Report

HW#2 Single-Cycle MIPS

Computer Architecture

Dpt. of Mobile System Engineering
32192530 양윤성

2023.06.01.

● Contents

1. Introduction

Single Cycle은 컴퓨터 내에서 명령어 처리를 한 Clock Cycle 내에 완료하는 방식이다. 명령어의 종류와 상관 없이 하나의 Clock에서 하나의 명령어를 고정적으로 수행하므로 다음 명령어가 실행되기 전 기존 명령어의 처리가 완료되어야 한다. Single Cycle은 컴퓨터 구조에서 중요한 개념 중 하나로 가장 기초적인 명령어 처리 구조이다. 이 개념을 이해하면 컴퓨터의 작동원리에 대한 보다 깊은 이해를 할 수 있으며, 이를 바탕으로 어떻게 프로그램에서 명령어 처리 과정을 최적화할지와 하드웨어를 개발할지에 대한 답을 얻을 수 있다. 따라서 CPU의 Architecture을 이해하기 위해 Single Cycle의 원리를 파악하는 것은 필수이다.

Simple Calculator에 이은 두번째 프로젝트가 바로 MIPS 기반 Single Cycle 에뮬레이터이다. MIPS binary 파일을 읽어서 실행한 뒤 binary 파일의 정보를 메모리에 저장한다. 32비트의 명령어들은 MSB부터 8비트씩 메모리 배열에 순차적으로 저장되는데, 이는 MIPS와 동일한 Big Endian이다. 이 에뮬레이터에는 r0부터 ra(r31)까지 총 32개의 레지스터를 지원하며, 레지스터와 메모리의 상호작용으로 명령어가 5가지 단계 (Fetch, Decode, Execution, Memory Access, Write Back)를 거쳐 처리된다. 사용자는 하나의 명령어가 실행될때마다 PC와 레지스터, 메모리의 값이 변화하는 과정을 확인할 수 있다. 명렁어 처리는 PC값이 0xFFFF:FFFF가 되기 전까지 반복하며 최종 Return Value는 v0(r2) 레지스터에 저장된다.

이번 레포트에서는 Single Cycle을 이해하기 위한 MIPS와 관련된 개념들을 먼저 소개하고 5단계 명령어 처리 순서와 Data path 및 명령어 처리에 사용된 부품에 대해 설명할 것이다. 배경지식 소개 뒤에는 에뮬레이터의 전체 회로도를 소개하고 본격적인 Single Cycle 구현 코드를 결과 사진과 함께 안내한다. 마지막으로 프로젝트를 위한 개발 환경과 컴파일 과정에 대해 설명한 뒤 구현을 하며 생각했던 과정과 개인적으로 배웠던 점에 대해 말할 것이다.

2. Background

2-1. ISA (Instruction Set Architecture)

