# 컴퓨터구조및모바일프로세서 과제2 보고서


과목: 컴퓨터구조및모바일프로세서 2분반
교수명: 유시환 교수님
학과: 모바일시스템공학과
학번: 32224020
이름: 전은지
**Rest Freedays: 1

# Index

## I. Introduction

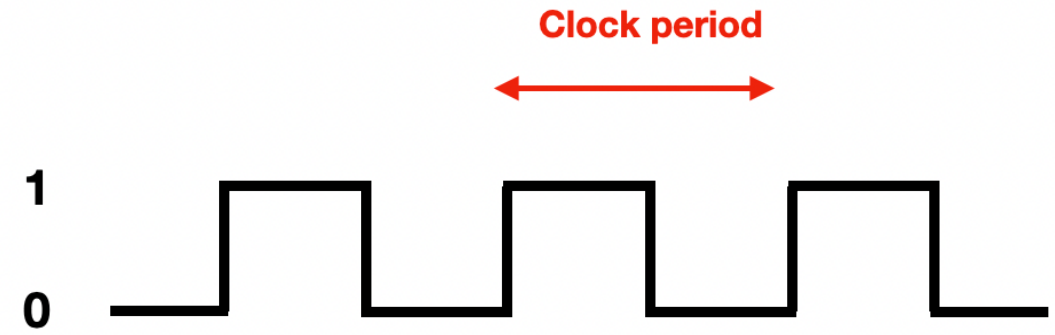
이번 프로젝트에서는 Microarchitecture의 구조를 이해하고 Single Cycle의 작동 방식을 직접 코드로 구현함으로써 CPU에서 명령어들이 처리되는 메커니즘에 대하여 학습한다. 구체적으로, 실행 가능한 바이너리 파일로부터 명령어를 읽어 처리하는 MIPS 프로세서 시뮬레이터를 개발하여, 단일 사이클 내에서 명령어를 실행하고 결과를 도출한다. 이 과정을 통해, 명령어가 Data path를 따라 이동하면서 레지스터에 데이터가 저장되고 값들이 처리되는 과정을 이해할 수 있다.

## II. Body

### 1. Background information

#### 1) Single Cycle Processor

CPU는 내부 회로를 동작 시키기 위해 일정한 주기로 규칙적인 전기 신호를 발생시키는데 이를 Clock이라고 한다. 즉 1 clock cycle은 한 번의 전기 신호를 말하고, 한 클럭에 걸리는 시간을 클럭 주기(Clock Period 또는 Clock cycle time)이라고 한다. 싱글 사이클 프로세서는 그 사이클 동안 하나의 명령어를 처리하는 프로세서를 의미한다.




#### 2) Instruction Cycle

명령어 주기는 CPU가 부팅 후 컴퓨터가 종료되어 명령어를 처리하기 위해 따라는 주기로 fetch 단계, decode단계, execute단계로 구성된다.

(1) Fetch

Fetch 단계는 CPU가 메모리에서 명령어를 읽어오는 과정으로 프로그램 카운터(PC)가 가리키는 메모리 주소에서 명령어를 가져온다. PC는 현재 CPU가 실행할 명령어의 위치를 가리키며, 명령어를 가져온 후에는 다음 명령어의 주소로 자동으로 증가한다. 명령어를 CPU로 전달하면 다음 단계인 decode 단계로 넘어간다.

