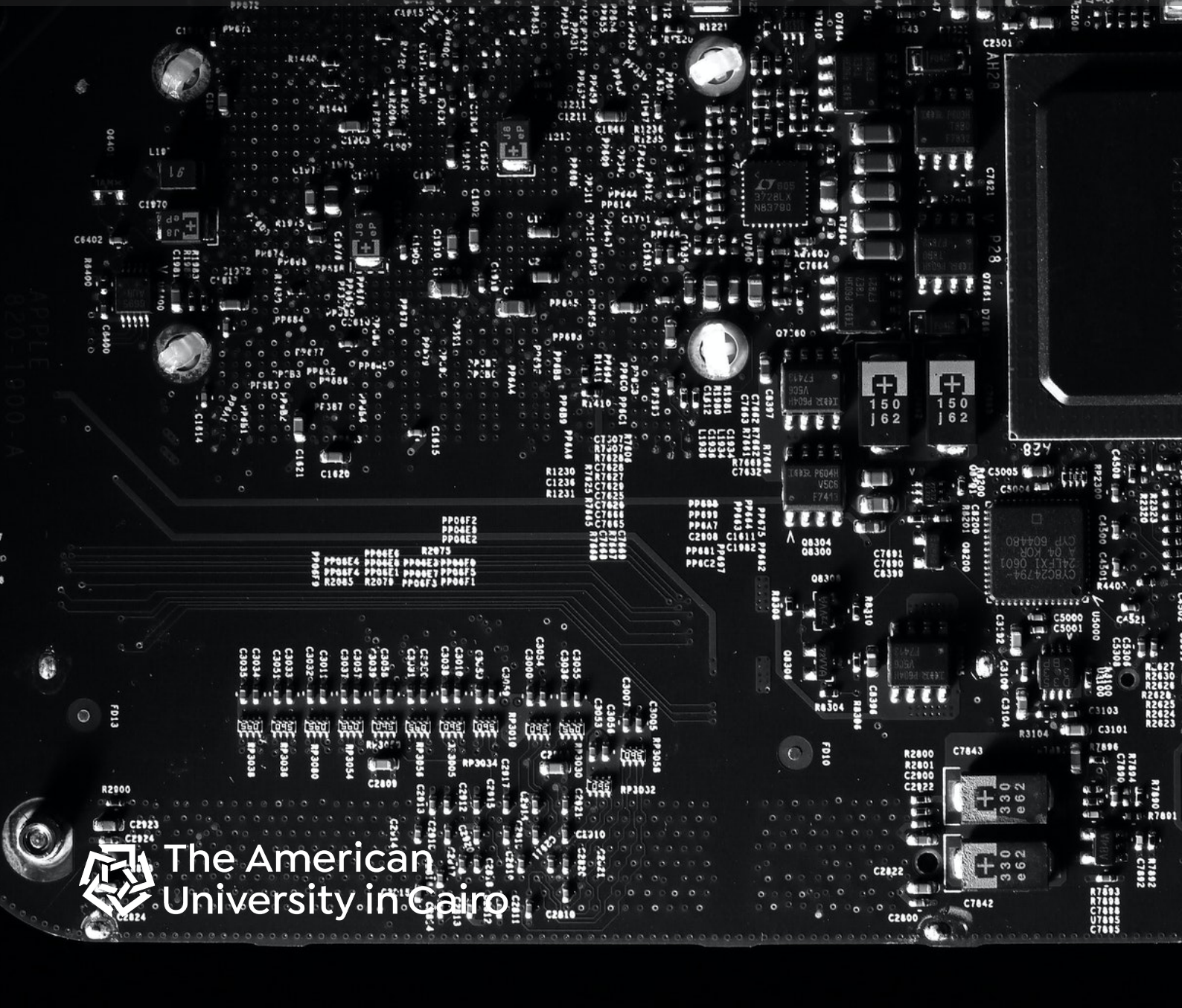
Pipelined RISC-V Processor

Final Submission

CSCE 3301 – Computer Architecture - F'22

Abdelaaty Rehab
Rana El Gahawy



The American University in Cairo

Pipelined RISC-V Processor

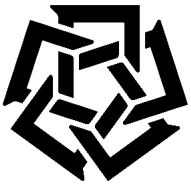
Final Submission

by

Abdelaaty Rehab
Rana El Gahawy

Student Name	Student ID
Rehab	900204245
Elgahawy	900202822

Faculty: School of Science and Engineering, AUC
Instructor: Dr. Cherif Salama | Fall 2022



