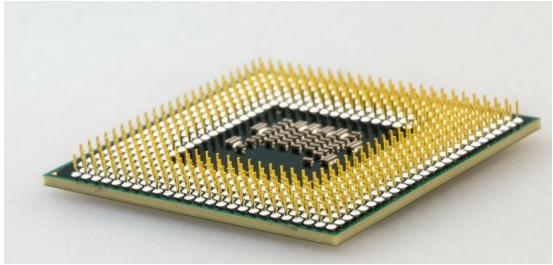
Final Project Presentation
ECE 174
Dr. Razouk

Carlos Lopez and Puya Fard

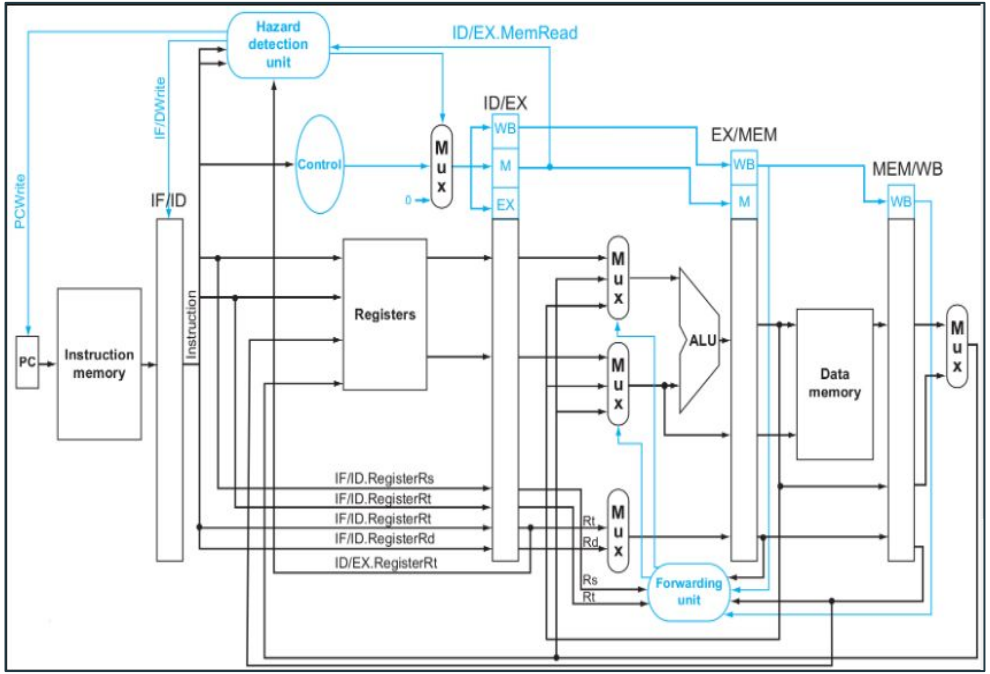


Project objective

The objective of this project is to integrate and prototype the datapath and the control units of the simple 32-bit MIPS processor with five pipeline stages.

This processor should be written in Verilog hardware language and must be able to perform arithmetic/logic, data movement, and flow control instructions.

The learning outcomes of this project will result in students having hands- on experience on building a computer processor and implementing it on hardware.



Project approach

In order to successfully implement the final design, we have decided to break the project apart into multiple sections. These sections include:

- Implementation of individual modules.
- Implementation of single-cycle processor.
- Implementation of Forwarding - Hazard control units
- Implementation of Final Pipelined design
- Tests and Validations

Design and modules: Single-Cycle

Our processor design consisted of the following modules:

- Program Counter
- Instruction memory block
- Register file
- ALU
- Data Memory

Following with several control modules:

- Control Unit (general purpose)
- ALU Control
- Flush unit

Following with Adders required:

- PC Adder
- Branch Adder
- Jump Calculation
- Sign Extend

Following with several MUXes

- PC Mux
- Register file Mux
- Write back Mux
- Operan Mux

And finally a Processor_Top (for single-cycle)

How do we upload instructions?

Using a instruction.txt file saved in the same directory as our processor modules will be necessary for this to work.

Then we will change our Instruction_memory module from (1) to (2). This way we read instructions from outside world instead of hardcoding it in our processor.

