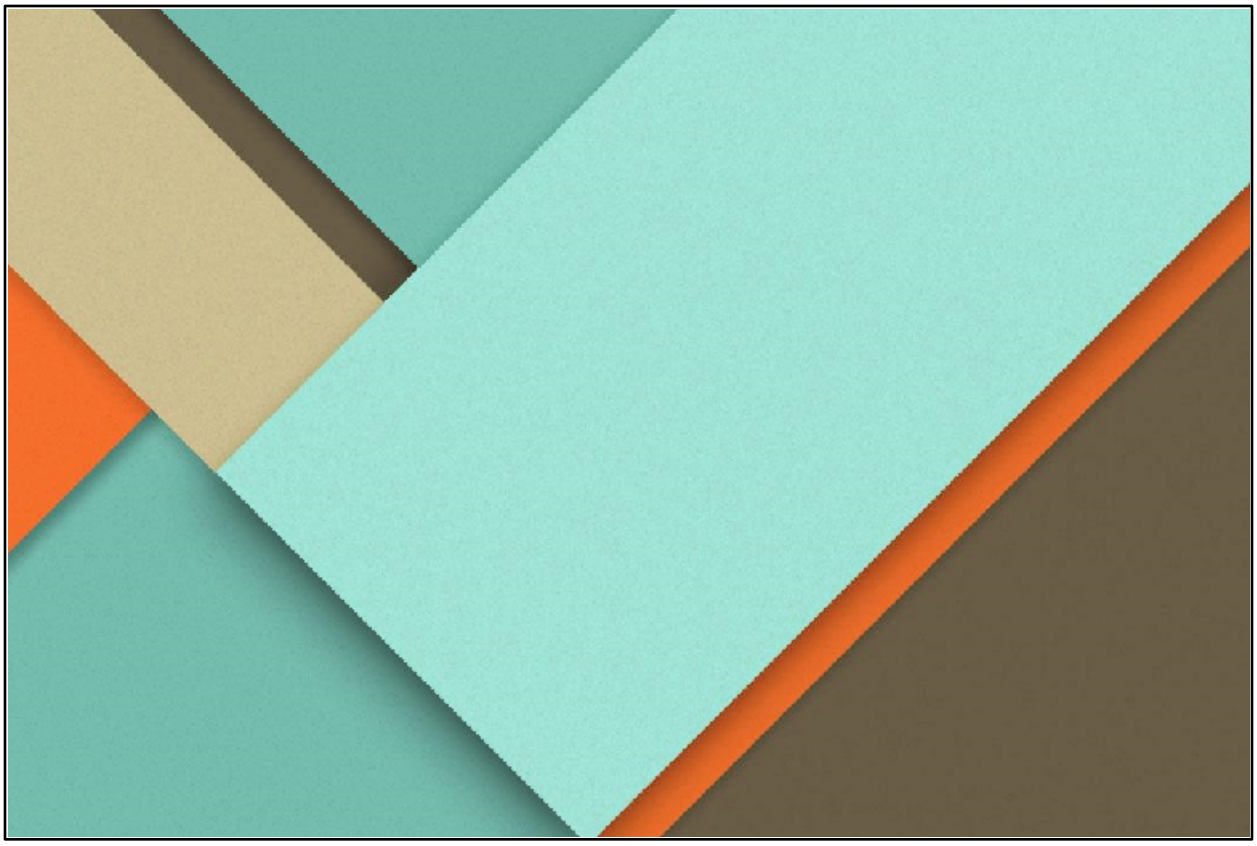


Computer Architecture

2023-05-30

—

엄현호

32192587@dankook.ac.kr

Contents

6.Lesson

1.Introduction

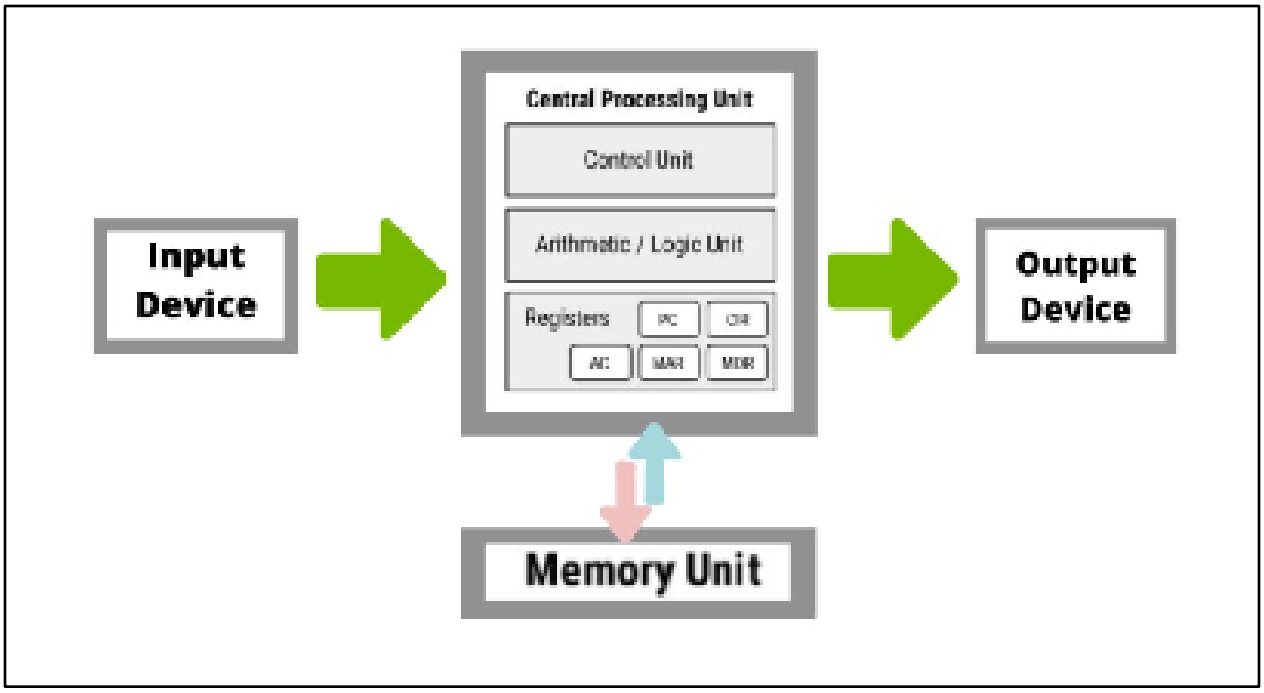
컴퓨터 구조와 MicroArchitecture 은 현대 컴퓨터 시스템의 핵심개념으로 이러한 개념을 이해하고 구현하는 것은 컴퓨터 구조에대한 높은 이해와 학습력을 요구합니다. 따라서 이번프로젝트에서는 지금까지 배운 컴퓨터 구조에대한 이해를 바탕으로 C 언어를 사용하여 직접 MIPS 프로세서 싱글 사이클을 구현해 으로써, 컴퓨터의 구조에 대한 이해도를 시험합니다.

이 보고서에는 싱글 사이클 구현함에 있어 필수적인 개념은 폰-노이만 구조, Instruction Set Architecture(ISA), MIPS, DataPath 등에 대해 설명하고, 어떤 방식 및 과정으로 구현 하였는지에

대해 설명합니다. 그과정에서 만든 세부적인 Data Path 의 설계나 들어가는 unit 들을 설명하고 이것들을 바이너리 파일이 어떻게 걸쳐 코드가 실행되고 결과가 나오는대해 알려줍니다.

2.Background

2-1. important concepts



1)VonNeuuman Architecture Stored Program

폰 노이만 구조는 프로그램 및 데이터를 저장하는 메모리와, 중앙 처리 장치(**Central Processing Unit, CPU**)로 구성됩니다. **CPU** 는 연산을 수행하고 명령어를 해석하는 역할을 담당합니다. 폰 노이만 구조에서는 메모리와 **CPU** 가 서로 다른 장소에 위치하며, 이들 간에 데이터 및 명령어가 주고받을 수 있도록 통신이 이루어집니다.

