2025 Spring

CSED311

컴퓨터구조

Lab 2 report

Team ID: 15

팀원 1: 20230345 이성재

팀원 2: 20230355 정지성

목차

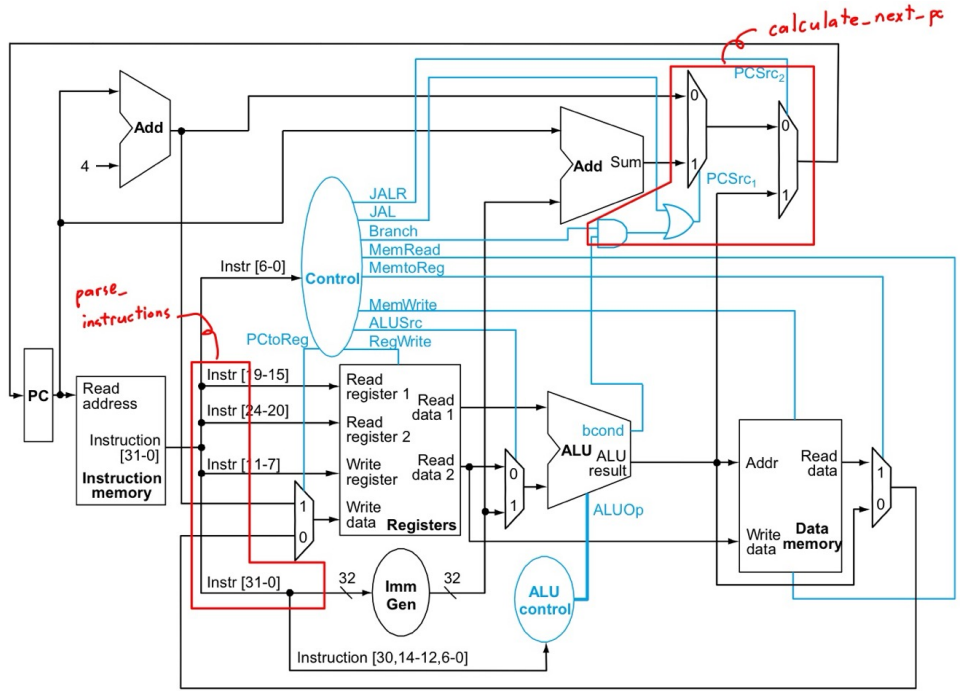
I. 서론

이 과제에서 우리는 Single cycle cpu를 베릴로그로 구현했다. CPU를 구성하는 기본 모듈들을 설계하고 구현하였으며 주어진 테스트를 이용해 확인한 결과 RIPES 시뮬레이터 상의 레지스터 값들과 동일한 레지스터 값을 가졌다. 이후 과제들을 통해 더 최적화 된 형태의 cpu를 개발하게 될텐데, 이번 과제를 통해 기본적인 cpu 구조를 명확히 이해함으로써 이후 더 개선된 구조의 cpu를 구현하는데 있어 어려움이 없도록 하는 것이 목표라고 할 수 있다.

II. 디자인

우선 각 모듈의 기능을 설명한 후, 모듈을 synchronous와 asynchronous module로 구분하겠다.

이 과제에서 기본적인 모듈들은 강의 슬라이드 CH5 의 34 페이지를 참고했다. 대부분의 모듈(instruction & data memory, registers, control units, etc.)은 아래 그림에 따라 구현했다. 새로 디자인한 모듈들도 cpu.v 파일을 정리하기 위해 로직들은 모듈로 캡슐화한 것이라고 생각할 수 있는데, 그것들도 아래 그림에서 빨간 박스를 통해 표시했다.



모듈의 기능 설명

이제부터 각 모듈의 기능을 대략적으로 설명하겠다. 데이터의 흐름을 최대한 반영하여 순서를 배치했다.

A. pc

Input

- **reset**, **clk** (synchronous logic을 위한 정보)
- **next_pc**

Output: **current_pc**

- 매 사이클 읽을 instruction의 주소를 업데이트한다.

B. instruction_memory

Input

- **reset**, **clk** (synchronous logic을 위한 정보)
- **addr** (명령어의 주소)

Output: **dout** (현재 명령어)

- PC의 메모리에 따라 현재 명령어를 출력해 주는 모듈이다.

C. parse_instructions

Input: **instruction**

Output: **opcode**, **funct3**, **funct7**, **rs1**, **rs2**, **rd**

- Instruction을 RISC-V cpu의 모듈이 읽기 좋은 형태의 정보로 잘라 주는 역할을 하는 간단한 모듈이다.

