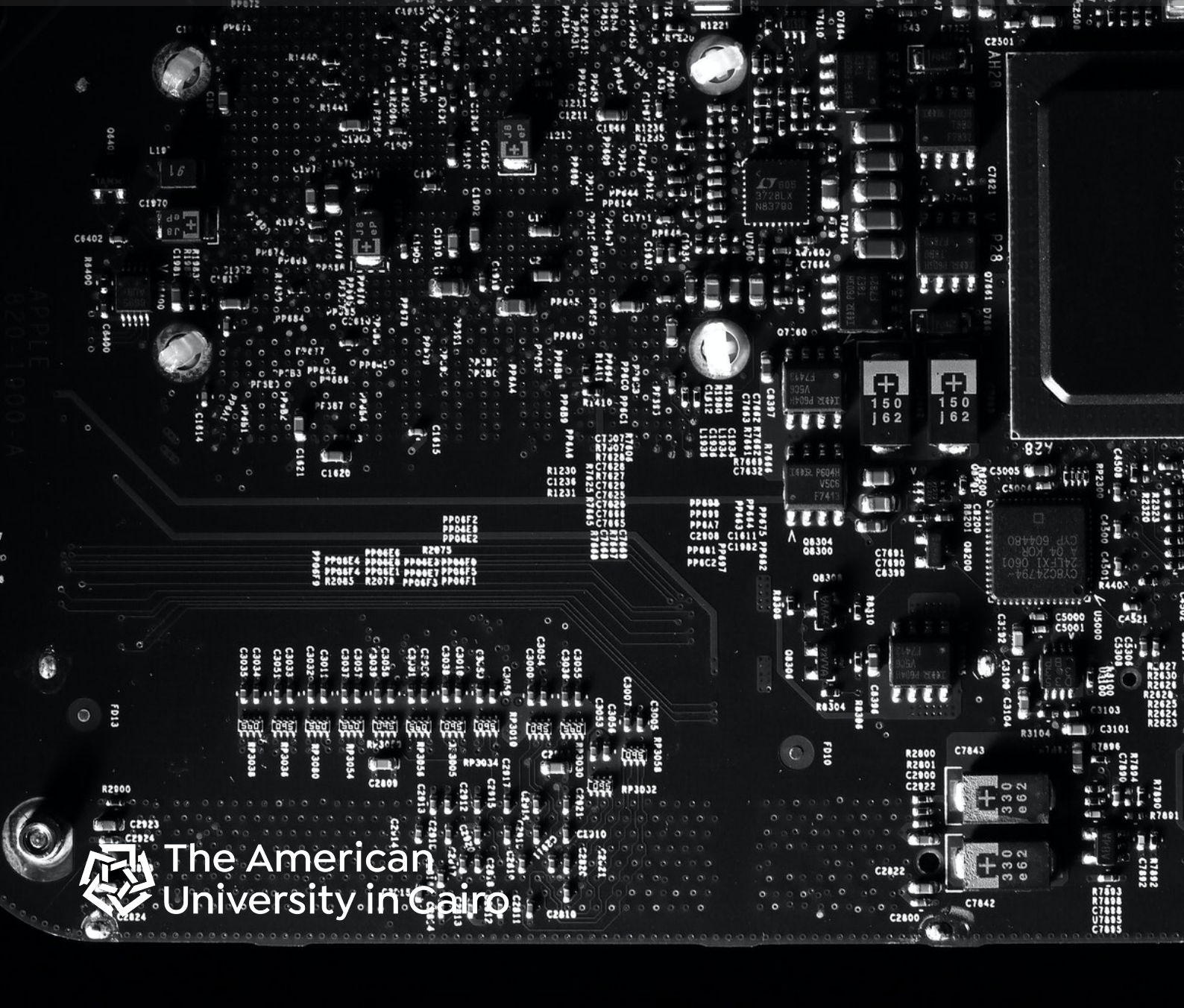
Single Cycle RISC-V Processor

Milestone 2

CSCE 3301 – Computer Architecture - F'22

Abdelaaty Rehab
Rana El Gahawy



The American University in Cairo

Single Cycle RISC-V Processor

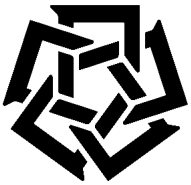
Milestone 2

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 single cycle processor supporting the rv32i user instruction sets (40 instructions).

Abdelaaty Rehab
Rana El Gahawy
November 2022

Summary

The single processor is the simplest implementation of a processor as it considers that every instruction consumes only one cycle in the processor to be executed. This is considered the basis of implementing pipelined processors which are to be discussed in the upcoming milestones of this project. Uptil now, the project includes a simple implementation of the single cycle processor that supports the full integer user instruction set.

