# DIGITAL SYSTEM DESIGN LABORATORY

# REPORT LAB 5

## SINGLE CYCLE MICROPROCESSOR DATAPATH DESIGN

Name: Bùi Gia Bảo
ID: ITITIU22019

# I. LAB OBJECTIVES

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

# II. DESCRIPTION

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

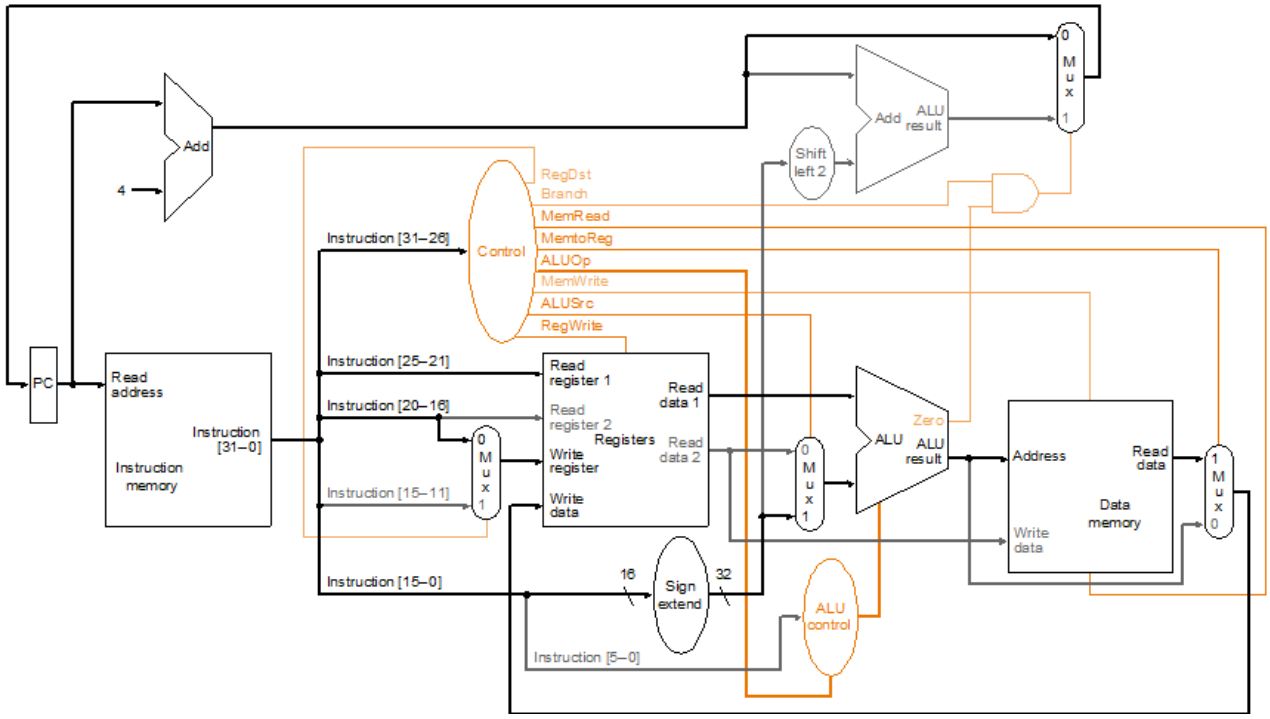



**Figure 2.1: Single Cycle Microprocessor DataPath**

# III. LAB PROCEDURE

## III.1 EXPERIMENT NO. 1

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

**register number of MIPS compiler conventions**

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

## MIPS Assembly Language Sumarize:

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

Operation code (Op code) sumarize:

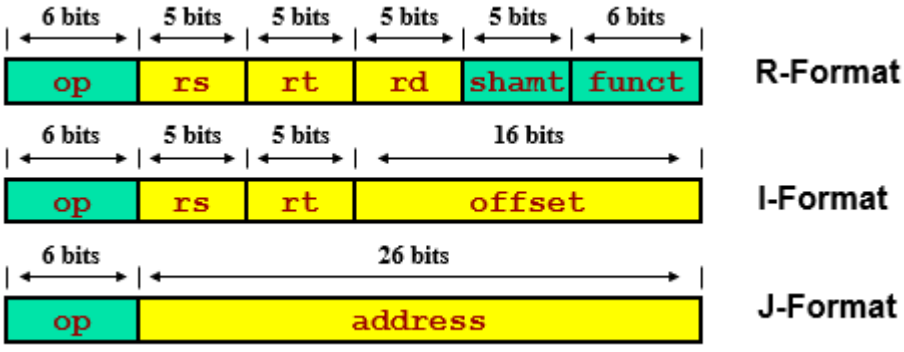
| Op | Opcode name | Value |
|---|---|---|
| 000000 | R-format | Rformat1 |
| 000010 | jmp | JUMP1 |
| 000100 | beq | BEQ1 |
| 100011 | lw | Mem1 |
| 101011 | sw | Mem1 |

