

Single Cycle CPU

20210115 이세규 / 20210706 이지원

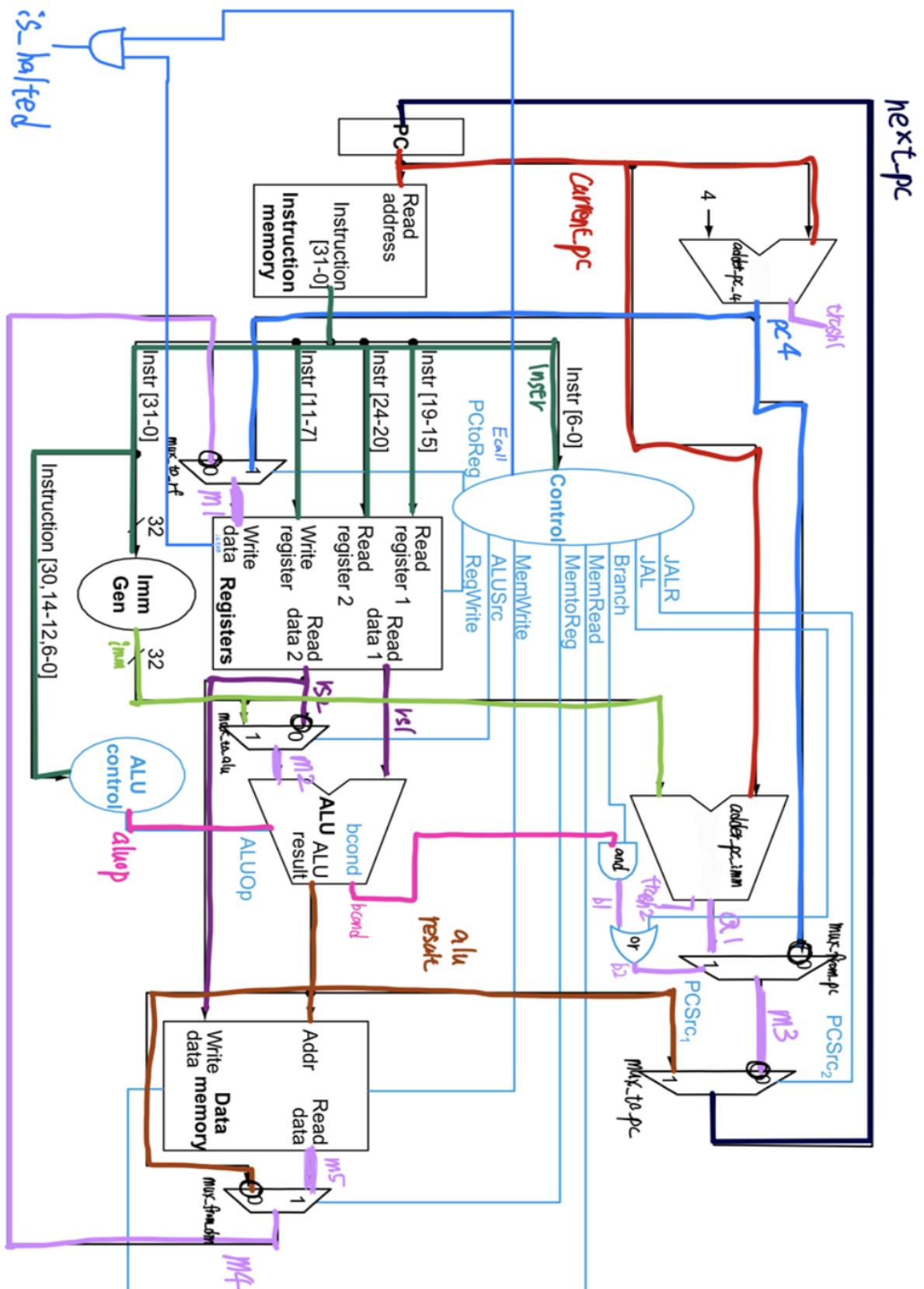
1. Introduction

이번 과제의 목표는 Vivado를 통해 single-cycle RISC-V CPU를 구현하며 RISC-V의 구조를 정확히 이해하고 single-cycle CPU가 어떻게 동작하는지 알아보는 것이다. 주어진 스켈레톤 코드를 활용하여 코딩하였고, 교재와 강의 PPT에서 나온 single-cycle CPU의 구조도를 기반으로 Modularization을 중심으로 구현하였다.

2. Design

single cycle cpu를 아래와 같이 설계하였고 이 장에서는 우리가 구현한 single cycle cpu의 설계와 구현의 전반에 대한 설명을 한 뒤 single cycle cpu가 가질 수 있는 state에 대해 예를 들어 설명하겠다.

cpu는 가장 먼저 pc을 참고하여 어떤 연산을 수행할지 결정한다. 32bit로 인코딩 된 연산에서 하위 7비트의 opcode를 통해 어떤 종류의 연산을 할 지 결정한다. 한 클락에 하나의 연산이 실행되도록 구현하였기 때문에 opcode를 해석하여 control unit에서 각 모듈로 작동 여부 또는 데이터 흐름 제어를 할 수 있는 신호를 보낸다. 이 신호를 바탕으로 레지스터 파일에 write를 하는지, memory read 또는 write를 하는지, mux 선택은 어떻게 되는지 등을 전반적인 논리 흐름을 결정한다. control 신호만 적절히 인가하면 주어진 instruction에 있는 데이터는 올바른 경로를 따라 의미에 맞게 처리된다. 이때 불필요한 연산이 수행될 수는 있으나, control signal이 이를 관리하여 불필요한 연산은 무시하고 해당 instruction을 수행하는데 필요한 부분만 취하여 혼선을 막는다. cpu관점에서의 control 신호에 대해 살펴보았다. ALU는 다양한 역할을 하기 때문에 ALU에게도 이와 유사한 역할을 하는 control unit이 필요하다. 이는 ALU control 모듈에서 담당한다. opcode와 funct3, funct7 부분을 활용하여 alu control 신호를 결정한다.



