CSED311 Lab02

-Single Cycle CPU-

Team ID: 2

20180740 염재후

1. Introduction

이번 랩의 목표는 Single Cycle CPU의 구성요소의 동작원리와 동작과정을 이해하는 것이다. 큰 목표는 이해한 내용을 바탕으로, Single Cycle CPU를 구현하는 것이다. 이 때, CPU의 구성 요소인 register file, PC, Memory, ALU, Control Unit, Immediate Generator를 비롯한 다양한 모듈의 동작 원리를 이해할 수 있다. 또한 IF, ID, EX, MEM, WB의 single cycle CPU의 전체 과정을 이해할 수 있다.

2. Design

Single Cycle CPU의 submodule로는 크게 PC, Instruction Memory, Memory, Register File, ALU, ALU Control Unit, Control Unit, Immediate Generator로 나눌 수 있다.

CPU는 크게 Instruction Fetch, Instruction Decoding, Execution, Memory access, Write Back 총 5가지 stage로 이루어진다. Instruction type에 따라서 관여하는 stage는 다르지만, 모든 Instruction을 다루기 위해 다섯 stage가 필요하다는 것은 부정할 수 없다.

IF Stage는 Instruction memory address를 input으로 하여 instruction memory에서 instruction을 가지고 오는 단계이다. 이 때, instruction memory address는 pc에 저장되어 있으며, pc에서 나온 값을 instruction memory에 접근하여, 우리가 사용하게 될 instruction을 가지고 오게 된다.

ID Stage는 가지고 온 instruction을 각 모듈에 필요한 field를 넣어주어서 각 모듈에서 어떤 작업을 수행하는지 결정해주는 단계이다. 대표적으로 Register file에서는 번호에 맞는 레지스터 값을 가지고 오며, immediate generator는 opcode에 따라서

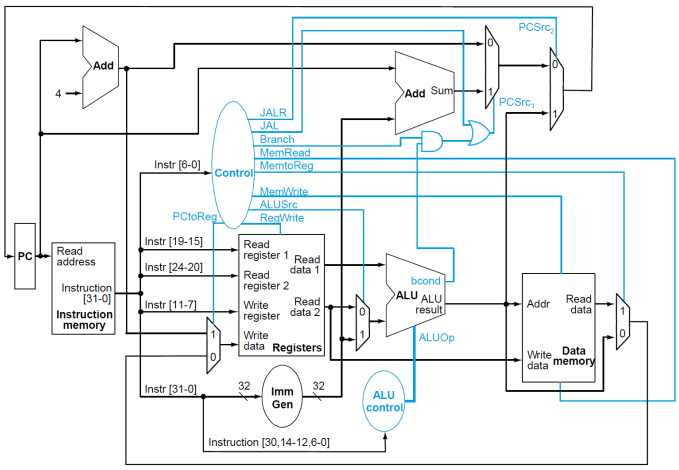
알맞은 immediate를 생성해준다.

EX stage는 CPU의 연산장치인 ALU가 여러 control bit에 맞춰서 연산을 실행하고, 실행된 연산 결과를 필요한 부분으로 보내주게 된다. Data memory에 접근하지 않는 instruction들은 대개 이 stage에서 종료가 된다.

MEM stage는 Data Memory에 접근하는 단계로, 메모리에 데이터를 쓰거나, 혹은 읽는 작업을 수행한다. S-Type이나 I-Type중 Load 연산 등이 해당 stage까지 오게 된다.

마지막으로 WB stage에서는 load한 instruction을 register로 불러오는 단계이다. Load 연산에서 주로 사용되는 단계로, 마지막 단계까지 오는 만큼, 다른 명령에 비해서 더 오랜 시간이 걸린다. 이는 결국 single cycle cpu의 한계를 여실히 보여주는 단계라고도 할 수 있다.

a. CPU



Lecture note에 나온 cpu의 설계가 정말 cpu를 잘 표현한다고 생각했기 때문에 위의 디자인을 따라서 구현을 하게 되었다. 각 module은 각 module에 정의된 일을 단순하게 수행한 이후에, 그 결과가 옳든, 옳지 않든, control unit에 의해서 옳은 값만 사용될 수 있게 설계가 되어 있으므로, modularization과 wire에 충실하여 디자인을 했다. 따라서, synchronous or asynchronous logic이라고 할 만한 logic part가 없다.