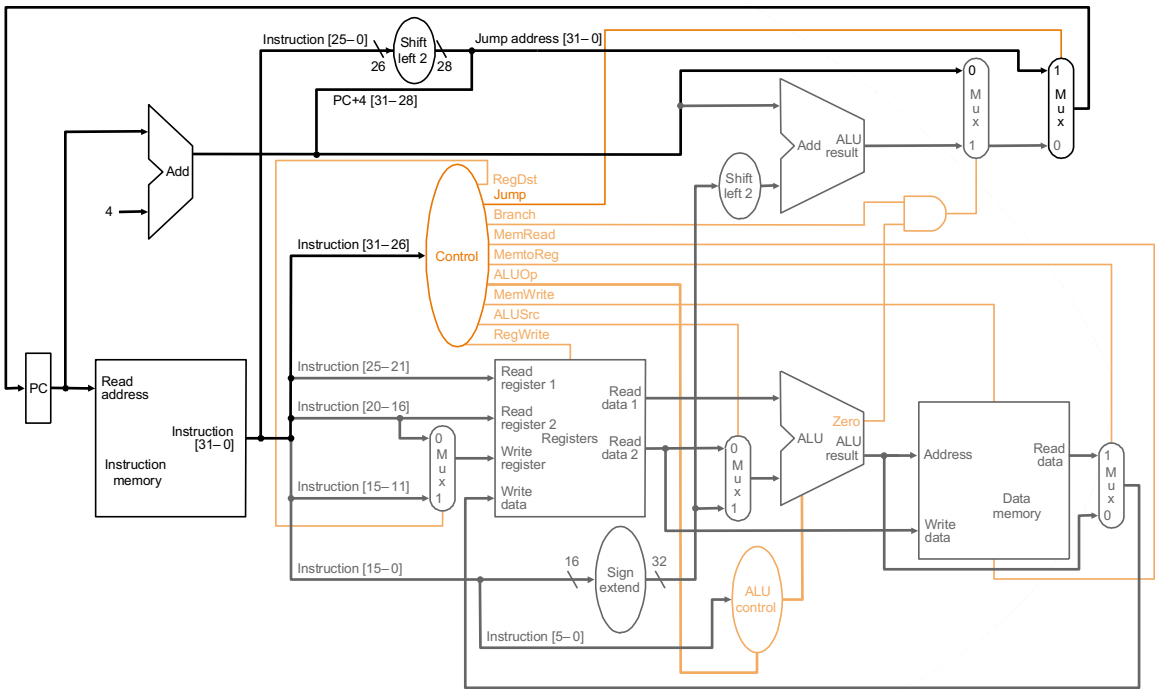
(2) Decode

Decode 단계는 패치 된 명령어를 해석하고 이해하는 과정이다. 이 단계에서 CPU는 명령어를 구성하는 비트들을 분석하여, 어떤 연산을 수행할지, 어떤 레지스터나 메모리 주소가 필요한지 등을 결정한다. 이 정보는 명령어 내에 있는 opcode와 operand들을 통해 알 수 있다.

(3) Execute

Execute 단계는 Decode 된 명령어에 따라 실제 연산을 수행하는 과정이다. 이 단계에서는 ALU(산술 논리 장치)가 활성화되어 덧셈, 뺄셈, 비교 등의 산술 연산 또는 논리 연산을 수행한다. 또한, 필요에 따라 데이터 메모리에 접근하여 값을 읽거나 쓰는 작업도 포함된다.

3) Datapath



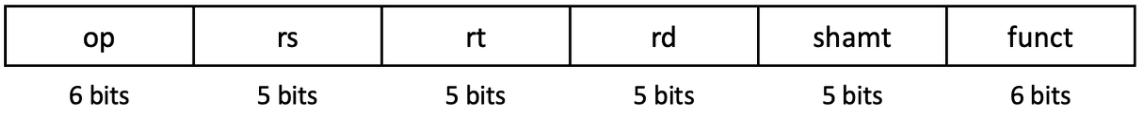

Single Cycle의 Data path는 5개의 단계로 나눌 수 있다.

