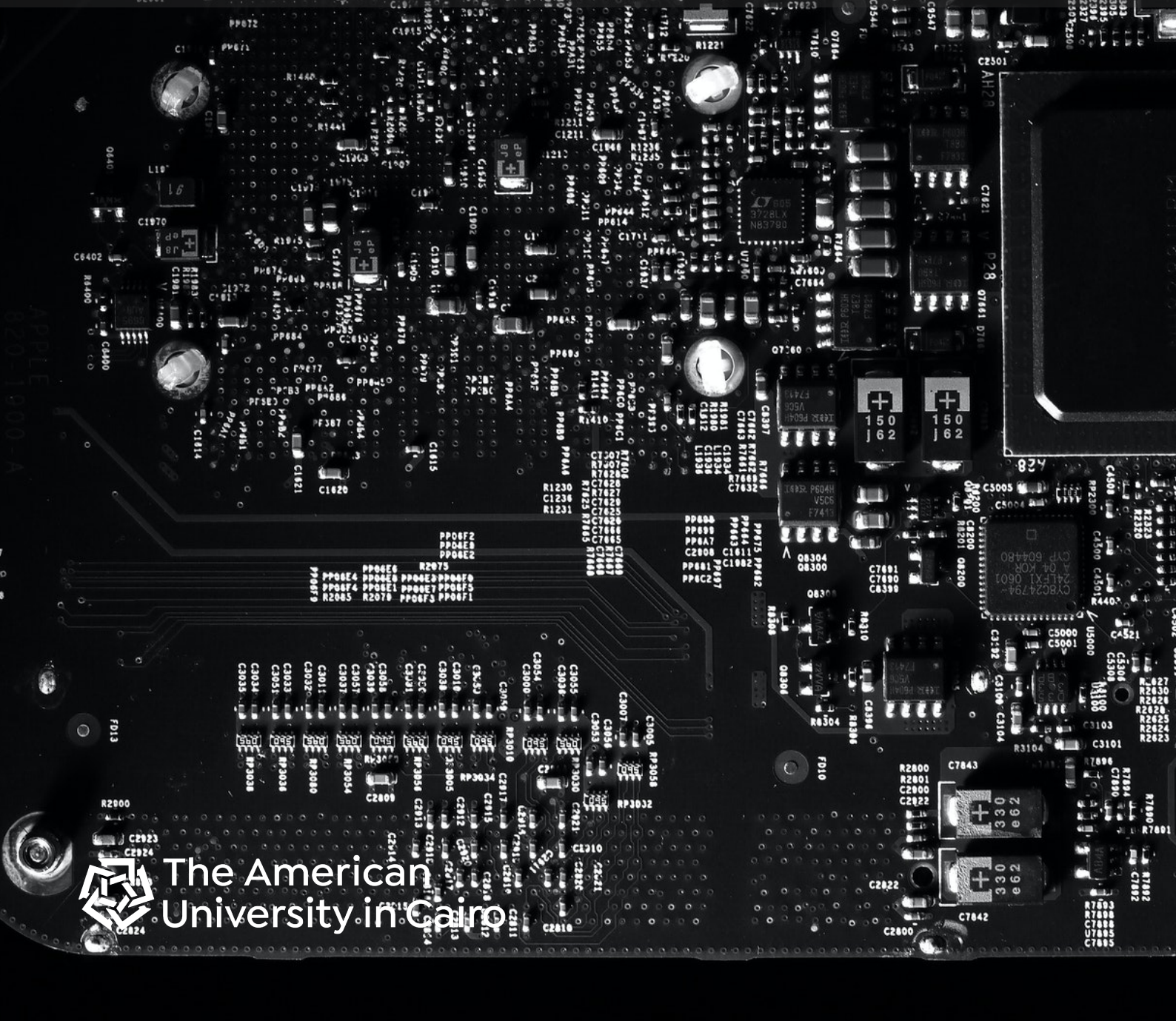
Pipelined RISC-V Processor

Milestone 3

CSCE 3301 – Computer Architecture - F'22

Abdelaaty Rehab
Rana El Gahawy



The American University in Cairo

Pipelined RISC-V Processor

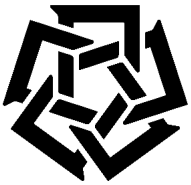
Milestone 3

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).

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 without hazard detection. More enhancements are to be done in the next milestone.