## ALU opcode and Function

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

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

Instruction Formats



## III.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] |

### 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 |

## 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

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

**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:**

Assembly Code

```
add $s0, $s1, $s2
```

Field Values

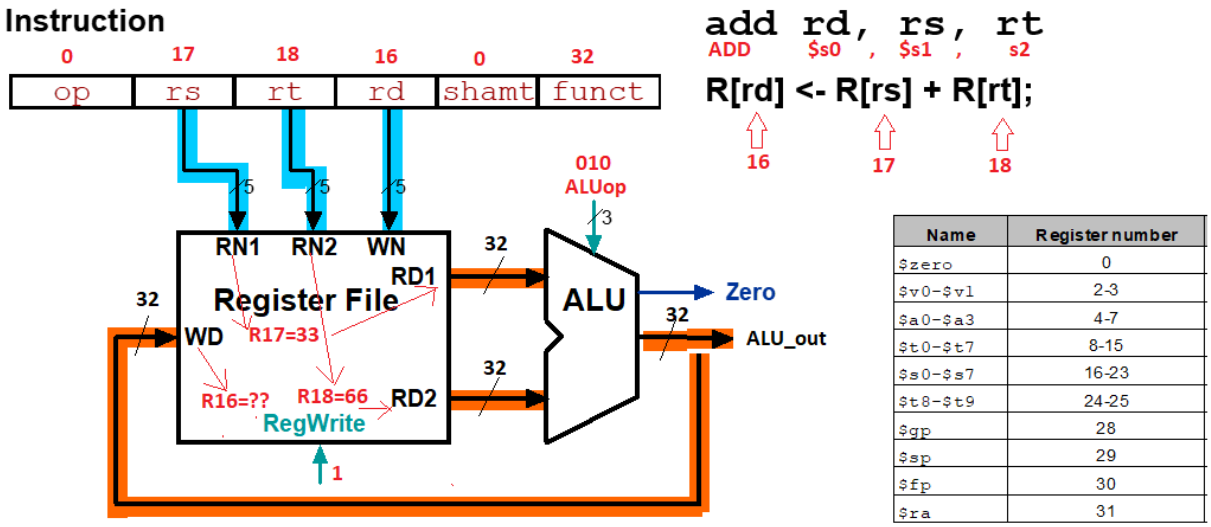
| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 0 | 17 | 18 | 16 | 0 | 32 |

Machine Code

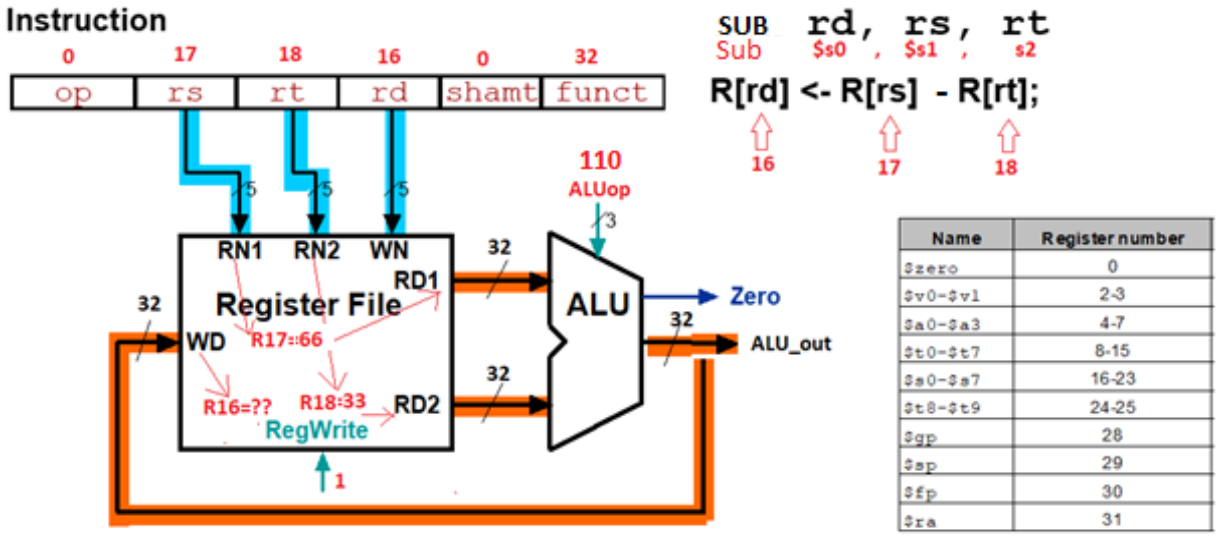
| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 000000 | 10001 | 10010 | 10000 | 00000 | 100000 |

