

# Assignment 3 Report: Multi-cycle CPU

TEAM 28

20180154 전제민 [jeminjeon@postech.ac.kr](mailto:jeminjeon@postech.ac.kr)

20180916 임경빈 [kbini@postech.ac.kr](mailto:kbini@postech.ac.kr)

2023/04/14

## 1. Introduction

이번 Lab3을 수행하며 Multi-cycle CPU를 RTL로 구현하였다. Multi -cycle CPU는 하나의 인스트럭션이 여러 개의 사이클에 걸쳐 수행되는 방식을 가지고 있는 CPU를 말한다. 이번 보고서에서는 Multi-cycle CPU의 논리 회로 설계를 다루고자 한다.

- **Design** **파트에서는** 각 모듈의 설계, 모듈 별 Synchronization 여부, Microcode controller design, Resource Reuse 설계에 대해 설명하였다.
- **Implementation** **파트에서는** 각 모듈의 구체적인 구현 (ControlUnit 코드 포함)을 설명하였다.
- **Discussion** **파트에서는** Single-cycle 그리고 Multi-cycle의 차이점을 성능과 비용 측면에서 설명하였다. 또한 basic\_mem.txt와 loop\_mem.txt를 각각 실행했을 때 사이클 수와 실행 결과를 표현했다.
- **Conclusion** **파트에서는** 이번 구현을 통해 새롭게 알거나 깨닫게 된 점을 간단히 정리하였다.

## 2. Design

### (1) Design of Each Modules

- **cpu** : 각 Multi-cycle CPU를 이루는 각 하위 모듈들을 연결시키는 모듈이다. 또한

Control Unit에서 나온 control flag에 따라 MUX의 output을 지정하는 로직도 포함하고 있다. top.v에서 발생시키는 clk cycle과 reset signal을 전달하며 cpu.v에서 output으로 나온 is\_halted에 따라 cpu를 종료시킨다.

- **PC** : 클럭 사이클에 따라 동기화되는 Synchronous 모듈이며, next\_pc를 입력받아 PC를 업데이트하는 역할을 하는 모듈이다.
- **Memory** : 클럭 사이클에 따라 동기화되는 Synchronous 모듈이며, MemRead 또는 MemWrite 컨트롤 플래그에 따라서 입력 받은 주소에 해당하는 데이터를 출력하거나 입력 받은 주소에 입력 데이터를 작성하는 역할을 하는 모듈이다. 이전 Single-cycle 을 구현할 당시에는 Instruction과 Data Memory를 분리하였으나 이번 구현에서는 Resource Reuse를 위해 둘을 통합하였다.
- **RegisterFile** : 클럭 사이클에 따라 동기화되는 Synchronous 모듈이며, 입력 받은 번호에 해당하는 레지스터 값을 출력하거나, 또는 RegWrite 컨트롤 플래그에 따라 입력 받은 데이터를 destination 레지스터에 작성하는 역할을 한다. 또한 Ecall 인스트럭션이 들어올 것을 대비해 x17 레지스터의 값을 확인하는 역할도 수행한다.
- **ALU** : Asynchronous 모듈이며 두 개의 input을 이용해 연산을 한다. 이때 ALUControl 모듈에서 가져온 연산 코드를 이용해 적절한 연산이 이루어지도록 한다. 만약 Branch 인스트럭션이 들어왔을 경우 두 레지스터의 값을 비교하여 적절하게 bcond 플래그를 set하는 역할도 수행한다.
- **ALUControl** : Asynchronous 모듈이며 opcode를 이용해 인스트럭션마다 적절한 ALU 연산이 이루어지도록 어떤 연산을 해야 하는지 결정하는 로직이 존재하는 모듈이다.
- **ImmediateGenerator** : Asynchronous 모듈이며 인스트럭션을 읽어 타입에 따라 적절하게 Immediate 값을 만들어내는 모듈이다.
- 이외에 Opcode.v 파일이 존재하나 모듈은 아니고 구현을 편리하게 하기 위한 매크로가 저장되어 있다. ControlUnit 모듈은 아래 (3) Microcode Controller 파트에서 자세하게 설명할 예정이다.

## (2) Synchronous and Asynchronous Modules

- **Synchronous Modules** : PC, RegisterFile, Memory, ControlUnit 모듈은 clk cycle에 따라 동기화되는 Synchronous Module이다. 이중 PC, RegisterFile, Memory 모듈은 Programmer visible state인 PC와 레지스터, 메모리의 값을 write하는 역할을 수행하

기 때문에 CPU가 실행 중일 때 데이터의 consistency와 integrity를 보장하기 위해 positive clock edge에서만 PVS가 write 되도록 하기 위해 클럭 사이클에 동기화가 필요하다. 한편 Control Unit에서는 Multi-cycle CPU를 구현하기 위해 사이클이 달라질 때마다 State가 달라지므로 이를 동기화하기 위해 클럭 사이클을 input으로 넣어 Synchronous하게 관리할 필요가 있다.

- **Asynchronous Modules** : 위에서 언급하지 않은 ALU, ALUControl, ImmediateGenerator 모듈은 클럭 사이클을 input으로 하지 않으며 다른 input들의 Combinational Logic에 의해 output이 결정되는 Asynchronous 모듈들이다.