imm[20 10:1 11 19:12]				rd	1101111
imm[11:0]		rs1	000	rd	1100111
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011
imm[11:0]		rs1	010	rd	0000011
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011
imm[11:0]		rs1	000	rd	0010011
imm[11:0]		rs1	100	rd	0010011
imm[11:0]		rs1	110	rd	0010011
imm[11:0]		rs1	111	rd	0010011
0000000	shamt	rs1	001	rd	0010011
0000000	shamt	rs1	101	rd	0010011
0000000	rs2	rs1	000	rd	0110011
0100000	rs2	rs1	000	rd	0110011
0000000	rs2	rs1	001	rd	0110011
0000000	rs2	rs1	100	rd	0110011
0000000	rs2	rs1	101	rd	0110011
0000000	rs2	rs1	110	rd	0110011
0000000	rs2	rs1	111	rd	0110011
000000000000000		00000	000	00000	1110011
				ECALL - <u>프로그램 종료</u>	

control 은 cpu 전반의 데이터 흐름을 결정하기 때문에 위 표에 있는 opcode 에 따라 R type, I type, SB type, L type, S type, UJ type, JALR, ECALL 로 총 8 가지 출력을 하는 반면 ALU control 의 경우는 control 과 유사하게 위 표에 따라 22 개의 출력을 한다. ALUcontrol 의 경우, 해당 모듈 내부에서 alu 가 어떤 연산을 할지 결정하는 역할을 하기 때문에 alu 의 입력으로는 중복된 상태처럼 보일 수 있으나 ALUcontrol 모듈의 역할을 생각해보면 22 개의 출력을 하는 것으로 보는 것이 적절하다. 지금까지 cpu 전체의 설계와 구현에 대한 논의를 하였다. 추가로, is_halted 신호를 처리하기 위하여, register file 에서 10 번 레지스터 내부의 값이 10 인지를 판단하는 1 비트 신호를 받아와 이를 control 유닛의 ECALL 신호와 논리곱하였다. ECALL instruction 이외의 instruction 의 구체적인 동작은 아래 그림을 통해 설명하겠다. 그리고 여기서 다루지 않은 memory 나 register, ALU, ImmGen 등의 모듈들에 대한 세부적인 내용은 이후 implementation 장에서 다루겠다.

<p>JAL "JAL 일가"</p> <p>I (UJ-type (JAL))</p> <p>O (나머지)</p>	<p>JALR "JALR 일가"</p> <p>I (I-type (JALR))</p> <p>O (나머지)</p>	<p>Branch "ALU에 ALU bcond를 쓸 것인가?"</p> <p>I (B-type)</p> <p>O (나머지)</p>
<p>MemRead - "메모리를 읽는가"</p> <p>I (I-type (LW))</p> <p>O (R-type I-type (ADDI, JALR) S-type B-type UJ-type)</p> <p>R: 00000/00100 I(imm): 00000/01100 SB: 00100/00000 L-type: 00011/01110 S-type: 00000/11000 UJ: 10000/00110 JALR: 01000/01110</p>	<p>Mem to Reg - "메모리에서 읽은 값을 레지스터로 보내는가." (ALU연산값 대신)</p> <p>I (I-type (LW))</p> <p>O (I-type (imm) R-type I-type (JALR) S-type • B-type UJ-type)</p>	

MemWrite - "메모리에 write 하는가?"

1 { S-type

0 { R-type
I-type
B-type
UJ-type

ALUSrc - "ALU 아래 부분에 넣는 값이 Imm인가?"

1 { I-type
S-type

0 { R-type
B-type

X { UJ-type

Reg write - "Register에 write 하는가?"

1 {

- R-type (ADD)
- I-type (ADDI, LW, JALR)
- UJ-type (JAL)

0 {

- S-type (SW)
- B-type (BEQ)

PC to Reg - "PC+4를 저장하는가?"

1 {

- I-type [JALR]
- UJ-type [JAL]

c {

- R-type
- I-type (imm, LD)

X {

- S-type
- B-type

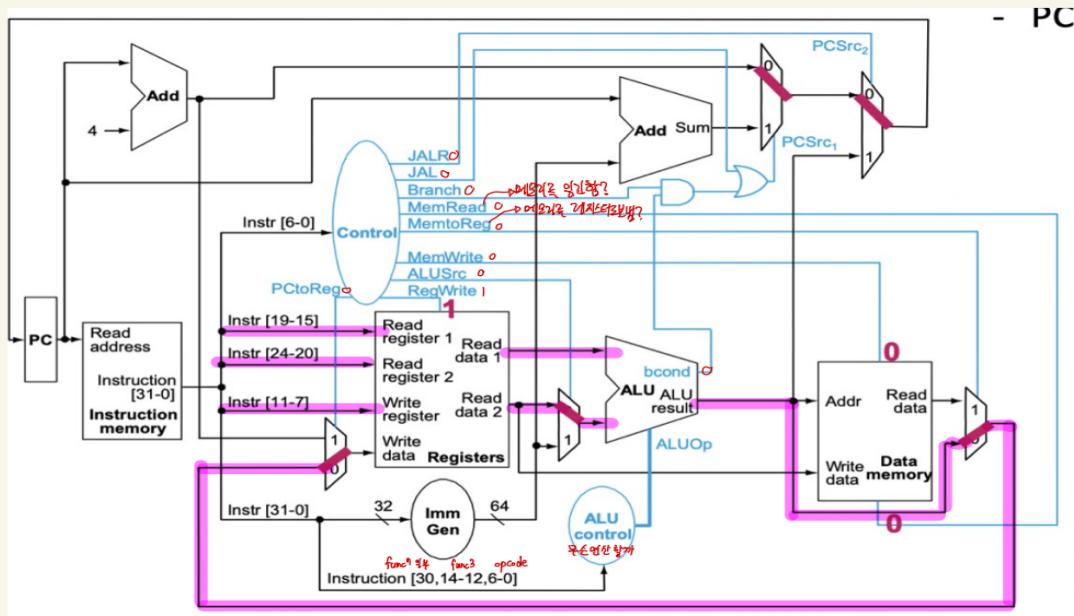
→ 이 그림은 연산에 따른 control unit의 출력을 나타낸다.