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)




| Name | Register number |
|---|---|
| $zero | 0 |
| $v0-$v1 | 2-3 |
| $a0-$a3 | 4-7 |
| $t0-$t7 | 8-15 |
| $s0-$s7 | 16-23 |
| $t8-$t9 | 24-25 |
| $gp | 28 |
| $sp | 29 |
| $fp | 30 |
| $ra | 31 |




| Name | Register number |
|---|---|
| $zero | 0 |
| $v0-$v1 | 2-3 |
| $a0-$a3 | 4-7 |
| $t0-$t7 | 8-15 |
| $s0-$s7 | 16-23 |
| $t8-$t9 | 24-25 |
| $gp | 28 |
| $sp | 29 |
| $fp | 30 |
| $ra | 31 |

### III.2.2 CODE

```
module
lab5_ex2(KEY,SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,
HEX7);
        input[3:0] KEY;
        input  [17:0]        SW;
        output[17:0]         LEDR;
        output[7:0]   LEDG;

        output [0:6] HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7;
        wire [31:0] w_hex;
        assign LEDR =SW;

        wire [2:0] op;
```

```
        //assign op = SW[17] ? 3'b010 : 3'b110;

        DatapathR_Type_Add(.rs(SW[14:10]), .rt(SW[9:5]), .rd(SW[4:0]),
                                        .ALUop(SW[17:15]),
                                        .Zero(LEDG[7]),
                                        .ALU_out(w_hex),
                                        .reset(KEY[0]),
                                        .clk(KEY[1]),
                                        .regWrite(1'b1));

        HEX_7SEG_DECODE H0(.BIN(w_hex[3:0]), .SSD(HEX0));
        HEX_7SEG_DECODE H1(.BIN(w_hex[7:4]), .SSD(HEX1));
        HEX_7SEG_DECODE H2(.BIN(w_hex[11:8]), .SSD(HEX2));
        HEX_7SEG_DECODE H3(.BIN(w_hex[15:12]), .SSD(HEX3));
        HEX_7SEG_DECODE H4(.BIN(w_hex[19:16]), .SSD(HEX4));
        HEX_7SEG_DECODE H5(.BIN(w_hex[23:20]), .SSD(HEX5));
        HEX_7SEG_DECODE H6(.BIN(w_hex[27:24]), .SSD(HEX6));
        HEX_7SEG_DECODE H7(.BIN(w_hex[31:28]), .SSD(HEX7));


endmodule

module DatapathR_Type_Add(rs, rt, rd,ALUop, Zero ,ALU_out, reset, clk,
regWrite);
        input[4:0] rs, rt, rd;
        input[2:0] ALUop;
        output Zero;
        output[31:0] ALU_out;
        wire[31:0] w_rd_1, w_rd_2, w_rd_11, w_rd_22;
        input clk;
        input reset;
        input regWrite;

        assign w_rd_11 = 32'h0000_0003;
        assign w_rd_22 = 32'h0000_0001;

        Register_File ( .read_addr_1(rs),
                                        .read_addr_2(rt),
                                        .write_addr(rd),
                                        .read_data_1(w_rd_1),
                                        .read_data_2(w_rd_2),
```