D. control_unit

Input: **opcode**

Output: **is_jal**, **is_jalr**, ... (다양한 플래그)

- 현재 instruction의 opcode를 이용해서, 다양한 모듈의 동작을 조절해주는 플래그를 여럿 출력하는 모듈이다.

E. register_file

Input

- **reset**, **clk** (synchronous logic을 위한 정보)
- **rs1, rs2** (read 레지스터 선택)
- **rd** (write 레지스터 선택)
- **rd_din** (writeback 시 rd에 저장되는 값)
- **write_enable** (writeback 시 1로 설정)

Output

- **rs1_dout, rs2_dout** (선택된 레지스터의 값들)
- **ecall_reg_cond, print_reg** (프로그램 종료 및 레지스터 상태 출력)

- 32개의 레지스터 중 최대 두 개를 선택하여 값을 보여줄 수 있다.
- 32개의 레지스터 중 하나를 선택하여 값을 저장할 수 있다.

F. immediate_generator

Input: **instruction**

Output: **imm_gen_out**

- 현재 instruction의 opcode를 이용해서, 그 명령어에 해당하는 immediate값을 decode해 주는 모듈이다.

G. alu_control_unit

Input: **funct3**, **funct7**, **opcode**

Output: **alu_op** (ALU가 실행해야 할 연산의 종류를 알려주는 값)

- alu 모듈이 정확히 어떤 연산(e.g., ADD, LLS, etc)을 해야 할지 정해서 alu에게 알려주는 역할을 수행한다.
    - 연산의 종류는 funct3, funct7, opcode를 알아야 결정할 수 있으므로 입력을 위와 같이 받고 4비트로 이루어진 alu_op를 출력한다. alu_op에 대한 규칙은 alu_func.v에 정의되어 있다.

H. alu

Input

- **alu_op** (연산의 종류)
- **alu_in_1, alu_in_2** (연산당할 값)

Output: **alu_result, alu_bcond** (bcond: branch일 경우 jump 여부)

- Lab1에서 제작했던 alu.v와 비슷한 구조로 작성되었지만 BEQ 등 branch를 할 수도 있는 연산을 처리할 경우 alu_bcond의 값을 결정하는 로직이 추가되었다.

I. calculate_next_pc

Input

- **current_pc_plus_4**, **branch_jal_address**, **jalr_address**, (다음 pc 값들)
- **is_jalr**, **is_jal**, **branch**, **bcond** (위의 값 중 하나를 선택하기 위한 정보)

Output: **next_pc**

- 현재 pc값과 instruction을 바탕으로 만들어진 "다음 pc값들"을 input으로 받는다.
- 또한, 저 세 pc값 중 하나를 결정하기 위한 control값들도 모조리 받은 뒤 이 값을 이용해 세 개 중 하나의 값을 내부적으로 선택한다.

J. data_memory

Input

- **reset, clk** (synchronous logic을 위한 정보)
- **addr** (명령어의 주소)
- **din** (write을 할 경우의 데이터)
- **mem_read, mem_write** (read / write 결정 신호)

Output: **dout** (선택된 주소의 값)

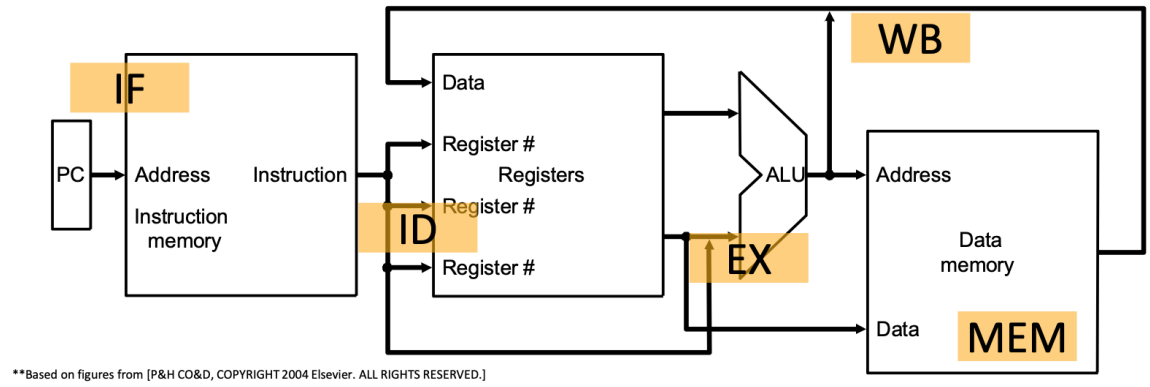
- 메모리에 접근해 저장된 데이터를 레지스터에 저장하거나, 레지스터에 저장된 데이터(din)을 메모리에 저장하는 모듈이다.

