program\_Counter C1(.clk(clk),.reset(reset),.PC\_in( W\_PC\_plus\_1),.PC\_out(W\_PC\_out));  Adder32Bit C2(.input1(W\_PC\_out),.input2(32'd1),.out(W\_PC\_plus\_1));  Mux\_32\_bit C3(.in0(W\_PC\_plus\_1),.in1(W\_m2),.mux\_out(W\_m4),.select(PCsrc));  Mux\_32\_bit C4(.in0(W\_m4),.in1(W\_jump\_PC),.mux\_out(W\_PC\_in),.select(jump));  Adder32Bit C5(.input1(W\_PC\_plus\_1),.input2(W\_Branch\_add),.out(W\_m2));  Instruction\_Memory C6(.read\_address(W\_PC\_out),.instruction(instruction),.reset(reset));  Register\_File\_16bit C7(.clk(clk),.read\_addr\_1(instruction[25:21]),.read\_addr\_2(instruction[20:16]),.write\_addr(W\_m3),.write\_data(W\_MemtoReg),.RegWrite(RegWrite),.read\_data\_1(W\_RD1),.read\_data\_2(W\_RD2));  Sign\_Extension\_16 C8(.sign\_in(instruction[15:0]),.sign\_out(W\_Branch\_add));  Mux\_16\_bit C9(.in0(W\_RD2),.in1(instruction[15:0]),.mux\_out(W\_m1),.select(ALUsrc));  Mux\_16\_bit C10(.in0(W\_RD1),.in1(instruction[15:0]),.mux\_out(W\_m5),.select(shift));  ALU\_16bit C11(.ALU\_Sel(ALUop),.A(W\_m5),.B(W\_m1),.ALU\_Out(W\_ALUout),.zero(zero));  Data\_Memory\_16bit C12(.clk(clk),.addr(W\_ALUout),.write\_data(W\_RD2),.read\_data(W\_RDm),.MemRead(MemRead),.MemWrite(MemWrite));  Mux\_16\_bit C13(.in0(W\_ALUout),.in1(W\_RDm),.mux\_out(W\_MemtoReg),.select(MemtoReg));  Sign\_Extension\_26 C14(.sign\_in(instruction[25:0]),.sign\_out(W\_jump\_PC));  Control C15(.clk(clk),.Op\_intstruct(instruction[31:26]),.ints\_function(instruction[5:0]), .RegDst(RegDst),.Branch(Branch),.MemRead(MemRead),.MemtoReg(MemtoReg),.ALUOp(ALUop),.MemWrite(MemWrite),.ALUSrc(ALUsrc),.RegWrite(RegWrite),.Zero(zero),.Jump(jump),.Shift(shift));  Mux\_5\_bit C16(.in0(instruction[20:16]),.in1(instruction[15:11]),.mux\_out(W\_m3),.select(RegDst));  and C17(PCsrc,Branch,zero);  endmodule  // Control  module Control(clk, Op\_intstruct, ints\_function, RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite, Zero, Jump,Shift);  input clk,Zero;  input [5:0] ints\_function;  input [5:0] Op\_intstruct;  output reg RegDst,Branch,MemRead,MemtoReg,Jump,MemWrite,ALUSrc,RegWrite,Shift;  output reg [3:0] ALUOp;  always @(\*)  begin  case (Op\_intstruct)  6'b000000: begin // R-Type Instruction  RegDst = 1;  Jump = 0;  Branch = 0;  MemRead = 0;  MemtoReg = 0;  MemWrite = 0;  ALUSrc = 0;  RegWrite = 1;  Shift =0;  case (ints\_function)  6'b100000: ALUOp = 4'b0000; // Addition  6'b100010: ALUOp = 4'b0001; // Subtraction  6'b011000: ALUOp = 4'b0010; // Multiplication  6'b011010: ALUOp = 4'b0011; // Division  6'b100100: ALUOp = 4'b0100; // Logical AND  6'b100101: ALUOp = 4'b0101; // Logical OR  6'b100110: ALUOp = 4'b0110; // Logical XOR  6'b100111: ALUOp = 4'b0111; // Logical NOR  6'b101000: ALUOp = 4'b1000; // Logical NAND  6'b101010: ALUOp = 4'b1001; // Logical XNOR  6'b000000: begin // Logical Shift Left  ALUOp = 4'b1010;  Shift = 1;  end  6'b000010: begin // Logical Shift Right  ALUOp = 4'b1011;  Shift = 1;  end  6'b111000: begin // Rotate Left (ROL)  ALUOp = 4'b1100;  Shift = 1;  end  6'b110000: begin // Rotate Right (ROR)  ALUOp = 4'b1101;  Shift = 1;  end  default: ALUOp = 4'b0000;  endcase  end  // I-Type Instruction  6'b000100: begin // BEQ  RegDst = 0;  Jump = 0;  Branch = (Zero == 1) ? 