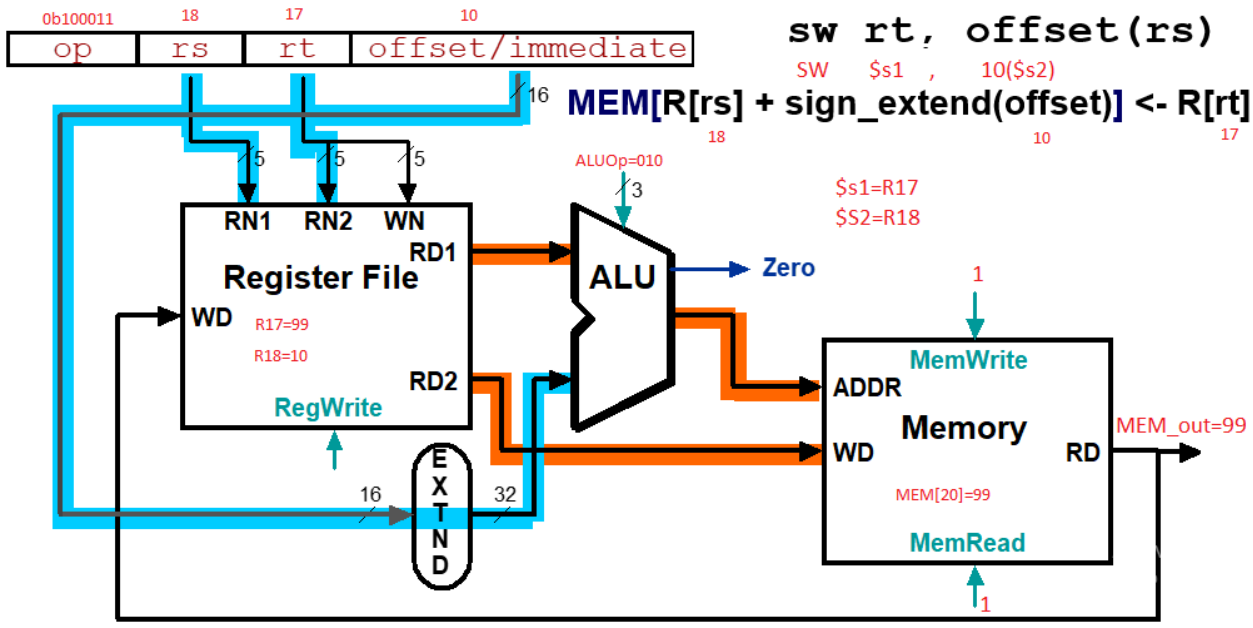
```
                                    .write_data(ALU_out),
                                    .RegWrite(regWrite),
                                    .clk(clk),
                                    .reset(reset));

        alu(    .alufn(ALUop),
                        .ra(w_rd_1),
                        .rb_or_imm(w_rd_2),
                        .aluout(ALU_out),
                        .zero(Zero));


endmodule



module HEX_7SEG_DECODE(BIN, SSD);
  input [3:0] BIN;
  output reg [0:6] SSD;
  always begin
    case(BIN)
      0:SSD=7'b0000001;
      1:SSD=7'b1001111;
      2:SSD=7'b0010010;
      3:SSD=7'b0000110;
      4:SSD=7'b1001100;
      5:SSD=7'b0100100;
      6:SSD=7'b0100000;
      7:SSD=7'b0001111;
      8:SSD=7'b0000000;
      9:SSD=7'b0001100;
      10:SSD=7'b0001000;
      11:SSD=7'b1100000;
      12:SSD=7'b0110001;
      13:SSD=7'b1000010;
      14:SSD=7'b0110000;
      15:SSD=7'b0111000;
    endcase
  end
endmodule
```

```
module Register_File (read_addr_1, read_addr_2, write_addr, read_data_1,
read_data_2, write_data, RegWrite, clk, reset);
	input [4:0] read_addr_1, read_addr_2, write_addr;
	input [31:0] write_data;
	input clk, reset, RegWrite;
	output [31:0] read_data_1, read_data_2;

	reg [31:0] Regfile [31:0];
	integer k;

	//assign Regfile[17] = 32'h33;
	//assign Regfile[18] = 32'h66;

	assign read_data_1 = RegWrite ? Regfile[read_addr_1] : 32'b0;

	assign read_data_2 = RegWrite ? Regfile[read_addr_2] : 32'b0;

	always @(posedge clk or negedge reset)
	begin
		if (reset==1'b0)
		begin
			for (k=0; k<32; k=k+1)
			begin
				Regfile[k] = 32'b0;
			end
		end

		else begin
				if (RegWrite == 1'b1) Regfile[write_addr] = write_data;
				Regfile[17] = 32'h33;
				Regfile[18] = 32'h66;
			 end


	end

endmodule


module alu(
	input [2:0] alufn,
```

```
        input [31:0] ra,
        input [31:0] rb_or_imm,
        output reg [31:0] aluout,
        output wire zero);

parameter    ALU_OP_ADD     = 3'b010,
                    ALU_OP_SUB     = 3'b110,
                    ALU_OP_AND     = 3'b000,
                    ALU_OP_OR              = 3'b001,
                    ALU_OP_NOT_A= 3'b100,
                    ALU_OP_LW              = 3'b101,
                    ALU_OP_SW              = 3'b011,
                    ALU_OP_BEQ     = 3'b111;

assign  zero = ((alufn == ALU_OP_BEQ) && (ra==rb_or_imm))?1'b1:1'b0;
always @(*)
begin
            case(alufn)
                    ALU_OP_ADD     : aluout = ra + rb_or_imm;
                    ALU_OP_SUB     : aluout = ra - rb_or_imm;
                    ALU_OP_AND     : aluout = ra & rb_or_imm;
                    ALU_OP_OR              : aluout = ra | rb_or_imm;
                    ALU_OP_NOT_A: aluout = ~ ra;
                    ALU_OP_LW              : aluout = ra + rb_or_imm;
                    ALU_OP_SW              : aluout = ra + rb_or_imm;
                    ALU_OP_BEQ     : aluout = ra - rb_or_imm;
            endcase
end
endmodule
```