Clock Synchronous Module vs Asynchronous Module

- 기본적으로는 sequential logic으로 구현된 모듈은 clock synchronous하고, 그렇지 않은 모듈은 asynchronous하다고 볼 수 있다.
- 이에 따르면 우리의 cpu 디자인에서 clock synchronous한 모듈에는 **`pc`**와 **`register_file`** (레지스터), **`data_memory`**와 **`instruction_memory`** (메모리)가 있다.
    - 이 경우에도 저장된 값을 변화시키는 경우 (pc를 업데이트하는 경우, 메모리에 값을 write하는 경우 등)에만 synchronous하고, 저장된 값을 이용(출력)하는 것은 asynchronous하다.
- 나머지 모듈들은 전부 asynchronous한 combinational logic이다.

III. 구현

Single cycle cpu의 5가지 단계를 거치며 어느 모듈이 사용되는지 설명하는 방식으로 구현 방법을 설명하도록 하겠다.



1. IF

- **주요 모듈: `pc`, `instruction_memory`, `calculate_next_pc`**

IF(Instruction Fetch)는 PC값을 기반으로 instruction을 만들어 내는 단계이다.

`calculate_next_pc`: 전 instruction을 통해 계산된, 업데이트해야 할 next_pc의 값이다. Jalr, (jal or branch), 평소의 세 가지 상황으로 나누어 할당한다.

```
always @(*) begin
    next_pc = current_pc_plus_4;
    if (is_jalr) begin
        next_pc = jalr_address;
    end

    else if (branch && bcond || is_jal) begin
        next_pc = branch_jal_address;
    end
end
```

`pc`: 평범한 register 파일이다. Positive edge일 때마다 input으로 들어오는 값으로 register의 값을 업데이트한다.

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

`instruction_memory`: pc로부터 주소를 받으면, 모듈에 저장된 명령어를 다음 stage로 전달한다.

```
// TODO
// Asynchronously read instruction from the memory
// (use imem_addr to access memory)

assign dout = mem[imem_addr];
```

2. ID

- **주요 모듈: `parse_instructions`, `immediate_generator`, `register_file`, `control_unit`**

ID(Instruction Decode)는 instruction에 담긴 여러 정보들을 decode해서 연산에 필요한 정보들을 준비해 놓는 단계이다.

`parse_instruction`: 말 그대로 32비트짜리 instruction을 다른 모듈이 쓰기 좋은 형태로 잘라 주는 모듈이다.

```
assign opcode = instruction[6:0];
assign funct3 = instruction[14:12];
assign funct7 = instruction[31:25];
assign rs1 = instruction[19:15];
assign rs2 = instruction[24:20];
assign rd = instruction[11:7];
```

`immediate_generator`: opcode에 따라서 immediate 값을 decode해 주는 모듈이다. 나중에 계산에서 상수 값을 사용해야 할 때 그 상수값을 계산해 준다.

```
always @(*) begin
    case(opcode)
        `JAL : imm_gen_out = {{12{instruction[31]}}, instruction[19:12], instruction[20], instruction[30:21], 1'b0};
        `JALR : imm_gen_out = {{20{instruction[31]}}, instruction[31:20]};
        `BRANCH : imm_gen_out = {{20{instruction[31]}}, instruction[7], instruction[30:25], instruction[11:8], 1'b0};
        `LOAD : imm_gen_out = {{20{instruction[31]}}, instruction[31:20]};
        `STORE : imm_gen_out = {{20{instruction[31]}}, instruction[31:25], instruction[11:7]};
        `ARITHMETIC_IMM : imm_gen_out = {{20{instruction[31]}}, instruction[31:20]};
        default: imm_gen_out = 32'b0;
    endcase
end
```