따라서, 각 Instruction memory, register files같은 module을 각각 구현한 이후, 이 module을 wire로 연결하는 방식을 통해서 전체 CPU를 구현하였다. 다만, ALU control의 경우는 조금 다르게 구현되었는데, 이는 ALU Control Unit

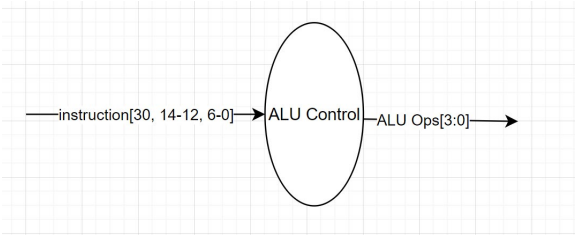
part에서 다시 다루도록 하겠다.

b. ALU Control Unit

ALU control unit의 경우, textbook에서는 opcode에 따라서 3bit ALU op code를 정한 후, 이후, Instruction format에서 다시 funct3 혹은 funct7에 따라서 specific한 alu control input을 정하는 방식으로 서술되어 있었다. 하지만, lecture note의 ppt에서는 이러한 과정이 modularization이 되어 있지 않았기 때문에 ALU control unit에서 모든 과정을 처리하도록 설계하였다.

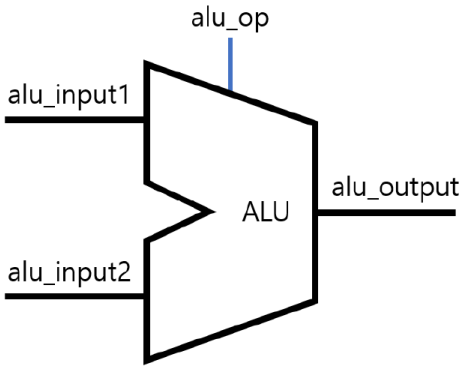
input으로는 2nd bit of funct7, funct3, opcode 위치의 bit를 받아왔다.

output으로는 asynchronous하게 각 bit의 combination에 따라서 알맞은 ALU op을 부여한 이후, 나머지 funct7, funct3에 따라서 ALU가 어떤 연산을 수행할지 결정하게 했다. 이 때, 각 ADD, SUB, SRL, SLL등 연산의 종류마다 binary number를 할당해주어서 그 정의해둔 값에 맞게 ALU에서 연산을 할 것이다.



c. ALU

ALU의 경우, 3가지의 input이 있고, 2가지의 output이 있다. Input으로는 rs1에 해당하는 register value가 alu_in_1로 들어오고, rs2에 해당하는 register value와 immediate generator에서 온 value를 mux를 선택해서 alu_in_2로 들어오게 된다. 이후, 4bit ALU OPS code에 따라서 수행할 연산을 결정해서, asynchronous하게 연산을 수행해준다. 그 결과를 alu_result로 내보내주고, branch인 경우 bcond에 알맞은 값을 할당하여 준다.



d. Control Unit

Control unit의 경우 instruction[6:0], 즉 opcode를 input으로 받은 이후, asynchronous하게 미리 define된 코드에 맞게, 값을 할당해준다. 값 할당은 lecture note의 control bit calculation table을 사용했다. 단, PCSrc1 signal의 경우에는, bcond와 같이, opcode에 포함되지 않은 정보가 있기 때문에, JAL인지 확인하는 wire만 뽑아낸 이후, cpu module 안에서 값을 continuous assign해주는 방식으로 디자인하였다.

	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

e. Immediate Generator

Immediate Generator의 경우, instruction code 전체를 input으로 받은 이후, asynchronous하게 opcode에 따라서 알맞게 bit slicing을 통해 값을 할당하여 32bit wire로 output을 지정해줬다.

f. PC

PC의 경우, 매우 간단한 module이었다. 외부에서 current pc +4, GPR[rs1]+ imm, PC+imm 한 값들을 control unit에서 나온 signal을 통해서 mux로 잘 선정한다면, 간단하게 next pc값을 input을 받게 되는데, 매 positive edge of clock마다 갱신해주면 된다.