## III.3 EXPERIMENT NO. 3

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




### III.3.2 CODE

```
module lab5_ex3(KEY,SW,LEDG,LEDR,HEX0,HEX1,HEX2);
        input[3:0] KEY;
        input  [17:0]          SW;
        output[17:0]           LEDR;
        output[7:0]   LEDG;

        output [0:6] HEX0,HEX1,HEX2;
        wire [7:0] w_hex,w_hex_2;
        assign LEDR =SW;

        //assign op = SW[17] ? 3'b010 : 3'b110;

        SW_datapath (       .rs(SW[2:0]),
                                .rt(SW[5:3]),
                                .offset(SW[10:6]),
                                .ALUop(SW[13:11]),
                                .MemWrite(SW[14]),
                                .MemRead(SW[15]),
```

```
        .Mem_out(w_hex),
        .reset(SW[17]),
        .clk(KEY[1]),
        .regWrite(1'b1));


    HEX_7SEG_DECODE H0(.BIN(w_hex[3:0]), .SSD(HEX0));
    HEX_7SEG_DECODE H1(.BIN(w_hex[7:4]), .SSD(HEX1));



endmodule

module SW_datapath (rs, rt, offset,ALUop,MemWrite,MemRead,Mem_out, reset,
clk, regWrite);

    input[2:0] rs, rt;
    input[2:0] ALUop;
    input[4:0] offset;
    input MemWrite, MemRead;
    output[7:0] Mem_out;

    wire[7:0] ALU_out;
    wire[7:0] w_rd_1, w_rd_2, w_rd_11, w_rd_22;
    wire[7:0] w_mrd, w_offset_ext;

    input clk;
    input reset;
    input regWrite;
```

```
Register_File (      .read_addr_1(rs),
                     .read_addr_2(rt),
                     .write_addr(rt),
                     .read_data_1(w_rd_1),
                     .read_data_2(w_rd_2),
                     .write_data(8'b00000000),
                     .RegWrite(regWrite),
                     .clk(clk),
                     .reset(reset));

alu(    .alufn(ALUop),
        .ra(w_rd_1),
        .rb_or_imm(w_offset_ext),
        .aluout(ALU_out),
        .zero(Zero));

Data_Memory (     .addr(ALU_out),
                  .write_data(w_rd_2),
                  .read_data(Mem_out),
                  .clk(clk),
                  .reset(reset),
                  .MemRead(MemRead),
                  .MemWrite(MemWrite));

Sign_Extension (.sign_in(offset), .sign_out(w_offset_ext));
```

```
endmodule



module HEX_7SEG_DECODE(BIN, SSD);
  input [3:0] BIN;
  output reg [0:6] SSD;
  always begin
    case(BIN)
      0:SSD=7'b0000001;
      1:SSD=7'b1001111;
      2:SSD=7'b0010010;
      3:SSD=7'b0000110;
      4:SSD=7'b1001100;
      5:SSD=7'b0100100;
      6:SSD=7'b0100000;
      7:SSD=7'b0001111;
      8:SSD=7'b0000000;
      9:SSD=7'b0001100;
      10:SSD=7'b0001000;
      11:SSD=7'b1100000;
      12:SSD=7'b0110001;
      13:SSD=7'b1000010;
      14:SSD=7'b0110000;
      15:SSD=7'b0111000;
    endcase
  end
endmodule
```

```
module Register_File (read_addr_1, read_addr_2, write_addr, read_data_1,
read_data_2, write_data, RegWrite, clk, reset);
    input [2:0] read_addr_1, read_addr_2, write_addr;
    input [7:0] write_data;
    input clk, reset, RegWrite;
    output [7:0] read_data_1, read_data_2;

    reg [7:0] Regfile [7:0];
    integer k;


    assign read_data_1 = RegWrite ? Regfile[read_addr_1] : 8'b0;

    assign read_data_2 = RegWrite ? Regfile[read_addr_2] : 8'b0;

    always @(posedge clk or negedge reset)
    begin
        if (reset==1'b0)
        begin
            for (k=0; k<8; k=k+1)
            begin
                Regfile[k] = 8'b0;
            end
        end

        else begin
                if (RegWrite == 1'b1) Regfile[write_addr] = write_data;
                Regfile[3] = 8'h1;
                Regfile[5] = 8'h2;
                Regfile[6] = 8'h3;
                Regfile[4] = 8'h2;
```