R-type:

$ADD\ rd, rs1, rs2$

Semantics (ADD)

- $GPR[rd] \leftarrow GPR[rs1] + GPR[rs2]$
- $PC \leftarrow PC + 4$

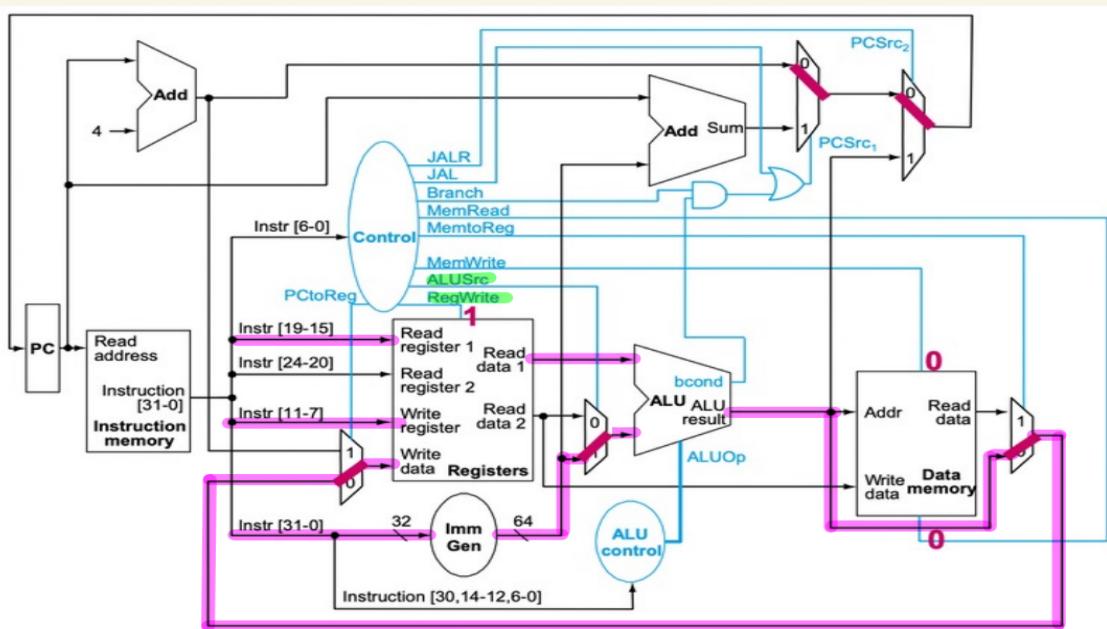


I type of ALU

$ADDI\ rd, rs1, imm_{12}$

Semantics

- $GPR[rd] \leftarrow GPR[rs1] + \text{sign-extend}(imm)$
- $PC \leftarrow PC + 4$

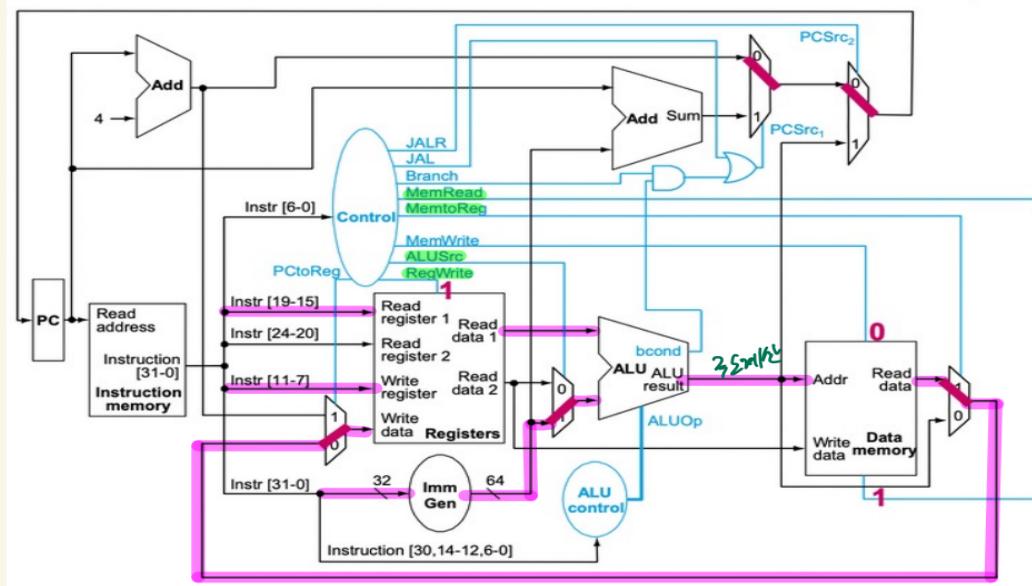


I type of LD

LW rd, offset₁₂(base)
imm rs1

Semantics:

- byte_addr₆₄ = sign-extend(offset₁₂) + GPR[base]
- GPR[rd] \leftarrow MEM₆₄[byte_addr₆₄]
- PC \leftarrow PC + 4



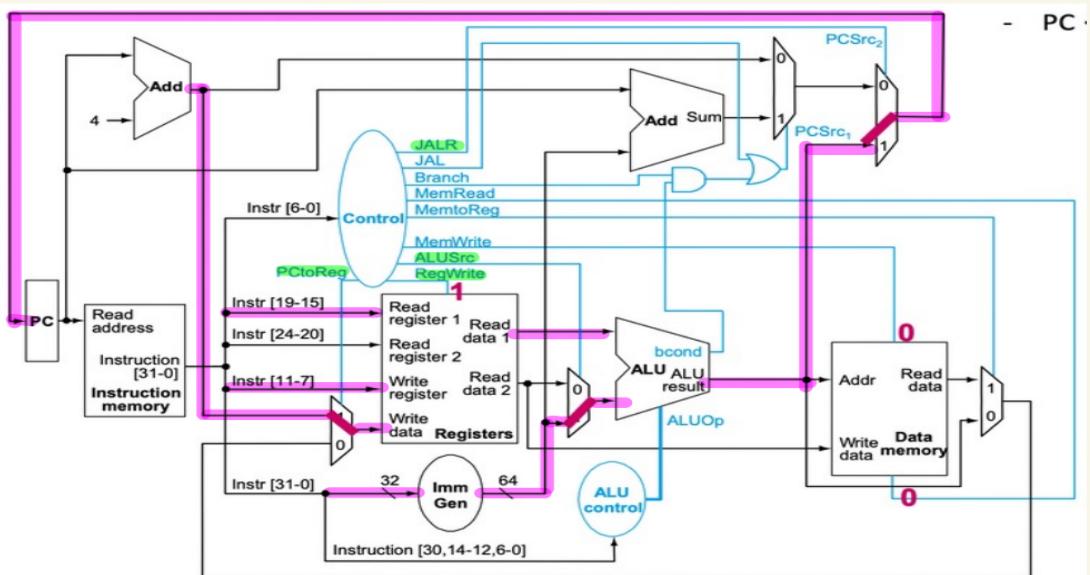
I-type of JALR

JALR rd, imm₁₂(rs1)