1. IF(Instruction Fetch): PC가 가리키는 메모리 주소에서 다음에 실행할 명령어를 읽어오는 단계이다. 보통 명령어를 가져온 후 PC는 다음 명령어의 주소(PC + 4)로 업데이트 된다.
2. ID/FR(Instruction Decode/Register Fetch): fetch된 instruction을 분석하여 CPU가 이해할 수 있는 형태로 해석하는 단계이다. 명령어를 구성하는 비트들을 분석하고, 해당 명령어에 필요한 연산을 결정한다. 또한, 필요한 데이터를 레지스터 파일에서 읽어온다.
3. EX/AG(Execute/Address Generate): Decode 단계에서 결정된 연산을 실제로 수행하는 단계이다. ALU(산술 연산 장치)가 레지스터 값이나 상수를 사용하여 덧셈, 뺄셈 등의 산술 연산 또는 AND, OR 등의 논리 연산을 수행한다. 조건 분기나 비교 연산, 메모리 주소 계산도 이 단계에서 처리된다. 계산된 메모리는 다음 메모리 접근 단계에서 사용 된다.
4. MEM(Memory Access): 실행 결과를 메모리에 저장하거나 메모리에서 데이터를 불러오는 단계이다. 'LW' (Load Word) 명령어는 메모리에서 데이터를 불러와 레지스터에 저장하고, 'SW' (Store Word) 명령어는 레지스터의 데이터를 메모리에 저장한다.
5. WB(Write Back): 연산 결과나 메모리에서 읽은 데이터를 레지스터 파일에 쓰는 단계이다. EX 또는 MEM 단계에서 생성된 결과를 대상 레지스터에 저장하고 명령어 처리를 마무리 한다.

4) MIPS Instruction

MIPS 아키텍처에서 모든 명령어는 32비트 길이를 가진다. 32비트를 기준으로 0~31까지 32개의 레지스터가 존재한다. 다음은 MIPS Instruction의 3가지 유형이다.

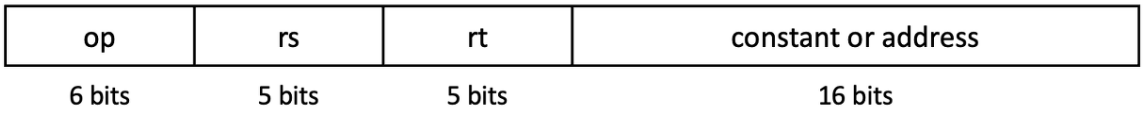
(1) R-type




R-type 명령어는 3개의 레지스터 연산자를 사용하며, 주로 산술 및 논리 연산에 사용된다. 사용되는 필드는 다음과 같다.

- opcode(6 bits): 명령어가 실행할 연산의 종류를 나타내며, R-type의 경우 0이다.
- rs(5 bits): 연산을 위해 사용되는 첫 번째 레지스터이다.
- rt(5 bits): 연산을 위해 사용되는 두 번째 레지스터이다.
- rd(t bits): 대상 레지스터로 연산의 결과가 저장된다.
- shamt(5 bits): 시프트 연산 시 시프트의 양을 나타내고, 사용되지 않는다면 0으로 고정된다.
- funct(6 bits): op 필드에서 연산의 종류를 나타내고, funct 필드에서 구체적인 연산을 지정한다.

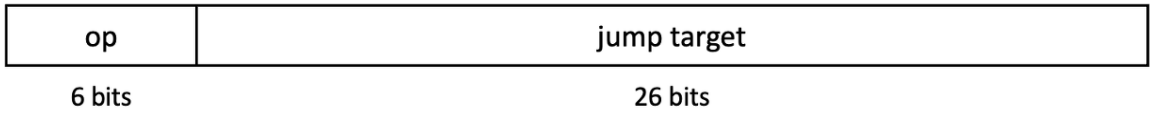
(2) I-type




I-type 명령어는 2개의 레지스터와 하나의 즉시 값을 사용하며 데이터 전송, 분기 및 즉시 값 연산에 주로 사용된다. 사용되는 필드는 다음과 같다.

- opcode(6 bits): 명령어가 실행할 연산의 종류를 나타낸다.
- rs(5 bits): 연산을 위해 사용되는 첫 번째 레지스터이다.
- rt(5 bits): 연산의 결과가 저장되는 대상 레지스터 또는 두 번째 레지스터이다.
- immediate(16 bits): 즉시 값, 주소 오프셋 또는 데이터 값이다.

(3) J-type




J-type 명령어는 프로그램의 실행 흐름을 제어하기 위해 사용되고 메모리의 특정 위치로 점프를 가능하게 한다. 사용되는 필드는 다음과 같다.

- opcode(6 bits): 명령어가 실행할 연산의 종류를 나타낸다.
- address(26 bits): jump 할 타겟 주소(상대 주소) 값이 저장된다.