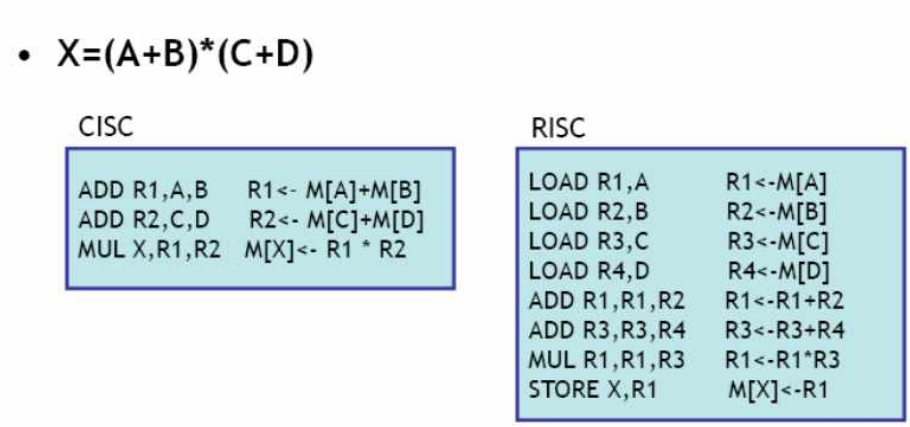


그림1. CISC와 RISC의 차이

ISA (Instruction Set Architecture)는 명령의 집합과 실행 방식, 명령어 실행 시 시스템의 상태가 어떻게 바뀌는지에 대해 정의하는 인터페이스이다. ISA에는 명령어의 길이, 메모리 주소 지정 방식, 데이터의 유형 등과 같은 명령어와 관련한 모든 세부사항을 포함한다. 그래서 프로그래머들은 ISA를 통해 프로그램과 CPU 사이의 상호작용을 이해할 수 있다. ISA를 통해 하드웨어와 소프웨어가 서로 호환되도록 보장할 수 있으며, 이는 컴퓨터 시스템 전체의 성능과 안정성에 영향을 미친다. 여기서 명령어를 어떤 방식으로 처리하냐에 따라 RISC와 CISC로 나뉜다. RISC는 명령어의 수를 간소화하고 명령어 처리에 필요한 하드웨어를 최소화 하여 처리 속도를 높이는데에 집중한 방식이다. 이러한 이유로 RISC에서는 명령어의 구조가 단순하고, 하나의 명령어 실행을 위한 Clock Cycle이 적어 명령어 처리 속도가 빠르다. 반면 CISC는 명령어의 기능을 다양화하는 것에 초점을 두었다. 기능이 다양하기에 명령어의 구조가 RISC에 비해 복잡하여 Piplining(병렬) 구현과 확장이 어렵다. 또한 요구하는 Clock Cycle도 많아 명령어 처리 속도가 느리다. 대신 프로그래머가 작성할 코드는 오히려 간결해지고 메모리를 RISC보다 절약할 수 있다는 장점이 있다. 후술할 MIPS는 RISC 디자인을 채택했다.

2-2. Byte Addresses

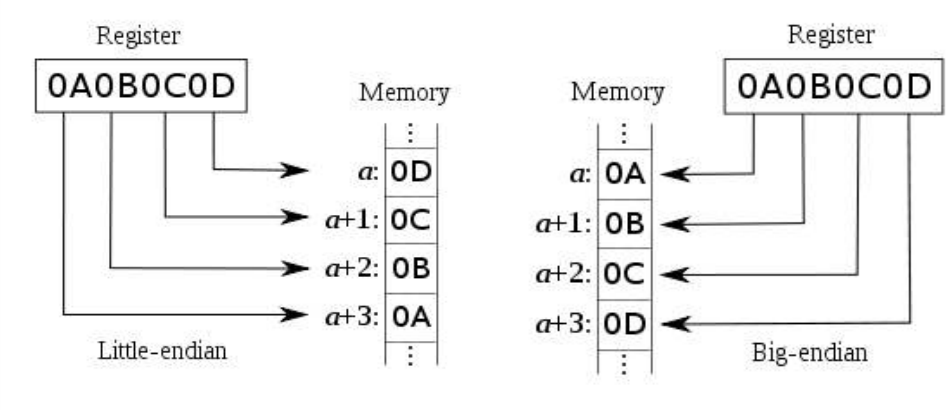


그림2. 메모리 주소 지정을 위한 2가지 방식

ISA에서 명령어나 데이터 등을 메모리에 저장할 때 저장하는 순서에 따라 Little-Endian과 Big-Endian으로 나뉜다. Little-Endian은 데이터에서 가장 하위의 바이트(LSB)부터 기록된다. 그림1의 왼쪽에서 LSB인 0D를 메모리의 가장 낮은 번지수인 a에다가 두는 것을 볼 수 있다. 그러나 Big-Endian은 데이터에서 가장 상위의 바이트(MSB)부터 기록된다. 역시 그림1의 오른쪽을 보면 MSB인 0A를 메모리의 a에 위치시키고, LSB인 0D는 메모리의 a+3에 위치시킨다. 대부분의 시스템은 Little-Endian을 사용하지만, 네트워크 프로토콜이나 일부 프로세서에서는 Big-Endian을 사용하는 경우도 많다. MIPS Architecture은 기본적으로 Big-Endian을 사용한다.