input으로는 next pc가 들어오고, 매 positive edge of clock마다, current pc에 next pc를 할당해주면 된다.

output으로는 current pc value가 나가게 된다.

g. Instruction Memory

Instruction memory를 비롯해 memory, register file의 경우, 대부분 skeleton code에서 제공된 내용을 제외한다면, asynchronous하게 들어온 주소 값에 맞는 데이터를 읽어서 내보내는 역할을 하는 단순한 module이다.

h. Memory

input으로는 mem write 혹은 mem read에 맞는 signal bit가 들어오고, 읽을 데이터의 주소 혹은 쓸 데이터가 들어오게 되는데, 이 데이터들은 control signal에 알맞게 취하게 된다. 이 때, 데이터를 읽는 과정은 asynchronous하게 작용하지만 data를 쓰는 경우는 synchronous하게 positive edge에만 수행된다.

output으로는 memory에서 읽은 값들이 나가게 된다.

i. Register File

Register file에서는 각 계산에서 사용할 register의 번호 3개, 그리고 register에 저장할 데이터가 들어오게 된다. 물론 연산의 종류에 따라서 데이터를 쓰지 않을 수도, 혹은 2개의 register만 사용할 수도 있지만, 이는 regwirte, pc_to_reg와 같은 control signal 역시 input으로 들어와서 알맞은 선택이 가능하다. 결론적으로 asynchronous하게 register value를 읽어서 내보낸다. 동시에 synchronous하게 positive edge에 register value를 update해준다.

3. Implementation

a. CPU

앞서 Design section에서 내가 생각했던 CPU는 여러 모듈과 와이어로 이루어져 있는 모듈이었다. 코드는 각 모듈과 와이어 선언으로 인해 길기 때문에, 상세한 코드를 첨부 할 수 없었지만, 결국 CPU module 내부에서는 다른 reg나 combinational logic 혹은 synchronous logic 없이 각 모듈과 와이어사

이의 상호작용을 통해 기능이 작동하도록 구현하였다.

```
module CPU(input reset,       // positive reset signal
           input clk,         // clock signal
           output is_halted); // Whehther to finish simulation

  /***** Wire declarations *****/
        // wire for PC & Inst Memory
        wire [`WordSize -1: 0] wire_next_pc;
        wire [`WordSize -1 :0] wire_current_pc;
        wire [`WordSize -1: 0] wire_inst;

        assign pc_inst_out = wire_next_pc;

        // wire for RegisterFiles
        wire [`WordSize -1: 0] wire_rs1_dout;
        wire [`WordSize -1: 0] wire_rs2_dout;
        wire [`WordSize -1: 0] wire_rd_din;
        wire [`WordSize -1: 0] wire_semi_write_data;

        // wire for Immediate Generator
        wire [`WordSize -1: 0] wire_imm_gen_out;

        // wire for ALU and ALU Control
        wire [3:0] wire_alu_op;
        wire wire_alu_bcond;
        //wire [`WordSize -1: 0] wire_alu_in_1;
        wire [`WordSize -1: 0] wire_alu_in_2;
        wire [`WordSize -1: 0] wire_alu_result;

        // wire for Memory
        wire [`WordSize -1: 0] wire_constant_4;
        wire [`WordSize -1: 0] wire_dout;

        assign wire_constant_4 = 32'd4;

        // wire for Adder
        wire [`WordSize -1: 0] wire_add4_nxt_pc;
        wire [`WordSize -1: 0] wire_jal_nxt_pc;
        wire [`WordSize -1: 0] wire_jalr_nxt_pc;

        assign wire_jalr_nxt_pc = (wire_alu_result & 32'hfffffffe);

        // wire for MUXs
        wire [`WordSize -1: 0] wire_semi_nxt_pc;

        // wire for Control Unit
        wire wire_is_jal;
        wire wire_is_jalr;
        wire wire_branch;
```

```
  /***** Module Instance declarations *****/
        //adders
        adder #(.data_width(`WordSize)) adder_add4_nxt_pc(.a(wire_
        adder #(.data_width(`WordSize)) adder_jal_nxt_pc(.a(wire_i

        //MUXs
                //MUXs for PC
        mux2to1 #(.data_width(`WordSize)) mux_semi_nxt_pc(.input0(

        mux2to1 #(.data_width(`WordSize)) mux_nxt_pc(.input0(wire_
                //MUX for ALU
        mux2to1 #(.data_width(`WordSize)) mux_alu_input(.input0(wi
                //MUXs for Write in Reg
        mux2to1 #(.data_width(`WordSize)) mux_semi_write_data(.inp

        mux2to1 #(.data_width(`WordSize)) mux_write_data(.input0(w


  // ---------- Update program counter ----------
  // PC must be updated on the rising edge (positive edge) of the
  PC_pc(

  // ---------- Instruction Memory from Memory.v ----------
  InstMemory imem(

  // ---------- Register File from RegisterFile.v ----------
  RegisterFile reg_file (


  // ---------- Control Unit ----------
  ControlUnit ctrl_unit (


  // ---------- Immediate Generator ----------
  ImmediateGenerator imm_gen(

  // ---------- ALU Control Unit ----------
  ALUControlUnit alu_ctrl_unit (

  // ---------- ALU ----------
  ALU alu (

  // ---------- Data Memory from Memory.v----------
  DataMemory dmem(
endmodule
```

