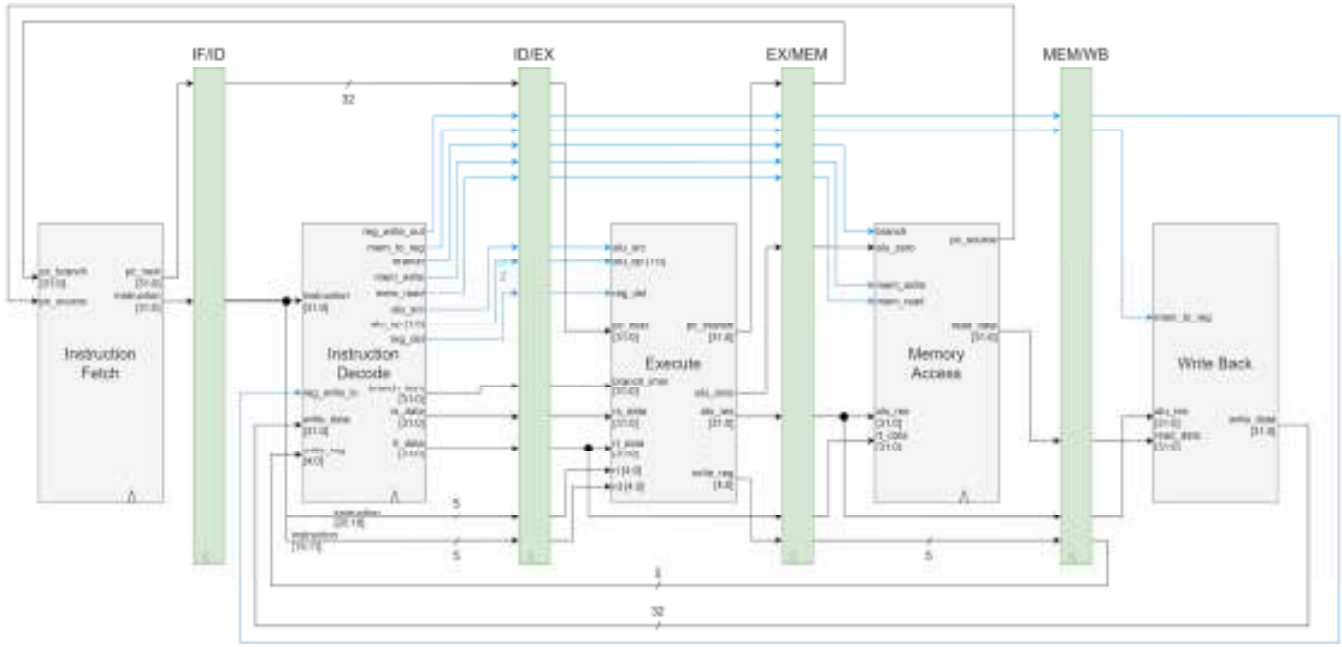
BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI K. K. BIRLA Goa Campus
First Semester 2020 - 2021
CS F342 Computer Architecture
Lab test Submission Date: 3rd Nov 2020 between 3 to 4:50 p.m.

The task is to implement a 5 stage pipelined processor. It incorporates pipeline registers that slice the datapath into 5 stages of similar latencies. The schematic of the processor is shown below -



The provided .zip folder contains all the files required for the task. The module for the processor has already been written and interfaced with the testbench, and need not be modified.

Problem Statement

The task for this lab is to write the code for each stage shown above. Only the following files need to be modified to complete the design of the processor -

1. Instruction Fetch (instr_fetch.v)
2. Instruction Decode (instr_decode.v)
3. Execute (execute.v)
4. Memory Access (mem_access.v)
5. Write Back (write_back.v)

The circuit diagrams for each stage have been given in this document. The modules that would be used in each stage (for example, mux2to1 from mux.v for Write Back) have already been included.

* All filenames, module names, and wire names are in lower case, separated by underscores if required.
* The following command should be used while compiling the testbench :

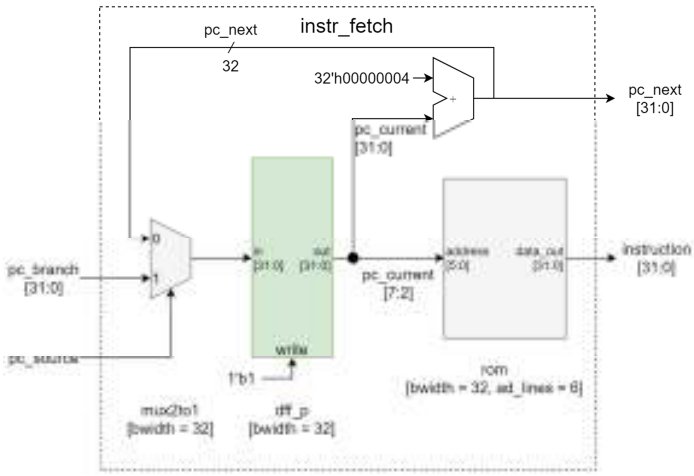
```
> iverilog -s testbench -o YourID_Lab5.vvp testbench.v
```