(Jump Indirect Instruction)

◆ Semantics:

- target = GPR[rs1] + sign-extend(imm₁₂)
- target &= 0xFFFF...FFFE
- GPR[rd] \leftarrow PC + 4
- PC \leftarrow target

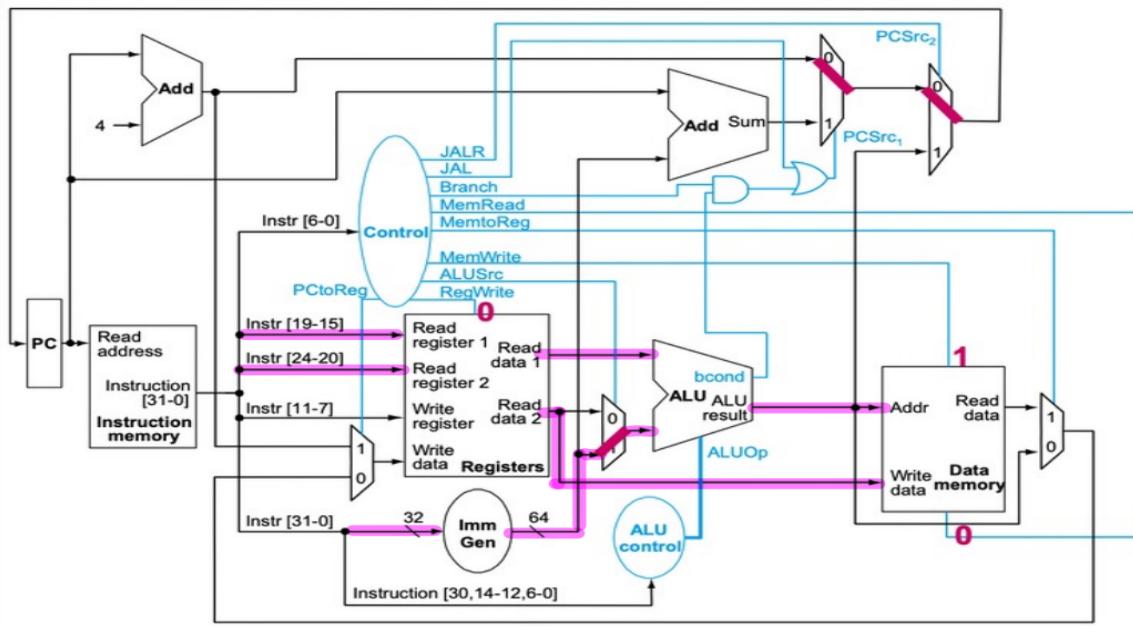


S-type

SW rs2, offset₁₂ (base)

Semantics:

- $\text{byte_address}_{64} = \text{sign-extend}(\text{offset}_{12}) + \text{GPR}[\text{base}]$
- $\text{MEM}_{64}[\text{byte_address}_{64}] \leftarrow \text{GPR}[rs2]$
- $\text{PC} \leftarrow \text{PC} + 4$

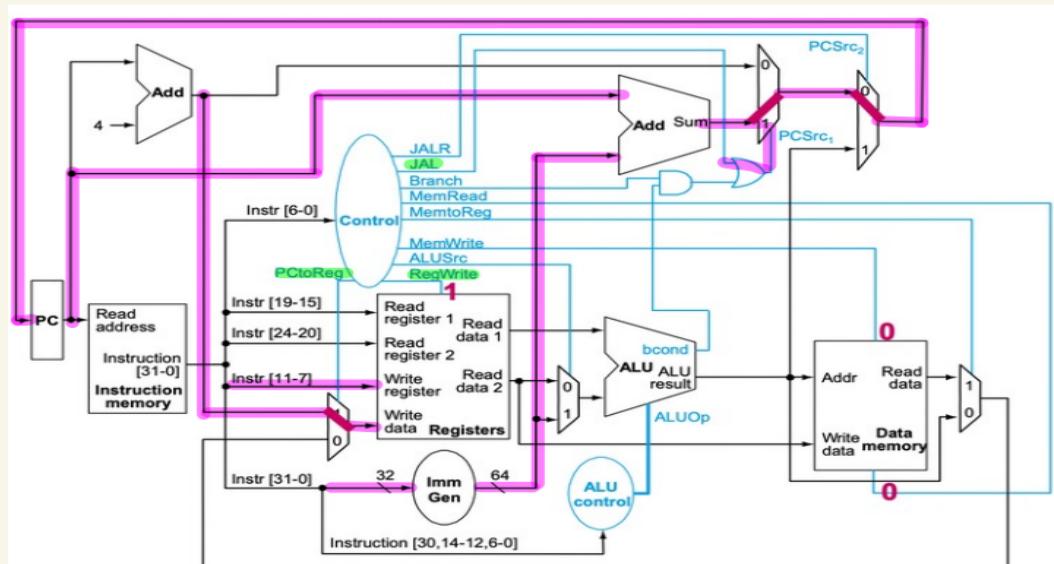


UJ-type

JAL rd, imm₂₁

Semantics:

- target = $\text{PC} + \text{sign-extend}(\text{imm}_{21})$
- $\text{GPR}[rd] \leftarrow \text{PC} + 4$
- $\text{PC} \leftarrow \text{target}$