일반적으로 폰 노이만 구조에서는 컴퓨터의 동작을 순차적으로 수행하는 프로그램을 사용합니다. 프로그램은 메모리에 저장되어 **CPU** 에 의해 차례로 실행됩니다. **CPU** 는 메모리에서 명령어를 가져와 해석한 후, 해당 명령어에 맞게 데이터를 처리하거나 다음 명령어의 위치를 결정합니다. 이러한 프로세스는 컴퓨터가 복잡한 작업을 수행하는 데에도 적용될 수 있도록 설계되어 있습니다.

2)ISA

ISA 는 프로세서가 지원하는 데이터 타입, 레지스터 세트, 명령어 형식, 명령어 집합 등을 명시합니다. 소프트웨어 개발자는 **ISA** 에 맞게 명령어를 작성하고 프로그램을 개발할 수 있습니다. **ISA** 는 하드웨어 개발자에게도 중요한 역할을 합니다. **ISA** 를 기반으로

하드웨어가 설계되며, 이를 통해 프로세서의 구조, 레지스터, 데이터 경로, 제어 신호 등이 결정됩니다.

ISA 의 종류에는 여러 가지가 있습니다. 가장 일반적인 ISA 는 CISC(Complex Instruction Set Computer)와 RISC(Reduced Instruction Set Computer)입니다. CISC ISA 는 명령어 세트가 다양하고 복잡한 특징을 가지며, 한 명령어로 복잡한 작업을 처리할 수 있습니다. 반면, RISC ISA 는 명령어가 간단하고 규모가 작으며, 실행 속도가 빠르고 예측 가능한 특징을 가지고 있습니다. 이외에도 VLIW(Very Long Instruction Word), EPIC(Explicitly Parallel Instruction Computing) 등 다양한 ISA 가 존재합니다.

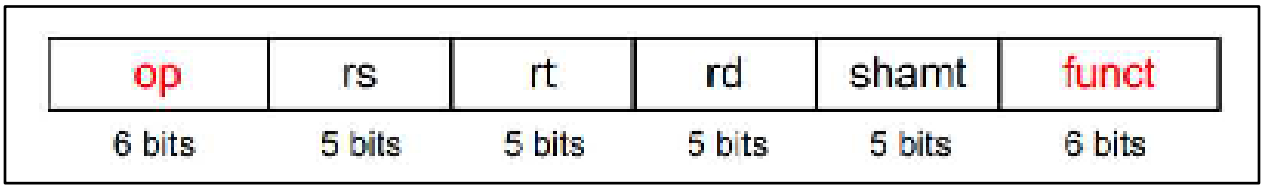
용될 수 있도록 설계되어 있습니다.

3) MIPS(Microprocessor without Interlocked Pipeline Stages)

MIPS 는 RISC(Reduced Instruction Set Computer) 아키텍처의 대표적인 예시중 하나입니다. MIPS ISA 는 MIPS 프로세서를 위한 명령어 집합을 정의하며, 간결하고 효율적인 설계를 특징으로 합니다. MIPS ISA 는 32 비트길이의 고정된 크기의 명령어를 갖고 있으며, 총 3 가지 타입의 명령어로 구성됩니다. 가장 기본적인 형식은 R 형식으로, 레지스터 간 산술 연산이 이루어지는 명령어입니다. 다음으로 I 형식은 레지스터와 상수간 연산을 수행하거나, 메모리와 데이터 간의 이동을 처리하는 명령어입니다. J 형식은 점프 명령어로, 프로그램 카운터(PC)를 다른 위치로 비교적 빠르게 분기할 수 있도록 지원합니다.

MIPS ISA 는 고정된 길이(32bit)의 명령어를 사용하기 때문에, 명령어의 해석(Decode)과 실행(Execution)이 단순화되어 매우 효율적인 하드웨어 구현이 가능합니다. 추가로, MIPS ISA 는 균일한 명령어 형식과 규칙적인 명령어 집합을 갖고 있어 프로그램의 복잡성을 낮추고 다음 프로젝트로 진행할 '파이프라이닝'과 같은 고성능 기법을 구현하는 데에 유리합니다.또한, MIPS ISA 는 32 개의 범용 레지스터를 제공하며, 주 연산이 레지스터 끼리 진행되기 때문에 프로그램 실행속도를 비약적으로 높일 수 있습니다. 물론, 필요한 경우 메모리에 데이터를 저장하거나 불러올 수 있도록 LW(Load Word)와 SW(Store Word)명령어 역시 지원합니다. 이를 통해 데이터를 메모리로부터 가져오거나 메모리에 저장할 수 있습니다.그렇다면 MIPS 의 세 타입의 명령어를 간단하게 살펴보겠습니다.

① R Type Instruction



R 타입은 총 6 개의 필드로 구분됩니다.

a) OPCode(6bits) : 6 비트로 이루어진 OPCode 로, R 타입에서는 항상 0 값을 가집니다.

b) RS(5bits) : 첫 번째 소스 레지스터의 index 입니다. 0~31 값을 가집니다.

c) RT(5bits) : 두 번째 소스 레지스터의 index 입니다.

d) RD(5bits) : Register Destination 의 약자로, 산술연산의 결과를 입력할 레지스터의 index 입니다.

e) shamt(5bits) : Shift Amount 의 약자로, Shift 연산시 몇 비트 shift 할지 정해주는 역할을 합니다.

f) funct(6bits) : R 타입에서, 실제로 어떤 산술&논리 연산을 할지 정해주는 역할을 합니다

② I Type Instruction

I 타입은 총 4 개의 필드를 가집니다.

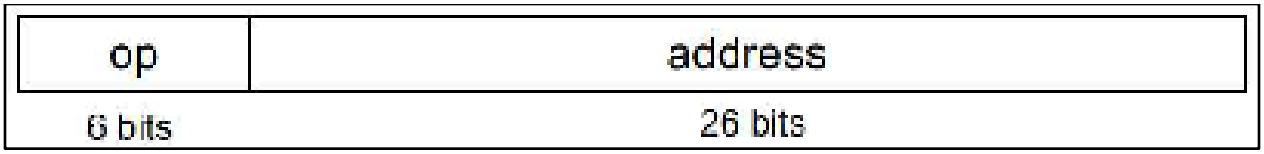
a) OPcode(6bits) : 어떤 산술연산을 할지 정해주는 역할을 합니다.

b) RS(5bits) : R 타입과 같이, Source Register Index 입니다.

c) RT(5bits) : Target Register 의 Index 를 담고 있습니다.

d) Immediate(16bits) : 주로 상수값 혹은 메모리 주소를 담고 있습니다. RS 와 같이 연산에 사용됩니다.

③ J Type Instruction



J 타입은 J(0x2), JAL(0x3)를 OPcode 로 갖는 명령어 두 가지만 있으며, 두가지 필드를 가집니다.

a) OPcode(6bits) : 단순히 Jump 할지, Jump and Link 를 할지 구분하는 역할을 합니다.

b) Address(26bits) : 어떤 Address 로 Jump 할지에 대한 메모리 주소를 담고 있습니다.