b. ALU control unit

ALU control unit은 10bit input을 받게 되는데, 이 10bit는 각각 funct7의 2nd bit와, funct3, opcode로 이루어져 있다. 그리고, ALU control unit은 주어진 input에 따라서, ALU가 수행할 연산을 선택하는 모듈이기 때문에, 주어진 signal을 사용하여, 분류해야 한다.

따라서, 먼저 7bit opcode로, 어떤 Type의 명령어인지 대분류를 한 이후에, (e.g. R-Type, I-Type, UJ-Type...etc) 각 타입마다, funct3 혹은 funct7의 사용 유무나, 특징이 다르기 때문에, 이 특징을 참고하여 소분류를 진행하였다.

```
        always@(*) begin
                case(part_of_inst[6:0])
                        `ARITHMETIC : begin
                                case (part_of_inst[9:7])
                                        `FUNCT3_AND : alu_op = `ALU_AND;
                                        `FUNCT3_OR : alu_op = `ALU_OR;
                                        `FUNCT3_XOR : alu_op = `ALU_XOR;
                                        `FUNCT3_ADD : alu_op = (part_of_inst[10] == 0)? `ALU_ADD : `ALU_SUB;
                                        `FUNCT3_SLL : alu_op = `ALU_SLL;
                                        `FUNCT3_SRL : alu_op = `ALU_SRL;
                                endcase
                        end

                        `ARITHMETIC_IMM : begin
                                case(part_of_inst[9:7])
                        end

                        `LOAD : begin

                        `JALR : begin


                        `STORE : begin

                        `BRANCH : begin
                                case(part_of_inst[9:7])
                        end

                        `JAL : begin

                endcase

        end
```

c. ALU

ALU의 구현은 비교적 간단하다. Modularization을 통해 어떤 연산을 할지 결정하는 로직을 ALU control unit에서 구현하였기 때문에, ALU control unit에서 오는 signal을 case문을 사용하여 어떤 연산을 할 지 결정하고 수행하면 되기 때문이다.

주의해야 할 점으로는, 먼저 shift 연산에서, rs2의 값 중 5bit만 slicing해서 사용해야 한다는 점이다. 두 번째로는, BGE, BLT 같은 연산의 경우, signed 연산을 통해 비교해야 한다. Verilog docs에 의하면, wire, reg와 같은 type들은 default로 unsigned가 되기 때문에, 주의하지 않는다면, -1이 BGE 연산에서 양수보다 크다고 판단될 수 있다.

```
`,
    input [3:0]alu_op;
    input [`WordSize -1: 0]alu_in_1;
    input [`WordSize -1: 0]alu_in_2;
    output reg [`WordSize -1: 0]alu_result;
    output reg alu_bcond;

    always@(*) begin
            case (alu_op)
                    `ALU_AND : alu_result = (alu_in_1 & alu_in_2);
                    `ALU_OR : alu_result = (alu_in_1 | alu_in_2);
                    `ALU_ADD : alu_result = (alu_in_1 + alu_in_2);
                    `ALU_SUB : alu_result = (alu_in_1 - alu_in_2);
                    `ALU_SLL : alu_result = (alu_in_1 << alu_in_2[4:0]);
                    `ALU_SRL : alu_result = (alu_in_1 >> alu_in_2[4:0]);
                    `ALU_XOR : alu_result = (alu_in_1 ^ alu_in_2);
                    `ALU_BEQ : begin
                            alu_result = alu_in_1 - alu_in_2;
                            if(alu_result == 0 )
                                    alu_bcond = 1;
                            else
                                    alu_bcond = 0;
                            end
                    `ALU_BNE : begin
                            alu_result = alu_in_1 - alu_in_2;
                            if(alu_result != 0)
                                    alu_bcond = 1;
                            else
                                    alu_bcond = 0;
                            end

                    `ALU_BLT : begin
                            alu_result = alu_in_1 - alu_in_2;
                            if($signed(alu_result) < 0)
                                    alu_bcond = 1;
                            else
                                    alu_bcond = 0;
                            end
                    `ALU_BGE : begin
                            alu_result = alu_in_1 - alu_in_2;
                            if($signed(alu_result) >= 0)
                                    alu_bcond = 1;
                            else
                                    alu_bcond = 0;
                            end
            endcase
    end
```

