Practical – 10

Aim: Implement 16-bit single-cycle MIPS processor in Verilog HDL.

- **VERILOG**
  - Verilog is a **HARDWARE DESCRIPTION LANGUAGE (HDL)**.
  - It is a language used for describing a digital system like a network switch or a microprocessor or a memory or a flip−flop.
  - It means, by using a HDL we can describe any digital hardware at any level.
  - Designs, which are described in HDL are independent of technology, very easy for designing and debugging, and are normally more useful than schematics, particularly for large circuits.

- **MIPS Processor**
  - MIPS (**Microprocessor without Interlocked Pipelined Stages**) is a **reduced instruction set computer (RISC) instruction set architecture (ISA)** developed by MIPS Computer Systems, now MIPS Technologies, based in the United States.
  - MIPS processors are used in embedded systems such as residential gateways and routers.

- **16-bit single cycle MIPS Processor**
  - The **Instruction Format** and **Instruction Set Architecture** for 16-bit simple cycle MIPS are as follows :

Name	Fields					Comments
Field size	3 bits	3 bits	3 bits	3 bits	4 bits	All MIPS-L instructions 16 bits
R-format	op	rs	rt	rd	funct	Arithmetic instruction format
I-format	op	rs	rt	Address/immediate		Transfer, branch, immediate format
J-format	op	target address				Jump instruction format

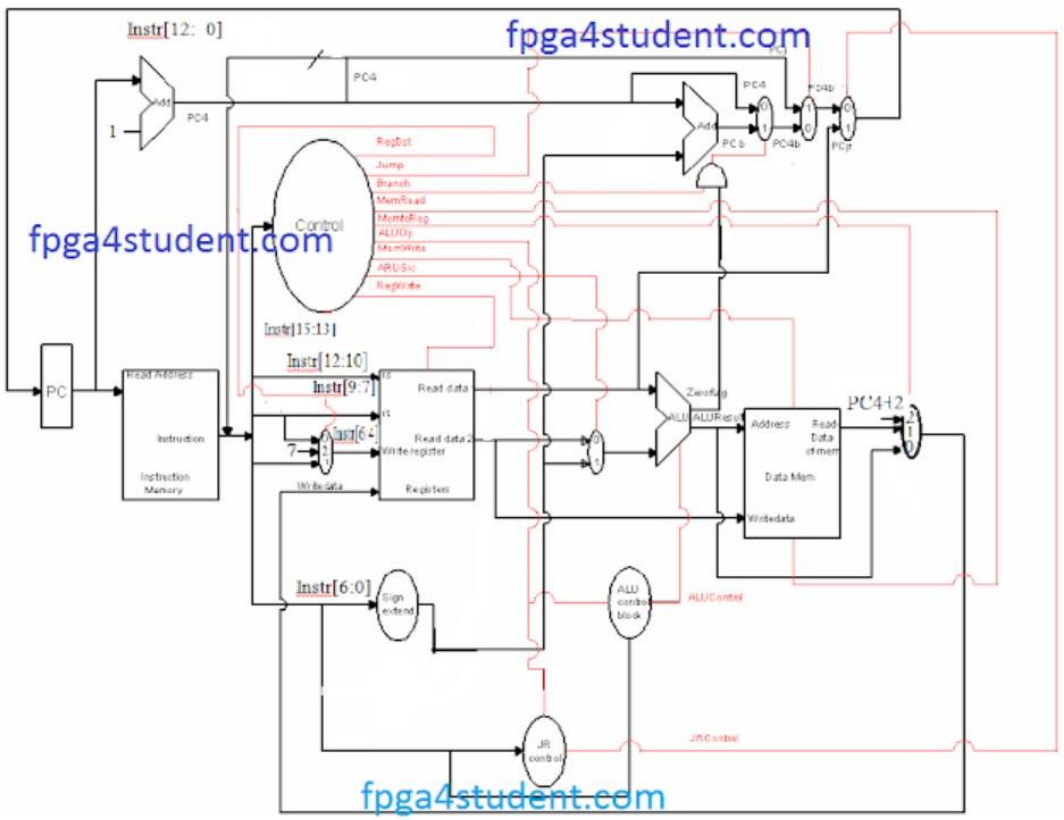
Name	Format	Example					Comments
		3 bits	3 bits	3 bits	3 bits	4 bits	
add	R	0	2	3	1	0	add $1,$2,$3
sub	R	0	2	3	1	1	sub $1,$2,$3
and	R	0	2	3	1	2	and $1,$2,$3
or	R	0	2	3	1	3	or $1,$2,$3
slt	R	0	2	3	1	4	slt $1,$2,$3
jr	R	0	7	0	0	8	jr $7
lw	I	4	2	1	7		lw $1, 7 ($2)
sw	I	5	2	1	7		sw $1, 7 ($2)
beq	I	6	1	2	7		beq $1,$2, 7
addi	I	7	2	1	7		addi $1,$2, 7
j	J	2	500				j 1000
jal	J	3	500				jal 1000
slti	I	1	2	1	7		slti $1,$2, 7