2-3. MIPS Architecture

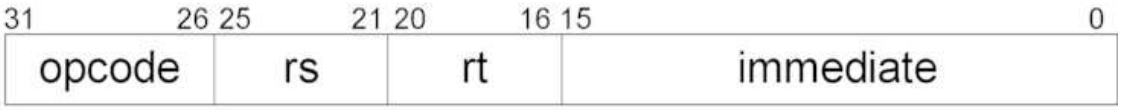
MIPS는 Microprocessor without Interlocked Pipeline Stages의 약자로, MIPS Technologies에서 개발한 RISC 계열의 32비트 ISA이다. RISC의 장점이 충실히 반영되어, 교착 상태가 없는 Pipline 구조 구현이 가능하기에 고성능을 발휘한다. 주로 Load와 Store 명령어로 데이터 전송을 처리하고 O32 ABI라는 32비트 CPU를 위한 레지스터 호출 규약을 사용한다. O32 ABI는 함수 호출 시 a0 ~ a3 레지스터에 4개의 매개변수(인자)를 전달한다. 만약 매개변수가 5개 이상이라면 메모리 내의 Stack을 통해 전달한다. 함수 호출의 반환값은 v0과 v1레지스터에 저장된다. 이 외에도 gp 레지스터는 Global(전역) 포인터 값, sp 레지스터는 메모리의 stack 포인터 값, fp 레지스터는 frame 포인터 값, ra 레지스터는 주소 반환값을 저장한다. 나머지 레지스터들은 주로 명령어 처리 시 일시적으로 사용한다.

CPU가 명령어들을 처리하는 과정은 제각각이고, 명령어마다 필요한 데이터들도 모두 다르다. 이에 MIPS는 총 3가지의 명령어 포맷을 지원한다. 레지스터 연산이 주가 되는 R-Type, 레지스터와 상수(Immediate)값의 연산이 주가 되는 I-Type, 프로그램의 제어 흐름을 변경(Jump)하는 J-Type으로 구성된다. 구조는 아래와 같다.

① R-Type

opcode (6bit) : 명령어의 종류를 구분하는 역할이지만 R-Type에선 값이 모두 0이다.
rs, rt (5bit) : 각각 1번째, 2번째 Source Register
rd (5bit) : 최종 연산 데이터가 저장될 Write Register
shamt (5bit) : Shift 연산 시에만 사용되며 얼마나 shift할지를 결정
function (6bit) : R-Type은 opcode가 모두 0이므로 function으로 명령어 종류를 구분

② I-Type



opcode (6bit) : 명령어의 종류를 구분
rs (5bit) : Source Register
rt (5bit) : 최종 연산 데이터가 저장될 Write Register
Immediate (16bit) : R-Type의 2번째 Source Register 역할을 대신하는 상수값