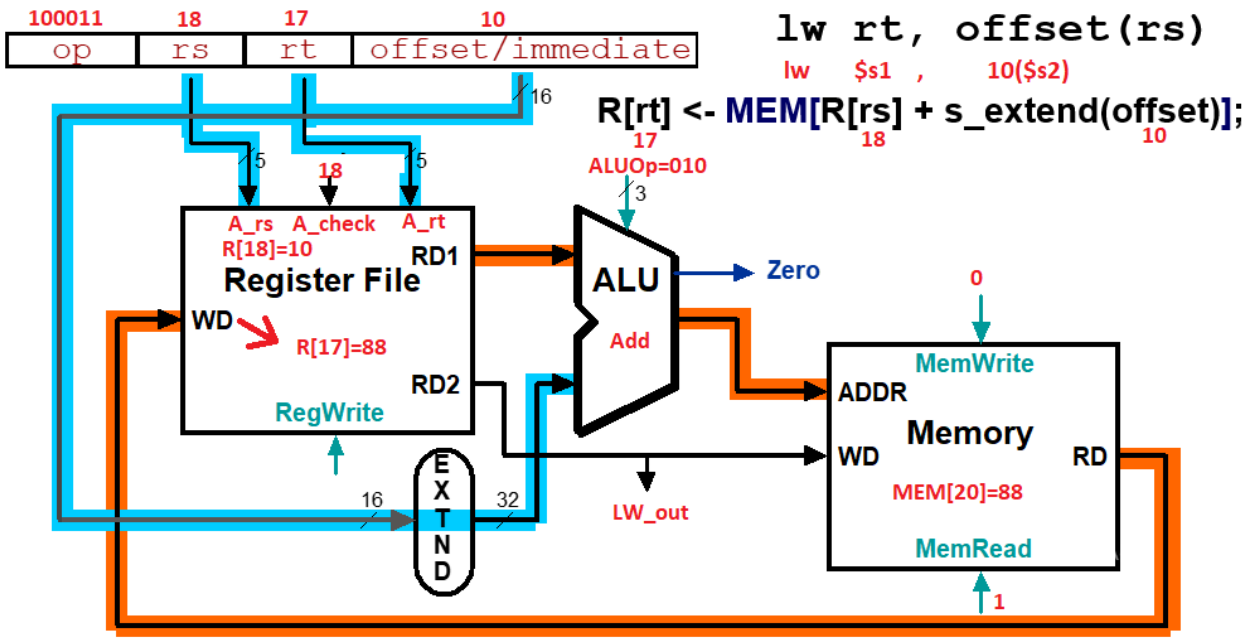
```
                end


        end

endmodule



module alu(
        input [2:0] alufn,
        input [7:0] ra,
        input [7:0] rb_or_imm,
        output reg [7:0] aluout,
        output wire zero);

        parameter   ALU_OP_ADD    = 3'b010,
                    ALU_OP_SUB    = 3'b110,
                    ALU_OP_AND    = 3'b000,
                    ALU_OP_OR           = 3'b001,
                    ALU_OP_NOT_A= 3'b100,
                    ALU_OP_LW           = 3'b101,
                    ALU_OP_SW           = 3'b011,
                    ALU_OP_BEQ    = 3'b111;

        assign  zero = ((alufn == ALU_OP_BEQ) && (ra==rb_or_imm))?1'b1:1'b0;
        always @(*)
        begin
                    case(alufn)
                            ALU_OP_ADD    : aluout = ra + rb_or_imm;
```

```
                        ALU_OP_SUB      : aluout = ra - rb_or_imm;
                        ALU_OP_AND      : aluout = ra & rb_or_imm;
                        ALU_OP_OR       : aluout = ra | rb_or_imm;
                        ALU_OP_NOT_A    : aluout = ~ ra;
                        ALU_OP_LW       : aluout = ra + rb_or_imm;
                        ALU_OP_SW       : aluout = ra + rb_or_imm;
                        ALU_OP_BEQ      : aluout = ra - rb_or_imm;
                endcase
        end
endmodule

module Data_Memory (addr, write_data, read_data, clk, reset, MemRead,
MemWrite);
        input [7:0] addr;
        input [7:0] write_data;
        output [7:0] read_data;
        input clk, reset, MemRead, MemWrite;
        reg [7:0] DMemory [7:0];
        integer k;
        assign read_data = (MemRead && ~MemWrite) ? DMemory[addr] : 8'bx;

        always @(posedge clk or negedge reset)
        begin
                if (reset == 1'b0)
                        begin
                                for (k=0; k<8; k=k+1)
  begin
                                        DMemory[k] = 8'b0;
                                end
                        end
                else
```

```
            if (MemWrite) DMemory[addr] = write_data;
        end
endmodule


module Sign_Extension (sign_in, sign_out);
        input [4:0] sign_in;
        output [7:0] sign_out;
        assign sign_out[4:0]=sign_in[4:0];
        assign sign_out[7:5]=sign_in[4]?{3{1'b1}}:3'b0;
endmodule
```