### (3) Microcode controller state design

- **Stage Design** : 이번 Multi-cycle CPU 구현을 위해 한 사이클이 인스트럭션 수행에 있어 하나의 스테이지만을 수행하도록 설계하였으며 퍼포먼스 향상을 위해 각 인스트럭션마다 모든 스테이지를 다 사용하는 것이 아닌 필요한 스테이지만을 경유하도록 하였다.
  - IF stage는 현재 PC 값을 주소로 이용해 메모리로부터 인스트럭션을 Fetch하는 역할을 한다.
  - ID stage는 인스트럭션을 읽어 어떤 타입의 명령어인지 확인하고 ControlUnit을 통해 적절한 컨트롤 플래그를 세팅한다. 또한 필요한 레지스터 값을 읽는다. 추가로, 성능 향상을 위해 Non-control flow 명령어의 경우 다음 PC (PC+4)를 계산해 PC를 미리 업데이트 하도록 하였다.
  - EX stage는 인스트럭션 타입마다 필요한 연산을 수행하도록 한다. 예를 들어 R 또는 Arithmetic I 타입의 명령어일 경우 지정된 연산을 수행하며, Load 또는 Store 일 경우 적절한 메모리 주소 연산을 수행한다. Branch 명령어는 rs1과 rs2를 비교하며, JAL 또는 JALR의 경우 jump 할 주소를 계산한다.
  - MEM stage는 메모리에 접근하여 값을 쓰거나 읽어오는 역할을 한다. Load와 Store 명령어만이 경유한다.
  - WB stage는 레지스터에 연산 결과를 write한다. 또한 Branch의 경우 EX stage의 연산 결과에 따라 다음 PC를 PC+4로 할지, PC+imm으로 할지 최종 결정하게 된다.
  - 각 명령어 타입 별 거치게 되는 Stage는 아래와 같다.

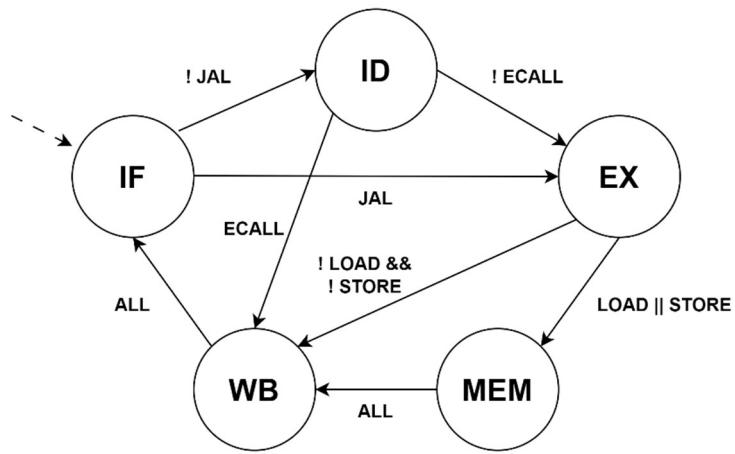


그림 1. 명령어 타입 별 경유하는 stage를 나타낸 그림

- Microcode Controller Design

- 한 사이클에 하나의 스테이지를 수행하도록 하기 위해 현재 어떤 스테이지를 수행하고 있는지를 표현할 Microprogram Counter를 두고, 이를 업데이트 해가면서 현재 stage에 따라 적절한 Datapath Control flag가 전달될 수 있도록 Microcode Controller를 설계하였다.
- Microcode Controller의 전체적 설계는 아래와 같다. 현재 state를 나타내는 MicroPC(아래 그림에서 Microprogram counter에 해당)가 있고, 이에 해당하는 Datapath Control이 세팅되어 output으로 출력된다. 클럭 사이클이 들어오면 현재 MicroPC에 1을 더한 뒤 Address Select Logic에 따라 다음 microPC를 정하게 된다. 이때 1을 더하는 Adder는 CPU의 ALU를 Reuse할수도 있었지만, 그렇게 되면 MicroPC를 업데이트 할 때마다 ALU를 사용해야 하기 때문에 불필요한 Cycle이 늘어나게 된다. 1을 더하는 작은 Adder를 하나 더하는 것으로 전체적인 수행에 필요한 cycle 수를 크게 줄일 수 있다고 판단하여 Adder는 따로 두게 되었다.

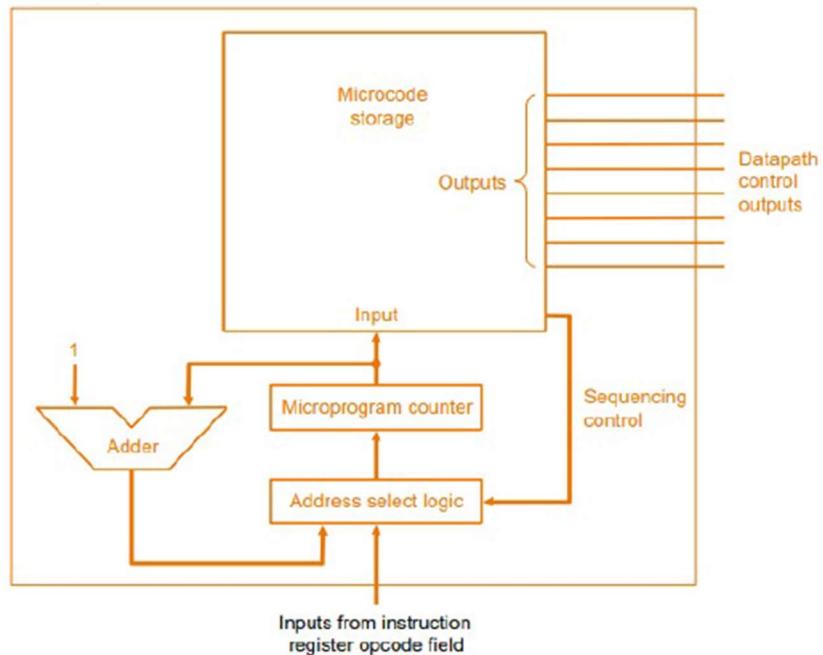


그림 2. Microcode Controller 설계도

- 구체적인 MicroPC는 아래와 같다. MicroPC는 6bit로 구성되며 앞 세자리는 인스트럭션 타입을, 뒤 세자리는 스테이지를 표현한다.