③ J-Type

opcode (6bit) : 명령어의 종류를 구분
Instr_index (26bit) : Jump할 Target의 메모리 주소를 의미

2-4. Sequential Execution

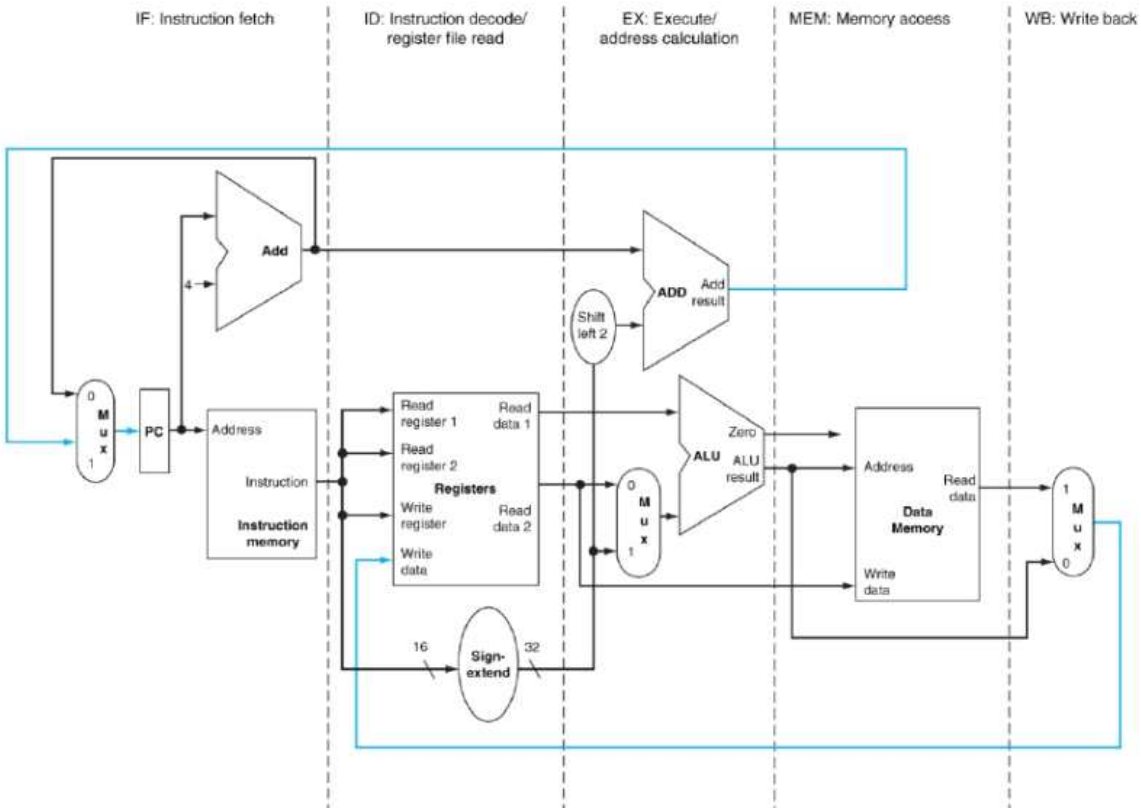


그림3. 명령어 처리의 5단계 Data Path

Single Cycle은 Sequential Execution 방식이다. 이는 메모리에 저장된 명령어를 순서대로 한 번에 한 개씩 실행하는 것을 의미한다. 전체적인 방식은 다음과 같다. 우선 PC를 통해 메모리에 저장되어 있는 명령어를 가져와 해석한다. 해석한 정보를 토대로 레지스터, ALU, 데이터 메모리가 서로 상호작용하며 명령어를 처리한다. 처리 후에는 다음 주소로 업데이트된 PC값으로 명령어를 가져오고 위의 과정을 반복한다. 일반적으로 PC는 PC+4값으로 업데이트 되고, Jump/Jar/Jr와 같은 Target Address로 Jump하거나, Branch 관련 명령어에서 Branch를 Taken한 경우에만 PC값이 예외적으로 업데이트된다. 이 Sequential Execution을 세분화하면 5가지 단계가 존재하며, 순서는 Fetch → Decode → Execute → Memory Access → Write Back이다. 일반적으로 Execute까지는 대부분의 처리 과정에서 필요하고, 명령어에 따라 Memory Access나 Write Back까지 진행한다.

① Fetch (Instruction Fetch, IF)

PC에서 명령어 주소를 가져오고 Instruction Memory에서 올바른 명령어를 찾아 Instruction Register(IR)에 저장하는 단계이다. IR에는 현재 처리할 32비트의 명령어가 존재하게 되고 PC는 다음 명령어 주소를 가리키게 된다. Instruction Memory는 프로세서가 실행해야할 명령어가 순차적으로 저장된 공간을 말한다.

② Decode (Instruction Decode, ID)

IR에 있는 명령어를 해석하는 단계이다. 32비트로 구성된 명령어를 Parsing하여 opcode와 레지스터 index, Immediate 값 등을 추출한다. 추출된 opcode는 Control Unit(CU)로 보내지고, CU는 명령어의 Format에 맞춰 Execute를 위한 Control Signal을 준비한다. 이때 Parsing된 정보는 32개x32bit 레지스터가 저장되어있는 Register File로 보내져 명령어 수행에 필요한 레지스터들이 결정된다.

③ Execute (EX)

해석한 명령어에 맞는 연산을 수행하는 과정으로, CU에서 제어한 신호에 맞춰 ALU 연산을 진행한다. R-Type은 레지스터와 레지스터의 연산이 이루어지는데, R-Type의 경우 opcode가 모두 0이므로 function code의 6bit에 따라 연산한다. I-Type은 레지스터와 상수의 연산을 진행한다.

④ Memory Access (MEM)

Data Memory에 접근하여 데이터를 Load하거나 Store하는 단계이다. Data Memory에서 읽어온 데이터를 Write Register에 전달해 주거나(LW), 레지스터에 저장된 값을 메모리에 작성한다(SW).

⑤ Write Back (WB)

EX나 MEM에서 계산한 결과값을 Write Register에 작성한다. Control Unit에서의 신호로 결과값 저장 유무를 제어한다.