* The processor is capable of executing the following MIPS instructions :

1. R-Type :
   a. `ADD  rd, rs, rt`
   b. `SUB  rd, rs, rt`
   c. `AND  rd, rs, rt`
   d. `OR   rd, rd, rt`
   e. `NOR  rd, rs, rt`
   f. `SLT  rd, rs, rt`
2. I-type :
   a. `LW   rt, offset(rs)`
   b. `SW   rt, offset(rs)`
   c. `BEQ  rs, rt, offset`
   d. `ADDI rt, rs, imm_data`

* The files instr.mem and data.mem contains the code and data that are being simulated.

1. IF Stage (instr_fetch.v)



Input Ports

1. clk
2. reset
3. pc_branch [31:0]
4. pc_source

Output Ports

1. pc_next [31:0]
2. instruction [31:0]

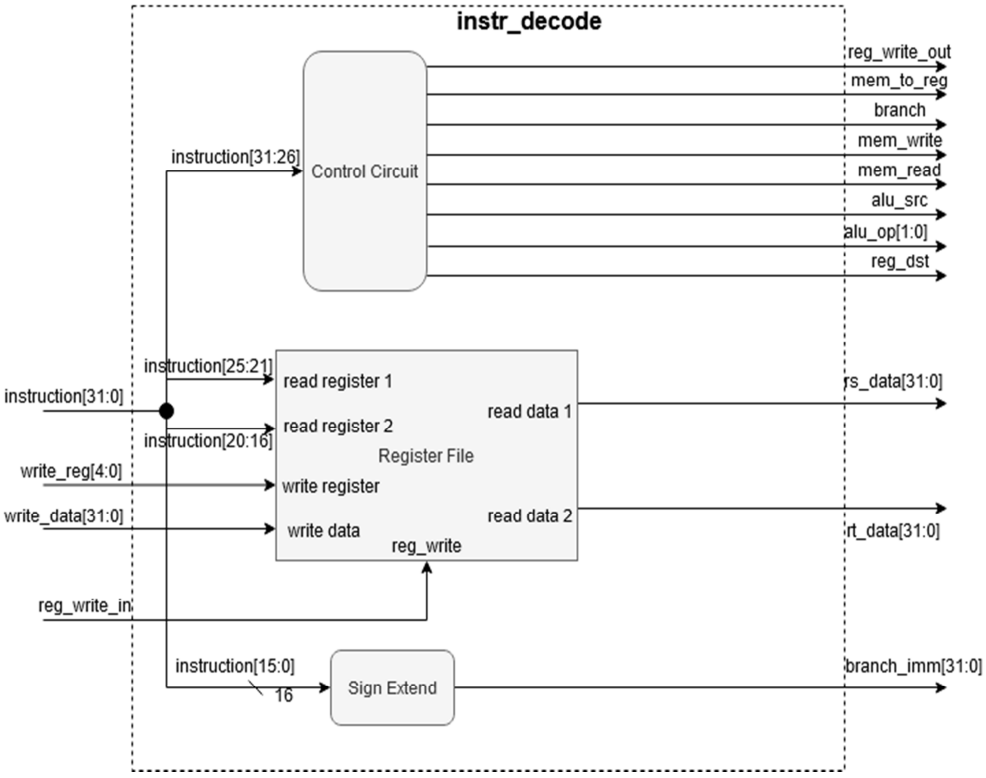
Modules Required

1. A 32 bit 2:1 mux (mux.v → mux2to1).
2. A 32 bit register to keep the program count

   (dff.v → dff_p).

   * Instantiate the DFF with the name - “program_counter”.
3. A block of Read Only Memory (rom), 32 bits wide, with 6 address lines.
   * Instantiate the ROM with the name - “instr_memory”.
4. A module for addition. An assign statement can be used here.