R-type	3'b000	IF	3'b000
I-type(Arithmetic)	3'b001	ID	3'b001
Load	3'b010	EX	3'b010
JALR	3'b011	MEM	3'b011
Store	3'b100	WB	3'b100
Branch	3'b101		
JAL	3'b110		
Ecall	3'b111		

- Address Select Logic : 현재 microPC에 1을 더하여, 뒤 3자리가 3'b101일 경우 3'b000으로 바꾸어 IF가 될 수 있도록 한다. 뒤 3자리가 3'b001일 경우 현재 opcode에 따라 앞 세자리를 업데이트 해준다. 만약 뒤 3자리가 3'b011 즉 MEM 스테이지인데 Load나 Store가 아닐 경우 3'b100, 즉 WB 단계로 넘어간다. ECALL의 경우 ID 다음 WB을, JAL의 경우 IF 다음 EX로 가도록 한다.

#### (4) Resource Reuse Design

- 이번 구현에서는 이전 Single-cycle 과 달리 하나의 인스트럭션이 여러 사이클에 걸쳐 수행되기 때문에 메모리를 instruction과 data로 분리하거나 주소를 계산하기 위한

별도의 Adder를 둘 필요가 없게 되었다. 따라서 비용 절감을 위해 Resource Reuse를 하도록 설계하였다.

- **Memory Resource Reuse** : 메모리 자원을 재활용하기 위해 stage마다 메모리로부터 Instruction을 가져올지 Data를 가져올지 결정할 컨트롤 플래그 (IorD)를 적절하게 세팅하여 메모리에 접근할 수 있도록 하였다. 대신 인струк션과 데이터를 저장할 레지스터 Instruction register, Memory data register를 각각 두어 활용하였다.
- **ALU Resource Reuse** : 연산 자원을 재활용하기 위해 각 인струк션 별로 연산이 필요할 경우 다른 스테이지에서 할 수 있도록 하였다. 먼저 Non-control flow 인струк션인 R/I(arithmetic), Store, Load 의 경우 ID 단계에서 미리 PC+4를 계산하여 PC를 업데이트 해 두었다. (이번 구현은 Pipeline이 적용되지 않았기 때문에 PC+4에 대한 예측이 틀릴 경우는 제외할 수 있다.) Branch 인струк션의 경우 연산이 3번 필요하다. 우선 ID 단계에서 PC+4를 계산해서 PC에 넣어둔다. 다음 EX 단계에서 bcond를 set할지 여부를 결정한다. 마지막으로 WB 단계에서 bcond가 set 되었을 경우 jump 해야 할 최종 PC를 다시 계산한다. JALR과 JAL은 각각 ID, EX에서 jump 할 주소를 계산하고 WB에서 레지스터에 PC+4를 작성한다.

### 3. Implementation

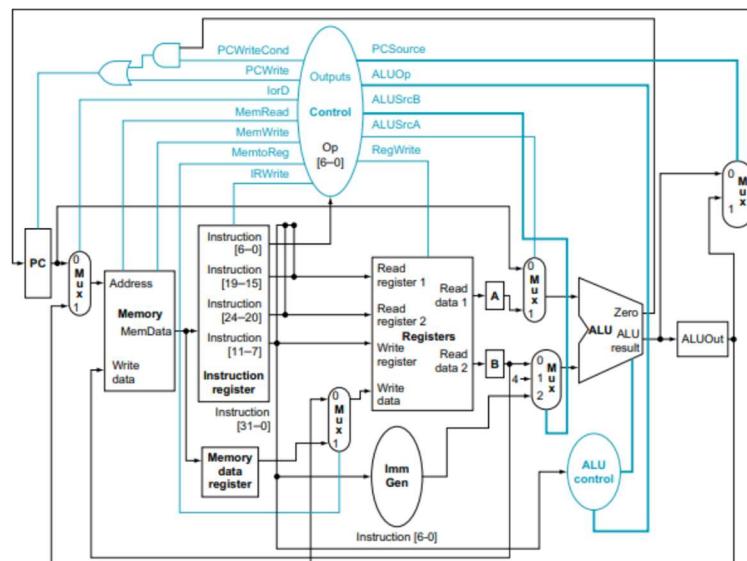


그림 3. multi cycle cpu

- **CPU**

우리가 구현한 multi cycle cpu의 Top module 이다. 하위 모듈들인 PC모듈, Memory 모듈, RegisterFile 모듈, ControlUnit 모듈, Immediate Generator 모듈, ALUControlUnit 모듈, ALU 모듈, DataMemory 등 하위모듈들의 연결을 여기서 구현하였다.

Reset signal, clock signal을 input으로 받는다. 그리고 reset signal, clock signal을 다른 모듈들로 전달함으로써 초기화한다.

- 1) **A,B register**

Register file에서 읽어온 값을 임시로 저장하는 레지스터이다. A에는 rs1에서 읽은 값을, B에는 rs2에서 읽어온 값을 저장한다.