3. Implementation

3-1. Data Path

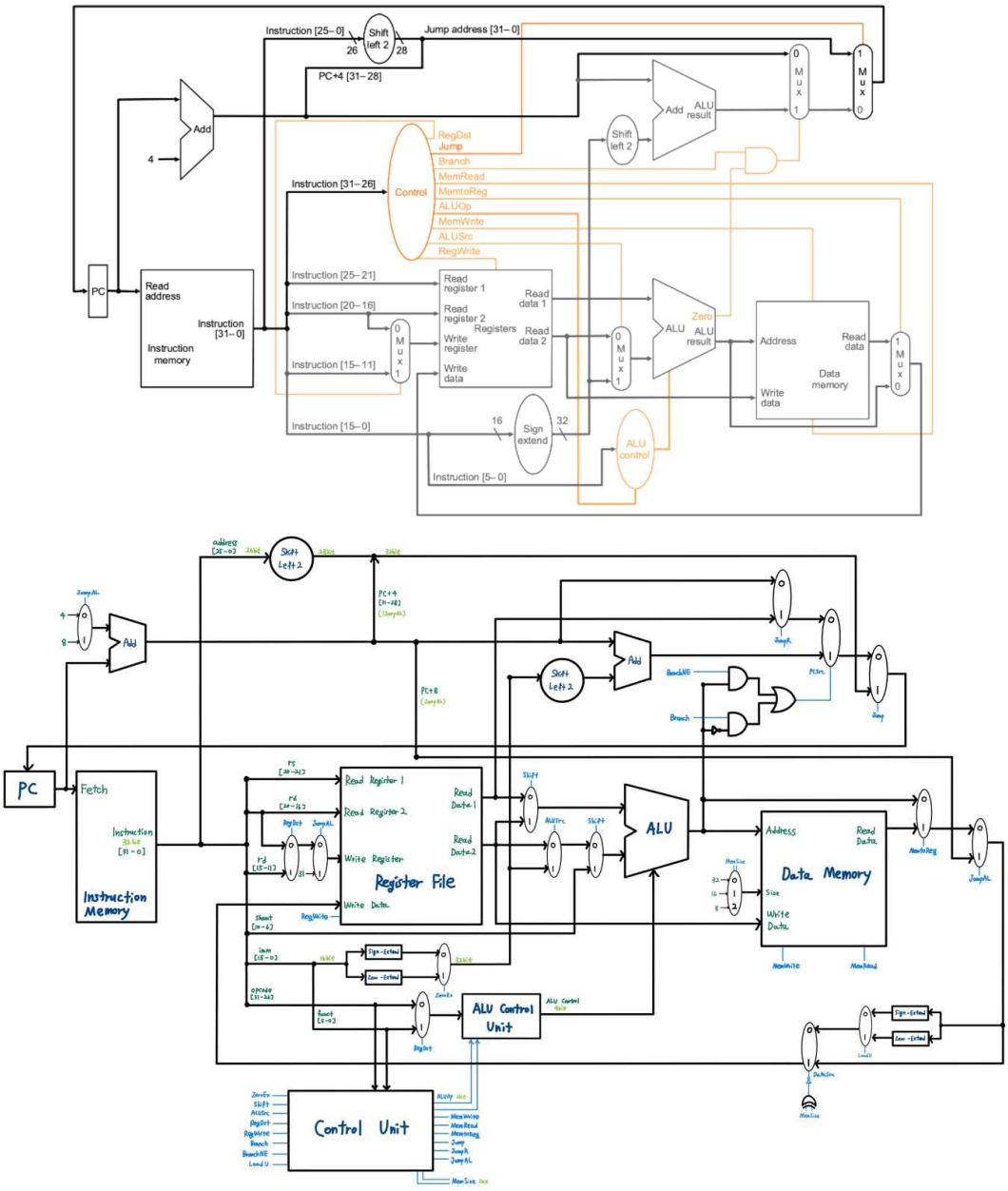


그림4. Single-Cycle 전체 Data Path (위는 기존, 아래는 직접 작성)

이번에 구현한 Single-Cycle MIPS Emulator의 전체 Data Path이다. 기존의 Data Path에는 몇몇 명령어들 (JR, JALR, Byte, Halfword, BNE)등의 실행을 지원하지 못했기에 이에 맞춰 Control Signal과 특정 Logic Circuit을 추가하였다. 이렇게 설계한 Data Path의 논리 흐름에 따라 에뮬레이터 코드를 그대로 구현했다.