`register_file`: 레지스터의 값을 읽거나 쓰는 작업을 수행한다. (write 과정은 **WB** 단계에서 일어난다. 추후 설명 예정) read 과정은 주어진 rf 레지스터를 참조하는 것으로 구현할 수 있다. 또한 우리 cpu 디자인은 (ecall) && x17 == 10일 때 중지하는데, 이 로직을 구현하기 위해 **`ecall_reg_cond`**의 정보도 output으로 내놓는다.

```
// TODO
// Asynchronously read register file
assign ecall_reg_cond = rf[17] == 10 ? 1 : 0;
assign rs1_dout = rf[rs1];
assign rs2_dout = rf[rs2];
```

control_unit: 각 opcode에 따라 켜고 꺼야 할 flag들을 명시한 모듈이다. 지금 코드는 opcode를 케이스로 분류한 후 각 flag를 조작하는 방식이지만, 다 짜고 나니 (flag) = (이 flag를 켜야 하는 opcode들) 형태로 짜는 것이 더 간결할 것 같았다는 생각이 든다.

```
        is_jal = 1'b0;
        is_jalr = 1'b0;
        branch = 1'b0;
        mem_read = 1'b0;
        mem_to_reg = 1'b0;
        mem_write = 1'b0;
        alu_src = 1'b0;
        write_enable = 1'b0;
        pc_to_reg = 1'b0;
        is_ecall = 1'b0;
        case(opcode)
            `ARITHMETIC : write_enable = 1'b1;
            `ARITHMETIC_IMM : begin
                alu_src = 1'b1;
                write_enable = 1'b1;
            end
            `LOAD : begin
                mem_read = 1'b1;
                mem_to_reg = 1'b1;
                alu_src = 1'b1;
                write_enable = 1'b1;
            end
            `JALR : begin
                is_jalr = 1'b1;
                alu_src = 1'b1;
                write_enable = 1'b1;
                pc_to_reg = 1'b1;
            end
            `STORE : begin
                mem_write = 1'b1;
                alu_src = 1'b1;
            end
```

```
            `BRANCH : branch = 1'b1;
            `JAL : begin
                is_jal = 1'b1;
                write_enable = 1'b1;
                pc_to_reg = 1'b1;
            end
            `ECALL : is_ecall = 1'b1;

            default : begin
                is_jal = 1'b0;
                is_jalr = 1'b0;
                branch = 1'b0;
                mem_read = 1'b0;
                mem_to_reg = 1'b0;
                mem_write = 1'b0;
                alu_src = 1'b0;
                write_enable = 1'b0;
                pc_to_reg = 1'b0;
                is_ecall = 1'b0;
            end
        endcase
    end
```

3. EX

- **주요 모듈: `alu_control_unit, alu`**

EX(Execution)은 ALU가 연산을 실행하는 과정이다.

`alu_control_unit: alu`가 처리할 연산의 종류를 구분해준다. 가장 먼저 opcode를 이용해 연산의 종류가 arithmetic, arithmetic_imm, load,store, jalr, jal, branch인지 구분한다. 다음으로 funct3의 값을 이용해서 연산의 종류를 결정짓고, 몇가지 경우에선 funct7의 값까지 이용해야 연산의 종류를 결정지을 수 있으므로 삼항 연산자를 이용해 구분을 지어줬다. 코드는 아래와 같다.

```
always @(*) begin
    case(opcode)
        `ARITHMETIC : begin
            case(funct3)
                `FUNCT3_ADD : alu_op = (funct7 == `FUNCT7_SUB) ? `FUNC_SUB : `FUNC_ADD;
                `FUNCT3_SLL : alu_op = `FUNC_LLS;
                `FUNCT3_XOR : alu_op = `FUNC_XOR;
                `FUNCT3_AND : alu_op = `FUNC_AND;
                `FUNCT3_OR : alu_op = `FUNC_OR;
                `FUNCT3_SRL : alu_op = (funct7 == `FUNCT7_SUB) ? `FUNC_ARS : `FUNC_LRS;
                default : alu_op = `FUNC_ZERO;
            endcase
        end
        `ARITHMETIC_IMM : begin
            case(funct3)
                `FUNCT3_ADD : alu_op = `FUNC_ADD;
                `FUNCT3_SLL : alu_op = `FUNC_LLS;
                `FUNCT3_XOR : alu_op = `FUNC_XOR;
                `FUNCT3_OR : alu_op = `FUNC_OR;
                `FUNCT3_AND : alu_op = `FUNC_AND;
                `FUNCT3_SRL : alu_op = (funct7 == `FUNCT7_SUB) ? `FUNC_ARS : `FUNC_LRS;
                default : alu_op = `FUNC_ZERO;
            endcase
        end
        `LOAD, `STORE, `JALR, `JAL: begin
            alu_op = `FUNC_ADD;
        end
        `BRANCH : begin
            case(funct3)
                `FUNCT3_BEQ : alu_op = `FUNC_BEQ;
                `FUNCT3_BNE : alu_op = `FUNC_BNE;
                `FUNCT3_BLT : alu_op = `FUNC_BLT;
                `FUNCT3_BGE : alu_op = `FUNC_BGE;
                default : alu_op = `FUNC_ZERO;
            endcase
        end
        default : alu_op = `FUNC_ZERO;

    endcase
end
```