- 2) **Halt control**

Ecall 이 1이고 ecall 이 10일때 is\_halted가 1이 되어 cpu가 멈추도록 구현하였다.

- 3) **IRWrite Control**

Instruction Register에 저장할지 말지를 결정하기 위한 부분이다. IRWrite가 1일 때만 메모리에서 가져온 인스트럭션 값을 가져와서 인스트럭션 레지스터(IR)에 저장하도록 구현하였다. IRWrite는 인스트럭션 페치 스테이지에서만 1이 되므로 즉, Instruction Fetch stage에서 인스트럭션을 fetch 할 때만 인스트럭션 레지스터에 인스트럭션을 저장할 수 있도록 구현하였다.

- 4) **ALUSrcA MUX**

Control Unit으로부터 나온 ALUSrcA로 ALU의 첫 번째 인풋을 결정하기 위한 MUX이다. 1bit MUX로 구현하였다. ALUSrcA가 0이면 ALU의 첫 번째 인풋으로 현재 PC값을 선택한다.. 0이 아닐 때는 레지스터에서 읽어온 값을 ALU의 첫 번째 인풋으로 선택하도록 구현하였다.

- 5) **ALUSrcB MUX**

Control Unit으로부터 나온 ALUSrcB로 ALU의 두 번째 인풋을 결정하기 위한 MUX이다. 2bit MUX로 구현하였다. ALUSrcB가 00이면 ALU의 두 번째 인풋으로 레지스터에서 읽어온 값을 선택한다. ALUSrcB가 01이면 ALU의 두 번째 인풋으로 4를 선택한다. ALUSrcB가 10이면 immediate value를 ALU의 두 번째 인풋으로 선택한다.

- 6) **MemtoReg MUX**

Control Unit으로부터 나온 MemtoReg로 Register에 저장할 값 즉 Register의 write data를

선택하기 위한 MUX이다. 만약 MemtoReg가 0이라면 register의 write data로 ALU 결과가 선택된다. 만약 MemtoReg가 1이라면 메모리 데이터 레지스터값(mem\_data\_reg)이 write data로 선택된다. Rtype의 instruction의 경우에는 MemtoReg값이 0이 될 것이므로 ALU 결과를 destination register에 저장한다. 로드 인스트럭션 같은 경우에는 MemtoReg 값이 1이 될 것이므로 메모리 데이터 레지스터값을 destination register에 저장한다.

## 7) IorD MUX

Control Unit으로부터 나온 IorD로 메모리 주소로 어떤 값을 쓸지 결정하기 위한 MUX이다. IorD가 0일 때는 현재 PC값이 메모리 주소로 쓰이게 되고 0이 아닐 때는 ALUOut 결과를 메모리 주소로 결정하도록 구현하였다.

- Control Unit

```
module ControlUnit(
    input clk,
    input reset,
    input [6:0] opcode,
    input [3:0] current_state,
    input in_bcond,
    output reg PCWrite,
    output reg PCWriteNotCond,
    output reg IorD,
    output reg MemRead,
    output reg MemWrite,
    output reg IRWrite,
    output reg MemtoReg,
    output reg PCSource,
    output reg [1:0] ALUSrcB,
    output reg ALUSrcA,
    output reg RegWrite,
    output reg Ecall,
    output reg [3:0] next_state,
    output reg [5:0] microPC
);

reg [5:0] next_microPC;
reg [5:0] temp_microPC;
reg bcond;
```

그림 4. control unit input, output, registers

Control Unit 모듈은 크게 두 가지 역할을 한다. 이 모듈과 연결된 다른 모듈들에 특정 제어 신호를 줘서 다른 모듈들의 행동을 제어하는 역할과 microPC를 제어하는 역할을 하도록 구현하였다.

```

    always @(posedge clk) begin
        if (reset == 1) begin
            microPC <= `MICRO_PC_INIT;
            PCWrite <= 0;
            IorD <= 0;
            MemRead <= 1;
            MemWrite <= 0;
            IRWrite <= 1;
            RegWrite <= 0;
            PCSource <= 0;
            Ecall <= 0;
        end
        else begin
            microPC <= next_microPC;
        end
    end

```

그림 5. microPC update

Reset signal이 1로 들어온 경우에는 모든 제어 신호들을 초기화하고 MemRead만 1로만 들어 준다. MemRead가 1이 되어야 pc값을 메모리에서 읽어와서 instruction fetch가 이뤄질 수 있기 때문이다. 그리고 reset이 0일 때는 계산된 next\_micro PC값을 micro PC에 write 하여 micro PC를 업데이트하는 과정이 clock synchronous하게 일어나도록 구현하였다.

```

always @(*) begin
    if(!reset) begin
        // Selecting Next MicroPC Logic
        temp_microPC = microPC + 1;

        // if current microPC indicates the WB state, next microPC is for IF
        if(temp_microPC[2:0] == 3'b101)
            temp_microPC[2:0] = 3'b000;

        // if next microPC is ID, microPC[5:3] is new instruction type
        if(temp_microPC[2:0] == `MICRO_PC_ID) begin
            if(opcode == `ARITHMETIC)
                temp_microPC = 6'b000001;
            else if(opcode == `ARITHMETIC_IMM)
                temp_microPC = 6'b001001;
            else if(opcode == `LOAD)
                temp_microPC = 6'b010001;
            else if(opcode == `JALR)
                temp_microPC = 6'b011001;
            else if(opcode == `STORE)
                temp_microPC = 6'b100001;
            else if(opcode == `BRANCH)
                temp_microPC = 6'b101001;
            else if(opcode == `JAL)
                temp_microPC = 6'b110001;
            else if(opcode == `ECALL)
                temp_microPC = 6'b111001;
        end

        // if microPC indicates MEM state but the instruction is not S or L, advance it to WB state.
        if(temp_microPC[2:0] == `MICRO_PC_MEM) begin
            if(temp_microPC[5:3] != `MICRO_PC_S && temp_microPC[5:3] != `MICRO_PC_L)
                temp_microPC[2:0] = `MICRO_PC_WB;
        end

        // if current instruction is JAL and microPC indicates ID state, advance it to EX state.
        if(temp_microPC[5:3] == `MICRO_PC_JAL && temp_microPC[2:0] == `MICRO_PC_ID)
            temp_microPC = 6'b110010;

        if(temp_microPC[5:3] == `MICRO_PC_ECALL && temp_microPC[2:0] == `MICRO_PC_EX)
            temp_microPC = 6'b111100;

        next_microPC = temp_microPC;
    end
end