3-2. base.h (Definitions)

```c
/* Define : J-Type Instruction Opcodes */
#define J 0x02
#define JAL 0x03

/* Define : Other Numbers */
#define MAX_SIZE 0x1000000

/* Instruction Formats */
typedef struct Instruction{
    uint8_t opcode;
    uint8_t rs;
    uint8_t rt;
    uint8_t rd;
    uint8_t shamt;
    uint8_t funct;
    uint32_t imm;
    uint32_t address;
}Instruction;


/* Control Unit Components */
// Functions
void CU_init(); // 신호들의 값을 0으로 초기화해준다.
void CU_setting(uint8_t opcode, uint8_t funct); // opcode에 따라 CU의 신호들을 설정한다.
// Variables
bool Jump;
```

그림5. base.h의 일부

우선 가장 많이 쓰이는 레지스터 번호나 명령어의 opcode, funct필드 등에 사용되는 숫자들을 매크로로 지정했다. 이런 방식을 사용하면 각각의 c코드에서 숫자를 쓰는것 대신 실제 이름으로 작성이 가능하다. 가령 0x08은 ADDI, 31은 ra로 쓸 수 있고 이로 인해 switch문에서의 가독성이 훨씬 좋아지게 된다. 또한 Single-Cycle 내의 모든 Logic Curcuit들이 한 Cycle에서 하나의 명령어를 같이 처리하므로, 대부분의 변수들을 전역 변수로 정의하여 구현했다. 지역변수로 명령어를 처리하면 하나의 함수에 너무 많은 매개변수가 들어갈 수 있어 코드가 복잡해지기 때문이다. 여기서 변수의 타입은 stdint.h를 include하여 unsigned int를 가장 많이 사용했다. 일반적으로 int를 사용하는 것 보다 훨씬 세밀하게 구현할 수 있기 때문이다. 대표적으로 Instruction Memory나 Data Memory에는 한 주소에 최대 8비트가 저장되므로 8비트의 자료형인 uint8_t를 사용하여 배열을 선언해 주었다. 이 두 배열은 정말 큰 메모리를 차지하는데, 만약 uint8_t가 아닌 4 Byte의 int를 사용했다면 메모리 낭비가 상당했을 것이다. 명령어 구조체 역시 각각의 크기에 최대한 맞게 선언해 주었고, 여기서 immediate는 후에 필요한 sign/zero extend를 고려하여 32bit로 했다.

base.h의 뒷부분에 각 코드에서 주로 사용하는 전역변수와 Function들이 묶어 정리되어있다. Function들과 Control Signal 변수들은 이후 파트별로 자세히 설명할 예정이고, 여기서는 뒤의 내용을 이해하기 위한 전역변수들을 소개할 것이다.

변수 이름	데이터 타입	설명
cycle	int	에뮬레이터에서 실행한 총 Cycle의 수
InsMemory	uint8_t[MAX_SIZE]	명령어들이 저장되는 배열
PC	uint32_t	Program Counter
IR	uint32_t	Instruction Register
ins	Instruction*	Decode하여 만들어낸 명령어 구조체
R_cnt	int	R Format 명령어 수
I_cnt	int	I Format 명령어 수
J_cnt	int	J Format 명령어 수
Brc_cnt	int	Branch 성공한 총 횟수
Mem_cnt	int	메모리에 접근한 총 횟수
Register	uint32_t[32]	r0~r31의 상태를 저장하는 배열
ReadData1	uint32_t	첫번째 source 레지스터
ReadData2	uint32_t	두번째 source 레지스터
size	int	메모리에서 load/store할 bit의 크기
DataMemory	uint8_t[MAX_SIZE]	데이터가 저장되는 배열
ReadData	uint32_t	메모리에서 load한 데이터
outData	uint32_t	DM에서 나오는 1차적인 write data
ALUResult	uint32_t	ALU 연산결과값
ERROR	bool	에러 감지 관련 변수, 감지 시 true

.