1)

```
module instruction_memory (
  input             clk,        // clock input (active-high)
  input      [31:0] address,
  output reg [31:0] instruction
);

  reg [7:0] mem[0:4095];

  initial begin

      // R-TYPE
      {mem[3 ],mem[2 ],mem[1 ],mem[0 ]}    =   32'h00430820; // add
      {mem[7 ],mem[6 ],mem[5 ],mem[4 ]}    =   32'h00222020; // add
      {mem[11],mem[10],mem[9 ],mem[8 ]}    =   32'h00430825; // or
      {mem[15],mem[14],mem[13],mem[12]}    =   32'h00430827; // nor
      {mem[19],mem[18],mem[17],mem[16]}    =   32'h00430822; // sub
      {mem[23],mem[22],mem[21],mem[20]}    =   32'h0043082A; // slt
      {mem[27],mem[26],mem[25],mem[24]}    =   32'h0043081A; // div
      {mem[31],mem[30],mem[29],mem[28]}    =   32'h00430818; // mul
      {mem[35],mem[34],mem[33],mem[32]}    =   32'h00000810; // mfhi
      {mem[39],mem[38],mem[37],mem[36]}    =   32'h00000812; // mflo

    // I-Type
      {mem[43],mem[42],mem[41],mem[40]}    =   32'h20410003; // addi
      {mem[47],mem[46],mem[45],mem[44]}    =   32'h10850019; // beq
      {mem[51],mem[50],mem[49],mem[48]}    =   32'h8c410064; // lw
      {mem[55],mem[54],mem[53],mem[52]}    =   32'hAC410064; // sw

    // J-Type
    {mem[59],mem[58],mem[57],mem[56]}    =   32'h08000101; // j

  end

  always @* begin
    instruction = {mem[address+3], mem[address+2], mem[address+1], mem[address+0]};
  end
```

2)

```
module instruction_memory (
    input             clk,
    input      [31:0] address,
    output reg [31:0] instruction
);

    reg [7:0] mem[0:4095];

    integer fp;
    integer status;
    integer i;

    initial begin
        fp = $fopen("instructions.txt", "r");
        if (fp != 0) begin
            for (i = 0; i < 4096; i = i+1) begin
                status = $fscanf(fp, "%h", mem[i]);
                if (status == 0) begin
                    $display("Error: Could not read instruction at address %d", i*4);
                    $finish;
                end
            end
            $fclose(fp);
        end
        else begin
            $display("Error: Could not open file 'instructions.txt'");
            $finish;
        end
    end

    always @(address) begin
        instruction = {mem[address+0], mem[address+1], mem[address+2], mem[address+3]};
    end

endmodule
```

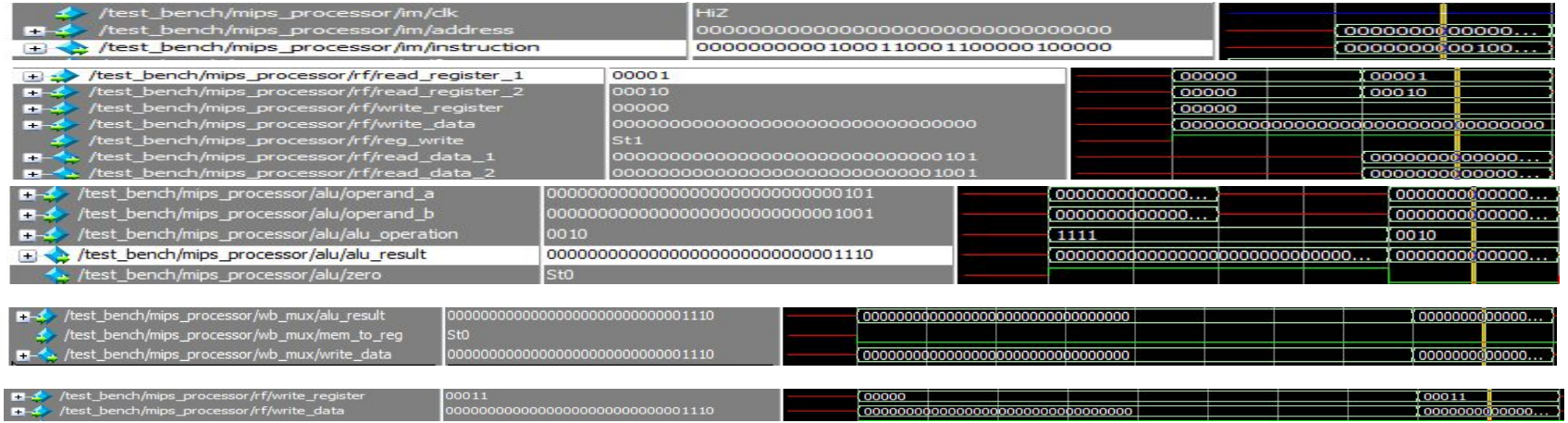
Module breakdown

How does the system work for a simple instruction, lets breakdown two instructions to understand. 1 R-type and 1 I-Type and simple Jump.

