# 2024 Spring CSED311 Lab 2 Report

Team ID: 67735
20220312 박준혁, 20220871 홍지우

**명예서약 (Honor Code)**

나는 이 프로그래밍 과제를 다른 사람의 부적절한 도움 없이 완수하였습니다.

I completed this programming task without the improper help of others.

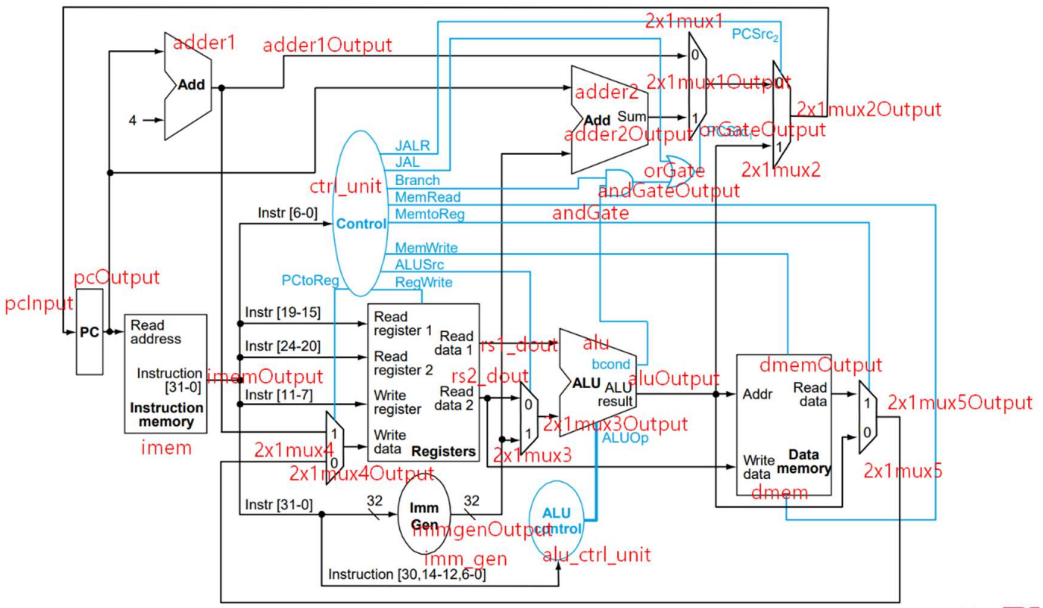
## 1 Introduction

RISC-V instruction 실행 가능한 Single-Cycle CPU를 verilog로 구현하였다.
모든 타입의 연산을 한 사이클에 실행 가능하다.

## 2 Design

### 2.1 전체 구조

기본적으로 Single-Cycle CPU 교안의 마지막에 나온 구조를 사용하였다.
PC, Inst memory, Register file, Data memory, imm generator, alu control ,control unit, ALU, Adder 두개와 Mux 5개, AND와 OR 게이트로 구성되어 있다.
코드에서 사용된 모듈과 wire 이름을 빨간 글씨로 적어 두었다.



## 2.2 모듈 설계.

cpu.v에 정의된 모듈들은 다음과 같다. 위의 그림대로 와이어를 연결하였다.

```
pc pc(
  .reset(reset),       // input (Use reset
  .clk(clk),          // input
  .next_pc(twomux2Output),     // input
  .current_pc(pcOutput)   // output
);
```

Program Counter 모듈

```
// ---------- Instruction Memory ----------
instruction_memory imem(
  .reset(reset),   // input
  .clk(clk),     // input
  .addr(pcOutput),    // input
  .dout(imemOutput)     // output
);
```

Instruction Memory 모듈

```
// ---------- Register File ----------
register_file reg_file (
  .reset(reset),        // input
  .clk(clk),          // input
  .rs1(imemOutput[19:15]),          // input
  .rs2(imemOutput[24:20]),          // input
  .rd(imemOutput[11:7]),           // input
  .rd_din(twomux4Output),       // input
  .write_enable(write_enable), // input
  .is_ecall(is_ecall), //input
  .rs1_dout(regfileOutputData1),     // output
  .rs2_dout(regfileOutputData2),     // output
  .print_reg(print_reg),  //DO NOT TOUCH THIS
  .is_halted(is_halted)        // output
);
```