```

그림 6. microPC control

컨트롤 유닛의 microPC 컨트롤을 구현하였다. 먼저 microPC는 6비트로 나타나는데 앞 3비트는 이 인스트럭션의 어떤 인스트럭션이지(ex R-type이면 000 I-type이면 001처럼 인스트럭션 타입별로 다른 state를 가짐) 나타내고 뒤 3비트는 이 인스트럭션의 현재 상태를 나타낸다. Instruction Fetch state인지, Instruction Decode state인지 등을 나타내는 것이다. IF state는 000, ID state는 001, EX state는 010, MEM state는 011, WB state는 100을 나타내도록 구현하였다. 기본적인 경우에는 IF->ID->EX->MEM->WB-> 다시 IF -> ... 과 같이 이뤄지도록 구현하였다. 그리고 해당 스테이트를 거칠 필요가 없는 인스트럭션들은 해당 스테이트를 건너뛰어서 진행하도록 구현하였다. 예를 들면 LOAD, STORE를 제외한 다른 인스트럭션들은 MEM state를 건너뛰고 바로 WB state로 가는 등 해당 인스트럭션이 필요로 하는 스테이트만 거치도록 구현하였다. 그리고 만약 하위 3비트 부분이 ID state라면 opcode를 통해 해당 인스트럭션을 파악하여 각 인스트럭션에 대응되는 micro\_pc\_opcode를 micro\_pc 상위 3비트에 넣어주도록 구현하였다.

Temp\_Micro\_pc값을 1씩 더해준다. (temp\_micro\_pc는 최종적으로 계산이 끝나면 현재 micro\_pc 다음의 micro\_pc이다) Next micro PC가 ID stage를 가리키면 opcode로 이 인스트럭션이 무슨 인스트럭션인지 구분하고 다음 micro PC를 결정하도록 구현하였다.

### 1) IF state

```
always @(microPC) begin
    if(!reset) begin
        //////////////////////////////// IF
        if(microPC[2:0] == `MICRO_PC_IF) begin
            PCWrite = 0;
            IorD = 0;
            MemRead = 1;
            MemWrite = 0;
            IRWrite = 1;
            RegWrite = 0;
            PCSource = 0;
            Ecall = 0;
        end
    end

```

그림 7. IF

이렇게 micro PC값이 변경되면(state가 바뀌면) micro PC의 하위 3비트를 통해 state를 파악하고 해당 state에 적합하게 control signal을 제어한다.

IF state에서는 인스트럭션 타입과 상관없이 메모리로부터 인스트럭션을 읽어와서 인스트럭션 레지스터에 write 해야 하므로 MemRead = 1, IRWrite = 1, 이 된다. 나머지 신호들은 PCWrite = 0, IorD = 0, MemWrite = 0, RegWrite = 0, Ecall = 0로 제어하였다.

### 2) ID state

```

/////////////////////////////// ID
else if(microPC[2:0] == `MICRO_PC_ID) begin
    PCWrite = 1;
    MemRead = 0;
    MemWrite = 0;
    IRWrite = 0;
    RegWrite = 0;
    PCSource = 0;|
    if(microPC[5:3] == `MICRO_PC_ECALL)
        Ecall = 1;

    if(microPC[5:3] == `MICRO_PC_JALR) begin
        ALUSrcA = 1;
        ALUSrcB = 2'b10;
    end
    else begin
        ALUSrcA = 0;
        ALUSrcB = 2'b01;
    end
end

```

그림 8. ID

ID state에서는 그리고 ID state는 register file에서 값을 읽어오는 state이므로 MemRead = 0, MemWrite = 0, IRWrite = 0, RegWrite = 0, PCSourece = 0이 된다. 그리고 ID state에서 미리 next PC를 계산하여 업데이트하도록 구현하였다. 이때 PC 계산은 ALU resource reuse로 계산하도록 구현하였다. 이를 위해 ALU를 ID state에서 사용해야 하므로 ALU 인풋을 결정하는 ALUSrcA, ALUSrcB를 제어한다. JALR 인스트럭션은 레지스터 A값을 ALU의 첫 번째 입력으로 받으므로 ALUSrcA= 1로 제어하고 나머지 인스트럭션들은 ALU의 첫 번째 입력으로 PC값이 들어가도록 ALUSrcA = 0으로 제어하였다. 그리고 JALR은 ALU의 두 번째 입력으로 immediate value 가 들어가도록 ALUSrcB =10으로 제어하였고 나머지 인스트럭션들은 ALU의 두 번째 입력으로 4가 들어갈 수 있게 ALUSrcB = 01로 제어하였다.