R-Type ADD for adding register r1 + r2, saving result into r3. **ADD r3, r2, r1.**

1) First convert to Binary, then HEX to use in instruction.txt file to input it into our Instruction_memory module

Binary: 000000 00001 00010 00011 00000 100000 = Hex: 00221820

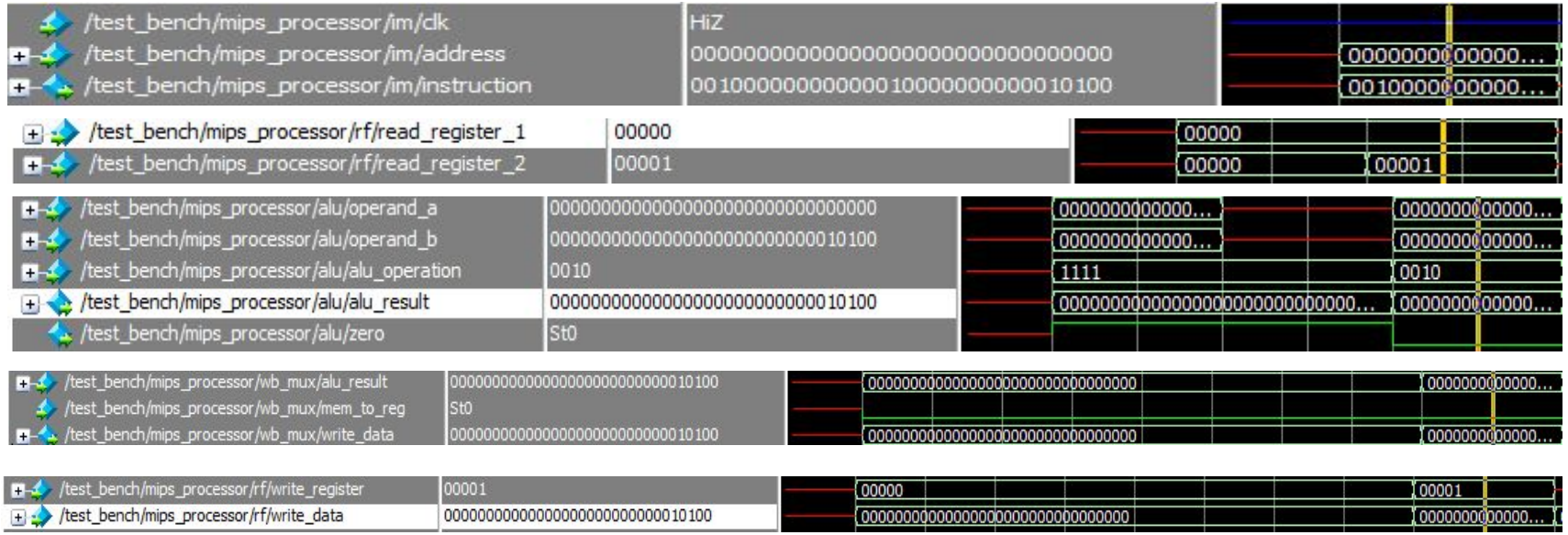


Module breakdown

I-Type ADDI for adding register r0 + imm(20) , saving result into r1. **ADDI r1, r0, 20**

1) First convert to Binary, then HEX to use in instruction.txt file to input it into our Instruction_memory module

Binary: 001000 00001 00000 0000000000010100 = Hex: 20200014



Module breakdown

J-Type Jump instruction Jumping to target address **101**

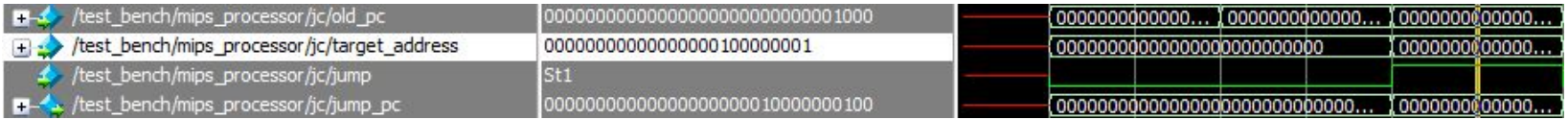
1) First convert to Binary, then HEX to use in instruction.txt file to input it into our Instruction_memory module