Register File 모듈

```
// ---------- Control Unit ----------
control_unit ctrl_unit (
  .part_of_inst(imemOutput[6:0]),  // input
  .is_jal(is_jal),        // output
  .is_jalr(is_jalr),       // output
  .branch(branch),        // output
  .mem_read(mem_read),      // output
  .mem_to_reg(mem_to_reg),    // output
  .mem_write(mem_write),     // output
  .alu_src(alu_src),       // output
  .write_enable(write_enable),  // output
  .pc_to_reg(pc_to_reg),     // output
  .is_ecall(is_ecall)       // output (ecall inst)
);
```

Control Unit 모듈

```
// ---------- Immediate Generator ----------
immediate_generator imm_gen(
  .inst(imemOutput),  // input
  .imm_gen_out(immgenOutput)    // output
);
```

Immediate Generator 모듈

```
// ---------- ALU Control Unit ----------
alu_control_unit alu_ctrl_unit (
  .opcode(imemOutput[6:0]),  // input
  .funct3(imemOutput[14:12]),  // input
  .funct7_5(imemOutput[30]),  // input
  .alu_op(alu_op),          // output
  .btype(btype)          // output
);
```

ALU Control Unit 모듈

```
// ---------- ALU ----------
alu alu (
  .alu_op(alu_op),      // input
  .btype(btype),      // input
  .alu_in_1(regfileOutputData1),    // input
  .alu_in_2(twomux3Output),    // input
  .alu_res(aluOutput),  // output
  .alu_bcond(alu_bcond)    // output
);
```

ALU 모듈

```
// ---------- Data Memory ----------
data_memory dmem(
  .reset(reset),      // input
  .clk(clk),        // input
  .addr(aluOutput),       // input
  .din(regfileOutputData2),        // input
  .mem_read(mem_read),   // input
  .mem_write(mem_write),  // input
  .dout(dmemOutput)        // output
);
```

Data Memory 모듈

```
adder adder1(
  .x1(pcOutput),
  .x2(32'b100),
  .y(adder1Output)
);
adder adder2(
  .x1(pcOutput),
  .x2(immgenOutput),
  .y(adder2Output)
);
```

```
andGate andGate(
  .x1(branch),
  .x2(alu_bcond),
  .y(andGateOutput)
);
orGate orGate(
  .x1(is_jal),
  .x2(andGateOutput),
  .y(orGateOutput)
);
```

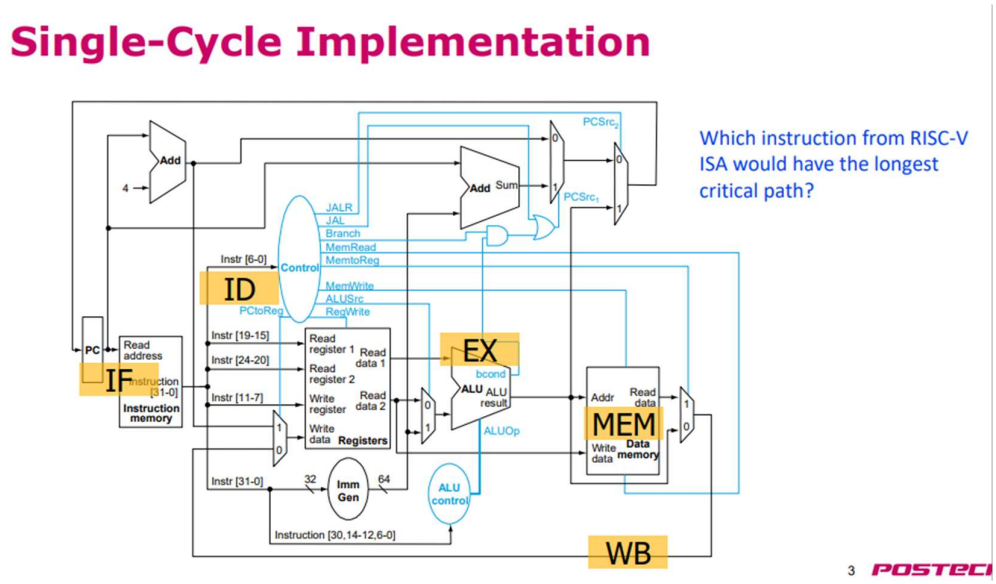
Adder, AND, OR 모듈

```
twomux twomux1(
  .x0(adder1Output),
  .x1(adder2Output),
  .sel(orGateOutput),
  .y(twomux1Output)
);
twomux twomux2(
  .x0(twomux1Output),
  .x1(aluOutput),
  .sel(is_jalr),
  .y(twomux2Output)
);
twomux twomux3(
  .x0(regfileOutputData2),
  .x1(immgenOutput),
  .sel(alu_src),
  .y(twomux3Output)
);
```

```
twomux twomux4(
  .x0(twomux5Output),
  .x1(adder1Output),
  .sel(pc_to_reg),
  .y(twomux4Output)
);
twomux twomux5(
  .x0(aluOutput),
  .x1(dmemOutput),
  .sel(mem_to_reg),
  .y(twomux5Output)
);
```