이때 R-type 연산의 경우 ADD와 SUB을 funct3만으로는 구분할 수 없어 funct7이 무슨 값인지를 확인한다. 비슷하게 SRL과 SRA, 그리고 I-type의 SRL과 SRA도 funct7 값을 알아야 하기 때문에 삼항 연산자를 이용했다.

LOAD, STORE, JALR, JAL의 경우 모두 alu가 ADD 연산을 하면 된다(점프도 결국 주소를 더하는 것이기 때문). 따라서 위 네 가지 경우 모두에서 FUNC_ADD로 alu_op를 설정해줬다.

BRANCH의 경우 alu는 모두 동일하게 SUB 연산을 한다고 볼 수도 있지만, SUB 결과에 따라 bcond를 어떻게 설정할지는 달라지기 때문에 각각 다른 function code를 할당해줬다. 이때 alu_func에서 사용하지 않는 값들을 주석처리 하고 해당 값을 각각의 function code에 할당했다.

alu: 이 모듈은 Lab1에서 작성한 alu 모듈에 약간의 수정을 가해 완성했다. 우선 사용하지 않는 명령들을 주석처리 했다. 이후 branch 명령어들에 대해 언제 bcond를 1로 설정할지에 대한 코드를 추가했다. 완성된 alu 코드는 아래와 같다.

```
always @(*) begin
    alu_result = 32'b0;
    alu_bcond  = 1'b0;
    case(alu_op)
        `FUNC_ADD : alu_result = alu_in_1 + alu_in_2;
        `FUNC_SUB : alu_result = alu_in_1 - alu_in_2;
        `FUNC_BEQ : alu_bcond = (alu_in_1 == alu_in_2);
        `FUNC_BNE : alu_bcond = (alu_in_1 != alu_in_2);
        `FUNC_BLT : alu_bcond = ($signed(alu_in_1) < $signed(alu_in_2));
        `FUNC_BGE : alu_bcond = ($signed(alu_in_1) >= $signed(alu_in_2));
        // `FUNC_ID : alu_result = alu_in_1;
        // `FUNC_NOT : alu_result = alu_in_1;
        `FUNC_AND : alu_result = alu_in_1 & alu_in_2;
        `FUNC_OR : alu_result = alu_in_1 | alu_in_2;
        // `FUNC_NAND : alu_result = ~(alu_in_1 & alu_in_2);
        // `FUNC_NOR : alu_result = ~(alu_in_1 | alu_in_2);
        `FUNC_XOR : alu_result = alu_in_1 ^ alu_in_2;
        // `FUNC_XNOR : alu_result = ~(alu_in_1 ^ alu_in_2);
        `FUNC_LLS : alu_result = alu_in_1 << alu_in_2;
        `FUNC_LRS : alu_result = alu_in_1 >> alu_in_2;
        // `FUNC_ALS : alu_result = alu_in_1 <<< 1'b1;
        `FUNC_ARS : alu_result = alu_in_1 >>> 1'b1;
        // `FUNC_TCP : alu_result = alu_in_1 + 1'b1;
        `FUNC_ZERO : alu_result = 0;
        default : ;
    endcase

end
```

4. MEM

- **주요 모듈: `data_memory`**

Memory 접근은 명령이 LOAD이거나 STORE인 경우에만 일어나며, 각각 메모리에 있는 값을 레지스터에 저장하거나 레지스터에 있는 값을 메모리에 저장하는 과정이 필요하다. 이는 **`data_memory`** 모듈에서 처리한다.