2. Program Description

1) Instruction fetch (IF)

```
uint32_t fetch() {
    // Read in big-endian
    instruction = (memory[pc] << 24) |
                  (memory[pc+1] << 16) |
                  (memory[pc+2] << 8) |
                  memory[pc+3];
    return instruction;
}
```

fetch() 함수에서는 프로그램 카운터(pc)가 가리키는 위치에서 메모리를 읽어 현재 명령어(instruction)를 가져온다. 명령어는 big-endian 방식으로 메모리에서 읽혀지도록 설정하였다. 이는 주소 값이 높은 바이트부터 낮은 바이트 순서로 메모리에서 명령어를 조합한다.

2) Instruction Decode / Register Fetch (ID/RF)

```
void decode(uint32_t instruction) {
    uint32_t opcode = instruction >> 26;
    if (opcode == 0x00) { // R-type
        r_type_count++;
    } else if (opcode == 0x02 || opcode == 0x03) { // J-type
        j_type_count++;
    } else { // I-type
        i_type_count++;
    }
    execute(instruction);
}
```

decode() 함수에서는 명령어의 opcode를 추출하고, 해당 명령어가 R-type, I-type, J-type 중 어느 것인지 판단한다. Opcode의 조건에 따라 타입별로 카운팅하고, 해당 명령어를 실행하는 execute() 함수로 명령어를 전달한다.

3) Execute / Address Generation (EX/AG)

```
void execute(uint32_t instruction) {
    uint32_t opcode = instruction >> 26;
    uint32_t rs = (instruction >> 21) & 0x1F;
    uint32_t rt = (instruction >> 16) & 0x1F;
    uint32_t rd = (instruction >> 11) & 0x1F;
    uint32_t shamt = (instruction >> 6) & 0x1F;
    uint32_t funct = instruction & 0x3F;
    int16_t immediate = instruction & 0xFFFF;
    uint32_t address = instruction & 0x3FFFFFF;
    int32_t sign_extended_immediate = (int32_t)immediate;
    uint32_t value, mem_address;
```

execute() 함수에서는 명령어 타입과 세부 기능 코드에 따라 실제 연산을 수행한다. 연산을 하기 전 연산에 필요한 필드 값을 설정한다.

```
switch (opcode) {
    case 0x00: // R-type instructions
        switch (funct) {
            case 0x08: // jr
                pc = reg[rs];
                break;
            case 0x21: // addu
                value = reg[rs] + reg[rt];
                writeBack(rd, value);
                break;
        }
        break;
    case 0x02: // J
        pc = (pc & 0xF0000000) | (address << 2);
        break;
    case 0x03: // JAL
        reg[31] = pc;
        pc = (pc & 0xF0000000) | (address << 2);
        break;
    case 0x23: // LW
        mem_address = reg[rs] + sign_extended_immediate;
        if (mem_address % 4 == 0 && mem_address < MEMORY_SIZE) {
            value = (memory[mem_address] << 24) | (memory[mem_address + 1] << 16) |
                    (memory[mem_address + 2] << 8) | memory[mem_address + 3];
            writeBack(rt, value);
        } else {
            printf("Memory access error: Invalid address %08X\n", mem_address);
        }
        memory_access_count++;
        break;
    case 0x2B: // SW
        memWrite(reg[rs] + sign_extended_immediate, reg[rt]);
        memory_access_count++;
        break;
```

decode()에서 추출된 opcode로 case를 구분하여 연산을 진행한다. R-type의 경우 func 필드로 한

번 더 구분하여 연산을 처리하였다.

** 예외처리

```
default:
    printf("Unsupported opcode: %X\n", opcode);
```

만약 opcode가 execute() 함수에 존재하지 않으면 해당 opcode를 출력하여 확인할 수 있게 설정하였다.