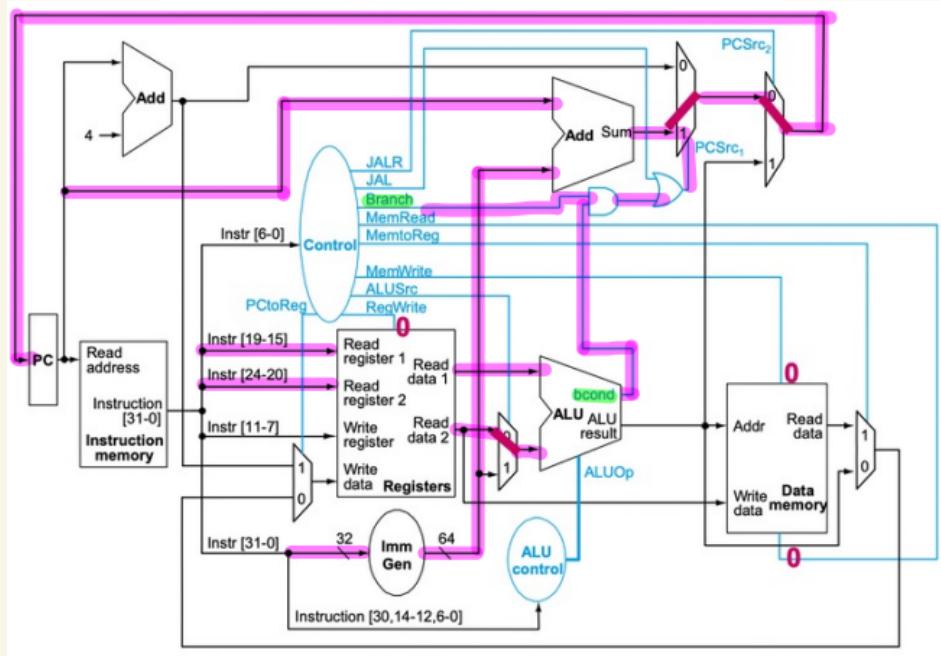


B-type

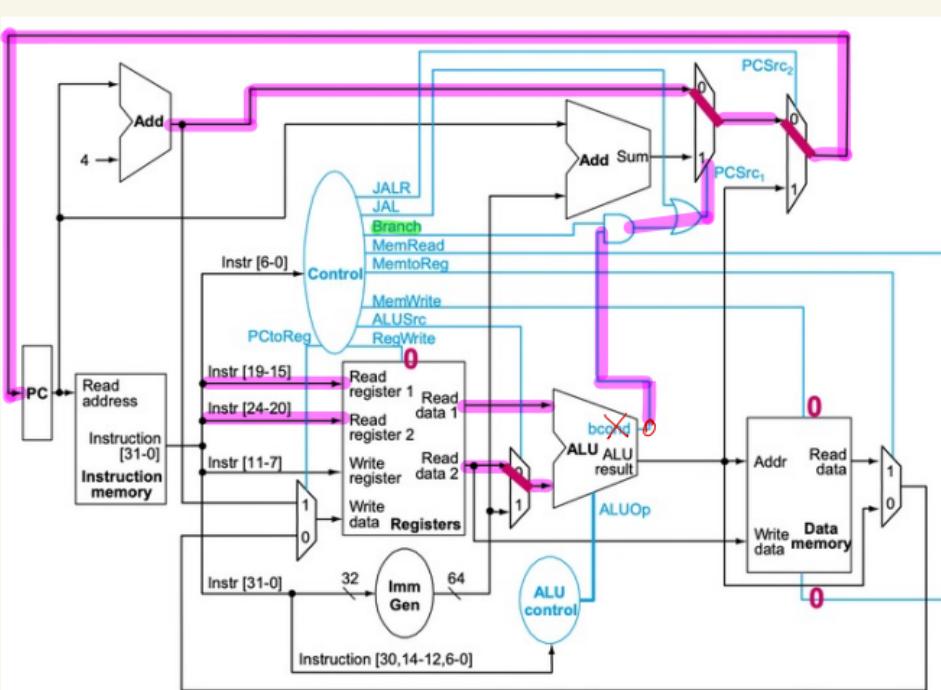
Semantics (BEQ):

- Target = PC + sign-extend(imm₁₃)
- If GPR[rs1] == GPR[rs2] then PC ← target
else PC ← PC + 4

조건지정



조건부지정



→ Control unit의 출력에 따라 주요 데이터의 흐름을 나타냈다. 연산 타입에 따라 datapath 가 달라지는 모습을 볼 수 있다.

이제 cpu 가 가질 수 있는 state에 대해 알아보자. 수업시간에 다룬 바와 같이 single cycle cpu 는 IF, ID, EX, MEM, WB 단계를 거쳐 하나의 연산을 처리한다. 모든 단계를 필요로 하는 Load instruction 을 예로 하여 state에 대해 알아보자. 우선 PC 모듈의 출력이 instruction memory 의 입력으로 들어가고 instruction memory 의 출력이 나온다. magic memory 를 가정하였으므로 이 과정은 combinational logic circuit 과 유사하게 동작한다. 이 단계가 instruction fetch 단계이다. fetch 해온 instruction 을 정해 둔 의미에 맞게 쪼개서 control, Registers, Imm Gen, ALU control 으로 보낸다. 그리고 registers file 에 있는 레지스터의 값을 읽고 그와 동시에 control signal 을 만들기도 하는 이 단계가 Instruction decode 단계이다. 그 다음 단계는 Execute 단계인데 이 때 decode 한 정보를 바탕으로 실질적인 연산이 수행한다. ALU 가 연산을 마치면 (store 와 load 의 경우) data memory 를 사용하게 되는데 메모리를 접근하는 Mem 단계를 거쳐 최종 결과를 다시 레지스터에 반영하는 Write Back 단계에 이르며 한 사이클이 마무리된다. instruction 의 종류에 따라 IF, ID, EX, MEM, WB 중 일부의 단계가 없을 수도 있는데 single cycle cpu 의 경우 특별한 조치를 취하지 않고 해당 단계를 무의미하게 훌려보낸다.