3-3. IM.c (Instruction Memory)

```c
#include "base.h"

void IM_init(){
    FILE* f = fopen("input1.bin", "r");
    uint8_t buffer; // 파일입출력용 버퍼
    int site = 0;
    int flag = 0;
    if (f == NULL){ // 예외처리 : 파일이 없을 경우
        printf("FILE ERROR\n");
        return;
    }
    // Big Endian 구조
    while(fread(&buffer, sizeof(uint8_t), 1, f) != 0){
        InsMemory[site + flag] = buffer & 0xff;
        flag++;
        if (flag > 3){
            flag = 0;
            site += 4;
        }
    }
    fclose(f);
}

uint32_t IM_fetch(uint32_t address){
    return (InsMemory[address] << 24) | (InsMemory[address + 1] << 16) |
           (InsMemory[address + 2] << 8) | (InsMemory[address + 3]);
}
```

그림6. IM.c

IM.c에서는 Instruction Memory 배열에 명령어들을 8비트씩 끊어 저장하는 IM_init()과 PC값에 맞춰 명령어를 찾는 IM_fetch()가 있다. IM_init()에서는 binary파일을 r모드로 열고 fread로 binary파일의 내용이 없을때까지 데이터를 읽는다. 읽어온 데이터는 buffer에 저장하고 0xff와 연산하여 8비트씩 끊어서 Instruction Memory 배열에 저장해준다. site와 flag 변수는 Big-Endian 형식을 지켜 저장하기 위해 사용하였다. 또한 파일이 존재하지 않을 경우 배열에 저장하는 과정으로 넘어가지 않고 바로 함수를 종료시킨다. IM_fetch()에서는 PC값을 매개변수로 받아오고 비트연산과 or을 이용하여 32비트의 완전한 명령어 데이터로 복구한 다음 이를 전역변수 IR로 return한다.

3-4. RF.c (Register File)

```c
#include "base.h"

void RF_init(){
    for (int i = 0; i < sp; i++){
        Register[i] = 0;
    }
    // r29 ~ r31 문제에 맞춰서 값 재설정
    Register[sp] = 0x1000000;
    Register[fp] = 0;
    Register[ra] = 0xffffffff;
}

void RF_read(uint8_t rs, uint8_t rt){
    ReadData1 = Register[rs];
    ReadData2 = Register[rt];
}

void RF_write(bool RegWrite, uint8_t write_r, uint32_t write_data){
    if (RegWrite == 1){
        Register[write_r] = write_data;
    }
}
```

그림7. RF.c

RF.c에서는 초기화 관련 함수인 RF_init()과 ReadData1, 2값을 설정해주는 RF_read(), Write Back 결과를 받아 Write Register에 작성하는 RF_write()가 있다. RF_init()에서는 sp와 ra를 제외한 모든 레지스터를 0으로 초기화하며, sp와 ra는 조건에 맞게 각각 0x1000000와 0xffffffff로 초기화해준다. RF_read()에서는 rs와 rt로 설정된 레지스터의 인덱스를 사용하여 해당 값으로 ReadData들을 정해준다. RF_write()에서는 RegWrite 신호가 1일때만 실행되며 Write Register의 index값인 write_r과 write data를 매개변수로 받아 올바른 위치의 레지스터에 데이터를 작성해준다. write_r와 write_data가 정해지는 조건은 3-9에서 후술한다.

3-5. CU.c (Control Unit)

```
void CU_setting(uint8_t opcode, uint8_t funct){
    Jump = (opcode == J) || (opcode == JAL);
    JumpR = ((opcode == 0) && (funct == JR)) || ((opcode == 0) && (funct == JALR));
    JumpAL = (opcode == JAL) || ((opcode == 0) && (funct == JALR));
    Branch = (opcode == BEQ);
    BranchNE = (opcode == BNE);
    ZeroEx = (opcode == ANDI) || (opcode == ORI);
    Shift = (opcode == 0) && ((funct == SLL) || (funct == SRL));
    RegDst = (opcode == 0);
    RegWrite = (opcode != BEQ) && (opcode != BNE) && (opcode != J) && (opcode != SB) &&
               (opcode != SH) && (opcode != SW) && (funct != JR);
    MemWrite = (opcode == SB) || (opcode == SH) || (opcode == SW);
    MemRead = (opcode == LB) || (opcode == LBU) || (opcode == LH) || (opcode == LHU) || (opcode == LW);
    MemtoReg = (opcode == LB) || (opcode == LBU) || (opcode == LH) || (opcode == LHU) || (opcode == LW);
    LoadU = (opcode == LBU) || (opcode == LHU);
    ALUSrc = (opcode != 0) && (opcode != BEQ) && (opcode != BNE);
```