4) MIPS ISA DataPath & Components / Single Cycle Stages

MIPS 의 Datapath 는 프로세서에서 데이터의 흐름을 처리하는 경로를 나타냅니다. 이는 MIPS 아

키텍처에서 명령어의 실행을 담당하는 핵심 구성 요소로, 다양한 하드웨어 구성 요소와 데이터 버

스로 구성됩니다. MIPS Datapath 의 주요 구성 요소는 다음과 같습니다.

a) Register File : 레지스터 파일은 데이터를 저장하는 레지스터의 집합입니다. 일반적으로

MIPS 아키텍처에서는 32 개의 32 비트 레지스터를 제공하며, 이는 Rs, Rt, Rd 와 같은 레지스터 번

호를 사용하여 데이터에 접근할 수 있습니다.

b) 산술 논리 장치(ALU) : ALU 는 산술&논리 연산을 수행하는 하드웨어입니다. MIPS Datapath

에서는 ALU 가 주로 산술 연산, 비교 연산, 논리 연산 등을 처리합니다.

c) 데이터 메모리 & 명령어 메모리 : 데이터 메모리는 데이터를 저장하는 메모리 공간이고, 명령

어 메모리는 명령어들이 저장되어 있는 메모리 공간입니다. MIPS Datapath 에서는 메모리에 접근

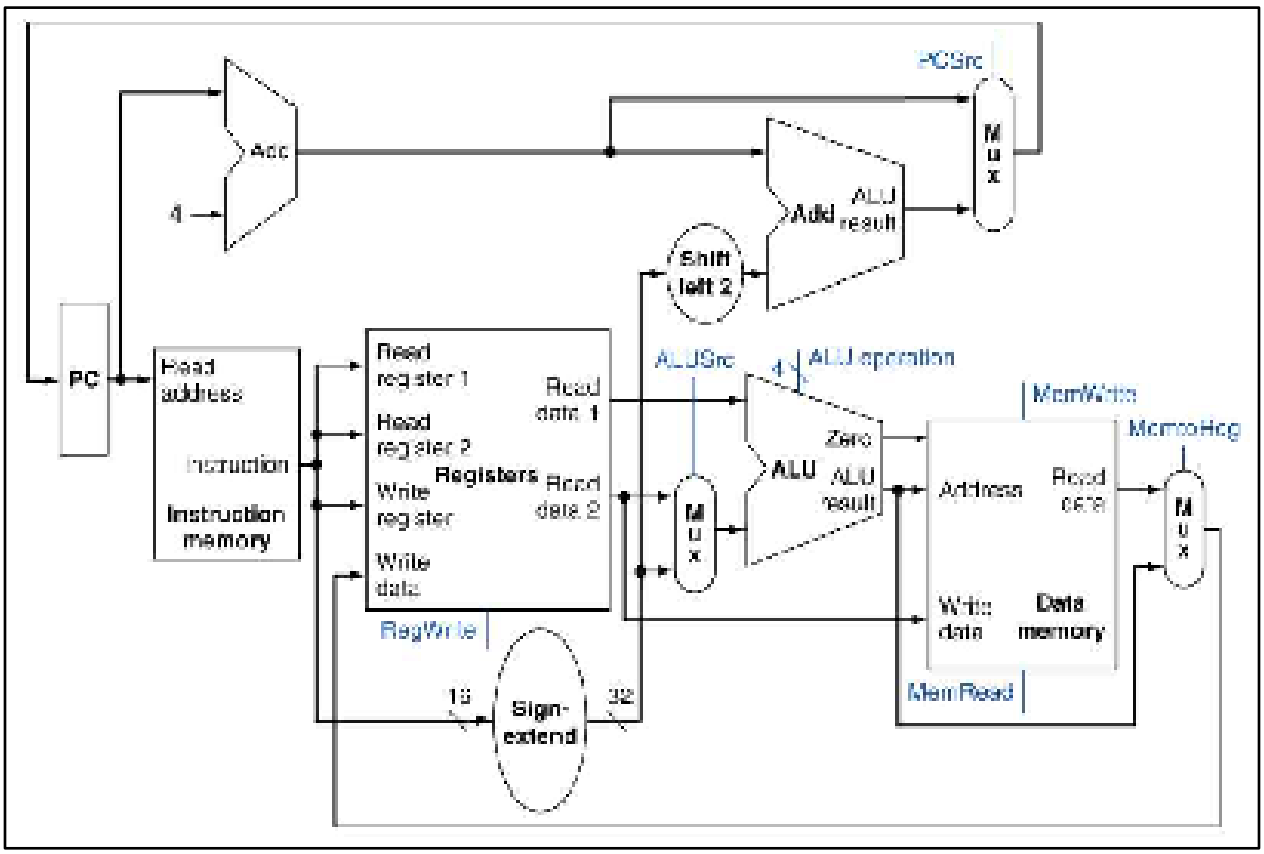
하여 데이터를 읽거나 쓰는 작업을 수행합니다.

※ 명령어 메모리에는 데이터를 읽기만 할 수 있을 뿐, 쓰는 작업은 할 수 없습니다.

d) 제어 유닛(Control Unit) : 제어 유닛은 명령어의 실행 흐름을 제어하는 역할을 합니다. 제어

유닛은 현재 실행 중인 명령어의 종류에 따라 필요한 제어 신호를 생성해 각 하드웨어 요소에 적절

한 신호를 보내, 명령을 작동시킵니다.



위 그림은 ControlUnit 을 일부 포함하여, Single Cycle MIPS 의 DataPath 를 간략하게 나타낸 것

입니다. 실제 구현에서는 일부 가필수정이 필요하지만 기본적인 흐름은 <그림 5>와 같습니다.

MIPS 에서는 명령어의 실행이 "IF - ID/RF - EX/AG - MEM

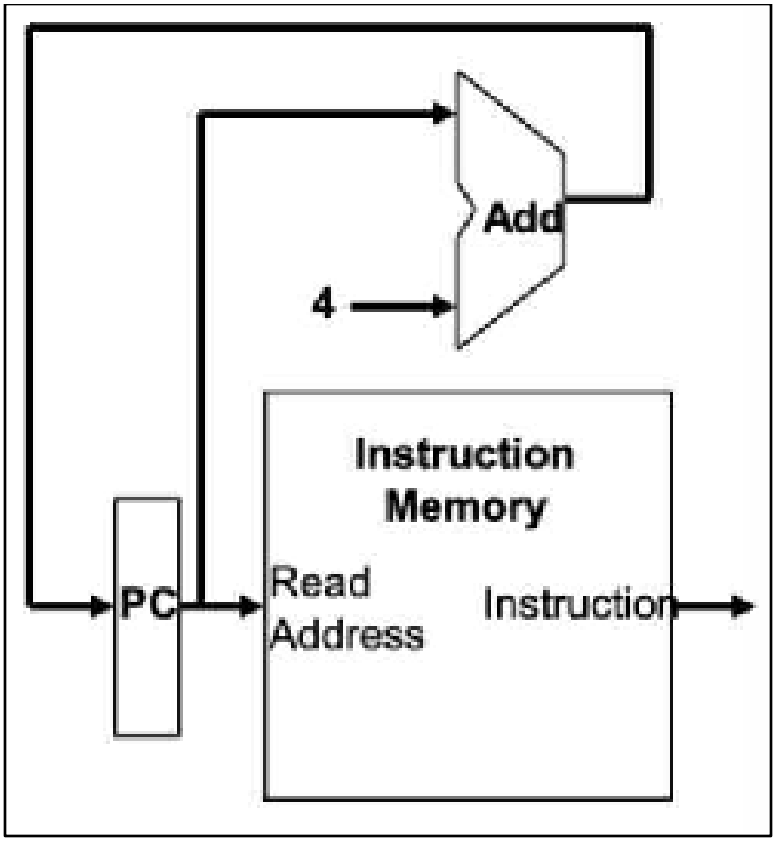
WB" 와 같이 크게 다섯 단계로

나누어 작동합니다. 각 단계는 다음과 같은 기능을 수행합니다.

a) IF (Instruction Fetch) : 이 단계에서는 명령어 메모리에서 명령어를 가져옵니다. 프로그램 카

운터 (PC)를 증가시키고, 해당 주소에서 명령어를 인출하여 명령어 레지스터(IR)에 저장합니다. IF

단계는 다음에 수행될 명령어의 주소를 결정하는 역할을 합니다.



a) IF (Instruction Fetch) : 이 단계에서는 명령어 메모리에서 명령어를 가져옵니다. 프로그램 카운터 (PC)를 증가시키고, 해당 주소에서 명령어를 인출하여 명령어 레지스터(IR)에 저장합니다. IF 단계는 다음에 수행될 명령어의 주소를 결정하는 역할을 합니다.