Binary: 000010 00000000000000000100000001 = Hex: 08000101

PC Before JUMP

PC After JUMP

Jump Calculation



Pipelining our design

In order to pipeline our design, we must enable forwarding logic to our design. We will implement general forwarding as practiced in class. For this design, we are required to have the following stage module:

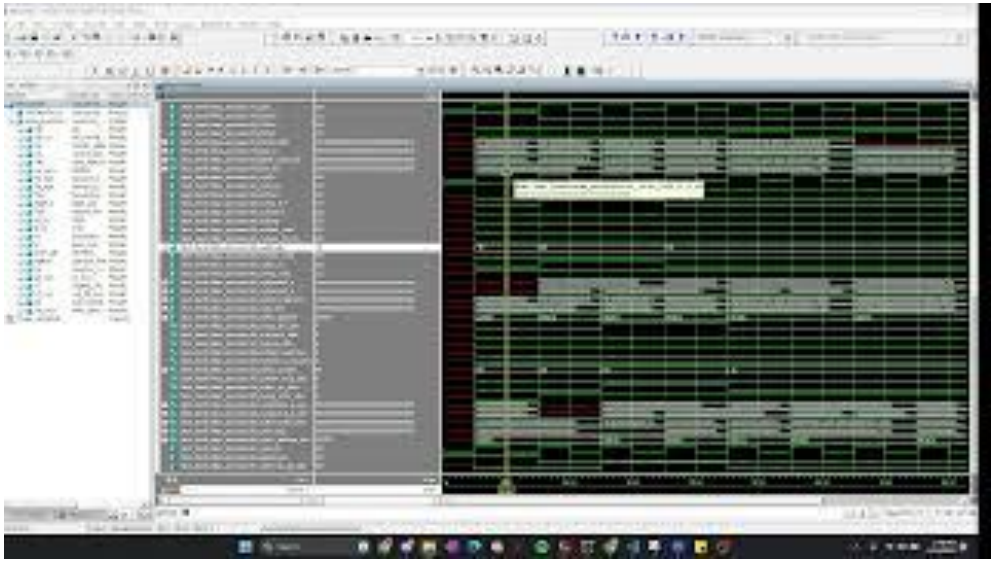
- IF/ID stage module
- ID/EX stage module
- EX/MEM stage module
- MEM/WB stage module

Control unit module for forwarding:

- Forwarding Control Unit

Along with the following MUXes:

- Forward_a_mux
- Forward_b_mux

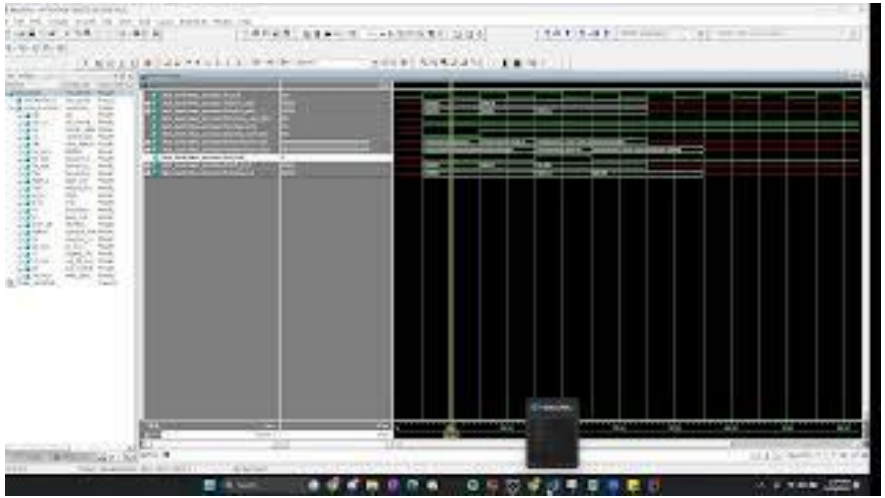


Detecting hazards

In order to successfully apply forwarding logic, we must make sure we detect the occurrence of hazards and handle them. Therefore, we implemented **Hazard_detection_unit** for our Pipelined system in order to Stall operations by **1**.