1 : 0;  MemRead = 0;  MemtoReg = 0;  ALUOp = 4'b1110;  MemWrite = 0;  ALUSrc = 0;  RegWrite = 0;  Shift =0;  end  6'b100011: begin // LW  RegDst = 0;  Jump = 0;  Branch = 0;  MemRead = 1;  MemtoReg = 1;  ALUOp = 4'b0000;  MemWrite = 0;  ALUSrc = 1;  RegWrite = 1;  Shift =0;  end  6'b101011: begin // SW  RegDst = 0;  Jump = 0;  Branch = 0;  MemRead = 0;  MemtoReg = 0;  ALUOp = 4'b0000;  MemWrite = 1;  ALUSrc = 1;  RegWrite = 0;  Shift =0;  end  6'b001000: begin // ADDI  RegDst = 0;  Jump = 0;  Branch = 0;  MemRead = 0;  MemtoReg = 0;  ALUOp = 4'b0000;  MemWrite = 0;  ALUSrc = 1;  RegWrite = 1;  Shift =0;  end  6'b000101: begin // BNE  RegDst = 0;  Jump = 0;  Branch = (Zero == 0) ? 1 : 0;  MemRead = 0;  MemtoReg = 0;  ALUOp = 4'b1111;  MemWrite = 0;  ALUSrc = 0;  RegWrite = 0;  Shift =0;  end  // J-Type Instruction  6'b000010: begin // JUMP  RegDst = 0;  Jump = 1;  Branch = 0;  MemRead = 0;  MemtoReg = 0;  ALUOp = 4'b0000;  MemWrite = 0;  ALUSrc = 1;  RegWrite = 0;  Shift =0;  end  6'b000011: begin // JUMP and LINK  RegDst= 1;  Jump = 1;  Branch = 0;  MemRead = 0;  MemtoReg = 1;  ALUOp = 4'b0000;  MemWrite= 0;  ALUSrc = 0;  RegWrite = 1;  Shift =0;  end  default: begin // Default case  RegDst = 0;  Branch = 0;  Jump = 0;  MemRead = 0;  MemtoReg = 0;  ALUOp = 4'b0000;  MemWrite = 0;  ALUSrc = 0;  RegWrite = 0;  Shift =0;  end  endcase  end  endmodule      // Register File 16bit  module Register\_File\_16bit (clk,read\_addr\_1, read\_addr\_2, write\_addr, write\_data, RegWrite,read\_data\_1, read\_data\_2);  input clk,RegWrite;  input [4:0] read\_addr\_1, read\_addr\_2, write\_addr;  input [15:0] write\_data;  output [15:0] read\_data\_1, read\_data\_2;  reg [15:0] Regfile [31:0];  integer i;  initial begin  for (i = 0; i < 32; i = i + 1) begin  Regfile[i] = 16'b0;  end  Regfile[19]=1; // $s3 = 1  Regfile[17]=3; // $s1 = 3  Regfile[20]=2; // $s4 = 2  Regfile[21]=5; // $s5 = 5  end  assign read\_data\_1 = Regfile[read\_addr\_1];  assign read\_data\_2 = Regfile[read\_addr\_2];  always @(posedge clk) begin  if (RegWrite) begin  Regfile[write\_addr] <= write\_data;  //$display("write\_addr=%h write\_data=%h",write\_addr,write\_data);  end  end  endmodule  // ALU 16bit  module ALU\_16bit(ALU\_Sel, A, B, ALU\_Out, zero);  input [3:0] ALU\_Sel;  input [15:0] A;  input [15:0] B;  output reg [15:0] ALU\_Out;  output reg zero;  parameter ALU\_OP\_ADD = 4'b0000, // Addition  ALU\_OP\_SUB = 4'b0001, // Subtraction  ALU\_OP\_MUL = 4'b0010, // Multiplication  