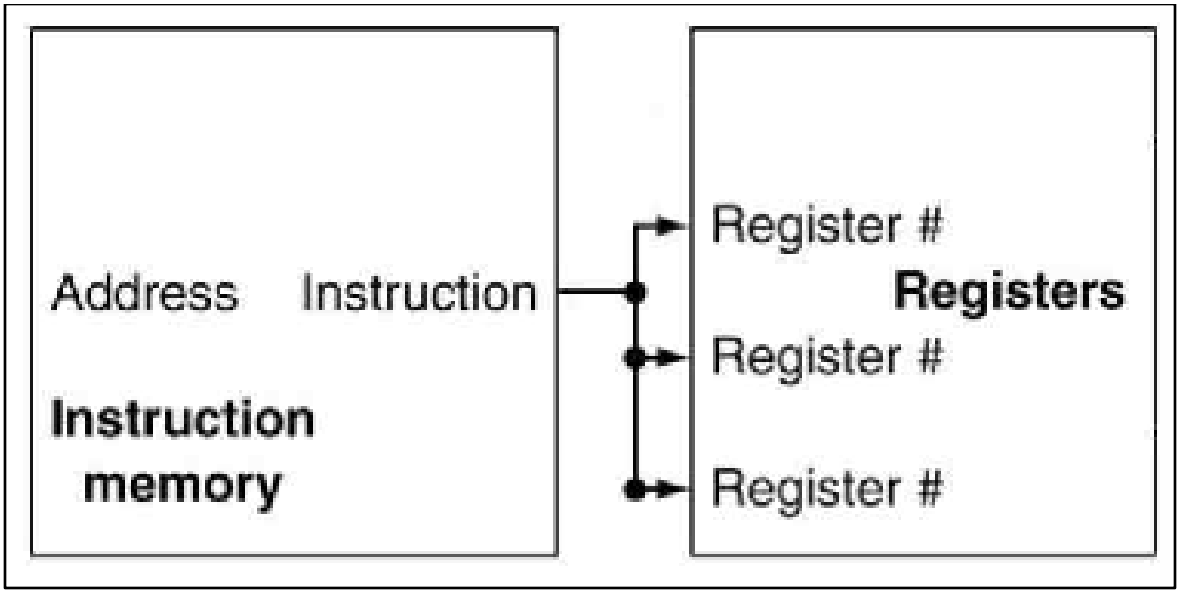
명령어 메모리(Instruction Memory)에서 PC 값에

해당하는 주소에 저장되어있는 4 바이트 크기의 Instruction 을 가

져오고, PC 의 값을 4 증가시켜주는 과정이 IF 단계입니다. 물론,

이후에 J 타입의 명령어나 I 타입의 Branch 명령어에 의해

PC=PC+4 이외의 다른 연산도 추가될 수 있으나, 가장 기본적인 IF 의 과정입니다.



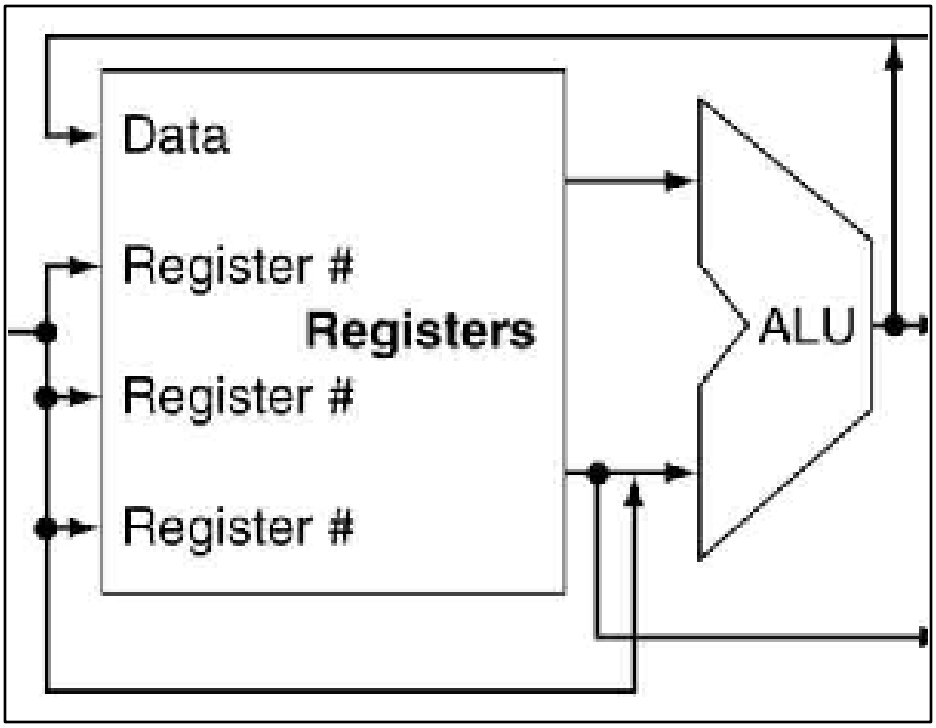
b) ID/RF (Instruction Decode/Register Fetch) : ID/RF 단계에서는 명령어를 해독하고 필요한 레지스터 값을 읽어옵니다. 명령어에서 필요한 제어 신호와 레지스터 번호 등을 추출하여 제어 유닛과 레지스터 파일에 전달합니다.

ID/RF 단계에서는 Instruction

Memory 에서 읽어드린 명령어를 RS, RT, RD,

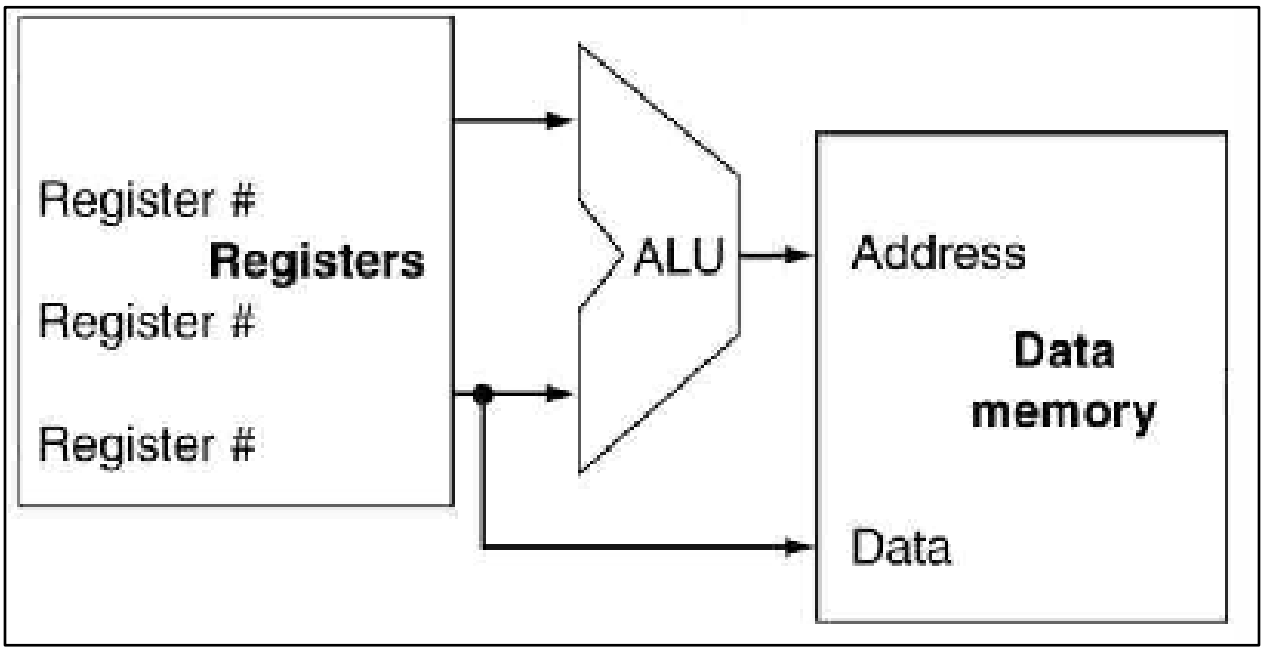
SHAMT, IMMED, FUNCT 등으로 분석하는 작업과,

RS, RT, RD 에 해당하는 레지스터 값을 읽습니다.