4) Memory Access (MEM)

```
void memWrite(uint32_t address, uint32_t value) {
    if (address < MEMORY_SIZE) {
        // Check if the address is a multiple of 4 (word-aligned)
        if (address % 4 == 0) {
            memory[address] = (value >> 24) & 0xFF;
            memory[address + 1] = (value >> 16) & 0xFF;
            memory[address + 2] = (value >> 8) & 0xFF;
            memory[address + 3] = value & 0xFF;
        } else {
            printf("Memory write error: Address is not word-aligned\n");
        }
    } else {
        printf("Memory write error: Address out of bounds\n");
    }
}
```

memWrite()의 경우 sw 연산을 실행할 때 필요하며 데이터를 4바이트 단위로 메모리에 기록한다. 먼저, 주소가 메모리 범위 내에 있는지 확인하고 범위를 벗어나면 에러 메시지를 출력한다. 메모리 주소가 4로 나누어 떨어지지 않는다면 이 역시 에러 메시지를 출력한다.

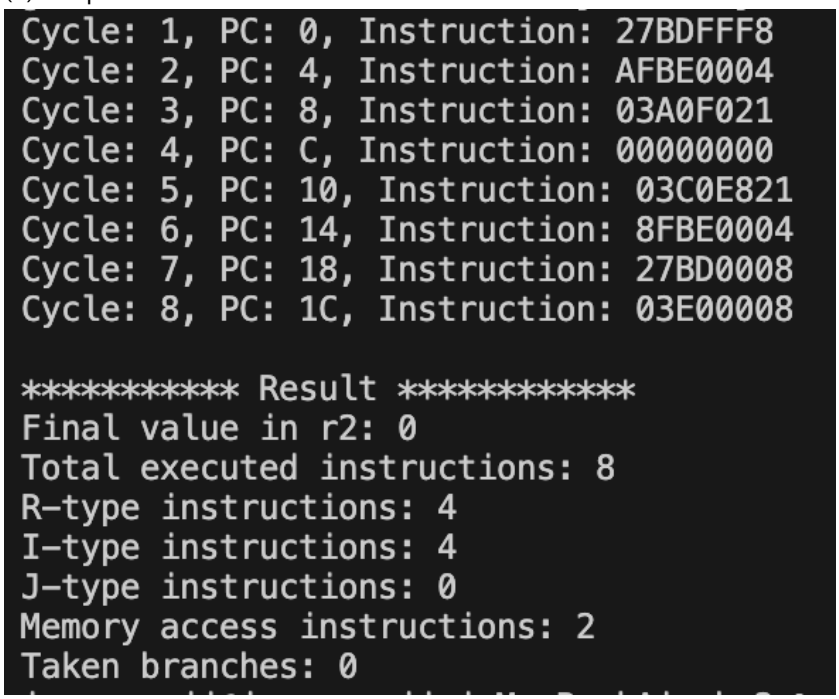
5) Write Back (WB)

```
void writeBack(uint32_t rd, uint32_t value) {
    reg[rd] = value;  // Write the value to the specified register
}
```

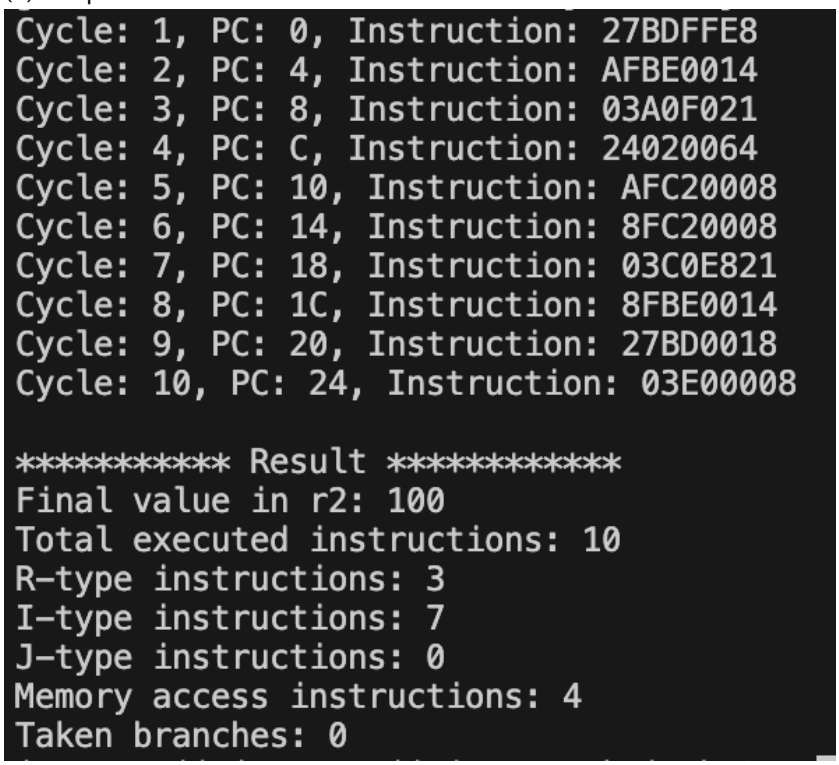
writeback()은 실행 결과로 나온 값을 명령어에서 지정된 레지스터에 쓰는 작업을 수행한다. 이 과정을 통해 연산 결과가 레지스터에 저장된다.