ALU\_OP\_DIV = 4'b0011, // Division  ALU\_OP\_AND = 4'b0100, // Logical and  ALU\_OP\_OR = 4'b0101, // Logical or  ALU\_OP\_XOR = 4'b0110, // Logical xor  ALU\_OP\_NOR = 4'b0111, // Logical nor  ALU\_OP\_NAND = 4'b1000, // Logical nand  ALU\_OP\_XNOR = 4'b1001, // Logical xnor  ALU\_OP\_SHL = 4'b1010, // Logical shift left  ALU\_OP\_SHR = 4'b1011, // Logical shift right  ALU\_OP\_ROL = 4'b1100, // Rotate left  ALU\_OP\_ROR = 4'b1101, // Rotate right  ALU\_OP\_BEQ = 4'b1110, // Equal comparison  ALU\_OP\_BNE = 4'b1111; // Not Equal comparison  always @(\*)  begin  case(ALU\_Sel)  ALU\_OP\_ADD : begin  zero=1'bx;  ALU\_Out = A + B;  end  ALU\_OP\_SUB : ALU\_Out = A - B;  ALU\_OP\_MUL : ALU\_Out = A \* B;  ALU\_OP\_DIV : ALU\_Out = A / B;  ALU\_OP\_AND : ALU\_Out = A & B;  ALU\_OP\_OR : ALU\_Out = A | B;  ALU\_OP\_XOR : ALU\_Out = A ^ B;  ALU\_OP\_NOR : ALU\_Out = ~(A | B);  ALU\_OP\_NAND : ALU\_Out = ~(A & B);  ALU\_OP\_XNOR : ALU\_Out = ~(A ^ B);  ALU\_OP\_SHL : ALU\_Out = A<<1;  ALU\_OP\_SHR : ALU\_Out = A>>1;  ALU\_OP\_ROL : ALU\_Out = {A[6:0],A[7]};  ALU\_OP\_ROR : ALU\_Out = {A[0],A[7:1]};  ALU\_OP\_BEQ : begin  zero = (A==B)?1'b1:1'b0;  ALU\_Out = (A==B)?16'd1:16'd0;  end  ALU\_OP\_BNE : begin  zero = (A!=B)?1'b1:1'b0;  // zero here is different from the above zero  // zero here is equal to 1, when A is different from B  // zero here is equal to 0, when A is the same as B  ALU\_Out = (A!=B)?16'd1:16'd0;  end  endcase  end  endmodule  // Sign Extension 16 bit  module Sign\_Extension\_16 (sign\_in, sign\_out);  parameter N = 16;  input [N-1:0] sign\_in;  output [31:0] sign\_out;  assign sign\_out = {{(32-N){sign\_in[N-1]}}, sign\_in};  endmodule  // Sign Extension 26 bit  module Sign\_Extension\_26 (sign\_in, sign\_out);  parameter N = 26;  input [N-1:0] sign\_in;  output [31:0] sign\_out;  assign sign\_out = {{(32-N){sign\_in[N-1]}}, sign\_in};  endmodule  // Adder 32 bit  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  // Program Counter  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) begin  if(reset)  PC\_out <= 32'b0;  else  PC\_out <= PC\_in;  end  endmodule  // Mux 32\_bit  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  // Mux 16 bit  module Mux\_16\_bit (in0, in1, mux\_out, select);  parameter N=16;  input [N-1:0] in0, in1;  output [N-1:0] mux\_out;  input select;  assign mux\_out = select ? in1 : in0;  endmodule  // Mux 5 bit  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  //Data Memory 16 bit  module Data\_Memory\_16bit (clk,addr,write\_data,read\_data,MemRead,MemWrite);  input clk;  input [15:0] addr;  input [15:0] write\_data;  output reg [15:0] read\_data;  input MemRead;  input MemWrite;  integer i;  reg [15:0] DMemory [0:255];  initial begin  for (i = 0; i < 256; i = i + 1) begin  DMemory[i] = 16'h0000;  end  end  always @(posedge clk) begin  if (MemWrite) begin  