Preface

This report includes a survey of simple implementation of a RISC-V 5-staged pipelined processor supporting the rv32i user instruction sets (40 instructions) accommodating data forwarding and hazard handling.

Abdelaaty Rehab
Rana El Gahawy
November 2022

Summary

A further enhancement was done on the project done in the previous milestone where the single cycle implementation was modified to be a 5-staged pipelined one to maximize the throughput of the processor. The processor supports the full RISC-V integer instruction set including both data forwarding and hazard handling.

Contents

1

Overview

1.1. Technical Summary

This project includes the verilog description of a pipelined implementation of the $rv32i$ RISC-V ISA. The processor was designed according to the following specifications:

1.1.1. Pipeline Stages

The data path was separated into 5 stages each two stages operate within two clock cycles. That is if it were a 3 stages pipelined implementation in which each stage includes two stages of the first implementation. This approach was adopted to solve structural hazards introduced due to memory accessing or control hazards introduced due to mispredicted branch outcomes.

1.1.2. Clock Cycles

As mentioned earlier, the approach to avoid several hazards was to include two stages of the pipeline within two cycles. This means that the original clock has to be slower by a factor of 2 in order to accommodate such approach. One solution is to use the rising and the falling edges of the original clock as the clock of the new stages. Another solution, which was adopted by this design, is to divide the clock by a factor of 2 using a clock divider.

1.1.3. Hazard Handling

As mentioned earlier, the only hazards that can occur with this implementation are the data hazards (RAW ones in particular). This can be solved by introducing a forwarding unit that forwards the results from the memory stage once they are ready. Notice that we always forward from the memory stage since the new design always has the memory stage of a preceding instruction right before the execution stage of a following instruction providing a great advantage to simplify the combinational logic of the forwarding unit.

2

Data paths

We can note that the data paths of the different instruction types did not change except that a pipeline register was added between each two consecutive stages to ensure that the control signals of the instruction are not lost during execution in a pipelined implementation.

2.1. R, I instruction type

The data path for this kind of instruction are similar where the instruction is fetched, decoded and then pass by the ALU which calculates the result according to the given instruction. Finally, the result of the ALU is used to update the register file with the corresponding values.

2.2. U instruction type

This type includes LUI and $AUIPC$ whose datapath does not pass by the ALU for simplicity. The LUI depends on forwarding the output of the immediate generator to the register file directly. Similarly, the $AUIPC$ adds the generator output to the PC for further storage in the register file.

2.3. J instruction type

The datapath for this kind of instruction required the introduction of a new control signal $JUMP$ which determines whether it is a branch or a jump instruction. This kind of instruction calculates the target address from the ALU result storing the value linked to the program counter to the register file and then jumping to the corresponding target address by updating the program counter.

2.4. S instruction type

This type depended on the data memory for storing data. It also requested to store bytes, words and half words in the memory using different modes (signed and unsigned). This was fixed by doing logical or arithmetic extension when loading values from the data memory.

2.5. B instruction type

This type of instruction includes different variations of the branch analogy. This was implemented using different flags that were determined as outputs from the ALU. Those flags were used as selection lines to determine what kind of branch to be done yet used to calculate the target address. The datapath depended on the result of the ALU which was used to select whether to add 4 to the program counter or add the offset generated by the immediate value generator.