Contents

Preface i

Summary ii

1 Overview 1
1.1 Technical Summary 1

2 Data paths 2
2.1 R, I instruction type 2
2.2 U instruction type 2
2.3 J instruction type 2
2.4 S instruction type 2
2.5 B instruction type 2

3 CPU Module Description 3

4 Simulation Testing and Results 5

5 Conclusion 7

1

Overview

1.1. Technical Summary

Within this project, we used the pipelined implementation which was done in the lab attaching to it a few modifications as follows:

- Dividing the data path to five separate stages introducing a pipeline register between every two consecutive stages.
- Transmitting the needed signals in the later stages along the pipeline keeping in mind that the unneeded signals shall not be transmitted to minimize the memory cost.
- Merging the data memory and the instruction memory in a single ported memory that is accessed depending on the fed control signals coming from the Control 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 Module Description

This is the main description of the top module of the project. The other verilog codes are attached within the compressed file.

```
`timescale 1ns / 1ps

module rv32_CPU(input wire clk, rst);
    wire [31:0] data_out;
    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;
    wire [31:0] gen_out, ALUResult;
    wire [3:0] ALU_Sel;
    wire [31:0] MUX_ALU;
    wire zf, cf, vf, sf;
    reg [31:0] PC;
    wire [31:0] PC_SUM;
    wire [31:0] IF_ID_PC, IF_ID_Inst;
    wire [31:0] Add4;
    reg [31:0] next_PC;
    reg [12:0] SSD;

/*  Fetching Stage */
    wire JB;
    wire [31:0] MEM_WB_PC_SUM, MEM_WB_RegR1;
    always @(*) begin
        if (~((data_out[6:0] == 7'b1110011)&&(data_out[20] == 1'b1)))
            case(JB)
                2'b00: begin
                if ((data_out[6:0] == 7'b0001111) || ((data_out[6:0] == 7'b1110011)& (data_out[20] == 1'b0)))
                    next_PC = 0;
                else
                    next_PC = Add4;
                end
                2'b01: next_PC = (Branchflag) ? MEM_WB_PC_SUM : Add4;
                2'b10: next_PC = (data_out[3]) ?  MEM_WB_PC_SUM : MEM_WB_PC_SUM + MEM_WB_RegR1;
            endcase
    end

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

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

/*  Decoding 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 [6:2], Branch, MemRead, WhichReg, MemWrite, ALUSrc, RegWrite, Jump, ALUOp);

    registerFile #(32) RF (~clk, rst , RegWrite, IF_ID_Inst [19:15], IF_ID_Inst [24:20], IF_ID_Inst [11:7], 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[11:7]}, {ALUSrc, ALUOp,
    {21{1'b0}}, Jump, WhichReg, MemRead, MemWrite, RegWrite, Branch}, ReadData1, ReadData2,
    {{27{1'b0}},IF_ID_Inst[19:15]}, {{27{1'b0}},IF_ID_Inst[24:20]},
     {IF_ID_Inst[30],{28{1'b0}}, IF_ID_Inst[14:12]}, 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});

/*  Execution Stage */
    wire [31:0] EX_MEM_Ctrl, EX_MEM_ALU_out, EX_MEM_RegR1, EX_MEM_RegR2, EX_MEM_Imm, EX_MEM_PC_SUM;
    wire [31:0] EX_MEM_Rd;
    wire [3:0]  EX_MEM_Flags;
    wire [31:0]  EX_MEM_Func;
    ALU_ControlUnit ACU (ID_EX_Func[31], ID_EX_Ctrl[30:29], ID_EX_Func[2:0], ALU_Sel);

    assign MUX_ALU = (ALUSrc)? ID_EX_Imm : ID_EX_Rs2;
    prv32_ALU ALU(ID_EX_Rs1, MUX_ALU, ALU_Sel, ALUResult, zf, cf, vf, sf);

    assign PC_SUM = PC + ID_EX_Imm;
    n_bit_register #(260) EX_MEM ({ID_EX_RegR1, ID_EX_Func, ID_EX_Imm, {zf, cf, vf, sf}, ID_EX_Ctrl, PC_SUM, ALUResult,
    ID_EX_Rs2, ID_EX_Rd}, rst, 1'b1, clk, {EX_MEM_RegR1, EX_MEM_Func, EX_MEM_Imm, EX_MEM_Flags, EX_MEM_Ctrl, EX_MEM_PC_SUM,
```

```
    EX_MEM_ALU_out, EX_MEM_RegR2, EX_MEM_Rd});

/* Memory Stage */
    wire [31:0] MEM_WB_Mem_out, MEM_WB_ALU_out;
    wire [31:0] MEM_WB_Ctrl;
    wire [31:0] MEM_WB_Imm, MEM_WB_Rd;

    IDM mem(clk, EX_MEM_Ctrl[3], EX_MEM_Ctrl[2], (EX_MEM_Ctrl[3] | EX_MEM_Ctrl[2]) ? EX_MEM_ALU_out[7:0] : PC, EX_MEM_Func, EX_MEM_RegR2, data_out);

    Branch_CU BU(EX_MEM_Func, EX_MEM_Flags[3], EX_MEM_Flags[2], EX_MEM_Flags[1], EX_MEM_Flags[0], Branchflag);

    n_bit_register #(224) MEM_WB({EX_MEM_RegR1, EX_MEM_Imm, EX_MEM_PC_SUM, EX_MEM_Ctrl, data_out, EX_MEM_ALU_out, EX_MEM_Rd}, rst, 1'b1, clk,
    {MEM_WB_RegR1, MEM_WB_Imm, MEM_WB_PC_SUM, MEM_WB_Ctrl, MEM_WB_Mem_out, MEM_WB_ALU_out, MEM_WB_Rd});

/* Write Back Stage */
    assign Add4 = 4 + PC;
    assign JB = {EX_MEM_Ctrl[6], EX_MEM_Ctrl[0]};

    always @(*) begin
        case(EX_MEM_Ctrl[5:4])
            3'b000: WriteData = MEM_WB_ALU_out;
            3'b001: WriteData = data_out;
            3'b010: WriteData = Add4;
            3'b011: WriteData = MEM_WB_Imm;
            3'b100: WriteData = MEM_WB_PC_SUM;
        endcase
    end
endmodule
```

