Course Name: Computer Architecture	**Course Code:** CMPE-421L
Assignment Type: Lab	**Dated:** 2nd October 2023
Semester: 7th	**Session:** 2020
Lab/Project/Assignment #: 5	**CLOs to be covered: CLO 2**
Lab Title: RISC-V Processor Design	**Teacher Name:** Engr. Afeef Obaid

Lab Evaluation

CLO 2	Understand the basics of RISC-V architecture, its assembly & design of basic Datapath components of a single cycle RISC-V processor.					
Levels (Marks)	**Level1**	**Level2**	**Level3**	**Level4**	**Level5**	**Level6**
(10)						
					Total	**/10**

Rubrics for Current Lab Evaluation

Scale	**Marks**	**Level**	**Rubric**
Excellent	**9-10**	L1	Submitted all lab tasks, BONUS task, have good understanding.
Very Good	**7-8**	L2	Submitted the lab tasks but have good understanding
Good	**5-6**	L3	Submitted the lab tasks but have weak understanding.
Basic	**3-4**	L4	Submitted the lab tasks but have no understanding.
Barely Acceptable	**1-2**	L5	Submitted only one lab task.
Not Acceptable	**0**	L6	Did not attempt

Lab # 5

Lab Goals

By reading this manual, students will be able to:

- Understand RISC-V (RV) Processor Design.
- Understand different phases of Processor Design.
- To implement R type instructions.

Equipment Required

- Computer system with ModelSim or Xcelium, installed on it.

RISC-V Processor Design

As discussed, a RISC V instruction normally goes through different phases starting with the instruction fetch phase. We will implement the necessary building blocks to perform the actions required at each phase.

Instruction Memory

Instruction memory can be viewed as a read-only memory buffer with address inputs and data outputs. Since we are following the RV 32I instruction set, all the instructions are encoded using 32-bit machine codes. As a result, the data bus width will be 32-bits. The address bus size depends on the size of the memory, however the addresses generated by the processor ALU will be 32-bits.

The following code example illustrates the instantiation of instruction memory and its initialization using the user provided file.

```
// Instruction memory instantiation and initialization XLEN is 32
logic   [`XLEN-1:0]          inst_memory[`IMEM_SIZE];

initial
begin
     // Reading the contents of imem.txt file to memory variable
     $readmemh("imem.txt", inst_memory);
end
```

Listing 4.1. Instruction memory instantiation and initialization.

Task:

- The above implementation makes the instruction memory word addressable. How would you modify it to byte-addressable memory? What will be the impact on the memory size?

Fetch Phase (F)

An instruction is fetched by providing an appropriate address to the instruction memory. The sample code that can perform this task is listed below.

```
// Asynchronous read operation of instruction memory
assign inst_machine_code = inst_memory[address];
```

Listing 4.2. Reading instruction memory.

Decode Phase (D)

The three key operations performed at this stage are:
1) Generating the control signals
2) Preparing the immediate values
3) Fetching the operands from the register file

Generating the Control Signals – The Decoder

The decoder is responsible to decode the machine code of an instruction and

```
case (instr_opcode)
    OPCODE_ARITH_INST  : begin

        id2exe_ctrl.alu_opr1_sel     = ALU_OPR1_REG;
        id2exe_ctrl.alu_opr2_sel     = ALU_OPR2_REG;

                       ...

        case (funct7_opcode)
            7'b0000000 : begin
                case (funct3_opcode)
                   3'b000 : id2exe_ctrl.alu_ops = ALU_OPS_ADD; // ADD
                   3'b001 : id2exe_ctrl.alu_ops = ALU_OPS_SLL; // Shift left logical
                        ...
```

Listing 4.3. Reading instruction memory.

Register File

A register file consists of thirty-two 32-bit registers which can be read and written by supplying the address of the registers. The register file is going to have two read ports and one write port. So the register file is going to take three addresses i.e two registers to be read and one write register along with the control signal and the data to be written for the write operation at the input and it is going to provide the value of the read operation at the output.