3

CPU Components

3.1. A single-ported instruction/data memory

The implementation required the usage of a single memory for both data and instructions. This led to the possibility of the occurrence of a structural hazard resulting from trying to read and write to the memory simultaneously. One solution to this was to adopt another pipelining method that groups every two stages into a new bigger stages taking the same number of clock cycles. Another solution which was also implemented is to convert the memory from being a **single-ported single** memory into **dual-ported single** memory. This guarantees that both reading and writing can occur simultaneously without causing any hazards as the write address and he read address are not passed on two different ports. As for the differentiation between the instructions and the data being outputted, we relied on the fact that exactly one of the two signals $MemRead$ or $MemWrite$ has to be high if we need to access the data memory. Otherwise, we have to fetch as instruction which is represented at the address of the current program counter.

3.2. A triple-ported register file

The design uses a 32-bit register file which has 3 ports designed to be 2 read ports supporting the instructions that require reading two register addresses at the same time and 1 write port to write to the register. The register is designed to read at the positive edge of the clock and write at the negative edge of the clock. This reduces the delay of stalling for an extra cycle during the write back stage since the values will be written at the positive edge of the clock.

4

Simulation Testing and Results

The program was tested on the following RISC-V program and was found to behave correctly as expected.

```
1  lw x1, 0(x0)
2  lw x2, 4(x0)
3  lw x3, 8(x0)
4  or x4, x1, x2
5  beq x4, x3, 4
6  add x3, x1, x2
7  add x5, x3, x2
8  sw x5, 12(x0)
9  lw x6, 12(x0)
10 and x7, x6, x1
11 sub x8, x1, x2
12 add x0, x1, x2
13 add x9, x0, x1
```

The following is a screenshot from the simulation results obtained.

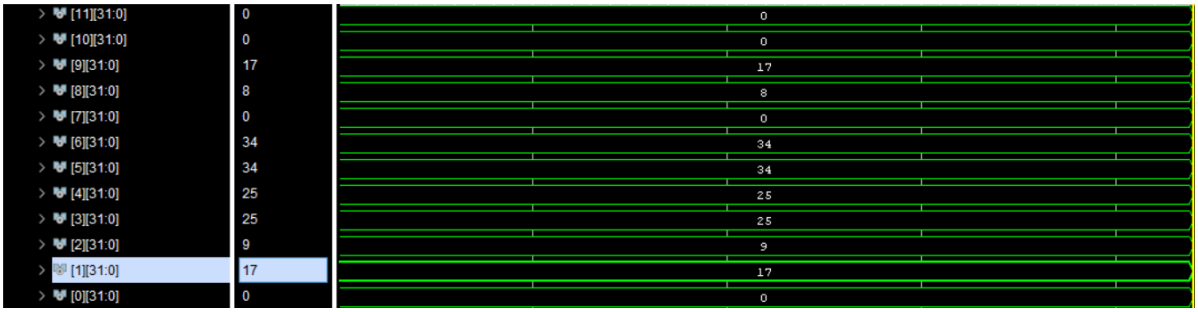


Figure 4.1: Results of simulating the above program on the given implementation.

5

Implemented Bonus Features

5.1. Supporting the integer multiplication/division full instructions

The design was extended to support the full integer multiplication and division instructions according to the RISC-V specifications.

5.2. Finding another solution for the structural hazard due to single memory

Instead of using a single ported memory which introduced a data hazard that was solved by increasing the CPU cycle time which downgraded the performance, we used another port for the memory write which eliminated the structural hazard we have in the memory. In this case, we guarantee that we can always write data back to memory and read from it simultaneously.

6

Conclusion

This report represented a quick survey of a trial to implement a pipelined RISC-V processor that supported the full unprivileged instruction set. The processor has been tested over the included test cases in the report. Errors, if existed, have been included within the report.