We will test our design to check if we detect hazards or not with the following two instruction:

ADD r3, r1, r2
ADD r4, r1, r3

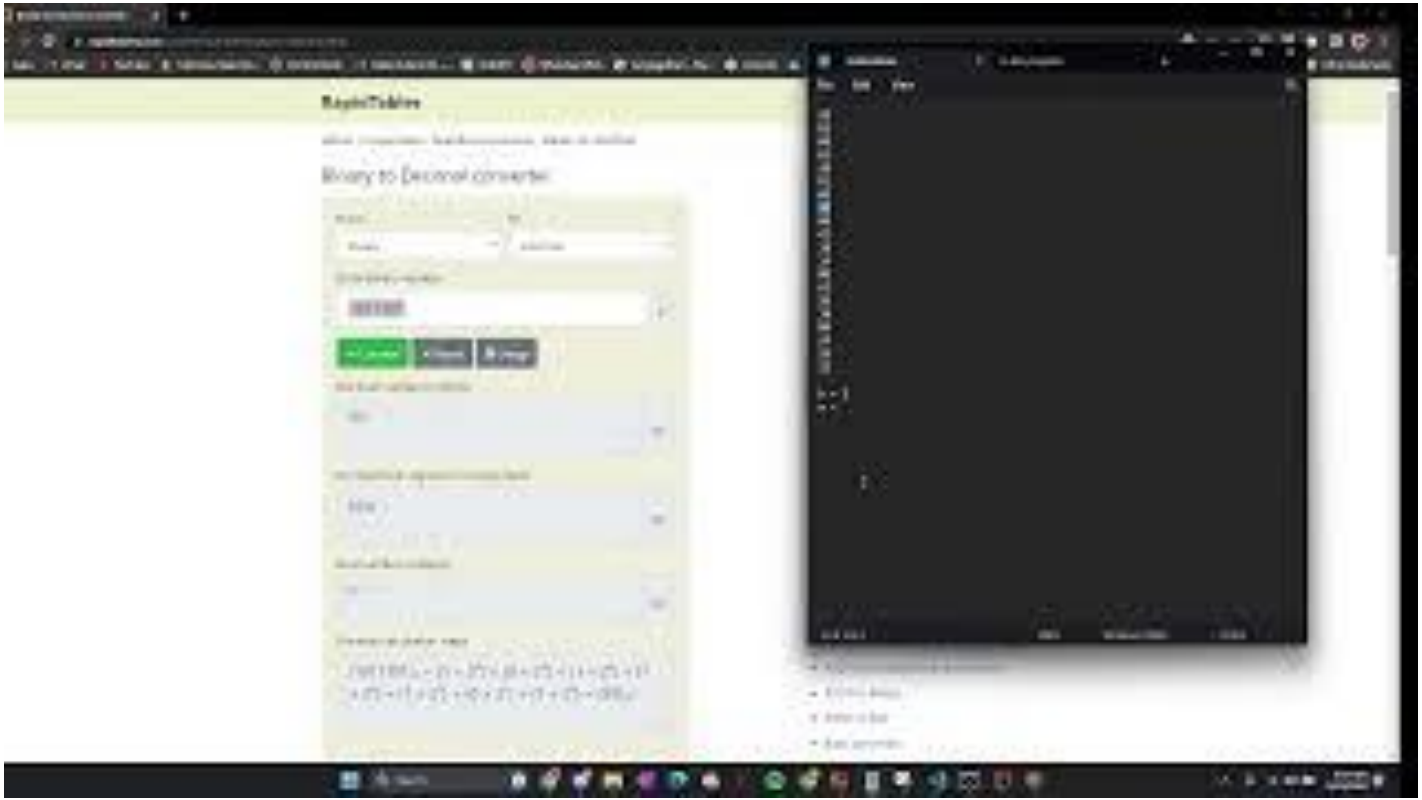


Tests and debugging

Throughout the process of implementing the final design there were several things that caused problem.

1) Every instruction in single-cycle processor worked **FINE**
2) Problems occurred when tried pipelining.
    a) At first, every instruction on its one worked fine, but then things started to breakdown
    b) Forwarding problems in **MUL** operation **input** and **output**.
    c) Forwarding problems in **I-Type** instructions.
    d) Had to go back and forth between our initial design and newly pipelined design to fix the problems.
3) Tests happened in two ways:
    a) First by single instruction tests.
    b) Second by a prototype program tests.

DEMO



Conclusion

As a conclusion we have integrated and prototype the datapath and the control units of the simple 32-bit MIPS processor with five pipeline stages. This processor is written in Verilog hardware language and able to perform arithmetic/logic such as AND, ADD, ADDI, OR, NOR, SUB, MUL, DIV, data movement such as LW, SW, MFHI, MFLO, and flow control such as J and BEQ instructions. Moreover, all instructions had passed the test successfully and were working accordingly. Finally, The learning outcomes of this project resulted in having hands- on experience on building a computer processor and implementing it on hardware.

Thank You :)