2. ID Stage (instr_decode.v)



Input Ports

1. clk
2. reset
3. instruction [31:0]
4. reg_write_in
5. write_data [31:0]
6. write_reg [4:0]

Output Ports

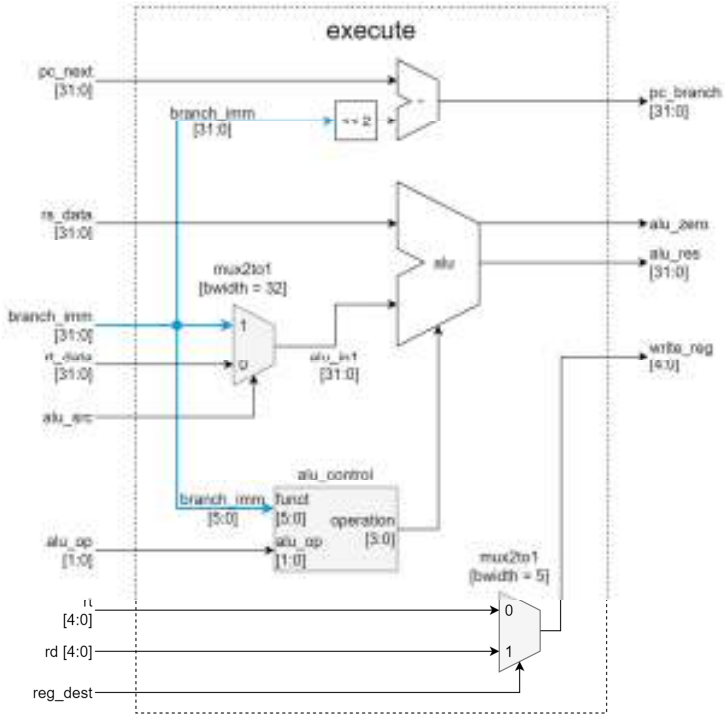
1. reg_write_out
2. mem_to_reg
3. branch
4. mem_write
5. mem_read
6. alu_src
7. alu_op [1:0]
8. reg_dst
9. branch_imm [31:0]
10. rs_data [31:0]
11. rt_data [31:0]

Modules Required

1. A 16 bit to 32 bit sign extension module (sign_ext).
2. The main register file (register_file.v → register_file)

   * Instantiate the register file with the name - “rf_main”.
3. The main control unit (control)

3. EX Stage (execute.v)



Input Ports

1. alu_src
2. alu_op [1:0]
3. reg_dst
4. pc_next [31:0]
5. branch_imm [31:0]
6. rs_data [31:0]
7. rt_data [31:0]
8. rt [4:0]
9. rd [4:0]

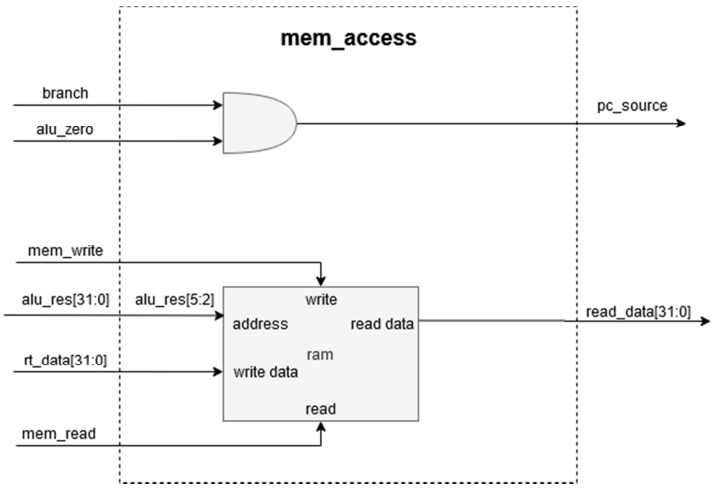
Output Ports

1. pc_branch [31:0]
2. alu_zero
3. alu_res [31:0]
4. write_reg [4:0]

Modules Required

1. A 32 bit 2:1 mux (mux.v → mux2to1).
2. The main alu (alu.v → alu).
3. The alu control unit (alu_control.v → alu_control).
4. A 5 bit 2:1 mux (mux.v → mux2to1).
5. A shift and add operator. An assign statement can be used here.