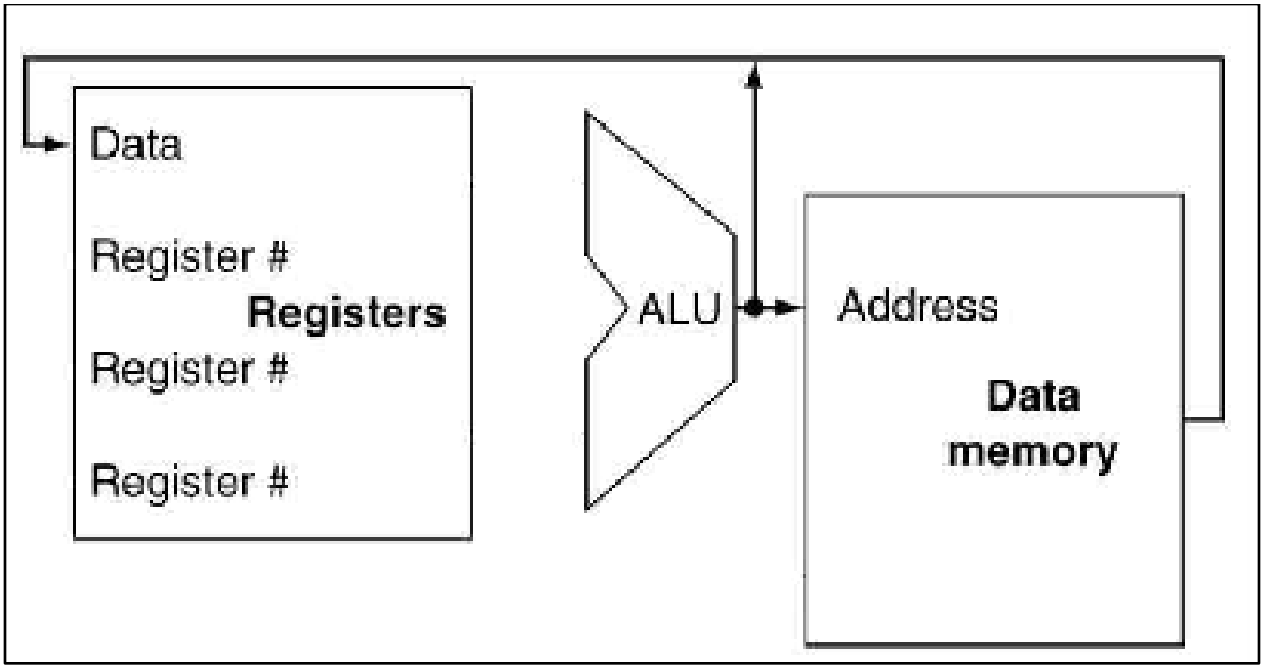
c) EX/AG (Execute/Address Generate) : EX/AG 단계에서는 산술 논리 연산과 주소 계산 등의 연산을 수행합니다. ALU (산술 논리 장치)에서 연산을 처리하고, 필요한 경우 주소 계산을 수행하여 데이터 메모리 주소를 생성합니다.

EX/AG 단계에서는 레지스터에서 읽어들인 값 및 IMMED 값 등과 산술 논리 연산을 진행합니다. 이때 결과값은 단순 데이터 값일 수도 있고 특정 주소(Address)값이 될 수도 있습니다



d) MEM (Memory Access) : MEM 단계에서는 데이터 메모리에 접근하여 데이터를 읽거나 쓰기 작업을 수행합니다. 주소 계산 결과로부터 데이터 메모리에 접근하고, 필요한 경우 데이터를 읽어올 수 있습니다.

MEM 단계에서는 ALU 에서 계산된 값을 Address 로 하는 메모리에 값을 쓰거나 읽어올 수 있습니다. 이는 메모리 접근과 관련된 명령어인 SW(Store Word)혹은 LW(Load Word)에 따라 결정됩니다.

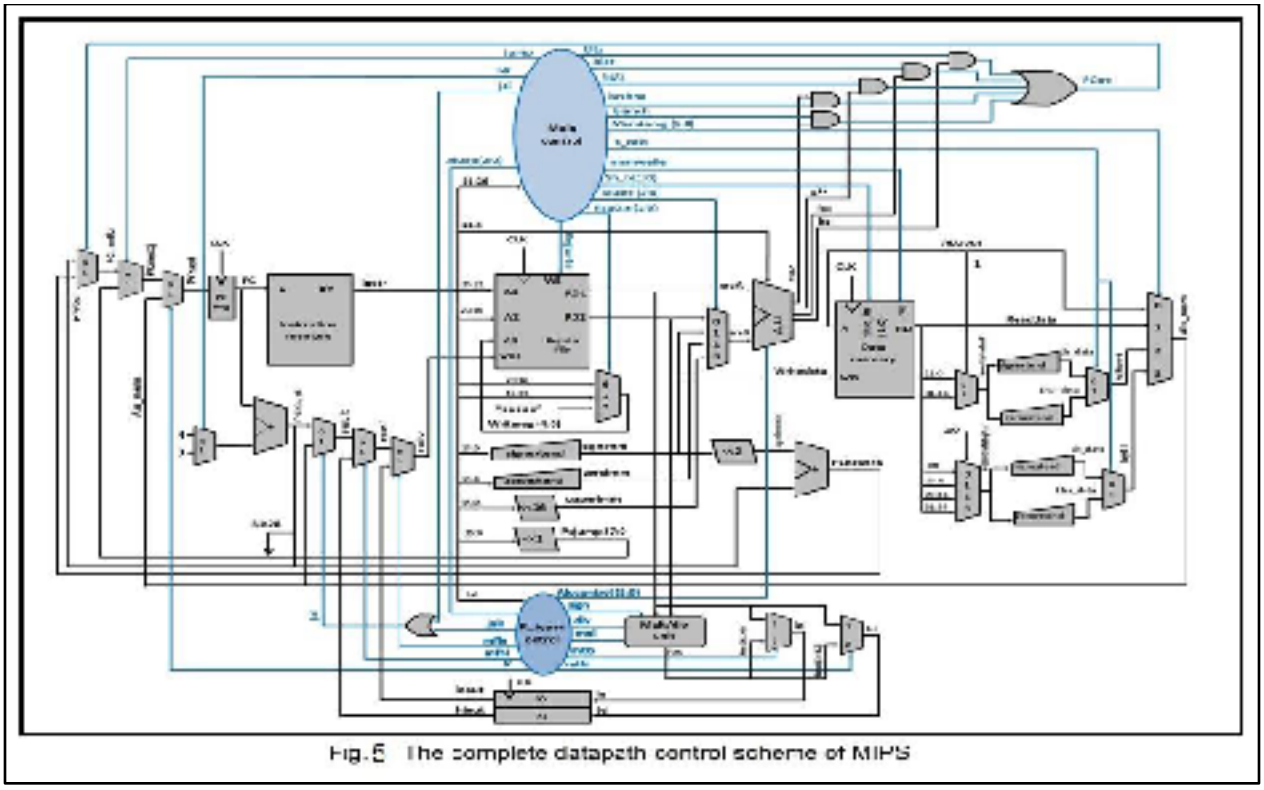


e) WB (Write Back) : WB 단계에서는 연산의 결과를 레지스터 파일에 기록합니다. EX/AG 또는 MEM 단계에서 계산된 결과나 데이터를 레지스터에 저장합니다.

ALU 에서 계산된 값 혹은 Data Memory 에서 읽어들인 값을 ID/RF 단계에서의 Rt 혹은 Rd 에 저장합니다.

3. Single Cycle MicroArchitecture Design

3-1. Datapath Implement



위사진은 제가 c 코드로 데이타패스 를 설계할때 쓰인 **Mips single Cycle DataPath** 입니다.