5개의 2:1 MUX 모듈

## 2.3 Stages




### 2.3.1 IF

Instruction Fetch: PC값을 기반으로 instruction을 불러오는 과정이다.
PC와 Instruction Memory에서 담당하는 역할이다.

### 2.3.2 ID

Instruction Decode: instruction에 정의된 값들을 해석해서 연산에 필요한 값을 생성하는 과정이다.

### 2.3.3 EX

Excute: ALU에서 연산하는 과정이다. 본 구조에서는 ALU가 하나 존재하며 ALU Control Unit 에서 제어한다.

### 2.3.4 MEM

Memory Access: 메모리에 접근하고 쓰는 과정이다.
load와 Store의 경우에만 사용되며, Control Unit에서 나오는 MemWrite, MemRead 신호에 따라 동작하게 된다.

### 2.3.5 WB

WriteBack: 연산 결과를 다음 과정으로 전달하는 과정이다.
ALU에서 연산한 값이거나 메모리에서 읽어온 값일 수 있다.

# 3 Implementation

## 3.1 Program Conter

```
always @(posedge clk) begin
    // Reset register file
    if (reset) begin
        current_pc <= 0;
    end
    else begin
        current_pc <= next_pc;
    end
end
```

Syncronous 하게 PC를 업데이트한다. 업데이트되는 값은 mux2 Output이다.

그 값은 PC+4이거나 PC+imm (JAL) 이거나 rs1+imm (JALR) 이다.

## 3.2 ALU

```
// calculate
always @(*) begin
    alu_res   = 0;
    alu_bcond = 0;
    case (alu_op)
        `FUNC_ADD: begin
            alu_res = alu_in_1 + alu_in_2;
        end
        `FUNC_SUB: begin
            alu_res = alu_in_1 - alu_in_2;
            // branch calculation
            case (btype)
                `BRANCH_EQ: alu_bcond = ((alu_res) == 0);
                `BRANCH_NE: alu_bcond = ((alu_res) != 0);
                `BRANCH_GE: alu_bcond = ((alu_res) >= 0);
                `BRANCH_LT: alu_bcond = ((alu_res) < 0);
                default: alu_bcond = 0;
            endcase
        end
        `FUNC_AND: alu_res = alu_in_1 & alu_in_2;  // AND
        `FUNC_OR:  alu_res = alu_in_1 | alu_in_2;  // OR
        `FUNC_XOR: alu_res = alu_in_1 ^ alu_in_2;  // XOR
        `FUNC_SLL: alu_res = alu_in_1 << alu_in_2;  // SLL
        `FUNC_SRL: alu_res = alu_in_1 >> alu_in_2;  // SRL
        default: begin
            alu_res   = 0;
            alu_bcond = 0;
        end
    endcase
end
```

Asynchrounous하게 주어진 연산을 한다.