DMemory[addr] <= write\_data;  //$display("Memory Write: Address=%h Data=%h", addr, write\_data);  end  end  always @(\*) begin  if (MemRead)  read\_data = DMemory[addr];  else  read\_data = 16'h0000;  end  endmodule  // Instruction Memory  module Instruction\_Memory (read\_address, instruction, reset);  input reset;  input [31:0] read\_address;  output [31:0] instruction;  reg [31:0] Imemory [256:0];  integer i;  assign instruction = Imemory[read\_address];  always @(posedge reset)  begin  for (i=0; i<32; i=i+1)  begin  Imemory[i] = 32'b0;  end  /\*Test R-type\*/  // add $s6, $s5, $s4 R22 = R21 + R20 = 0x7 (0x5 + 0x2)  Imemory[0] = 32'b000000\_10101\_10100\_10110\_00000\_100000;  // sub $s6, $s5, $s4 R22 = R21 - R20 = 0x3 (0x5 - 0x2)  Imemory[1] = 32'b000000\_10101\_10100\_10110\_00000\_100010;  // mult $s5, $s4 R21 \* R20 = 0xA (0x5 \* 0x2)  Imemory[2] = 32'b000000\_10101\_10100\_00000\_00000\_011000;  // div $s5, $s4 R21 / R20 = 0x2 (0x5 / 0x2)  Imemory[3] = 32'b000000\_10101\_10100\_00000\_00000\_011010;  // and $s6, $s5, $s4 R22 = AND(R21, R20) = 0x0 (0x5 AND 0x2)  Imemory[4] = 32'b000000\_10101\_10100\_10110\_00000\_100100;  // or $s6, $s5, $s4 R22 = OR(R21, R20) = 0x7 (0x5 OR 0x2)  Imemory[5] = 32'b000000\_10101\_10100\_10110\_00000\_100101;  // xor $s6, $s5, $s4 R22 = XOR(R21, R20) = 0x7 (0x5 XOR 0x2)  Imemory[6] = 32'b000000\_10101\_10100\_10110\_00000\_100110;  // nor $s6, $s5, $s4 R22 = NOR(R21, R20) = 0xFFFFFFF8 (NOR of 0x5 and 0x2)  Imemory[7] = 32'b000000\_10101\_10100\_10110\_00000\_100111;  // nand $s6, $s5, $s4 R22 = NAND(R21, R20) = 0xFFFFFFFD (NAND of 0x5 and 0x2)  Imemory[8] = 32'b000000\_10101\_10100\_10110\_00000\_101000;  // xnor $s6, $s5, $s4 R22 = XNOR(R21, R20) = 0xFFFFFFFE (XNOR of 0x5 and 0x2)  Imemory[9] = 32'b000000\_10101\_10100\_10110\_00000\_101010;  // sll $s6, $s5, $s4 R22 = R21 << R20 = 0x14 (0x5 << 0x2)  Imemory[10] = 32'b000000\_10101\_10100\_10110\_00000\_000000;  // srl $s6, $s5, $s4 R22 = R21 >> R20 = 0x1 (0x5 >> 0x2)  Imemory[11] = 32'b000000\_10101\_10100\_10110\_00000\_000010;  // rol $s6, $s5 R22 = R21 rotated 1 bit left = 0xA (rotate 0x5 left by 1)  Imemory[12] = 32'b000000\_10101\_00000\_10110\_00000\_111000;  // ror $s6, $s5 R22 = R21 rotated 1 bit right = 0x2 (rotate 0x5 right by 1)  Imemory[13] = 32'b000000\_10101\_00000\_10110\_00000\_110000;    /\*Test I-type\*/  // addi $s1, $s1, 0x05 R17 = 0x4E => R17 = R17 + 0x5 = 0x53  Imemory[14] = 32'b001000\_10001\_10001\_00000\_00000\_000101;  // addi $s1, $s1, 0x05 R17 = 0x53 => R17 = R17 + 0x5 = 0x58  Imemory[15] = 32'b001000\_10001\_10001\_00000\_00000\_000101;  // addi $s3, $s1, 0x10 R17 = 0x58 => R19 = R17 + 0x10 = 0x68  Imemory[16] = 32'b001000\_10001\_10011\_00000\_00000\_010000;  // sw $s3, 0x04($s1) Memory[$s1 + 0x04] = $s3 => Memory[0x58 + 0x04] = 0x68  Imemory[17] = 