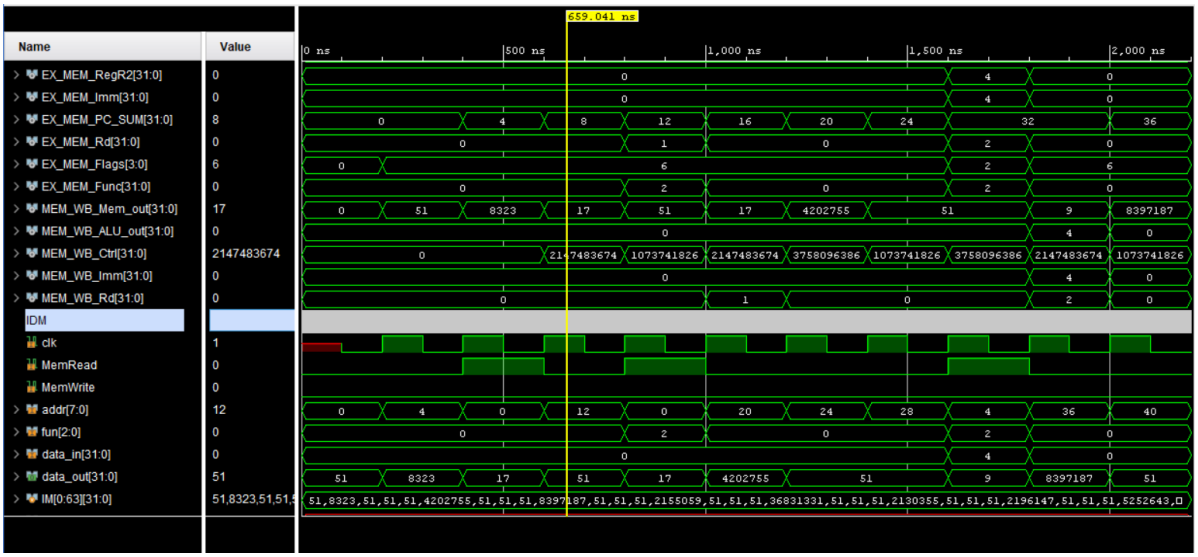
4

Simulation Testing and Results

The program was tested using the following simple code and was found to be functioning as expected. The program was found to have some control signal issues which are to be inspected in the upcoming milestone.

```
add x0, x0, x0
lw x1, 0(x0)
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
lw x2, 4(x0)
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
lw x3, 8(x0)
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
or x4, x1, x2
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
beq x4, x3, 16
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
add x3, x1, x2
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
add x5, x3, x2
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
sw x5, 12(x0)
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
lw x6, 12(x0)
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
and x7, x6, x1
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
sub x8, x1, x2
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
add x0, x1, x2
add x0, x0, x0
add x0, x0, x0
add x0, x0, x0
```

Note that NOP instructions were added to avoid the hazards which were not handled in this implementation. The results were shown to be as follows:


659.041 ns
0 ns
500 ns
1,000 ns
1,500 ns
2,000 ns
Name
Value
EX_MEM_RegR2[31:0]
EX_MEM_Imm[31:0]
EX_MEM_PC_SUM[31:0]
EX_MEM_Rd[31:0]
EX_MEM_Flags[3:0]
EX_MEM_Func[31:0]
MEM_WB_Mem_out[31:0]
MEM_WB_ALU_out[31:0]
MEM_WB_Ctrl[31:0]
MEM_WB_Imm[31:0]
MEM_WB_Rd[31:0]
IDM
clk
MemRead
MemWrite
addr[7:0]
fun[2:0]
data_in[31:0]
data_out[31:0]
IM[0:63][31:0]
51,8323,51,51,51,4202755,51,51,51,8397187,51,51,51,2155059,51,51,51,36831331,51,51,51,2130355,51,51,51,2196147,51,51,51,5252643,0

5

Conclusion

This report represented a quick survey of a trial to implement a pipelined RISC-V processor that supported the full unprivileged instruction set. More enhancements are to be done on this implementation in the upcoming milestones so that it supports the compressed instruction set as well as correcting some signals issues along with handling hazards and enabling data forwarding.