ALU Control Unit에서 나온 alu_op 값에 따라 동작한다.

또한 btype에 따라 branch 계산도 한다..

## 3.3 ALU Control Unit

```
module alu_control_unit (
    input [6:0] opcode,
    input [2:0] funct3,
    input funct7_5,
    output reg [2:0] alu_op,
    output reg [2:0] btype
);
```

입력으로는 opcode와 funct3, funct7 [5]를 받아서 alu_op와 btype(branch)을 출력한다. 계산 공식은 교안에 나와있는 대로 구현하였다.

| | Options | Equation |
|---|---|---|
| **ALUOp** | • ADD, SUB, AND, OR, XOR, NOR, LT, and Shift<br>• btype: EQ, NE, GE, LT | case opcode<br>RTypeALU : according to funct3, funct7[5]<br>ITypeALU : according to funct3 only (except shift)<br>LW/SW/JALR : Add<br>Bxx : Subtract and select btype |

```
// change control values
always@(*) begin
    alu_op = `NOT_FUNC;
    btype=`NOT_BRANCH;
    case(opcode)
        //based on funct3, funct7
        `ARITHMETIC: begin
            // sub
            if(funct7_5==1) begin
                alu_op = `FUNC_SUB;
            end
            // not sub
            else begin
            case (funct3)
                `FUNCT3_ADD: alu_op = `FUNC_ADD;
                `FUNCT3_SLL: alu_op = `FUNC_SLL;
                `FUNCT3_XOR: alu_op = `FUNC_XOR;
                `FUNCT3_OR: alu_op = `FUNC_OR;
                `FUNCT3_AND: alu_op = `FUNC_AND;
                `FUNCT3_SRL: alu_op = `FUNC_SRL;
                default: alu_op = `NOT_FUNC;
            endcase
            end
        end
        // based on funct3
        `ARITHMETIC_IMM: begin
            case (funct3)
            `FUNCT3_ADD: alu_op = `FUNC_ADD;
            `FUNCT3_SLL: alu_op = `FUNC_SLL;
            `FUNCT3_XOR: alu_op = `FUNC_XOR;
            `FUNCT3_OR: alu_op = `FUNC_OR;
            `FUNCT3_AND: alu_op = `FUNC_AND;
            `FUNCT3_SRL: alu_op = `FUNC_SRL;
            default: alu_op = `NOT_FUNC;
            endcase
        end
```

```
        `LOAD, `STORE, `JALR: begin
            alu_op=`FUNC_ADD;
        end
        // branch setting
        `BRANCH: begin
            alu_op = `FUNC_SUB;
            case (funct3)
                `FUNCT3_BEQ: btype = `BRANCH_EQ;
                `FUNCT3_BNE: btype = `BRANCH_NE;
                `FUNCT3_BLT: btype = `BRANCH_LT;
                `FUNCT3_BGE: btype = `BRANCH_GE;
                default: btype = `NOT_BRANCH;
            endcase
        end
        default: begin
            alu_op = `NOT_FUNC;
            btype=`NOT_BRANCH;
        end
    endcase
end
```

Asynchrounous 하게 연산한다.

산술연산은 funct3을 기반으로 결정하며 IMM이 아닌 경우에 funct7로 SUB를 구분한다 BRANCH는 일단 두 값의 차이를 구한 다음 ALU에게 어떤 비교를 해야 하는지 btype을 통해 제어한다.