d. Control Unit

Control Unit을 구현하는데 있어서도, 상당히 lecture note의 Design을 충실하게 따라서 구현하게 되었다. Single bit control signal을 이 모듈에서 생산하는데, 이 control signal들 중 하나만 사용되는 것이 아니라, CPU의 하위 모듈에 모두 사용된다. 따라서 case문을 사용하기 보다는 전체 signal들을 따로 계산해주어야 한다. 나의 경우에는 always 문을 사용하여 opcode가 바뀔 때마다 이 control signal들을 갱신해줬다.

```
always@(part_of_inst) begin
        write_enable = (part_of_inst != `STORE &&
                part_of_inst != `BRANCH && part_of_inst != `ECALL);

        alu_src = (part_of_inst != `ARITHMETIC &&
                part_of_inst != `BRANCH);

        mem_read = (part_of_inst == `LOAD);
        mem_write = (part_of_inst == `STORE);
        mem_to_reg = (part_of_inst == `LOAD);
        pc_to_reg = (part_of_inst == `JAL ||
                part_of_inst == `JALR);

        is_jal = (part_of_inst == `JAL);
        is_jalr =(part_of_inst == `JALR);
        branch = (part_of_inst == `BRANCH);

        is_ecall = (part_of_inst == `ECALL);
end
```

e. Immediate Generator

Immediate generator는 instruction을 받아서 32bit immediate를 output으로 내보내는데, 이 instruction type마다, immediate bit가 서로 다른 위치에 있기 때문에 case문과 opcode를 통해서 instruction type을 나눠준 이후에 각 type에 맞게 field를 재조합해서 내보냈다.

```
module ImmediateGenerator(
        part_of_inst,
        imm_gen_out
);
        input [`WordSize -1: 0] part_of_inst;
        output reg[`WordSize -1: 0] imm_gen_out;

        always@(*) begin
                case(part_of_inst [6:0])
                        `ARITHMETIC : imm_gen_out = 32'd0;

                        `ARITHMETIC_IMM : imm_gen_out = $signed(part_of_inst[31:20]);

                        `LOAD : imm_gen_out = $signed(part_of_inst[31:20]);

                        `JALR : imm_gen_out = $signed(part_of_inst[31:20]);

                        `STORE : imm_gen_out = $signed({part_of_inst[31:25], part_of_inst[11:7]});

                        `BRANCH : imm_gen_out = $signed({part_of_inst[31], part_of_inst[7], part_of_inst[30:25], part_of_inst[11:8], 1'b0});

                        `JAL : imm_gen_out = $signed({part_of_inst[31], part_of_inst[19:12], part_of_inst[20], part_of_inst[30:21], 1'b0});
                endcase
        end

endmodule
```

f. PC

PC의 경우, synchronous하게 next pc 값을 통해 current pc값을 갱신해주고 reset signal이 올 때, pc를 reset시켜주는 역할이 전부이다. Next pc value는 cpu에서 mux와 다른 control signal들이 알맞은 값을 정해서 input으로 넣어주기 때문에, 복잡한 모듈은 아니다.

```
        input reset;
        input clk;
        input [`WordSize -1 : 0] next_pc;
        output reg [`WordSize -1 : 0] current_pc;

        always@(posedge clk)begin
                if(reset)
                        current_pc <= `WordSize'd0;
                else
                        current_pc <= next_pc;
        end
endmodule
```