3. implementation

- ALU

ALU는 클락에 비동기적으로 작동한다. 우리가 구현한 ALU를 살펴보자. 구현의 가독성을 높이기 위해 ALU control signal을 아래와 같이 정의하였고, 이를 ALUop.v 파일로 추가적으로 만들었다.

```
`define BEQ 4'b0000
`define BNE 4'b0001
`define BLT 4'b0010
`define BGE 4'b0011

`define ADD 4'b0100
`define SUB 4'b0101
`define SLL 4'b0110
`define XOR 4'b0111
`define SRL 4'b1000
`define OR 4'b1001
`define AND 4'b1010

`define ALUNULL 4'b1111
```

LW와 SW의 경우 ALU에서 주소 계산을 하므로 LW와 SW 연산을 따로 두지 않고 ADD로 대신하였다. immediate value를 alu에서 사용하는 I-type의 경우에서도 연산을 R-type의 것으로 대체하였다. 이에 대한 자세한 구현은 ALUControlUnit에서 볼 수 있다. ALUNULL의 입력이 오면 alu는 모든 출력을 0으로 한다. 이는 ALU를 무의미하게 사용한 것이다.

ALU는 위에서 볼 수 있는 alu control signal에 따라 해당 연산을 수행하는데

```
module ALU(
    input [3:0] ALUop,
    input [31:0] A,
    input [31:0] B,
    output reg bcond,
    output reg [31:0] ALUresult
);
```

결과를 bcond와 ALUresult에 각각 저장하여 내보낸다. always begin을 활용하여 산술 연산과 논리 연산을 간단하게 구현할 수 있었다.

- ALUControlUnit

```
'include "opcodes.v"
`include "ALUOp.v"

module ALUControlUnit(input [31:0] part_of_inst, output reg [3:0] alu_op
);
always @(*) begin
    case(part_of_inst[6:0])
        `JALR: begin`JALR
            alu_op= ADD;
        end
        `BRANCH: begin
            if(part_of_inst[14:12]==`FUNCT3_BEQ)
                alu_op= BEQ;
            else if(part_of_inst[14:12]==`FUNCT3_BNE)
                alu_op= BNE;
            else if(part_of_inst[14:12]==`FUNCT3_BLT)
                alu_op= BLT;
            else if(part_of_inst[14:12]==`FUNCT3_BGE)
                alu_op= BGE;
            end
        `LOAD: begin
            alu_op= ADD;
        end
        `STORE: begin
            alu_op= ADD;
        end
        `ARITHMETIC_IMM: begin
            if(part_of_inst[14:12]==`FUNCT3_ADD)
                alu_op= ADD;
            else if(part_of_inst[14:12]==`FUNCT3_XOR)
                alu_op= XOR;
            else if(part_of_inst[14:12]==`FUNCT3_OR)
                alu_op= OR;
            else if(part_of_inst[14:12]==`FUNCT3_AND)
                alu_op= AND;
            else if(part_of_inst[14:12]==`FUNCT3_SLL)
                alu_op= SLL;
            else if(part_of_inst[14:12]==`FUNCT3_SRL)
                alu_op= SRL;
            end
        end
    `ARITHMETIC: begin
        if(part_of_inst[14:12]==`FUNCT3_ADD && part_of_inst[30]==0)
            alu_op= ADD;
        else if(part_of_inst[14:12]==`FUNCT3_SUB && part_of_inst[30]==1)
            alu_op= SUB;
        else if(part_of_inst[14:12]==`FUNCT3_XOR)
            alu_op= XOR;
        else if(part_of_inst[14:12]==`FUNCT3_XOR)
            alu_op= XOR;
        else if(part_of_inst[14:12]==`FUNCT3_OR)
            alu_op= OR;
        else if(part_of_inst[14:12]==`FUNCT3_AND)
            alu_op= AND;
        else if(part_of_inst[14:12]==`FUNCT3_SLL)
            alu_op= SLL;
        else if(part_of_inst[14:12]==`FUNCT3_SRL)
            alu_op= SRL;
        end
    default: begin
        alu_op = `ALUNULL;
    end
    endcase
endmodule
```

ALUControlUnit은 클락에 비동기적으로 작동한다. opcode(part_of_inst[6:0])에 따라 ALU의 사용 여부와 어떤 연산을 해야 하는지 정해지므로 case문으로 나누었다. BRANCH나 ARITHMETIC, ARITHMETIC_IMM의 경우에는 추가적으로 funct3 또는 funct3와 funct7을 같이 확인해야 하므로 이를 모두 분기를 나누어 적절한 연산을 취해주었다. ALU를 사용하지 않는 명령이 입력으로 들어오면 default로 ALUNULL을 출력하도록 하였다.

- ControlUnit

ControlUnit은 클락에 비동기적으로 작동한다. opcode를 입력으로 받아 cpu에서 제어가 필요한.

모듈들을 대상으로 제어 신호를 내보낸다. 제어 신호에는 이전 장 design 부분에서도 다뤘지만 다음과 같은 것들이 있다.

is_jal: instruction 이 JAL 인가

is_jalr: Instruction 이 JALR 인가

branch: ALU 의 출력 bcond 신호를 사용할 것인가(instruction 이 B type 인가)

mem_read: 메모리를 읽는가

mem_to_reg: ALU 연산값 대신 메모리에서 읽은 값을 레지스터로 보내는가

mem_write: 메모리에 write 하는가

alu_src: ALU 의 두 번째 입력에 넣는 값이 Imm 인가

reg_write: 레지스터에 write 하는가

pc_to_reg: PC+4 를 레지스터에 저장하는가

is_ecall: ECALL신호가 왔는가

```
// OPCODE
// R-type instruction opcodes
`define ARITHMETIC      7'b0110011
// I-type instruction opcodes
`define ARITHMETIC_IMM  7'b0010011
`define LOAD            7'b0000011
`define JALR            7'b1100111
// S-type instruction opcodes
`define STORE           7'b0100011
// B-type instruction opcodes
`define BRANCH          7'b1100011
// U-type instruction opcodes
//`define LUI             7'b0110111
//`define AUIPC          7'b0010111
// J-type instruction opcodes
`define JAL              7'b1101111

`define ECALL           7'b1110011
```

```
module ControlUnit (
    input [6:0] part_of_inst,
    output reg is_jal,
    output reg is_jalr,
    output reg branch,
    output reg mem_read,
    output reg mem_to_reg,
    output reg mem_write,
    output reg alu_src,
    output reg reg_write,
    output reg pc_to_reg,
    output reg is_ecall
);
```

정의된 opcode를 활용하여 역시 always begin에서 case문으로 각 출력에 위의 의미에 맞는 비트를 할당하여 구현하였다. 이 모듈에 대한 논리는 위 design 장에서 확인할 수 있다.