그리고 PCWrite=1로 하여 미리 계산된 nextPC로 current PC를 미리 업데이트하도록 구현하였다. Branch 인스트럭션 같은 경우에도 PC+4로 일단 업데이트하고 나중 state에서 branch가 확인된 뒤 다시 업데이트하도록 구현하였다. JAL은 ID state를 거치지 않으므로 EX state에서 PC를 업데이트하도록 구현하였다.

만약 이 인스트럭션이 ECALL state임이 확인되면 Ecall을 1로 내보내 프로그램을 종료하도록 구현하였다.

### 3) EX state

```

////////// EX
else if(microPC[2:0] == `MICRO_PC_EX) begin
    MemRead = 0;
    MemWrite = 0;
    IRWrite = 0;
    RegWrite = 0;
    PCSource = 0;

    if(microPC[5:3] == `MICRO_PC_R || microPC[5:3] == `MICRO_PC_B) begin
        PCWrite = 0;
        ALUSrcA = 1;
        ALUSrcB = 2'b00;
    end
    else if (microPC[5:3] == `MICRO_PC_I || microPC[5:3] == `MICRO_PC_L || microPC[5:3] == `MICRO_PC_S) begin
        PCWrite = 0;
        ALUSrcA = 1;
        ALUSrcB = 2'b10;
    end
    else if (microPC[5:3] == `MICRO_PC_JAL) begin
        PCWrite = 1;
        ALUSrcA = 0;
        ALUSrcB = 2'b10;
    end
    else if (microPC[5:3] == `MICRO_PC_JALR) begin
        PCWrite = 0;
    end
end

```

**그림 9. EX**

먼저 EX state에서는 Memory read, write 가 일어나지 않으므로 MemRead, Mem Write는 0으로 제어하였다. 그리고 Instruction Register에 Write 하지 않으므로 IRWrite=0, Register File에도 Write 하지 않으므로 RegWrite = 0으로 제어하였다.

JAL 인스트럭션은 next PC값을 EX state에서 계산하도록 구현하였다. Current PC + IMM 을 계산하기 위해 ALUSrcA = 0, ALUSrcB = 10으로 하여 ALU에 첫 번째 input 으로 PC값이, 두 번째 input으로 immediate value가 들어가도록 구현하였다. 그리고 PCWrite =1로 하여 바로 current PC 가 미리 업데이트되어있을 수 있게 구현하였다. 나머지 인스트럭션들은 모두 PCWrite=0으로 하여 PC를 업데이트하지 못하게 구현하였다.

R-type과 Branch instruction 들은 ALU의 입력으로 레지스터 파일에서 가져온 값이 들어가야 하므로 ALUSrcA = 1, ALUSrcB = 00으로 제어하였다.

I-type과 LOAD, Store instruction 들은 ALU의 첫 번째 입력으로 레지스터 파일에서 가져온 값, 두 번째 입력으로 immediate value가 들어가야 하므로 ALUSrcA = 1, ALUSrcB = 10으로 제어하였다.

이런 식으로 현재 micro pc값, opcode 등을 통해 다음 스테이트를 계산한 뒤 output register인 micro PC에 값을 쓰는 과정은 clock synchronous 하게 이뤄지도록 구현하였다.

#### 4) MEM state

```

////////// MEM
else if(microPC[2:0] == `MICRO_PC_MEM) begin
    PCWrite = 0;
    IorD = 1;
    IRWrite = 0;
    RegWrite = 0;
    PCSource = 0;

    if (microPC[5:3] == `MICRO_PC_L) begin
        MemRead = 1;
        MemWrite = 0;
    end
    else if (microPC[5:3] == `MICRO_PC_S) begin
        MemRead = 0;
        MemWrite = 1;
    end
end

```

그림 10. MEM

MEM State는 LOAD 인스트럭션은 MemRead = 1, MemWrite = 0으로 하여 메모리를 읽어올 수 있게 하였고 STORE 인스트럭션은 MemRead = 0, MemWrite = 1로 하여 메모리에 Write 할 수 있게 구현하였다. 그리고 LOAD, STORE 모두 memory에서 이용할 주소를 선택할 때 ALU 결과가 사용되도록 IorD = 1로 제어하였다.

## 5) WB state

```

////////// WB
else if(microPC[2:0] == `MICRO_PC_WB) begin
    MemRead = 0;
    MemWrite = 0;
    IRWrite = 0;

    bcond = in_bcond;

    if(microPC[5:3] == `MICRO_PC_R || microPC[5:3] == `MICRO_PC_I) begin
        PCWrite = 0;
        MentoReg = 0;
        PCSource = 0;
        RegWrite = 1;
    end
    else if(microPC[5:3] == `MICRO_PC_L ) begin
        PCWrite = 0;
        MentoReg = 1;
        PCSource = 0;
        RegWrite = 1;
    end
    else if(microPC[5:3] == `MICRO_PC_S ) begin
        PCWrite = 0;
        PCSource = 0;
        RegWrite = 0;
    end
    else if(microPC[5:3] == `MICRO_PC_JAL || microPC[5:3] == `MICRO_PC_JALR) begin
        PCWrite = 0;
        MentoReg = 0;
        PCSource = 0;
        RegWrite = 1;
        ALUSrcA = 0;
        ALUSrcB = 2'b01;
    end
end