- **Below is the description of instructions being implemented in Verilog :**
    1. Add : R[rd] = R[rs] + R[rt]
    2. Subtract : R[rd] = R[rs] - R[rt]
    3. And: R[rd] = R[rs] & R[rt]
    4. Or : R[rd] = R[rs] | R[rt]
    5. SLT: R[rd] = 1 if R[rs] < R[rt] else 0
    6. Jr: PC=R[rs]
    7. Lw: R[rt] = M[R[rs]+SignExtImm]
    8. Sw : M[R[rs]+SignExtImm] = R[rt]
    9. Beq : if(R[rs]==R[rt]) PC=PC+1+BranchAddr
    10. Addi: R[rt] = R[rs] + SignExtImm
    11. J : PC=JumpAddr
    12. Jal : R[7]=PC+2;PC=JumpAddr
    13. SLTI: R[rt] = 1 if R[rs] < imm else 0

- **Control Unit Design :**

Control signals									
Instruction	**Reg Dst**	**ALU Src**	**Memto Reg**	**Reg Write**	**MemRead**	**Mem Write**	**Branch**	**ALUOp**	**Jump**
R-type	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**00**	**0**
LW	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**11**	**0**
SW	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**11**	**0**
addi	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**11**	**0**
beq	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**01**	**0**
j	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**00**	**1**
jal	**2**	**0**	**2**	**1**	**0**	**0**	**0**	**00**	**1**
slti	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**10**	**0**

ALU Control				
ALU op	**Function**	**ALUcnt**	**ALU Operation**	**Instruction**
11	**xxxx**	**000**	**ADD**	**Addi,lw,sw**
01	**xxxx**	**001**	**SUB**	**BEQ**
00	**00**	**000**	**ADD**	**R-type: ADD**
00	**01**	**001**	**SUB**	**R-type: sub**
00	**02**	**010**	**AND**	**R-type: AND**
00	**03**	**011**	**OR**	**R-type: OR**
00	**04**	**100**	**slt**	**R-type: slt**
10	**xxxxxx**	**100**	**slt**	**i-type: slti**