7

Source Code

```
/*********************************************************************
*
* Module: rv32_CPU.v
* Author: Abdelaaty Rehab & Rana Elgahawy
*
*********************************************************************/
`timescale 1ns / 1ps
`include "defines.v"

module rv32_CPU(
    input wire clk,
    rst
);
    wire Branch, MemRead, RegWrite, MemWrite, ALUSrc, Jump, Branchflag;
    wire [2:0] WhichReg;
    wire [1:0] ALUOp;
    reg  [31:0] WriteData; //from mux
    wire [31:0] ReadData1, ReadData2, ReadData, EX_MEM_PC_Branch, EX_MEM_PC_JAL, MEM_WB_Rd, EX_MEM_Ctrl, EX_MEM_PC_JALR, EX_MEM_Func;
    wire [31:0] gen_out, ALUResult;
    wire [3:0] ALU_Sel;
    wire [31:0] MUX_ALU;
    wire zf, cf, vf, sf, flush_pipeline;
    reg  [31:0] PC;
    wire [31:0] MEM_WB_Ctrl;
    wire [31:0] BranchAddress, JALAddress, JALRAddress;
    wire [31:0] Add4;
    reg  [31:0] next_PC;
    reg  [12:0] SSD;
    wire [1:0] JB;

    assign JB = {EX_MEM_Ctrl[5], EX_MEM_Ctrl[0]};
    assign Add4 = 4 + PC;

    always @(*) begin
        if (~((ReadData[6:0] == 7'b1110011)&&(ReadData[20] == 1'b1)))
                case(JB)
                    2'b00: begin
                    if ((ReadData[6:0] == 7'b0001111) || ((ReadData[6:0] == 7'b1110011)& (ReadData[20] == 1'b0)))
                        next_PC = 0;
                    else
                        next_PC = Add4;
                    end
                    2'b01: next_PC = (Branchflag) ? EX_MEM_PC_Branch : Add4;
                    2'b10: next_PC = (EX_MEM_Func[30]) ?  JALAddress : JALRAddress;
                    default: next_PC = Add4;
                endcase
        else
            next_PC = PC;
    end

    always@(posedge clk or posedge rst) begin
     if (rst) PC = 0;
     else
        PC = next_PC;
    end

    wire [31:0] IF_ID_PC, IF_ID_Inst;

    n_bit_register #(64) IF_ID_REG ({ReadData, PC}, rst, 1'b1, ~clk, {IF_ID_Inst, IF_ID_PC});

    ////////////////////////////// end of fetching stage ////////////////////////////////////////////////

    wire [31:0] ID_EX_PC, ID_EX_RegR1, ID_EX_RegR2, ID_EX_Imm;
    wire [31:0] ID_EX_Ctrl;
    wire [31:0] ID_EX_Func;
    wire [31:0] ID_EX_Rs1, ID_EX_Rs2, ID_EX_Rd;

    Control_Unit CU (IF_ID_Inst, Branch, MemRead, WhichReg, MemWrite, ALUSrc, RegWrite, Jump, ALUOp);
    registerFile #(32) RF (~clk, rst , MEM_WB_Ctrl[1], IF_ID_Inst [`IR_rs1], IF_ID_Inst [`IR_rs2], MEM_WB_Rd, WriteData, ReadData1, ReadData2);
    rv32_ImmGen IG (IF_ID_Inst, gen_out);
    n_bit_register #(288) ID_EX_REG (
    {IF_ID_PC, {{27{1'b0}},IF_ID_Inst[`IR_rd]},
    flush_pipeline ? {32{1'b0}} : {ALUSrc, ALUOp, {21{1'b0}}, Jump, WhichReg, MemRead, MemWrite, RegWrite, Branch},
     ReadData1, ReadData2,
    {{27{1'b0}},IF_ID_Inst[`IR_rs1]},