```

그림 11. WB\_1

```

        else if(microPC[5:3] == `MICRO_PC_B) begin
            if (bcond == 1) begin
                PCWrite = 1;
                PCSource = 0;
                RegWrite = 0;
                ALUSrcA = 0;
                ALUSrcB = 2'b10;
            end
            else begin
                PCWrite = 0;
                PCSource = 0;
                RegWrite = 0;
            end
        end
    end

```

그림 12. WB\_2

WB state에서는 destination register에 값을 저장해야 하므로 각 인스트럭션 별로 destination register에 들어갈 데이터를 정하는 신호들을 제어한다. R-type, I-type 인스트럭션은 RegWrite = 1을 통해 register에 write 할 수 있도록 하고 ALU 결과가 destination register에 write 될 수 있도록 구현하였다. Load 인스트럭션은 RegWrite = 1, MemtoReg = 1로 제어함으로써 destination 레지스터에, 메모리에서 읽어온 값을 write 할 수 있도록 구현하였다. JAL, JALR은 ALUSrcA = 0, ALUSrcB = 01, RegWrite = 1로 제어함으로써 현재 PC + 4를 ALU로 계산한 뒤 destination register에 write 하도록 구현하였다. 그리고 Branch condition을 통해 Branch instruction의 next PC를 계산하고 PC를 업데이트하도록 구현하였다. Bcond가 1인 경우에는 ALUSrcA = 0, ALUSrcB = 10으로 제어하여 PC + immediate value가 ALU에서 계산되도록 구현하였고 PCwrite = 1로 제어하여 PC가 PC + immediate value로 업데이트될 수 있게 구현하였다. R-type, I-type instruction은 RegWrite = 1로 하여 ALU result가 destination register에 write 될 수 있도록 구현하였다.

- **Memory**

Memory 모듈은 Instruction memory, data memory로 나누지 않고 하나의 memory를 resource reuse하도록 구현하였다. IF stage에서는 pc를 주소로 받아서 Instruction memory처럼 쓰이고 MEM stage에서는 ALU 결과 혹은 레지스터에 저장된 주소를 메모리 주소로 받아서 이뤄지도록 구현하였다. Reset, CLK, 메모리 주소인 addr, 저장할 데이터, 메모리에서 데이터를 읽을지 여부를 나타내는 MemRead, 메모리에서 데이터를 쓸지 여부를 나타내는 MemWrite를 입력으로 받도록 구현하였다. MemRead, MemWrite는 Control Unit 모듈에서 온 값이다. MemRead가 1일 때만 Memory값을 읽을 수 있고 MemWrite가 1일 때만 Memory에 저장할 데이터 값을 저장할 수 있도록 구현하였다.

메모리를 초기화하는 과정은 reset이 1인 경우에 메모리값을 전부 0으로 초기화하고 주어진 경로에서 메모리를 가져오도록 구현하였다. 이 과정은 clock synchronous하게 일어나도록 구

현하였다. .

메모리 읽기 과정은 CLK asynchronous 하게 이뤄지도록 구현하였다. 이 모듈의 출력인 dout은 MemRead가 1이면 해당 addr에 저장된 데이터 값을 읽어서 dout으로 내보내고 MemRead가 0이면 0을 내보내도록 구현하였다.

메모리 쓰기 과정은 CLK synchronous 하게 이뤄지도록 구현하였다. MemWrite가 1일 때 저장할 데이터를 메모리에 write 한다.

- **RegisterFile module**

RegisterFile 모듈은 레지스터 파일을 읽고 쓰는 모듈이다. Input으로 clk, rs1, rs2, rd, rd\_din(input data for rd), write\_enable을 받도록 구현하였다. 그리고 output은 rs1의 데이터를 읽은 값인 rs1\_dout, rs2의 데이터를 읽은 값인 rs2\_dout, 17번 레지스터값을 읽은 값인 call을 가지도록 구현하였다. 레지스터파일에서 레지스터값을 읽는 과정은 clock asynchronous 하게 이뤄지도록 구현하였다. 그리고 레지스터 파일에 write 하는 과정은 clock0| rising edge일 때 write 하도록(clock synchronous 하게) 구현하였다.

- **ALU Control Unit**

어떤 연산을 해야 할지 결정하는 모듈이다. instruction, microPC를 입력받아 어떤 연산을 해야 하는지 구분한 뒤 alu\_op\_out으로 내보낸다. 먼저 opcode를 통해 어떤 타입의 인스트럭션인지 구분하고 그 뒤 funct3를 통해 어떤 instruction인지 세부적으로 구분한다. 예를 들어 opcode를 통해 B-type임을 판단 한 뒤에 funct3를 이용하여 BEQ, BNE, BLT 등을 구분하는 방식으로 구현하였다. ADD, SUB의 구분은 funct7을 통해 구분하도록 구현하였다. 여기서 출력한 alu\_op\_out은 ALU 모듈로 input으로 들어가게 구현하였다. 모든 과정은 CLK asynchronous 하게 이루어지게 구현하였다. 또한 ALU가 EX state 가 아닌 다른 state에서도 재사용 되므로 그 state에 적합한 alu\_op\_out을 가지도록 구현하였다. 예를 들면 Branch type 인스트럭션이 branch condition이 taken 됐을 때 WB stage에서는 ALU\_ADD를 통해 PC+immediate value를 계산해야 하므로 instruction의 opcode를 통해 얻은 alu\_op\_out이 아닌 state를 통해 얻은 ALU\_ADD를 alu\_op\_out으로 내보내서 ADD 연산을 이뤄질 수 있도록 구현하였다.