Following figure shows the block diagram of the register file.

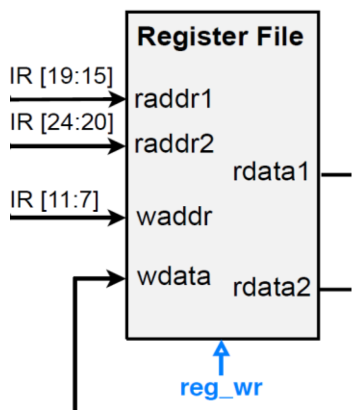

Figure 4.1. Block diagram of register file.

For designing the register file first we are going to create the header file which is going to define the parameters for the size of the register file, the width of the register file and the size of the registers inside the register file.

```
// UETRV_PCore_defs.svh
`ifndef UETRV_PCORE_SVH
`define UETRV_PCORE_SVH

//============================ CORE PARAMETERS =======================//

// Width of main registers and buses
`define XLEN               32

`define RF_AWIDTH           5
`define RF_SIZE            32

`endif // UETRV_PCORE_SVH
```

Listing 4.4. Header File Defining the core parameters

The register file is going to be instantiated as a multidimensional array along with the local signals which are going to check the validity of the input addresses for the read operation and the write operations. Asynchronous read operation for two register operands. The write operation is going to be performed based on the valid signal for the write operation.

```
// register file instantiation
logic   [`XLEN-1:0]         register_file[`RF_SIZE];

// control signals for validity of register file read/write operations
assign  rs1_addr_valid   = |id2rf_rs1_addr_i;
assign  rs2_addr_valid   = |id2rf_rs2_addr_i;
assign  rf_wr_valid      = |id2rf_rd_addr_i & id2rf_rd_wr_req_i;

// asynchronous read operation for two register operands
assign  rf2id_rs1_data_o = ( rs1_addr_valid )
                         ? register_file[id2rf_rs1_addr_i]
                         : '0;
assign  rf2id_rs2_data_o = (rs2_addr_valid)
                         ? register_file[id2rf_rs2_addr_i]
                         : '0;
```

Listing 4.5. Register file instantiation and asynchronous read operation.

```
// write operation
always_ff @( posedge clk) begin
    if ( rst_n ) begin
        register_file <= '{default: '0};
    end else if ( rf_wr_valid ) begin
```

```
        register_file[id2rf_rd_addr_i] <= id2rf_rd_data_i;
    end
end

endmodule : reg_file
```

Listing 4.6. Register file synchronous write operation.

Task:

- Perform the verification of the register file code shared in the manual by initializing the register file with a memory file using the $readmemh command. Use of $readmemh has been explained in the context of instruction memory.

Execution Phase (E) – The ALU

The required operation by the instruction is performed in the execution phase. The first step is to prepare the operands followed by the implementation of the operation to be performed by the ALU. The Listing 4.7 illustrates the preparation of the operands, Listing 4.8 shows the ALU operation implementation.

```
// Prepare the two operands
always_comb begin
   alu_operand_1 = (id2exe_ctrl.alu_opr1_sel  ==   ALU_OPR1_PC)
                 ? (id2exe_data.pc)                         // Operand 1 is PC
                 : (id2exe_data.rs1_data);                  // Operand 1 is register

   alu_operand_2 = (id2exe_ctrl.alu_opr2_sel  ==   ALU_OPR2_IMM)
                 ? (id2exe_data.imm)                        // Operand 2 is immediate
                 : (id2exe_data.rs2_data);                  // Operand 2 is register
end
                         ...
```

Listing 4.7. Preparing the operands for the ALU.

```
always_comb begin
   exe2mem_alu_result   =  '0;
   case (alu_operator)
      ALU_OPS_AND : begin
         exe2mem_alu_result = alu_operand_1 & alu_operand_2;
      end
                                  ...
```

Listing 4.8. Implementing the ALU operations.