3-2. Implementation definition of this Processor

[3-1] 에서의 싱글 사이글 Datapath 를 구현하기 위해, 이 프로그램에서 선언한 구조체와 각 변수에 대해 간략하게 설명합니다.

3-2-1 header file and function

저는 코드를 헤더파일과 c 파일로 나눠서 함수 정의와 구현을 따로했고

컴퓨터의 작동과 부품 그다음 결과를 출력하는세 부분을 component cpu utils 로 폴더를 나누어서 구현 하였습니다. 중요한 부분만 설명 하겠습니다

먼저 memory.h 부분을 보면

```c
#ifndef MEMORY_H
#define MEMORY_H

typedef unsigned char byte;

typedef struct Memory {
    int size;
    byte* data;
} Memory;

Memory* Memory_load(int size, const char* path);
int Memory_read(Memory* memory, int address, int memRead);
void Memory_write(Memory* memory, int address, int value, int memWrite);
void Memory_free(Memory* memory);

#endif
```

메모리 header 에서 메모리에 read write free 함수를 정의하고 메모리는 구조체로 load 를 통해 데이터를 저장하게 했습니다.

```c
#ifndef CONTROLUNIT_H
#define CONTROLUNIT_H

#include "../header/component/Memory.h"
#include "../header/utils/Logger.h"

typedef struct {
    Memory* memory;
    Logger* logger;
    Registers* registers;
    DecodeUnit* decodeUnit;
    ALUnit* alu;
    ControlSignal controlSignal;
} ControlUnit;

ControlUnit* ControlUnit_create(Memory* memory, Logger* logger);
void ControlUnit_destroy(ControlUnit* controlUnit);
int ControlUnit_process(ControlUnit* controlUnit);

#endif /* CONTROLUNIT_H */
```

위사진은 컨트롤유닛 헤더파일입니다 컨트롤유닛 생성과 파괴 동작 을 함수로 정의했고 필요한 구조체를 정의 했습니다.

```
class ControlSignal {
public:
    ControlSignal(Opcode opcode)
        : opcode(opcode),
          regDest(opcode.getType() == Opcode::Type::R),
          aluSrc((opcode.getType() != Opcode::Type::R)
                 && (opcode != Opcode::BEQ)
                 && (opcode != Opcode::BNE)),
          shift(opcode == Opcode::SLL),
          upperImm(opcode == Opcode::LUI),
          memToReg(opcode == Opcode::LW),
          regWrite((opcode != Opcode::SW)
                   && (opcode != Opcode::BEQ)
                   && (opcode != Opcode::BNE)
                   && (opcode != Opcode::J)
                   && (opcode != Opcode::JR)),
          memRead(opcode == Opcode::LW),
          memWrite(opcode == Opcode::SW),
          jump((opcode == Opcode::J) || (opcode == Opcode::JAL)),
          branch((opcode == Opcode::BNE) || (opcode == Opcode::BEQ)),
          jr(opcode == Opcode::JR),
          jal(opcode == Opcode::JAL),
          aluOp(opcode.getOperation()) {}
```

decode 유닛 헤더파일입니다 각각 opcode 또한 opcode.h 에 정의되어있고

헤더에서 불러와서 decode 에서 쓰는식입니다. opcode 에 정의에맞게

구현했습니다.

```
class ParsedInstruction {
public:
    ParsedInstruction(int pc, int instruction)
        : pc(pc),
          instruction(instruction),
          opcode(Opcode::of(instruction)),
          rs(instruction >> 21 & 0x1F),
          rt(instruction >> 16 & 0x1F),
          rd(instruction >> 11 & 0x1F),
          shiftAmt(instruction >> 6 & 0x1F),
          immediate(immediate(pc, instruction)),
          address(address(pc, instruction)) {}
```

```
    static int immediate(int pc, int instruction) {
        int originImm = instruction & 0xFFFF;
        switch (Opcode::of(instruction)) {
            case Opcode::ADDIU:
            case Opcode::ADDI:
            case Opcode::SLTI:
            case Opcode::SW:
            case Opcode::LW:
                return signExtension32(originImm);
            case Opcode::ORI:
                return zeroExtension32(originImm);
            case Opcode::BNE:
            case Opcode::BEQ:
                return pc + branchAddress(originImm);
            default:
                return originImm;
        }
    }

    static int address(int pc, int instruction) {
        int originAddress = instruction & 0x3FFFFFF;
        switch (Opcode::of(instruction)) {
            case Opcode::J:
            case Opcode::JAL:
                return jumpAddress(pc, originAddress);
            default:
                return originAddress;
        }
    }
};

ParsedInstruction DecodeUnit_parse(int pc, int instruction);
ControlSignal DecodeUnit controlSignal(Opcode opcode);
```

이건 decode.h 에있는 인스트럭션 구현부입니다

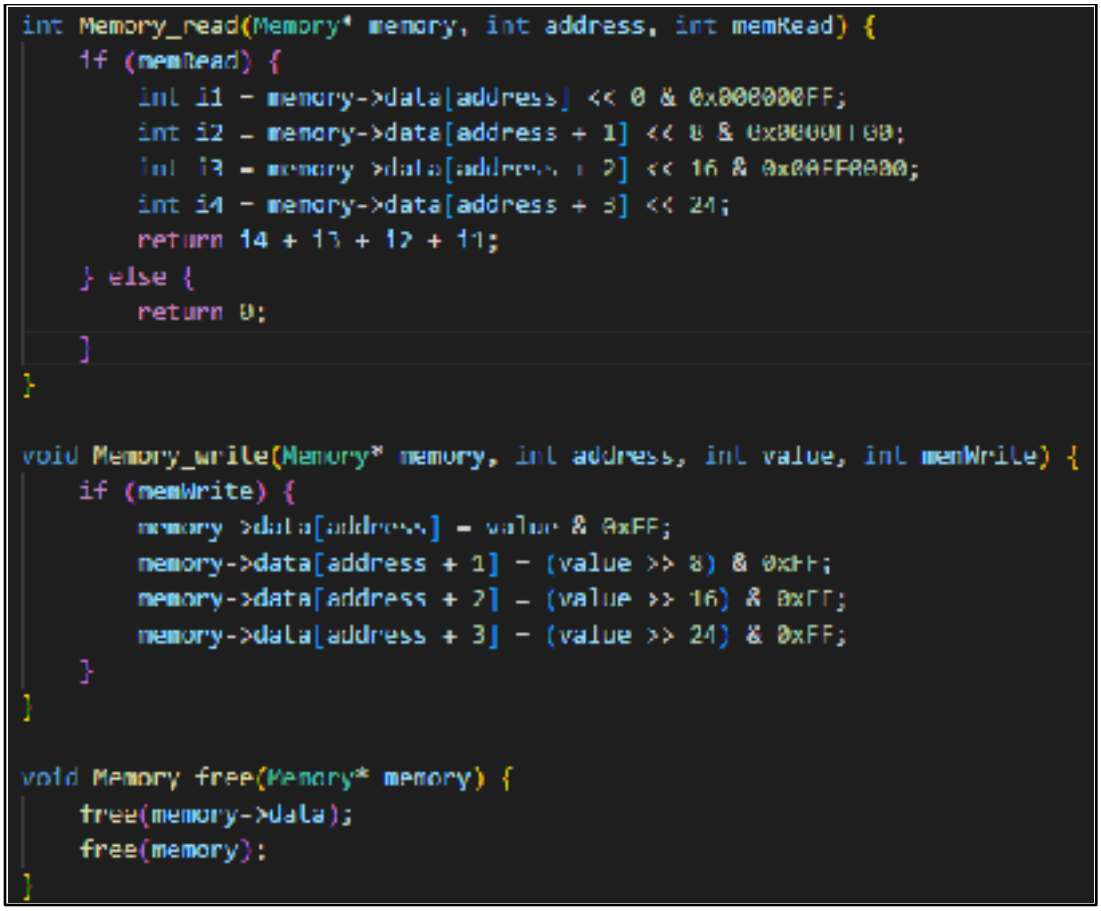
4.Implementation

4-1.memory.c

```
Memory* Memory_load(int size, const char* path) {
    FILE* file = fopen(path, "rb");
    if (file == NULL) {
        printf("파일을 열 수 없습니다: %s\n", path);
        return NULL;
    }

    Memory* memory = malloc(sizeof(Memory));
    memory->size = size;
    memory->data = malloc(sizeof(byte) * size);

    byte bytes[2048];
    int address = 0;
    size_t bytesRead;
    while ((bytesRead = fread(bytes, sizeof(byte), 2048, file)) > 0) {
        for (int i = 0; i < bytesRead; i += 4) {
            memory->data[address] = bytes[i + 3];
            memory->data[address + 1] = bytes[i + 2];
            memory->data[address + 2] = bytes[i + 1];
            memory->data[address + 3] = bytes[i];
            address += 4;
        }
    }

    fclose(file);
    return memory;
}
```