32'b101011\_10001\_10011\_00000\_00000\_000100;  // lw $s7, 0x04($s1) $s7 = Memory[$s1 + 0x04] = 0x68  Imemory[18] = 32'b100011\_10001\_10111\_00000\_00000\_000100;  // beq $s7, $s3, 0x08 R23 (0x68) == R19 (0x68) move to Imemory[22 + 8 + 1]  Imemory[19] = 32'b000100\_10111\_10011\_00000\_00000\_001000;  // addi $s1, $zero, 0x11 R17 = 0x11  Imemory[20] = 32'b001000\_00000\_10001\_00000\_00000\_010001;  // addi $s1, $zero, 0x22 R17 = 0x22  Imemory[21] = 32'b001000\_00000\_10001\_00000\_00000\_100010;  // addi $s1, $zero, 0x33 R17 = 0x33  Imemory[22] = 32'b001000\_00000\_10001\_00000\_00000\_110011;  // bne $s1, $s7, 0x04 (R17 (0x33) != R23 (0x68)) move to Imemory[26 + 4 + 1]  Imemory[23] = 32'b000101\_10001\_10111\_00000\_00000\_000100;  // addi $s2, $zero, 0x66 R18 = 0x66  Imemory[24] = 32'b001000\_00000\_10010\_00000\_00001\_100110;  // addi $s2, $zero, 0x77 R18 = 0x77  Imemory[25] = 32'b001000\_00000\_10010\_00000\_00001\_110111;    /\* Test J-type \*/  // Jump to Imemory[116] using j (JUMP)  Imemory[26] = 32'b000010\_00000\_00000\_00000\_00001\_110100; // j 0x74 (JUMP to Imemory[116])  // JAL instruction to target 0x74 (Jump and Link)  Imemory[27] = 32'b000011\_00000\_00000\_00000\_00001\_110100; // jal 0x74 (Jump and Link to Imemory[116])  // Another JUMP to Imemory[120] using j  Imemory[28] = 32'b000010\_00000\_00000\_00000\_00001\_111000; // j 0x78 (JUMP to Imemory[120])  // Another JAL instruction to target 0x78  Imemory[29] = 32'b000011\_00000\_00000\_00000\_00001\_111000; // jal 0x78 (Jump and Link to Imemory[120])  // JUMP back to Imemory[100] (looping behavior)  Imemory[30] = 32'b000010\_00000\_00000\_00000\_00001\_100100; // j 0x64 (JUMP to Imemory[100])  // JAL to Imemory[104] (jump and link behavior)  Imemory[31] = 32'b000011\_00000\_00000\_00000\_00001\_101000; // jal 0x68 (Jump and Link to Imemory[104])  // JUMP again to Imemory[108]  Imemory[32] = 32'b000010\_00000\_00000\_00000\_00001\_101100; // j 0x6C (JUMP to Imemory[108])  // JAL back to Imemory[112]  Imemory[33] = 32'b000011\_00000\_00000\_00000\_00001\_110000; // jal 0x70 (Jump and Link to Imemory[112])  // Final JUMP instruction to Imemory[116]  Imemory[34] = 32'b000010\_00000\_00000\_00000\_00001\_110100; // j 0x74 (JUMP to Imemory[116])  // END of the test program  Imemory[35] = 32'b000010\_00000\_00000\_00000\_00001\_111100; // j 0x7C (JUMP to Imemory[124])  end  endmodule |
| **Testbench** |
| module tb\_Single\_Cycle\_Processor;  reg reset, clk;  wire [31:0] W\_PC\_out, instruction, W\_m2;  wire [15:0] W\_RD1, W\_RD2, W\_m1, W\_ALUout;  reg [127:0] instr\_name;      Single\_Cycle\_Processor\_16bit dut(  .reset(reset),  .clk(clk),  .W\_PC\_out(W\_PC\_out),  .instruction(instruction),  .W\_RD1(W\_RD1),  .W\_RD2(W\_RD2),  .W\_m1(W\_m1),  .W\_m2(W\_m2),  .W\_ALUout(W\_ALUout)  );      always #5 clk = ~clk;      always @(instruction) begin  case (instruction[31:26])  6'b000000: begin  case (instruction[5:0]) // funct field  6'b100000: instr\_name = "ADD ";  6'b100010: instr\_name = "SUB ";  6'b011000: instr\_name = "MULT ";  6'b011010: instr\_name = "DIV ";  6'b100100: instr\_name = "AND ";  6'b100101: instr\_name = "OR ";  6'b100110: instr\_name = "XOR ";  6'b100111: instr\_name = "NOR ";  6'b101000: instr\_name = "NAND ";  6'b101010: instr\_name = "XNOR ";  6'b000000: instr\_name = "SLL ";  6'b000010: instr\_name = "SRL ";  6'b111000: instr\_name = "ROL ";  6'b110000: instr\_name = "ROR ";  default: instr\_name = "UNKNOWN\_R ";  endcase  end  6'b000100: instr\_name = "BEQ ";  6'b100011: instr\_name = "LW ";  6'b101011: instr\_name = "SW ";  6'b001000: instr\_name = "ADDI ";  6'b000101: instr\_name = "BNE ";  6'b000010: instr\_name = "J ";  6'b000011: instr\_name = "JAL ";  default: instr\_name = "UNKNOWN ";  endcase  end    always @(posedge clk) begin  $display($time,  " reset=%b | clk=%b | PC=%4d | Inst=%-10s | Op=%6b | rs=%2d | rt=%2d | rd=%2d | zero=%b | Imm=%4h | ALU=%4h | RD1=%4h | RD2=%4h",  reset, clk, W\_PC\_out, instr\_name, instruction[31:26],  instruction[25:21], instruction[20:16], instruction[15:11],  dut.C11.zero, instruction[15:0], W\_ALUout, W\_RD1, W\_RD2);  end    initial begin  clk = 0;  reset = 1;  instr\_name = "";      $display("\n\t\t\t\t----------TEST SINGLE-CYCLE PROCESSOR----------\n");  $display("\n\t\t\t\t----------R-TYPE INSTRUCTIONS TEST----------\n");    #10 reset = 0;    #140 $display("\n\t\t\t\t----------I-TYPE INSTRUCTIONS TEST----------\n");      #120 $display("\n\t\t\t\t----------J-TYPE INSTRUCTIONS TEST----------\n");    #100 $finish;  end  endmodule |
|  |
|  |

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

**Table

Description automatically generated**

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**

001000 00000 10010 0000000001010101

001000 00000 10011 0000000000100010

001000 00000 10101 0000000001110111

000000 10010 10011 10100 00000 100000

000000 10010 10011 10001 00000 100010

101011 10010 10001 0000000000000010

100011 10010 10110 0000000000000010

000101 10101 10100 0000000000000101

001000 00000 11000 0000000000010000

000100 10101 10100 0000000000000011

001000 00000 11000 0000000000100000

000010 00000000000000000000000001

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**

001000 00000 10010 0000000001010101

001000 00000 10011 0000000000100010

001000 00000 10101 0000000001110111

000000 10010 10011 10100 00000 100000

000000 10010 10011 10001 00000 100010

101011 10010 10001 0000000000000010

100011 10010 10110 0000000000000010

000100 10101 10100 0000000000000101

001000 00000 11000 0000000000010000

000101 10101 10100 0000000000000011

001000 00000 11000 0000000000100000

000010 00000000000000000000000001

**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.