## 3.4 Control Unit

각 항목별 assert 조건은 교안의 표를 참고하였다.

| | When de-asserted | When asserted | Equation |
|---|---|---|---|
| **RegWrite** | GPR write disabled | GPR write enabled | (opcode!=SW/SH/SB) && (opcode!=Bxx) |
| **ALUSrc** | 2nd ALU input from 2nd GPR read port | 2nd ALU input from immediate | (opcode!=isRtype) && (opcode!=isSBtype) |
| **MemRead** | Memory read disabled | Memory read port return load value | opcode==LW/LH/LB |
| **MemWrite** | Memory write disabled | Memory write enabled | opcode==SW/SH/SB |
| **MemtoReg** | Steer ALU result to GPR write port | Steer memory load to GPR write port | opcode==LW/LH/LB |
| **PCtoReg** | Steer above result to GPR write port | Steer PC+4 to GPR write port | opcode==JAL/JALR |
| **$PCSrc_1$** | Next PC = PC + 4 | Next PC = PC + immediate | opcode==JAL \|\| (opcode==isSBtype && bcond) |
| **$PCSrc_2$** | Next PC is determined by $PCSrc_1$ | Next PC = GPR + immediate | opcode==JALR |

```
// setting control values
always @(*) begin
  is_jal = (part_of_inst == `JAL);
  is_jalr = (part_of_inst == `JALR);
  branch = (part_of_inst == `BRANCH);
  mem_read = (part_of_inst == `LOAD);
  mem_to_reg = (part_of_inst == `LOAD);
  mem_write = (part_of_inst == `STORE);
  alu_src = (part_of_inst != `ARITHMETIC && part_of_inst != `BRANCH);
  write_enable = (part_of_inst != `STORE && part_of_inst != `BRANCH);
  pc_to_reg = (part_of_inst == `JAL || part_of_inst == `JALR);
  is_ecall = (part_of_inst == `ECALL);
end
```

Asynchrounous하게 Opcode를 받아서 이에 맞는 컨트롤 값 설정을 해준다.

## 3.5 Imm Generator

| | | | | | | |
|---|---|---|---|---|---|---|
| imm[20\|10:1\|11\|19:12] | | | | rd | 1101111 | JAL |
| imm[11:0] | | rs1 | 000 | rd | 1100111 | JALR |
| imm[12\|10:5] | rs2 | rs1 | 000 | imm[4:1\|11] | 1100011 | BEQ |
| imm[12\|10:5] | rs2 | rs1 | 001 | imm[4:1\|11] | 1100011 | BNE |
| imm[12\|10:5] | rs2 | rs1 | 100 | imm[4:1\|11] | 1100011 | BLT |
| imm[12\|10:5] | rs2 | rs1 | 101 | imm[4:1\|11] | 1100011 | BGE |
| imm[11:0] | | rs1 | 010 | rd | 0000011 | LW |
| imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 | SW |
| imm[11:0] | | rs1 | 000 | rd | 0010011 | ADDI |
| imm[11:0] | | rs1 | 100 | rd | 0010011 | XORI |
| imm[11:0] | | rs1 | 110 | rd | 0010011 | ORI |
| imm[11:0] | | rs1 | 111 | rd | 0010011 | ANDI |
| 0000000 | shamt | rs1 | 001 | rd | 0010011 | SLLI |
| 0000000 | shamt | rs1 | 101 | rd | 0010011 | SRLI |
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | ADD |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | SUB |
| 0000000 | rs2 | rs1 | 001 | rd | 0110011 | SLL |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 | XOR |
| 0000000 | rs2 | rs1 | 101 | rd | 0110011 | SRL |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 | OR |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 | AND |
| 000000000000 | | 00000 | 000 | 00000 | 1110011 | ECALL |

Imm의 경우에는 assignment instruction에 있는 표를 참고하였다.

opcode에 따른 instruction의 각 위치를 imm에 할당하였다.

sign extension을 해주었으며

필요한 경우 (branch, JAL) 0으로 고정하는 비트도 설정했다.