- **ALU**

ALUControlUnit에서 어떤 연산을 해야 하는지 구분해 alu\_op\_in을 입력으로 받고 이를 통

해 무슨 연산을 해야 하는지 판단 한 후 입력받은 두 입력값 input 1, input 2로 연산을 수행한다. 만약 Arithmetic 연산을 수행해야 한다면 input 1, input 2로 연산을 수행하고 연산 결과를 output reg [31:0] result\_out으로 내보내고 b\_con(branch condition이 True인지 False인지)을 0으로 내보낸다. 만약 연산이 branch condition을 확인하기 위한 연산(BEQ, BNE 등 을을 위한 연산) 이라면 branch conditon이 True일 때 b\_cond를 1로 내보내고 아니면 b\_cond를 0으로 내보내도록 구현하였다. 모든 과정은 CLK asynchronous 하게 이루어지게 구현하였다.

- **Immediate Generator**

인스트럭션을 통해 immediate value를 계산하는 모듈이다. 인스트럭션을 input으로 받고 이 인스트럭션을 통해 계산된 immediate value인 imm\_gen\_out을 output으로 가진다. 먼저 Opcode 부분에 해당하는 part\_of\_inst[6:0]로 타입을 구분한다. 그리고 각 타입에 맞는 immediate value 위치를 인스트럭션에서 찾아 immediate value를 계산한 뒤 sign extension 하여 imm\_gen\_out으로 내보낸다. 모든 과정은 CLK asynchronous 하게 이루어지게 구현하였다.

- **PC**

Program Counter 모듈이다. next\_pc, reset, CLK를 input으로 가진다. CLK가 rising edge일 때 reset이 1이면 current\_pc를 0으로 초기화하고 reset이 0이 아니라면 input으로 받은 계산된 next PC value를 current\_pc를 다음 pc값으로 업데이트해 준다. 이 과정은 clock synchronous 하게 이뤄지도록 구현하였다.

## 4. Discussion

- **Difference between Single-cycle CPU and Multi-cycle CPU:** 먼저 Single-cycle CPU는 하나의 클럭 사이클에 하나의 명령어를 실행한다. 모든 명령어 타입에 대해 동일한 시간이 걸리게 된다. 또한 가장 긴 시간이 걸리는 명령어에 맞추어 클럭 사이클을 정해야 하기 때문에 일부 명령어를 실행할 경우 실행이 끝났음에도 파이프라인이 idle 상태로 대기해야 하는 비효율이 발생한다. 반면 Multi-cycle CPU는 하나의 명령어를 여러 사이클에 걸쳐 실행한다. 대신 명령어 타입마다 필요한 수만큼의 사이클만큼 실행하기 때문에 명령어의 종류에 따라 실행시간이 유동적으로 조절되어 Single-cycle에 비해 실행 자원이 대기하는 시간이 줄어 좀 더 효율적인 방법이다. 다만 회로가 복잡하고 설계가 어려워질 수 있다. 또한 파이프라인이 적용되지 않았을 경우 한 사이클에 단 하나의 명령어만을 수행한다는 한계가 존재한다.

- Resource Reuse : 2. Desgin 파트에서 언급했던 것 처럼 이번 구현에서 메모리 자원과 ALU 자원을 재활용하여 비용을 절감하는 방향으로 설계할 수 있었다. 이러한 자원 재활용 또한 Single-cycle에 비해 Multi-cycle이 가지는 이점이라고 볼 수 있다.
- Number of cycles to run basic\_mem.txt and loop\_mem.txt

basic_mem.txt 실행 결과	loop_mem.txt 실행 결과
TOTAL CYCLE	121
0 00000000	0 00000000
1 00000000	1 00000000
2 00002ffc	2 00002ffc
3 00000000	3 00000000
4 00000000	4 00000000
5 00000000	5 00000000
6 00000000	6 00000000
7 00000000	7 00000000
8 00000000	8 00000000
9 00000000	9 00000000
10 00000013	10 00000000
11 00000003	11 00000000
12 ffffffd7	12 00000000
13 00000037	13 00000000
14 00000013	14 0000000a
15 00000026	15 00000009
16 0000001e	16 0000005a
17 0000000a	17 0000000a
18 00000000	18 00000000
19 00000000	19 00000000
20 00000000	20 00000000
21 00000000	21 00000000
22 00000000	22 00000000
23 00000000	23 00000000
24 00000000	24 00000000
25 00000000	25 00000000
26 00000000	26 00000000
27 00000000	27 00000000
28 00000000	28 00000000
29 00000000	29 00000000
30 00000000	30 00000000
31 00000000	31 00000000

## 5. Conclusion

이번 Lab에서는 Verilog를 이용해 Multi-Cycle CPU을 구현하였다. 확실히 이전 Single-cycle CPU에 비해 동기화나 데이터 유지와 관련하여 고려할 점들이 많았다. 한 인스트럭션의 여러 사이클에 걸쳐 수행되다 보니 유지해야 할 데이터와 새롭게 업데이트 해야 할 데이터를 관리하는 것이 까다로웠다. 다만 이전 과제의 경험들 덕분에 과제를 완료하는데 걸린 시간이 이전에 비해 짧아졌다. 앞으로 Pipelined CPU를 구현하기 위해 관련 내용을 충분히 숙지하는 것이 도움이 될 것 같다.