- **Verilog Code for 16-bit single cycle MIPS Processor :**

```
//fpga4student.com: FPGA projects, Verilog projects, VHDL projects

// Verilog project: Verilog code for 16-bit MIPS Processor //
Verilog code for 16 bit single cycle MIPS CPU module
mips_16( input clk,reset, output[15:0] pc_out, alu_result

//,reg3,reg4

);

reg[15:0] pc_current; wire
signed[15:0] pc_next,pc2;

wire [15:0] instr;

wire[1:0] reg_dst,mem_to_reg,alu_op;

wire jump,branch,mem_read,mem_write,alu_src,reg_write     ;

wire     [2:0]     reg_write_dest; wire
[15:0] reg_write_data; wire     [2:0]
reg_read_addr_1; wire     [15:0]
reg_read_data_1; wire     [2:0]
reg_read_addr_2; wire     [15:0]
reg_read_data_2;

wire [15:0] sign_ext_im,read_data2,zero_ext_im,imm_ext;
wire JRControl; wire [2:0] ALU_Control; wire [15:0]
ALU_out; wire zero_flag;

wire signed[15:0] im_shift_1, PC_j, PC_beq, PC_4beq,PC_4beqj,PC_jr;
wire beq_control; wire [14:0] jump_shift_1; wire [15:0]mem_read_data;
wire [15:0] no_sign_ext; wire sign_or_zero;

// PC

always @(posedge clk or posedge reset) begin

if(reset) pc_current <=
16'd0; else pc_current <=
pc_next;

end

// PC + 2 assign pc2 = pc_current +
16'd2; // instruction memory

instr_mem instrucion_memory(.pc(pc_current),.instruction(instr)); //
jump shift left 1

assign jump_shift_1 = {instr[13:0],1'b0};

// control unit
```

```
control control_unit(.reset(reset),.opcode(instr[15:13]),.reg_dst(reg_dst)

,.mem_to_reg(mem_to_reg),.alu_op(alu_op),.jump(jump),.branch(branch),.mem_read(mem_r ead),

.mem_write(mem_write),.alu_src(alu_src),.reg_write(reg_write),.sign_or_zero(sign_or_zero));

// multiplexer regdest

assign reg_write_dest = (reg_dst==2'b10) ? 3'b111: ((reg_dst==2'b01) ? instr[6:4] :instr[9:7]);

// register file

assign reg_read_addr_1 = instr[12:10]; assign
reg_read_addr_2 = instr[9:7];

register_file reg_file(.clk(clk),.rst(reset),.reg_write_en(reg_write),

.reg_write_dest(reg_write_dest),

.reg_write_data(reg_write_data),

.reg_read_addr_1(reg_read_addr_1),

.reg_read_data_1(reg_read_data_1),

.reg_read_addr_2(reg_read_addr_2),

.reg_read_data_2(reg_read_data_2));

//.reg3(reg3),

//.reg4(reg4)); // sign extend assign sign_ext_im
= {{9{instr[6]}},instr[6:0]}; assign zero_ext_im
= {{9{1'b0}},instr[6:0]};

assign imm_ext = (sign_or_zero==1'b1) ? sign_ext_im : zero_ext_im;

// JR control

JR_Control JRControl_unit(.alu_op(alu_op),.funct(instr[3:0]),.JRControl(JRControl)); //
ALU control unit

ALUControl

ALU_Control_unit(.ALUOp(alu_op),.Function(instr[3:0]),.ALU_Control(ALU_Control));

// multiplexer alu_src

assign read_data2 = (alu_src==1'b1) ? imm_ext : reg_read_data_2;

// ALU alu

alu_unit(.a(reg_read_data_1),.b(read_data2),.alu_control(ALU_Control),.result(ALU_out),.zero(
zero_flag)); // immediate shift 1

assign im_shift_1 = {imm_ext[14:0],1'b0};

//
```

```
assign no_sign_ext = ~(im_shift_1) + 1'b1;
// PC beq add
assign PC_beq = (im_shift_1[15] == 1'b1) ? (pc2 - no_sign_ext): (pc2 +im_shift_1);
// beq control
assign beq_control = branch & zero_flag;
// PC_beq
assign PC_4beq = (beq_control==1'b1) ? PC_beq : pc2;
// PC_j
assign PC_j = {pc2[15],jump_shift_1};
// PC_4beqj
assign PC_4beqj = (jump == 1'b1) ? PC_j : PC_4beq;
// PC_jr
assign PC_jr = reg_read_data_1;
// PC_next
assign pc_next = (JRControl==1'b1) ? PC_jr : PC_4beqj;
// data memory
data_memory datamem(.clk(clk),.mem_access_addr(ALU_out),
.mem_write_data(reg_read_data_2),.mem_write_en(mem_write),.mem_read(mem_read),
.mem_read_data(mem_read_data));
// write back
assign reg_write_data = (mem_to_reg == 2'b10) ? pc2:((mem_to_reg == 2'b01)?
mem_read_data: ALU_out);
// output assign pc_out =
pc_current;
assign alu_result = ALU_out;
endmodule
```