Asynchrounous하게 계산하여 전달해 준다.

```
always @(*) begin
    imm_gen_out = 0;
    case (inst[6:0]) //opcode
        `ARITHMETIC_IMM, `LOAD, `JALR: begin
            imm_gen_out[11:0] = inst[31:20];

            // sign extension
            for (i = 12; i < 32; i = i + 1) begin
                imm_gen_out[i] = inst[31];
            end
        end
        `STORE: begin
            imm_gen_out[11:5] = inst[31:25];
            imm_gen_out[4:0]  = inst[11:7];

            // sign extension
            for (i = 12; i < 32; i = i + 1) begin
                imm_gen_out[i] = inst[31];
            end
        end
```

```
        `BRANCH: begin
            imm_gen_out[12] = inst[31];
            imm_gen_out[10:5] = inst[30:25];
            imm_gen_out[4:1] = inst[11:8];
            imm_gen_out[11] = inst[7];
            // fix to zero
            imm_gen_out[0] = 0;

            // sign extension
            for (i = 13; i < 32; i = i + 1) begin
                imm_gen_out[i] = inst[31];
            end
        end
        `JAL: begin
            imm_gen_out[20] = inst[31];
            imm_gen_out[10:1] = inst[30:21];
            imm_gen_out[11] = inst[20];
            imm_gen_out[19:12] = inst[19:12];
            // fix to zero
            imm_gen_out[0] = 0;

            // sign extension
            for (i = 21; i < 32; i = i + 1) begin
                imm_gen_out[i] = inst[31];
            end
        end
        default: imm_gen_out = 0;
```

## 3.6 Instruction Memory

Asynchrounous하게 메모리에 접근하여 instruction을 읽어오는 역할을 한다.

```
// TODO
// Asynchronously read instruction from the memory
// (use imem_addr to access memory)
always @(*) begin
    dout = mem[imem_addr];
end
```

## 3.7 Data Memory

```
// TODO
// (use dmem_addr to access memory)
// Asynchrnously read data from the memory
always @(*) begin
    if(mem_read==1 && mem_write==0) begin
        dout = mem[dmem_addr];
    end
    else begin
        dout = 0;
    end
end

// Synchronously write data to the memory
always @(posedge clk) begin
    if (mem_write==1 && mem_read==0) begin
        mem[dmem_addr] <= din;
    end
end
```

Asynchrounous하게 메모리에서 읽어온다.

clk에 맞춰 메모리에 값을 쓴다.

control unit에서 나오는

mem_read, mem_write에 따라 동작한다.

### 3.8 Register File

```
// TODO
// Asynchronously read register file
always @(*) begin
    rs1_dout = rf[rs1];
    rs2_dout = rf[rs2];

    // Halting Check
    if(is_ecall==1) begin
        is_halted = (rf[17]==10);
    end
    else begin
        is_halted = 0;
    end
end

// Synchronously write data to the register file
always @(posedge clk) begin
    if (write_enable && rd!=0) begin
        rf[rd] <= rd_din;
    end
end
```

Asynchrounous하게 레지스터를 읽는다.
clk에 맞춰 레지스터에 값을 쓴다.

이 모듈 내부에서 halt 수행을 한다.
ecall 신호가 발생할 시, x17==10을 검사한 뒤 is_halted 신호를 발생시킨다.

Control Unit에서 나오는 write_enable에 맞추어 동작한다.
x0은 hardwired 0으로 고정이기 때문에, x0에 write하지 못하게 한다.

### 3.9 Mux, Adder, Gate Module

```
// simple 2:1 mux
module twomux (
    input [31:0] x0,
    input [31:0] x1,
    input sel,
    output [31:0] y
);
    assign y = sel? x1:x0;
endmodule
```

```
// simple adder module
module adder (
    input  [31:0] x1,
    input  [31:0] x2,
    output [31:0] y
);
    assign y = x1 + x2;
endmodule
```

```
// simple AND module
module andGate (
    input x1,
    input x2,
    output wire y
);
    assign y = x1 & x2;
endmodule
```

```
// simple orGate
module orGate(
    input x1,
    input x2,
    output wire y
);
    assign y = x1 | x2;
endmodule
```

구현에 필요한 모듈들이다. Asynchrounous하게 작동한다.