g. Instruction Memory

만약 instruction memory 전체를 구현해야 했다면, 까다로웠을 것이다. 하

지만, memory 주소를 계산해주는 작업이나, 혹은 memory를 초기화해주는 작업이 스켈레톤 코드에서 구현되어 있었기 때문에, 다음과 같이 immediate address가 갱신된 경우에, instruction을 내보내는 로직만을 구현하게 되었다.

```
// (use imem_addr to access memory)
      always@(*) begin
              if(!reset)
                      dout = mem[imem_addr];
      end
```

h. Memory

Memory의 경우 읽는 작업과 쓰는 작업 두 가지 종류의 작업이 있고, 각각 asynchronous, synchronous하기 때문에 서로 다른 always block을 쓰게 되었다.

```
  // TODO
  // Asynchrnously read data from the memory
        always@(*) begin
                if(mem_read) begin
                        dout = mem[dmem_addr];
                end
        end
  // Synchronously write data to the memory
  // (use dmem_addr to access memory)
        always@(posedge clk) begin
                if(mem_write) begin
                        mem[dmem_addr] <= din;
                end
        end
```

i. Register file

Register file 역시 memory와 비슷한 역할을 수행한다. Register를 읽고 쓰는 작업이다. 대신, 다른 점으로는 memory가 아니라 register이고, x0 register는 항상 0값을 가지고 있어야 하기 때문에, x0 register에는 다른 값을 써서는 안된다는 것이다.

```
// TODO
// Asynchronously read register file
      always@(*) begin
              rs1_dout = rf[rs1];
              rs2_dout = rf[rs2];
      end
// Synchronously write data to the register file
      always@(posedge clk) begin
              if (write_enable && rd != 0)
                      rf[rd] <= rd_din;
      end
```

4. Discussion

먼저, Instruction Memory, Memory, PC, Register file에서는 크게 문제가 될 만한 내용이 없었다. 들어온 주소/레지스터 번호에 알맞게 값을 읽어내면 되는 문제였기 때문이다.

이번 과제에서 단언컨대, 가장 애매하고, 어려웠던 파트는 ALU Control Unit이다.그 이유는 ALU Ops와 ALU Control input의 차이이다. ALU Ops는 2bit로 opcode에 의거해서 만들어지는 signal인데, 이 signal은 바로 다른 모듈로 가서 영향을 미치는 것이 아니라, 다시 funct code에 따라서 세부적인 alu control input으로 바뀌게 된다. 따라서 이를 정확히 알고 있지 않는다면, ALU Control unit을 구현하는데 있어서 많이 헷갈릴 수밖에 없다.

또한, 세부적인 디테일로는, JAL로 계산한 주소 값에는 0xfffffffe를 &연산을 통해서 홀수가 되지 않도록 값을 조정해줘야 한다는 것을 까먹으면 안된다는 것이 있다. 그리고, ALU에서 shift 연산을 실행할 때, alu_in_1 << alu_in_2 형태를 취하게 되는데, 이 때, alu_in_2의 경우, 5개의 bit만을 다시 slicing해주지 않으면, 값이 잘못될 수 있다는 점을 알아야만 한다. I-Type instruction의 ALU ops 결정은 funct3 code만으로 충분하지만, shift가 예외이기 때문이다. 물론, 이 lab에서는 SRAI를 다루지는 않지만, SRAI의 경우, imm[10] bit가 1로 표시되어 있기 때문에 잘못된 값을 야기할 수 있다.

5. Conclusion

결론적으로, Single Cycle CPU에서는 모든 instruction이 한 cycle안에 실행된다고 가정하기 때문에, 각 모듈의 제어 측면에서 multi cycle이나 pipeline에 비해서 간단하게 구현할 수 있다. 다만, JAL의 주소 계산이나, shift 연산의 경우와 같이 정확한 CPU의 역할을 구현하기 위해서는 CPU 자체의 동작 원리나 순서와 더불어 ISA에 대해서도 깊은 이해가 필요하다는 것을 알 수 있었다.