## III.4 EXPERIMENT NO. 4

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




### III.4.2 CODE

```
module lab5_ex4(KEY,SW,LEDG,LEDR,HEX0,HEX1,HEX2);
        input[3:0] KEY;
        input  [17:0]        SW;
        output[17:0]         LEDR;
        output[7:0]   LEDG;

        output [0:6] HEX0,HEX1,HEX2;
```

```
wire [7:0] w_hex,w_hex_2;
assign LEDR =SW;


LW_datapath (      .rs(SW[2:0]),
                        .rt(SW[5:3]),
                        .offset(SW[10:6]),
                        .ALUop(SW[13:11]),
                        .MemWrite(SW[14]),
                        .MemRead(SW[15]),
                        .LW_out(w_hex),
                        .reset(SW[17]),
                        .clk(KEY[1]),
                        .regWrite(1'b1));


HEX_7SEG_DECODE H0(.BIN(w_hex[3:0]), .SSD(HEX0));
HEX_7SEG_DECODE H1(.BIN(w_hex[7:4]), .SSD(HEX1));



endmodule

module LW_datapath (rs, rt, offset, ALUop,MemWrite,MemRead, LW_out, reset,
clk, regWrite);

        input[2:0] rs, rt;
        input[2:0] ALUop;
        input[4:0] offset;
        input MemWrite, MemRead;
        output[7:0] LW_out;

        wire[7:0] ALU_out;
        wire[7:0] w_rd_1, w_rd_2, w_rd_11, w_rd_22;
        wire[7:0] w_mrd, w_offset_ext;

        input clk;
        input reset;
        input regWrite;
```

```
        Register_File (     .read_addr_1(rs),
                                .read_addr_2(rt),
                                .write_addr(rt),
                                .read_data_1(w_rd_1),
                                .read_data_2(w_rd_2),
                                .write_data(LW_out),
                                .RegWrite(regWrite),
                                .clk(clk),
                                .reset(reset));

        alu(    .alufn(ALUop),
                    .ra(w_rd_1),
                    .rb_or_imm(w_offset_ext),
                    .aluout(ALU_out),
                    .zero(Zero));


        Data_Memory (   .addr(ALU_out),
                            .write_data(8'b0000000),
                            .read_data(LW_out),
                            .clk(clk),
                            .reset(reset),
                            .MemRead(MemRead),
                            .MemWrite(MemWrite));



        Sign_Extension (.sign_in(offset), .sign_out(w_offset_ext));


endmodule



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

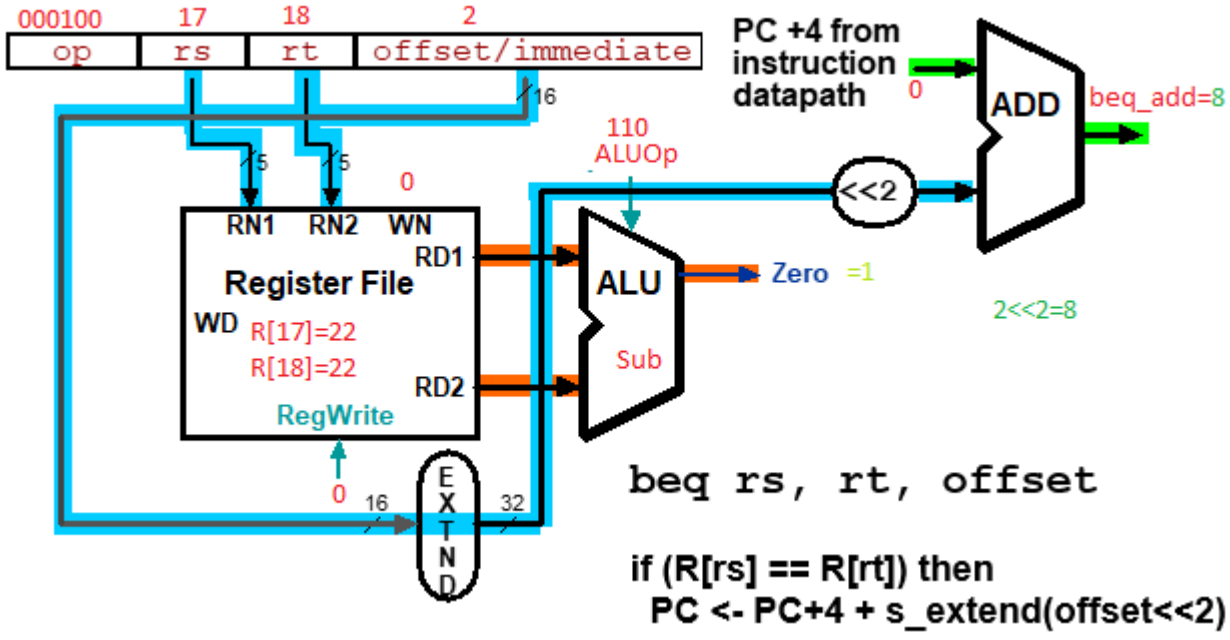
```
      1:SSD=7'b1001111;
      2:SSD=7'b0010010;
      3:SSD=7'b0000110;
      4:SSD=7'b1001100;
      5:SSD=7'b0100100;
      6:SSD=7'b0100000;
      7:SSD=7'b0001111;
      8:SSD=7'b0000000;
      9:SSD=7'b0001100;
      10:SSD=7'b0001000;
      11:SSD=7'b1100000;
      12:SSD=7'b0110001;
      13:SSD=7'b1000010;
      14:SSD=7'b0110000;
      15:SSD=7'b0111000;
    endcase
  end
endmodule