load 의 c 파일에 있는 함수 구현부입니다 메모리는 1 씩증가하게 했고

pc 는 4 씩증가하게 메모리를 구현했습니다.



```
int Memory_read(Memory* memory, int address, int memRead) {
    if (memRead) {
        int i1 = memory->data[address] << 0 & 0x000000FF;
        int i2 = memory->data[address + 1] << 8 & 0x0000FF00;
        int i3 = memory->data[address + 2] << 16 & 0x00FF0000;
        int i4 = memory->data[address + 3] << 24;
        return i4 + i3 + i2 + i1;
    } else {
        return 0;
    }
}

void Memory_write(Memory* memory, int address, int value, int memWrite) {
    if (memWrite) {
        memory->data[address] = value & 0xFF;
        memory->data[address + 1] = (value >> 8) & 0xFF;
        memory->data[address + 2] = (value >> 16) & 0xFF;
        memory->data[address + 3] = (value >> 24) & 0xFF;
    }
}

void Memory_free(Memory* memory) {
    free(memory->data);
    free(memory);
}
```

각각 read write free 함수 구현부입니다 읽는부분도 4 가증가하면 다음 명령어가실행되게 짯습니다.

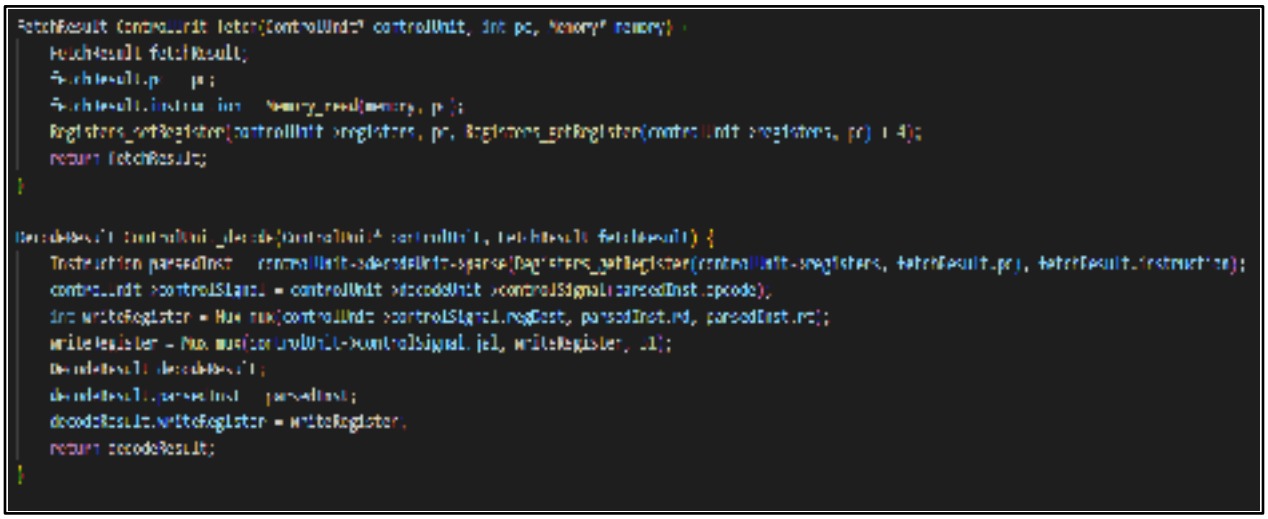
4-2. ControlUnit.c

```
ControlUnit* ControlUnit_create(Memory* memory, Logger* logger) {
    ControlUnit* controlUnit = (ControlUnit*)malloc(sizeof(ControlUnit));
    controlUnit->memory = memory;
    controlUnit->logger = logger;
    controlUnit->registers = Registers_create(32);
    controlUnit->decodeUnit = DecodeUnit_create();
    controlUnit->alu = ALUnit_create();
    return controlUnit;
}

void ControlUnit_destroy(ControlUnit* controlUnit) {
    Registers_destroy(controlUnit->registers);
    DecodeUnit_destroy(controlUnit->decodeUnit);
    ALUnit_destroy(controlUnit->alu);
    free(controlUnit);
}

int ControlUnit_process(ControlUnit* controlUnit) {
    int cycleCount = 0;
    FetchResult ifResult;
    DecodeResult idResult;
    ExecutionResult exResult;
    MemoryAccessResult maResult;
    WriteBackResult wbResult;
```

controlunit.c 입니다. 컨트롤 유닛의 생성 파괴 실행을 구현했습니다.

컨트롤 유닛 생성은 메모리와 로거에 저장 기록되고 다른 함수도 cpu 의 실행 과정에 맞게 실행되게 짰습니다.

4-3. fetch and decode



```
ParsedInstruction DecodeUnit_parse(int pc, int instruction) {
    ParsedInstruction parsedInst;
    parsedInst.pc = pc;
    parsedInst.instruction = instruction;
    parsedInst.opcode = Opcode_of(instruction);
    parsedInst.rs = (instruction >> 21) & 0x1F;
    parsedInst.rt = (instruction >> 16) & 0x1F;
    parsedInst.rd = (instruction >> 11) & 0x1F;
    parsedInst.shiftAmt = (instruction >> 6) & 0x1F;
    parsedInst.immediate = DecodeUnit_immediate(pc, instruction);
    parsedInst.address = DecodeUnit_address(pc, instruction);
    return parsedInst;
}
```

controlunit.c 와 decode.c 에있는 fetch 부분과 decode 실행 하는 코드입니다.pc+4 와 mux 신호를 디코드하는 부분 구현입니다.

4-4. Excute

```
int main() {
    Logger logger = initLogger();
    const char* fileToLoad = "HW2/input1/input.bin";
    Memory memory = loadMemory(20000000, fileToLoad);

    ControlUnit controlUnit;
    initControlUnit(&controlUnit, &memory, &logger);
    int processResult = process(&controlUnit);

    printProcessResult(&logger, processResult);

    return 0;
}

Logger initLogger() {
    Logger logger;
    logger.cycle = 1;
    logger.fetch = 1;
    logger.decode = 1;
    logger.execute = 1;
    logger.memoryAccess = 1;
    logger.writeBack = 1;
    logger.resultInformation = 1;
    logger.sleepTime = 0;

    return logger;
}
```

main.c 입니다 컨트롤 유닛이 생성 실행되고 함수가 컴퓨터 동작 흐름에따라 구현되게했습니다 마지막 결과로 logger 를 출력해 화면에 보여주는 식으로 짰습니다.

5. Build Environment for Testing Simulator

5-1. Code check mechanism & run program

실행방법은 먼저 환경이 리눅스 환경이어야합니다. 만약 리눅스 환경이아니라면 groomide 에서 리눅스환경을 만들고 파일을 업로드한후 실행해야 됩니다. 먼저 최상위 폴더에서 `const char* fileToLoad = "HW2/input4/input.bin";`

이부분의 경로와 맞게 터밀너에 gcc main.c 를 치고 exe 파일을 만든후 나오는 exe 파일을 실행하면 됩니다.