    {{27{1'b0}},IF_ID_Inst[`IR_rs2]},
    {IF_ID_Inst[30], IF_ID_Inst[3], {27{1'b0}}, IF_ID_Inst[`IR_funct3]},
     gen_out},
```

```
     rst, 1'b1, clk,
    {ID_EX_PC, ID_EX_Rd, ID_EX_Ctrl, ID_EX_RegR1, ID_EX_RegR2, ID_EX_Rs1, ID_EX_Rs2, ID_EX_Func, ID_EX_Imm});

    /////////////////////////// end of decoding stage ////////////////////////////////////////////////////

    wire [31:0] EX_MEM_ALU_out, EX_MEM_RegR2, EX_MEM_RegR1, ALUInput1, ALUInput2;
    wire [31:0] EX_MEM_Imm;
    wire [31:0] EX_MEM_Rd, EX_MEM_PC;
    wire [3:0]  EX_MEM_Flags;
    wire forwardA, forwardB;

    assign JALAddress = ID_EX_PC + ID_EX_Imm;
    assign JALRAddress = ID_EX_PC + ID_EX_RegR1 + ID_EX_Imm;
    assign BranchAddress = ID_EX_PC + ID_EX_Imm;
    assign ALUInput1 = (forwardA) ? WriteData : ID_EX_RegR1;
    assign ALUInput2 = (forwardB) ? WriteData : ID_EX_RegR2;
    assign MUX_ALU = (ID_EX_Ctrl[31])? ID_EX_Imm : ALUInput2;

    ALU_ControlUnit ACU (ID_EX_Func[31], ID_EX_Ctrl [30 : 29], ID_EX_Func [2:0], ALU_Sel);
    prv32_ALU ALU(ALUInput1, MUX_ALU, ALUResult, zf, cf, vf, sf, ALU_Sel);

    n_bit_register #(356) EX_MEM (
    {ID_EX_RegR1, ID_EX_Func, ID_EX_Imm, ID_EX_PC,
    {zf, cf, vf, sf}, ID_EX_Ctrl,
     BranchAddress, JALAddress, JALRAddress, ALUResult,
     ALUInput2, ID_EX_Rd},
     rst, 1'b1, ~clk,
    {EX_MEM_RegR1, EX_MEM_Func, EX_MEM_Imm, EX_MEM_PC, EX_MEM_Flags, EX_MEM_Ctrl, EX_MEM_PC_Branch,
     EX_MEM_PC_JAL, EX_MEM_PC_JALR, EX_MEM_ALU_out, EX_MEM_RegR2, EX_MEM_Rd});

    /////////////////////////// end of execution stage ////////////////////////////////////////////////////

    wire [31:0] MEM_WB_Mem_out, MEM_WB_ALU_out;
    wire [31:0] MEM_WB_Imm, MEM_WB_PC;

    Forwarding_Unit FWD (forwardA, forwardB, ID_EX_Rs1, ID_EX_Rs2, MEM_WB_Rd, MEM_WB_Ctrl[1]);
    Branch_CU BU(EX_MEM_Func[2:0], EX_MEM_Flags[3], EX_MEM_Flags[2], EX_MEM_Flags[1], EX_MEM_Flags[0], Branchflag);
    IDM MEM(clk, EX_MEM_Ctrl[3], EX_MEM_Ctrl[2], {EX_MEM_ALU_out[7:0] , PC}, EX_MEM_Func[2:0],EX_MEM_RegR2, ReadData);

    assign flush_pipeline =  EX_MEM_Ctrl[0] & Branchflag;

     n_bit_register #(192) MEM_WB(
     {EX_MEM_Imm, EX_MEM_Ctrl, EX_MEM_ALU_out, EX_MEM_Rd, ReadData, EX_MEM_PC},
      rst, 1'b1, clk,
     {MEM_WB_Imm, MEM_WB_Ctrl, MEM_WB_ALU_out, MEM_WB_Rd, MEM_WB_Mem_out, MEM_WB_PC});

    /////////////////////////// end of memory stage ////////////////////////////////////////////////////

    always @(*) begin
        case(MEM_WB_Ctrl[6:4])
            3'b000:  WriteData = MEM_WB_ALU_out;
            3'b001:  WriteData = MEM_WB_Mem_out;
            3'b010:  WriteData = MEM_WB_PC+4;
            3'b011:  WriteData = MEM_WB_Imm;
            3'b100:  WriteData = MEM_WB_PC + MEM_WB_Imm;
            default: WriteData = 0;
        endcase
    end
endmodule
```