`data_memory`: read가 실행되는 경우 **`mem`**의 **`dmem_addr`**번째 값을 출력한다. 만약 **`mem_read`**가 1로 설정되지 않았다면(= read할 상황이 아니라면) 0을 출력하도록 했다.

```
assign dout = mem_read ? mem[dmem_addr] : 0;
```

write이 실행되는 경우 **`din`**으로 입력받은 데이터를 **`mem`**의 **`dmem_addr`**번째 위치에 저장한다. 코드는 다음과 같다.

```
always @(posedge clk) begin
  if(mem_write) begin
    mem[dmem_addr] <= din;
  end
end
```

5. WB

- **주요 모듈: `register_file`**

WB 단계는 레지스터의 값을 변경해야 할 경우 일어나며, **`register_file`** 모듈에서 처리한다. 다시 말하지만 이 모듈은 read와 write을 처리하고 read과정은 ID단계에서 일어난다. WB에서 일어나는 write 과정은 input으로 들어온 **`write_enable`**이 1로 설정된 경우에만 일어난다. 또한 write을 하려고 하는 레지스터가 `x0`, 즉 항상 0으로 유지되어야 하는 레지스터가 아닌 경우에만 write을 할 수 있다. 따라서 코드는 다음과 같다.

```
// Synchronously write data to the register file
always @(posedge clk) begin
  if (write_enable && rd != 5'b00000) // Prevent writing to x0
    rf[rd] <= rd_din;

end
```

IV. 논의사항

- ALU 컨트롤의 "경제화" 가능성: 현재로써는 **`alu_control_unit`**에서 funct3를 ALU의 input으로 하나하나 따로 번역했는데, funct3의 값을 통째로 alu에 넘겨 주는 방법으로 디자인할 수도 있다고 생각했다. 이렇게 디자인하면 불필요한 rewiring을 줄일 수 있겠다는 생각이 든다.
- 현재는 cpu.v의 디자인을 깔끔하게 하기 위해 **`calculate_next_pc`**의 로직을 하나의 모듈로 묶었는데, 나중에 pipelined cpu로 확장을 하게 되면 이런 모듈을 다시 해체해야 할 수도 있다고 생각했다. 조금 귀찮긴 할 것 같다..

V. 결론

최종적으로 작성된 코드를 이용해 테스트를 해본 결과, 각 테스트에 대해서 아래와 같은 레지스터 값들을 얻었다.

```
### SIMULATING ###
TEST END
SIM TIME : 58
TOTAL CYCLE : 28
FINAL REGISTER OUTPUT
 0 00000000
 1 00000000
 2 00002ffc
 3 00000000
 4 00000000
 5 00000000
 6 00000000
 7 00000000
 8 00000000
 9 00000000
10 00000013
11 00000003
12 fffffd7
13 00000037
14 00000013
15 00000026
16 0000001e
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
```

<basic_mem.txt>

```
### SIMULATING ###
TEST END
SIM TIME : 446
TOTAL CYCLE : 222
FINAL REGISTER OUTPUT
 0 00000000
 1 00000000
 2 00002ffc
 3 00000000
 4 00000000
 5 00000000
 6 00000000
 7 00000000
 8 00000000
 9 00000000
10 00000000
11 00000000
12 00000000
13 00000000
14 0000000a
15 00000009
16 0000005a
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
```

<loop_mem.txt>

```
### SIMULATING ###
TEST END
SIM TIME : 80
TOTAL CYCLE : 39
FINAL REGISTER OUTPUT
 0 00000000
 1 00000000
 2 00002ffc
 3 00000000
 4 00000000
 5 00000000
 6 00000000
 7 00000000
 8 00000000
 9 00000000
10 0000000a
11 0000003f
12 fffffff1
13 0000002f
14 0000000e
15 00000021
16 0000000a
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
```

<non-controlflow_mem.txt>

Ripes simulator로 확인한 결과 세가지 테스트에서 모두 동일한 사이클 수와 레지스터 값을 얻을 수 있었다. 이를 통해 과제의 목표였던 single cylcle cpu를 정확히 구현하는 것에 성공했다고 결론 지을 수 있겠다.

처음 single cycle cpu에 대한 수업을 들을 때는 이해하기 어려웠다. 어떻게 각 모듈들이 동작하는 것인지 파악하기 힘들었고 각 신호가 무슨 역할을 하는지도 한눈에 들어오지 않았다. 하지만 이 과제를 끝내고 나니 해당 내용들에 대한 이해도가 매우 올라간 느낌이다. 컴퓨터 구조에 대한 내용을 단순히 이해하는 것에서 그치지 않고 실제 작동하는 cpu를 구현해 보는 경험을 통해 HDL 능력까지 키울 수 있었던 매우 흥미로운 과제였다고 생각한다.

다음 단계는 자연히 multi cycle cpu를 구현하는 것일 것이다. Single cycle cpu는 구조상 매우 느릴 수밖에 없다. 앞으로 남은 과제를 통해 사람들이 어떻게 cpu를 발전시켰는지 알아가보고 싶다.