Contents

1

Overview

1.1. Technical Summary

Within this project, we used the simple RISC-V single cycle implementation which was done in the lab attaching to it a few modifications as follows:

- Introducing the J type of instructions which depends on jumping to a certain address of the instruction memory saving the next program counter value to the destination register. It has different variations depending on the presence or the absence of an immediate value in the instruction.
- Supporting different kinds of branch instructions along with the BEQ using the flags that were available as outputs from the ALU. Those flags served as selection lines to select which kind of branch to be executed.
- Introducing the I type of instructions that are similar to the R ones except having the second source of the ALU to be the value generated by the immediate value generator.
- Modifying the memory to be byte addressable so that we can support the full S type of instruction along with the byte, halfword, word loading and storing.
- Supporting the environment calls like $FENCE$ and $EBREAK$ according to the assumption made in the project file description.

2

Data paths

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, ReadData;
    wire [31:0] gen_out, ALUResult;
    wire [3:0] ALU_Sel;
    wire [31:0] MUX_ALU;
    wire zf, cf, vf, sf;
    reg [7:0] PC;
    wire [7:0] Sum;
    wire [31:0] ShiftLeftOut;
    wire [31:0] Add4;
    reg [31:0] next_PC;
    reg [12:0] SSD;

    InstMem IM (PC[7:2], data_out);
    Control_Unit CU (data_out [6:2], Branch, MemRead, WhichReg, MemWrite, ALUSrc, RegWrite, Jump, ALUOp);
    registerFile #(32) RF (clk, rst , RegWrite, data_out [19:15], data_out [24:20], data_out [11:7], WriteData, ReadData1, ReadData2);
    rv32_ImmGen IG (data_out, gen_out);
    ALU_ControlUnit ACU (data_out[30], ALUOp, data_out [14:12], ALU_Sel);
    prv32_ALU ALU(ReadData1, MUX_ALU, ALUResult, zf, cf, vf, sf, ALU_Sel);
    DataMem DM (clk, MemRead,MemWrite, ALUResult[7:0], data_out[14:12], ReadData2, ReadData);
    n_bit_ShiftLeft #(32) SL (gen_out, ShiftLeftOut);
    Branch_CU BU(data_out[14:12], zf, cf, vf, sf, Branchflag);
    assign MUX_ALU = (ALUSrc)? gen_out : ReadData2;
    assign Sum = PC + gen_out;
    assign Add4 = 4 + PC;
    wire JB = {Jump, Branch};

    always @(*) begin
        case(WhichReg)
            3'b000: WriteData = ALUResult;
            3'b001: WriteData = ReadData;
            3'b010: WriteData = Add4;
            3'b011: WriteData = gen_out;
            3'b100: WriteData = PC + gen_out;
        endcase
    end

    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) ? Sum : Add4;
                2'b10: next_PC = (data_out[3]) ?  gen_out+ PC : gen_out + ReadData1 + PC;
            endcase

    end

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

endmodule
```

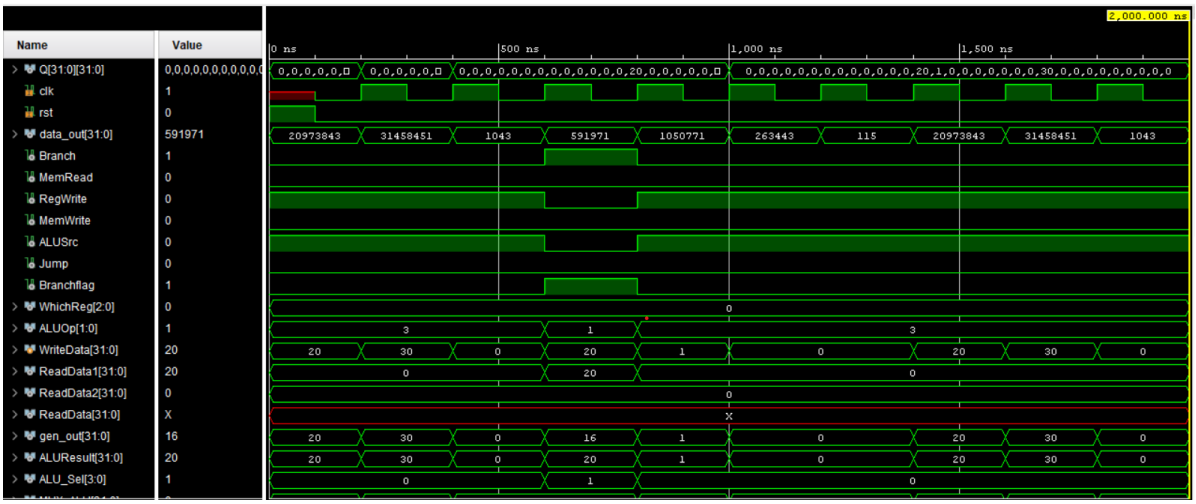
4

Simulation Testing and Results

The program was tested using the following simple code and was found to be functioning as expected. The full test cases are to be included within the compressed file as well.

```
addi        x18,x0,20
addi        x9,x0,30
addi        x8,x0,0
W: beq          x18,x0,L
add         x8,x8,x9
addi        x18,x18,-1
jal         x0,W
L:      addi x17,x0,1
addi        x10,x8,0
ecall
addi        x17,x0,10
ecall
```

The results were shown to be as follows:



More test cases were attached to the test bench so that we can uncomment sections to test different kinds of instructions.

5

Conclusion

This report represented a quick survey of a trial to implement a single cycle 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.