## 4 Discussion

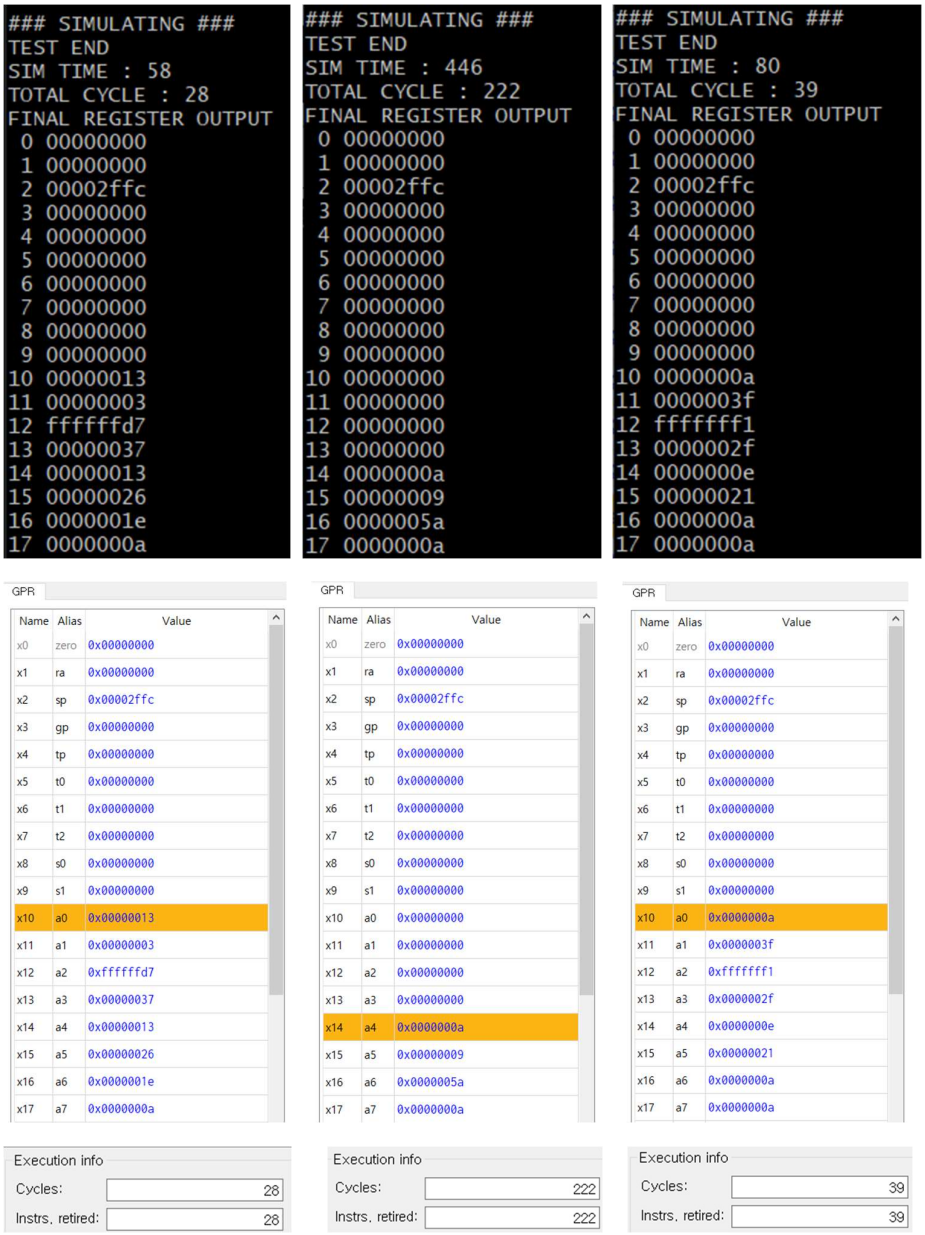
### 4.1

x0가 0으로 고정되어 있다는 것, imm에서 sign extension을 해야 한다는 것, 하위 비트를 0으로 고정하는 것을 신경써야 했다.

## 5 Conclusion

### 5.1 결과 비교 (다음 페이지)

x18부터는 사용하지 않기 때문에 사진에는 나와 있지 않다.




3개의 테스트벤치 basic_mem, loop_mem, non-controlflow에서 ripes와 전부 동일한 reg value와 사이클 수를 기록하였다.

이로써 Single-Cycle CPU를 성공적으로 구현하였다.