3. Result – 기본 구현

(1) simple.bin




(2) simple2.bin



(3) simple3.bin

```
Cycle: 1320, PC: 40, Instruction: AFC20008
Cycle: 1321, PC: 44, Instruction: 8FC20008
Cycle: 1322, PC: 48, Instruction: 00000000
Cycle: 1323, PC: 4C, Instruction: 28420065
Cycle: 1324, PC: 50, Instruction: 1440FFF3
Cycle: 1325, PC: 54, Instruction: 00000000
Cycle: 1326, PC: 58, Instruction: 8FC2000C
Cycle: 1327, PC: 5C, Instruction: 03C0E821
Cycle: 1328, PC: 60, Instruction: 8FBE0014
Cycle: 1329, PC: 64, Instruction: 27BD0018
Cycle: 1330, PC: 68, Instruction: 03E00008

********** Result **********
Final value in r2: 5050
Total executed instructions: 1330
R-type instructions: 409
I-type instructions: 920
J-type instructions: 1
Memory access instructions: 613
Taken branches: 101
```

## III. Conclusion

이번 과제를 진행하면서 다양한 오류를 마주하였다. simple.bin과 simple2.bin을 실행할 땐 메모리의 크기를 1024 * 1024로 설정하여도 결과가 도출되었으나 simple3.bin부터 제대로 된 결과가 도출되지 않고 무한루프에 걸리는 현상이 발생하였다. 메모리 크기를 0x4000000(64MB)로 키워 실행 했을 때 모든 파일에 대해 올바른 결과가 도출되었다. 하지만, 메모리의 적정 크기를 설정하는 방식에 대하여 좀 더 생각해보아야 할 필요성을 느꼈다. 그리고 simple3.bin을 실행할 때 r2값이 0, 1, 2가 반복되며 또 다시 무한루프에 빠지는 문제가 발생하였다. 문제를 해결하기 위해 명령어가 한 번 진행될 때마다 r2 값을 출력하도록 하였고 레지스터에 누적이 되지 않는 다는 것을 알게 되었다. 문제를 해결하기 위해 execute()의 여러 case들을 분석하였고 lw에서 메모리에 값을 로드하는데 문제가 있다는 사실을 알게 되었다. writeback()에 rt를 입력해야 하는데 rd를 입력하고 있어서 수정하였고, 주소가 4의 배수인지 확인하고 값을 32비트로 조합하여 입력하도록 하였다. 수정 후 값이 올바르게 도출됨을 확인 할 수 있었다. 마지막으로 sign_extended_immediate에 부호 확장 된 즉시 값을 할당 할 때 32비트로 확장하여 할당함으로써 정확한 값이 나오도록 하였다.

과제를 진행하며 생긴 문제들을 하나씩 해결하면서 명령어와 필드에 대한 이해도를 높일 수 있었고, 단일 사이클이 이루어지는 과정에 대하여 깊게 생각해볼 수 있었다. 앞으로 과제를 진행 할 때 먼저 큰 구조에 대하여 고민해보고 코드 작업을 진행해야겠다고 생각하였다.

## IV. References

ChatGPT – 코드의 전체적인 구조를 설정하고 에러를 고치는데 도움을 받았습니다.
MIPS 명령어 - https://jyukki.tistory.com