4. MEM Stage (mem_access.v)



Input Ports

1. clk
2. reset
3. branch
4. alu_zero
5. mem_write
6. mem_read

Output Ports

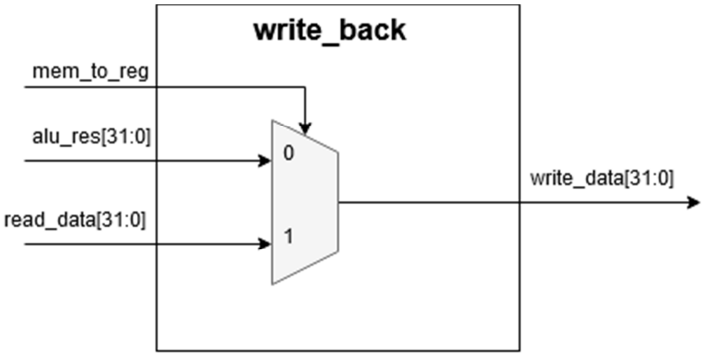
1. alu_res [31:0]
2. rt_data [31:0]
3. pc_source
4. read_data [31:0]

Modules Required

1. An AND gate.
2. A block of Random Access Memory (memory.v → ram), 32 bits wide with 4 address lines.

   *Instantiate the RAM with the name - “data_memory”

5. WB Stage (write_back.v)



Input Ports

1. mem_to_reg
2. alu_res [31:0]
3. read_data [31:0]

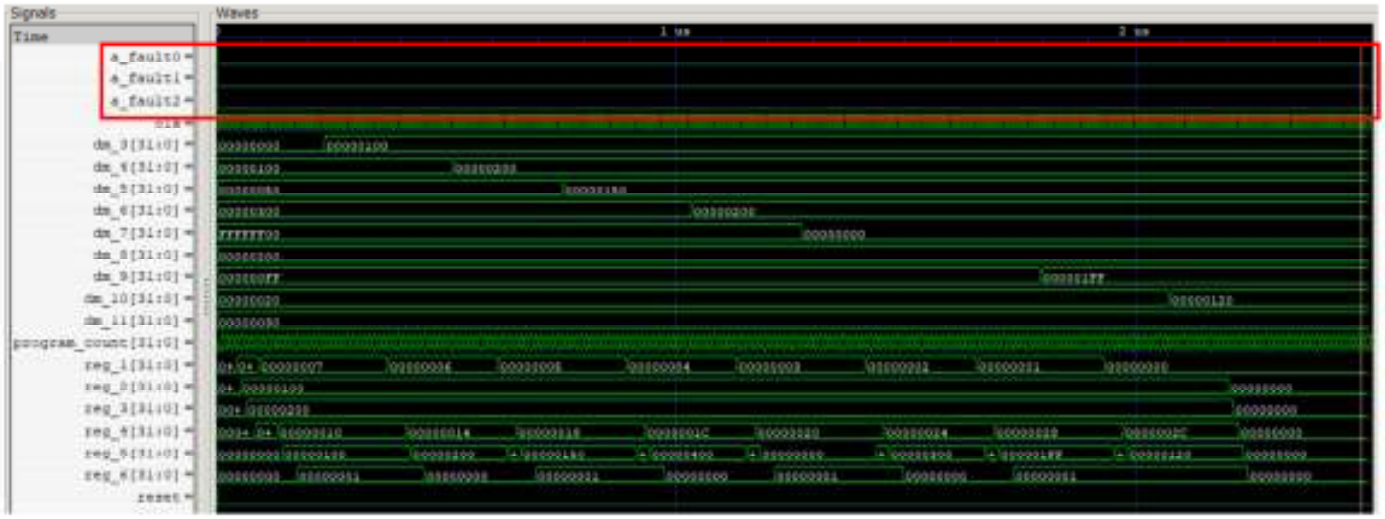
Output Ports

1. write_data [31:0]

Modules Required

1. One 32 bit 2:1 mux (mux.v → mux2to1).

Expected Outcome



The three fault signals should always be 0.

1. a_fault0 corresponds to the program counter being incremented normally.
2. a_fault1 corresponds to the registers being loaded with the correct data.
3. a_fault2 corresponds to the correct data being present in the registers and data memory at the end of the simulation.

The remaining signals have been provided for reference.

Marking Scheme

1. 3 mark - a_fault0
2. 3 mark - a_fault1
3. 4 marks - a_fault2

Submission Method

1. Change the name of the testbench file (testbench.v) to **YourID_Labtest.v**.
   (NOTE: Change yourID to your 13 digit BITS ID in the testbench)
2. Save the vcd dump file generated as **YourID_Labtest.vcd** (this will already be called **YourID_Labtest.vcd** since you have changed it in the testbench).
3. Save your GTKWave output as **YourID_Labtest.gtkw** using the 'Save As' option in File->Write.

All files (YourID_Labtest.v, YourID_Labtest.vcd, YourID_Labtestgtkw, and all the other verilog files) must be compressed into a .zip file which is to be submitted on Quanta.

* Do not create archives in other formats (rar, tar.gz, etc).

* Once uploaded on Quanta, remember to submit for grading. Do not leave it as a draft.