module Register_File (read_addr_1, read_addr_2, write_addr, read_data_1,
read_data_2, write_data, RegWrite, clk, reset);
        input [2:0] read_addr_1, read_addr_2, write_addr;
        input [7:0] write_data;
        input clk, reset, RegWrite;
        output [7:0] read_data_1, read_data_2;

        reg [7:0] Regfile [7:0];
        integer k;


        assign read_data_1 = RegWrite ? Regfile[read_addr_1] : 8'b0;

        assign read_data_2 = RegWrite ? Regfile[read_addr_2] : 8'b0;

        always @(posedge clk or negedge reset)
        begin
                if (reset==1'b0)
                begin
                        for (k=0; k<8; k=k+1)
                        begin
                                Regfile[k] = 8'b0;
```

```
                end
            end

            else begin
                    if (RegWrite == 1'b1) Regfile[write_addr] = write_data;

                end


    end

endmodule



module alu(
    input [2:0] alufn,
    input [7:0] ra,
    input [7:0] rb_or_imm,
    output reg [7:0] aluout,
    output wire zero);

    parameter   ALU_OP_ADD    = 3'b010,
                ALU_OP_SUB    = 3'b110,
                ALU_OP_AND    = 3'b000,
                ALU_OP_OR         = 3'b001,
                ALU_OP_NOT_A= 3'b100,
                ALU_OP_LW         = 3'b101,
                ALU_OP_SW         = 3'b011,
                ALU_OP_BEQ    = 3'b111;

    assign  zero = ((alufn == ALU_OP_BEQ) && (ra==rb_or_imm))?1'b1:1'b0;
    always @(*)
    begin
                case(alufn)
                    ALU_OP_ADD    : aluout = ra + rb_or_imm;
                    ALU_OP_SUB    : aluout = ra - rb_or_imm;
                    ALU_OP_AND    : aluout = ra & rb_or_imm;
                    ALU_OP_OR         : aluout = ra | rb_or_imm;
                    ALU_OP_NOT_A: aluout = ~ ra;
                    ALU_OP_LW         : aluout = ra + rb_or_imm;
```

```
                        ALU_OP_SW           : aluout = ra + rb_or_imm;
                        ALU_OP_BEQ      : aluout = ra - rb_or_imm;
                endcase
        end
endmodule

module Data_Memory (addr, write_data, read_data, clk, reset, MemRead,
MemWrite);
        input [7:0] addr;
        input [7:0] write_data;
        output [7:0] read_data;
        input clk, reset, MemRead, MemWrite;
        reg [7:0] DMemory [7:0];
        integer k;
        assign read_data = (MemRead && ~MemWrite) ? DMemory[addr] : 8'bx;

        always @(posedge clk or negedge reset)
        begin
                if (reset == 1'b0)
                        begin
                                for (k=0; k<8; k=k+1)
  begin
                                        DMemory[k] = 8'b0;
                                  end
                        end
                else begin
                        if (MemWrite) DMemory[addr] = write_data;
                        DMemory[5] = 8'h8;
                        DMemory[6] = 8'h99;
                        DMemory[7] = 8'h55;
                end
        end
endmodule

module Sign_Extension (sign_in, sign_out);
        input [4:0] sign_in;
        output [7:0] sign_out;
        assign sign_out[4:0]=sign_in[4:0];
        assign sign_out[7:5]=sign_in[4]?{3{1'b1}}:3'b0;
endmodule
```

## III.5 EXPERIMENT NO. 5

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




### III.5.2 CODE

module beq_Datapath (rs,rt,offset,ALUOp,PC_plus_4, Zero,beq_add)

endmodule

### III.5.3 LAB ASSIGNMENT

1) Write Verilog code to implement beq_Datapath module
2) Write testbenches to verify beq_Datapath module, simulate and verify the output data.

### III.10.3 LAB ASSIGNMENT

Write testbenches to verify above blocks and attach waveforms.

## 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.