Instruction	Operation
add rd, rs1, rs2	rd = rs1 + rs2
sub rd, rs1, rs2	rd = rs1 - rs2

sll rd, rs1, rs2	rd = rs1 << rs2[4:0]
slt rd, rs1, rs2	rd = $signed(rs1) < $signed(rs2)
sltu rd, rs1, rs2	rd = rs1 < rs2
xor rd, rs1, rs2	rd = rs1 ^ rs2
srl rd, rs1, rs2	rd = rs1 >> rs2[4:0]
sra rd, rs1, rs2	rd = rs1 >>> rs2[4:0]
or rd, rs1, rs2	rd = rs1 | rs2
and rd, rs1, rs2	rd = rs1 & rs2

Table 4.1. R-Type Instructions and operations.

Data Memory Access Phase (M)

For R-type instructions no memory access is required and as a result no activity is performed in this phase. We will discuss the data memory access phase and the required interface for this purpose in the next lab handout.

Writeback Phase (W)

During the write back phase the result of execution is written back to the destination register. The following code illustrates this step.

```
// Writeback MUX for output signal selection
always_comb begin
      wb2id_rd_data  = '0;
       case (mem2wb_ctrl.rd_wb_sel)
          RD_WB_ALU : begin
              wb2id_rd_data  =  mem2wb_data.alu_result;
          End

                    ...

          RD_WB_MEM : begin
              wb2id_rd_data  =  mem2wb_data.dmem_rdata;
          end
                                ...

          default :    wb2id_rd_data  = '0; // default case
      endcase
 end
```

Listing 4.9. The writeback operation to register file.

Tasks

- Implement the R-type instructions using the following datapath.

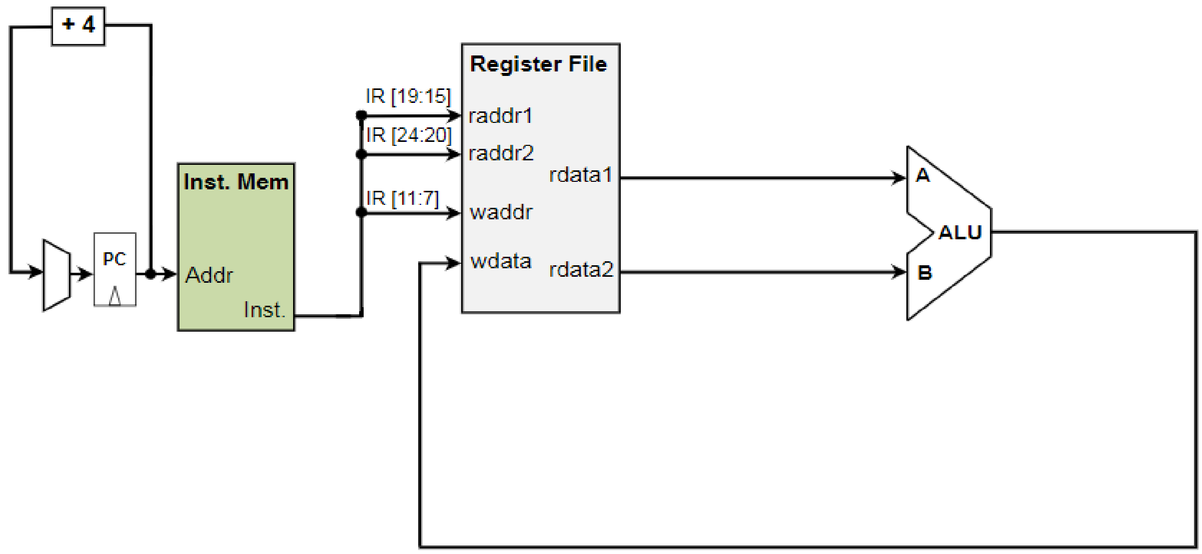

Figure 4.2. R-Type Instruction Data Path.

- Initialize the register file registers using $readmemh and write a simple assembly program that can test and verify the working of the implemented R-type instructions.