5-1. run screen

input1

```
int main() {
    int sum = 0;

    for (int i=0; i<10; i++)
        sum += 1;

    return sum;
}
```

```
[Write Back] R[29] <- 0x1000000
[PC Update] PC <- 0x00000064 + 0x00000004

cle[146] (pc: 0x64)
[Fetch Instruction] 0x03e00008
[Decode Instruction] type: R
        opcode: 0x0, rs: 0x1f (H[31] 0xffffffff), rt: 0x0 (H[0] 0x0), rd: 0x0 (0), shamt: 0x0, funct: 0x8
[Execute]
        alu src1: 0xffffffff, src2: 0x0, result: 0xffffffff
[PC Update] PC <- 0xffffffff + 0[31]: 0xffffffff

Return value (r2): 45
Total Cycle: 146
Executed 'R' instruction: 13
Executed 'I' instruction: 101
```

input2

```c
int foo(int index);

int main() {
    int index = 4;
    return foo(index);
}

int foo(int index) {
    if (index == 1)
        return 1;
    else
        return index + foo(index-1);
}
```

```
[Execute]
        alu src1: 0xfffffd8, src2: 0x28, result: 0xffffff8
[Load] W[28] <- Mem[0xffffffd8] = 0x0
[PC Update] PC <- 0x00000030 + 0x0000002c+4

cle[94] (pc: 0x30)
[Fetch Instruction] 0x2fbd0028
[Decode Instruction] type: I
        opcode: 0x8, rs: World (W[29] 0xffffd8), rt: World (W[29] 0xffffd8), imm: 0x28
[Execute]
        alu src1: 0xffffd8, src2: 0x28, result: 0x00000000
[Write Back] W[29] <- 0x00000000
[PC Update] PC: <- 0x00000034   0x00000030+4

cle[98] (pc: 0x34)
[Fetch Instruction] 0x03e00008
[Decode Instruction] type: R
        opcode: 0x0, rs: World (W[31] 0xffffffff), rt: 0x0 (W[0] 0x0), rd: 0x8 (8), shamt: 0x0, funct: 0x8
[Execute]
        alu src1: 0xffffffff, src2: 0x0, result: 0xffffffff
[PC Update] PC: <- 0xffffffff - W[31]: 0xffffffff

Return value (r2): 10
Total Cycles: 95
Executed 'R' instruction: 24
Executed 'I' instruction: 60
Executed 'J' instruction: 4
Number of Branch Taken: 4
Number of Memory Access Instruction: 36
root@guorm:/workspace/ringbuscpu/single cycle#
```

input3

```c
int fib(int n);

int main()
{
    int fib_n = 10;
    int result;

    result = fib(fib_n);
}

int fib(int n)
{
    int result;
    if (n <= 2) return 1;
    else {
        result = fib(n-1) + fib(n-2);
        return result;
    }
}
```

```
cle[33] (pc: 0x2c)
|Fetch instruction| 0x01bc0020
|Decode instruction| type: I
        opcode: 0x20, rs: 0x1d (R[29] 0xfffffd0), rt: 0x1e (R[30] 0xfffffd8), imm: 0x20
[Execute]
        alu res1: 0x[[[[d8, src2: 0x28, result: 0x[[[[[8
[Load] R[30] <- Mem[0x0fffffff]   0x8
[PC Update] PC <- 0x00000030   0x00000028<<1

cle[34] (pc: 0x30)
[Fetch instruction] 0x27bd0020
|Decode instruction| type: I
        opcode: 0x9, rs: 0x1d (R[29]=0xfffffd8), rt: 0x1d (R[29]=0xfffffc8), imm: 0x28
[Execute]
        alu res1: 0xFFFFfdN, src2: 0x2N, result: 0x[0000000
[Write Back] R[29] <- 0x10000000
[PC Update] PC <- 0x00000031 + 0x00000004<<1

cle[35] (pc: 0x34)
[Fetch instruction] 0x03e00008
|Decode instruction| type: R
        opcode: 0x0, rs: 0x1f (R[31]=0x[[[[[[[[), rt: 0x0 (R[0]=0x0), rd: 0x8 (8), shamt: 0x0, funct: 0x8
[Execute]
        alu res1: 0x[[[[[[[[, src2: 0x0, result: 0x[[[[[[[[
|PC Update| PC <  0x[[[[[[[[ = R[31]: 0x[[[[[[[[

Return value (r2): 10
Total Cycle: 35
Executed 'R' instruction: 24
Executed 'I' instruction: 00
Executed 'J' instruction: 4
Number of Branch Taken: 4
Number of Memory Access Instructions: 00
root@goorm:/workspace/singlecycle/single cycle#
```

input4

```c
int gcd(int a, int b);

void main()
{
    int a = 0x1298;
    int b = 0x09387;
    int res;

    res = gcd (a, b);
}

int gcd(int a, int b)
{
    if (a==b) return a;
    else if (a>b) return gcd(a-b, b);
    else return gcd(b-a, a);
}
```

```
cle[95] (pc: 0x70)
[Fetch Instruction] 0x[illegible]
[Decode Instruction] type: I
        opcode: 0x23, rs: 0x1d (R[29]=0xeffffdd0), rt: 0x1c (R[28]=0x10008000), imm: 0x70
[Execute]
        alu src1: 0xffffffd0, src2: 0x20, result: 0xfffffff0
[Load] R[28] <- Mem[0x10007ff0] = 0x0
[PC Update] PC <- 0x00000038 + 0x0000002c+4

cle[96] (pc: 0x38)
[Fetch Instruction] 0x27bd0020
[Decode Instruction] type: I
        opcode: 0x9, rs: 0x1d (R[29] 0xffffffd0), rt: 0x1d (R[29] 0xffffffd0), imm: 0x20
[Execute]
        alu src1: 0xffffffd0, src2: 0x20, result: 0x10000000
[Write Back] R[29] <- 0x10000000
[PC Update] PC <- 0x00000034   0x00000030+4

cle[97] (pc: 0x34)
[Fetch Instruction] 0x03e00008
[Decode Instruction] type: R
        opcode: 0x0, rs: 0x1f (R[31]=0xffffffff), rt: 0x0 (R[0]=0x0), rd: 0x0 (0), shamt: 0x0, funct: 0x8
[Execute]
        alu src1: 0xffffffff, src2: 0x0, result: 0xffffffff
[PC Update] PC <- 0xffffffff + R[31]: 0xffffffff

Return value (r2): 10
Total Cycle: 93
Executed 'R' instruction: 24
Executed 'I' instruction: 68
Executed 'J' instruction: 4
Number of Branch Taken: 0
Number of Memory Access Instruction: 30
root@goorm:/workspace/singlecycle/single_cycle#
```

6. Lesson

싱글 사이클 데이터 패스는 각 명령어를 단일 클럭 사이클 동안 처리하는 간단한 구조입니다. 이를 구현하면서 명령어의 실행 단계를 명확하게 이해할 수 있었고, 각 구성 요소의 역할과 상호 작용이 직관적으로 드러났습니다.데이터 패스의 제어 신호는 명령어의 유형에 따라 동적으로 변경됩니다. 이를 위해 제어 유닛이 필요하며, 제어 유닛은 제어 신호를 생성하고 각 구성 요소의 동작을 조정합니다. 제어 유닛의 역할과 중요성을 깨닫게 되었습니다.싱글 사이클 데이터 패스는 단순하고 직관적이지만, 복잡한 명령어를 처리할 때는 효율이 떨어질 수 있습니다. 성능을 향상시키기 위해서는 파이프라인, 분기 예측, 메모리 계층 구조 등을 고려해야 합니다. 따라서 성능과 설계의 상충 관계에 대해 이해하게 되었습니다.데이터 패스는 컴퓨터 시스템의 핵심 구성 요소 중 하나입니다. 데이터 패스를 구현하면서 레지스터, ALU, 메모리 등 다른 구성 요소들과의 상호 작용과 시스템의 전체적인 동작을 이해하게 되었습니다