예시로 R-type의 경우를 살펴보면 다음과 같다.

```
//R type
`ARITHMETIC: {is_jal, is_jalr, branch, mem_read, mem_to_reg, mem_write, alu_src, reg_write, pc_to_reg, is_ecall} = {1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b1,1'b0,1'b0};
```

- ImmediateGenerator

```
module ImmediateGenerator(input [31:0] part_of_inst, output reg [31:0] imm_gen_out
);
  always @(*) begin
    case (part_of_inst[6:0])
      'JAL: begin
        if(part_of_inst[31]==0)
          | imm_gen_out={11'b0000000000,part_of_inst[31],part_of_inst[19:12],part_of_inst[20],part_of_inst[30:21],1'b0};
        else
          | imm_gen_out={11'b1111111111,part_of_inst[31],part_of_inst[19:12],part_of_inst[20],part_of_inst[30:21],1'b0};
      end
      'JALR: begin
        if(part_of_inst[31]==0)
          | imm_gen_out={20'b00000000000000000000,part_of_inst[31:20]};
        else
          | imm_gen_out={20'b11111111111111111111,part_of_inst[31:20]};
      end
      'BRANCH: begin
        if(part_of_inst[31]==0)
          | imm_gen_out={19'b00000000000000000000,part_of_inst[31],part_of_inst[7],part_of_inst[30:25],part_of_inst[11:8],1'b0};
        else
          | imm_gen_out={19'b11111111111111111111,part_of_inst[31],part_of_inst[7],part_of_inst[30:25],part_of_inst[11:8],1'b0};
      end
      'LOAD: begin
        if(part_of_inst[31]==0)
          | imm_gen_out={20'b00000000000000000000,part_of_inst[31:20]};
        else
          | imm_gen_out={20'b11111111111111111111,part_of_inst[31:20]};
      end
      'STORE: begin
        if(part_of_inst[31]==0)
          | imm_gen_out={20'b00000000000000000000,part_of_inst[31:25],part_of_inst[11:7]};
        else
          | imm_gen_out={20'b11111111111111111111,part_of_inst[31:25],part_of_inst[11:7]};
      end
      'ARITHMETIC_IMM: begin
        if(part_of_inst[31]==0)
          | imm_gen_out={20'b00000000000000000000,part_of_inst[31:20]};
        else
          | imm_gen_out={20'b11111111111111111111,part_of_inst[31:20]};
      end
      default: begin
        imm_gen_out=32'b0;
      end
    endcase
  end
endmodule
```

ImmediateGenerator 모듈은 클락에 비동기적으로 작동한다. 명령에 담겨있는 상수를 추출하여 32비트로 변환해주는 ImmediateGenerator모듈은 어떤 명령인지에 따라 상수가 담긴 위치가 달라 이를 경우를 나누어야 한다. 명령에 상수가 포함된 경우는 위와 같이 jal, jalr, branch, load, stroe, arithmetic_imm이 있고 과제 설명 pdf의 표를 참고해가며 상수의 위치를 파악했다. {}중괄호문을 통해 각 비트를 합쳐주었고 jal, branch의 경우 주소의 이동 범위 문제로 한 비트를 왼쪽으로 시프트 해줘야 하기 때문에 이를 고려하여 맨 뒤에 1'b0을 추가하고 앞에 붙을 sign비트의 개수를 조정했다. sign비트 연장은 간단하게 if문으로 상수의 부호를 나타내는 instruction의 32번째 비트를 보고 0과 1 중 무엇으로 연장할지를 정하였다. 모든 경우에서 결과적으로 32비트의 수가 나오도록 하였다. 상수를 사용하지 않는 명령에 대해서는 default로 의미없이 32비트의 0을 출력하도록 하였다.

- InstMemory

InstMemory는 우리가 실질적으로 구현한 부분은 read뿐이다. read는 클락에 비동기적으로 작동

한다. 초기화 할 때는 InstMemory에 write도 가능한데 이때 클락에 동기적으로 작동한다.

```
// TODO  
// Asynchronously read instruction from the memory  
// (use imem_addr to access memory)  
assign dout = mem[imem_addr];
```

imem_addr는 입력으로 들어온 addr를 index형식으로 바꾸기 위해 4바이트씩 끊어 오른쪽으로 2칸 시프트를 해준 것이다. dout을 mem[imem_addr]로 연결해주면 InstMemory에서의 구현은 완료된다.

- DataMemory

DataMemory는 read는 클락에 비동기적이고 write는 클락에 동기적으로 작동한다.

```
assign dout = mem_read ? mem[dmem_addr]:32'b0 ;  
  
always @(posedge clk) begin  
if(mem_write)  
begin  
| mem[dmem_addr] <= din;  
| end  
end
```

삼항연산자를 통해 read가 필요한 상황인지 mem_read로 확인하고 mem_read가 1이라면 dout = mem[dmem_addr]로, 0이라면 32비트의 0으로 assign한다. dmem_addr는 InstMemory의 imem_addr처럼 addr를 4바이트씩 끊기 위해 오른쪽으로 2칸 시프트해준 값이다.

클락에 따라 write를 해주는데, mem_write의 신호에 따라서 if문으로 mem[dmem_addr]에 쓰고자 하는 값인 din을 넣는다.

- MUX

```
module MUX(  
    input [31:0] A,  
    input [31:0] B,  
    input select,  
    output reg [31:0] OUT  
);  
  
    always @(*) begin  
        if (select == 0 )  
            OUT = A;  
        else  
            OUT = B;  
    end  
endmodule
```

MUX는 32비트 2X1 멀티플렉서로 클락에 비동기적으로 작동한다. always @(*)이므로 모든 input의 변화에 따라 바로 OUT값을 바꾸어 출력한다. 간단히 if문으로 구현했다.

- PC

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

PC는 클락에 따라 state를 바꿔주며 동기적으로 작동한다. 매 클락마다 먼저 reset을 해야 하는지 확인하고 스켈레톤 코드에 나온 것처럼 reset한다면 current_pc의 값을 0으로 초기화해준다. reset이 아닌 경우 간단히 입력으로 들어온 next_pc를 current_pc에 넣어주며 다음 state로 넘어가도록 해준다.

- RegisterFile

RegisterFile은 Memory처럼 read는 비동기적으로 write는 동기적으로 한다. Data Memory와 살짝 다른 점은 read에서는 따로 control의 신호 없이 항상 read한다는 점이다.

```
always @(*) begin
    if(rf[17]==10) is_ten =1;
    else is_ten =0;
end
```

ecall을 구현하기 위해 항상 x17의 값을 확인하여 값이 10인지를 판단해주었다. 이를 is_ten신호로 출력한다.

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

read는 단순히 신호의 조건 없이 rs1_dout=rf[rs1], rs2_dout=rf[rs2]로 구현했다. write의 경우 0번

째 레지스터는 항상 자주 쓰는 0이 들어가야 하므로 수정하면 안되기 때문에 0번째가 아닌 다른 레지스터를 고르는 경우인지를 체크하고, write_enable 신호가 1인지를 체크하여 두 조건을 모두 만족할 때만 동기적으로 write가 가능하도록 하였다.

- cpu

앞서 구현한 모듈들의 선연결을 design 파트의 구조도대로 수행하여 cpu를 완성했다. 몇몇의 adder는 이미 구현한 ALU를 활용하였고, 여러 MUX나 and, or 게이트들도 주어진 구조도대로 완성하였다. 구조도에 없던 부분은 단 한가지인데, 레지스터파일의 17레지스터의 값이 10인지를 판단하는 is_ten신호와 ecall 입력이 들어왔는지를 판단하는 Ecall신호를 and 게이트로 묶어 ecall의 여부를 판단하는 is_halted 신호를 출력으로 내보내도록 하였다. cpu는 클락을 받아 주어진 연산코드에 맞는 연산을 통해 PVS(PC, Memory, Registers)를 동기적으로 변화시킨다.

4. discussion

- 모듈화를 통해 구현을 조직적으로 할 수 있었다. single cycle cpu를 처음 구현하고 동작이 예상 결과와 다르게 나오자 control unit, register file 등 각각의 구현을 살펴보며 편하게 디버깅 할 수 있었다.

```
testmodule.v  × testmodule_tb.v  × Untitled 1  ×
C:/Users/Jiwon/single_cycle_debug_0318/single_cycle_debug_0318.srcts/sim_1/new/testmodule_tb.v

Q | S | ← | → | X | E | F | X | // | B | ? |
57
58     ControlUnit controlunit_module
59     (
60         .part_of_inst(part_of_inst),    // input
61         .is_jal(is_jal),           // output
62         .is_jalr(is_jalr),          // output
63         .branch(branch),           // output
64         .mem_read(mem_read),        // output
65         .mem_to_reg(mem_to_reg),    // output
66         .mem_write(mem_write),      // output
67         .alu_src(alu_src),          // output
68         .reg_write(reg_write),      // output
69         .pc_to_reg(pc_to_reg),      // output
70         .is_ecall(is_ecall)        // output (ecall inst)
71     );
72
73     initial begin
74         #1
75         part_of_inst = `ARITHMETIC;
76         #1
77         part_of_inst = `ARITHMETIC_IMM;
78         #1
79         part_of_inst = `LOAD;
80         #1
```



제한적인 조건 아래에서 연산이 필요로 하는 모듈들을 각각 검사해보며 해당 모듈의 잘못된 구현에 대한 의심을 해소할 수 있었다. 각 모듈에 대한 올바른 구현이 보장된 이후, 오류를 발생시키는 부분을 제한하여 생각할 수 있었고 결과적으로, 모듈사이의 연결에서 문제가 있음을 빠르고 정확하게 파악할 수 있었다. 원하는 동작이 정확히 구현되었는지 단계별로 제한적으로 확인할 수 있는 것이 중요하다는 것을 다시 한 번 깨달았다.

- 메모리를 읽을 때, magic memory를 가정하고 구현한다는 점과 alu나 control 신호를 만들어내는 과정에서 delay가 없다는 점을 고려하여 data memory와 Register File을 구현할 때 write를 클락에 동기적으로 구현한 것이 기억에 남는다.
- 우리 팀은 Lab1에서 assignment에 대해 정확히 학습하지 못했고, 실제로 보고서에서 꽤 큰 감점을 당했다. 이번 Lab2과제에서 다양한 모듈이 특정 부분마다 동기적인지 비동기적인지가 다른 경우가 많았는데, 이를 고려하며 =과 <=를 정확히 구분해가며 사용했다. Lab1에서 잘 학습하지 못했던 것을 이번에 더 공부하며 보충했던 것 같다.
- MUX 모듈의 경우 사실 output에 register가 필요하지 않음에도 불구하고 사용하였다. 원래 MUX는 input에 따라 그저 output을 내보내기만 하며, 값을 저장하고 있는 경우가 없어야 하기 때문에 register가 필요없다. 그러나 우리처럼 구현한 경우에도 always @(*)를 통해 모든 input의 변화에 맞추어 바로 비동기적으로 값을 바꿔주도록 하였기 때문에 사실 거시적으로는 MUX의 기능을 완벽히 수행하며 문제점이 없다고 판단하여 굳이 수정하지 않았다.

5. conclusion

이론적으로 책을 읽고 수업을 듣는 것만으로는 RISC-V를 정확히 이해하지 못하였다. 허나 single-cycle CPU 구현을 위해서는 이를 정확히 알고 opcodes와 funct3, funct7에 따라 분기를 나눠 무슨 수행을 해야 할지를 정해야 했기 때문에 명령들의 비트構成을 세심하게 이해할 수 있었던 것 같다. single-cycle CPU를 구현하기 위해서는 명령의 분류에 따라 data path가 결정되므로 어떤 모듈이 사용되어야 하는지를 체크해야 한다는 것을 깨달았다. ControlUnit의 구현이 가장 힘들었던만큼 이 부분이 중요히 다뤄야 한다고 느꼈다.