그림8. CU.c의 CU_setting Function (1 bit control signals)

```
/* ALUOp */
if ((opcode == LB) || (opcode == LBU) || (opcode == LH) || (opcode == LHU) || (opcode == LW) ||
(opcode == SB) || (opcode == SH) || (opcode == SW)){
    ALUOp = 0b00;
}
else if ((opcode == BEQ) || (opcode == BNE)){
    ALUOp = 0b01;
}
else if ((opcode == 0)){
    ALUOp = 0b10;
}
else{
    ALUOp = 0b11;
}

/* Data Memory를 위한 MemSize 신호 */
if ((opcode == LH) || (opcode == LHU) || (opcode == SH)){
    MemSize = 0b01;
}
else if ((opcode == LB) || (opcode == LBU) || (opcode == SB)){
    MemSize = 0b10;
}
else{
    MemSize = 0b00;
}
```

그림9. CU.c의 CU_setting Function (2 bit control signals)

CU.c에는 Control Unit과 관련된 기능들이 모여있다. 사진에는 없지만 CU_init()는 모든 신호를 0으로 초기화해주는 Function이다. CU_setting()은 해석한 명령어에서 opcode 혹은 funct 필드에 따라 Control Signal들의 값을 설정해준다. 본래 Control Unit에는 opcode 6bit만 입력되는것이 기존 Data Path지만, 이번 프로젝트에서는 JR/JALR/Shift의 구분을 위해 funct 6bit가 필요했기에 funct 필드도 input으로 설정하였다. 이 신호들로 후의 Execute, Memory Access, Write Back단계의 진행을 결정하게 된다. 다음 13~14쪽에서 Control Signal의 Rules와 Table을 확인할 수 있다.

신호	0	1
Jump	-	26bit address 필드로 PC계산
JumpR	-	rs register로 PC계산
JumpAL	PC값 계산 Source : PC+4	Write Reg=r31, Write Data=PC+8
Branch	-	ALUResult==0 ? 16bit imm 필드로 PC계산 : PC=PC+4
BranchNE	-	ALUResult!=0 ? 16bit imm 필드로 PC계산 : PC=PC+4
ZeroEx	Sign-Extend (ID)	Zero-Extend (ID)
Shift	ALU op1=Read Data1 ALU op2=Read Data2	ALU op1=Read Data2 ALU op2=shamt field
RegDst	Write Reg=rt ALUCU input=opcode	Write Reg=rd ALUCU input=funct
RegWrite	Write Back X	Write Back O
MemWrite	Memory Write X	Memory Write O
MemRead	Memory Read X	Memory Read O
MemtoReg	Write Data=ALU Result	Write Data=DM Read Data
LoadU	Sign-Extend (WB)	Zero-Extend (WB)
ALUSrc	ALU op2=Read Data2	ALU op2=Extended 16bit imm

Control Signal Rules (1 bit)

신호	00	01	10	11
ALUOp	+ 연산	- 연산	R-Type 명령어 중 구분하여 연산	I-Type 명령어 중 구분하여 연산
MemSize	32bit Mem접근	16bit Mem접근	8bit Mem접근	

Control Signal Rules (2 bit)

신호	BEQ	BNE	J	JR	JAL	JALR	SLL/SRL	나머지 R-Type
Jump	0	0	1	0	1	0	0	0
JumpR	0	0	0	1	0	1	0	0
JumpAL	0	0	0	0	1	1	0	0
Branch	1	0	0	0	0	0	0	0
BranchNE	0	1	0	0	0	0	0	0
ZeroEx	0	0	0	0	0	0	0	0
Shift	0	0	0	0	0	0	1	0
RegDst	0	0	0	1	0	1	1	1
RegWrite	0	0	0	0	1	1	1	1
MemWrite	0	0	0	0	0	0	0	0
MemRead	0	0	0	0	0	0	0	0
MemtoReg	0	0	0	0	0	0	0	0
LoadU	0	0	0	0	0	0	0	0
ALUSrc	0	0	0	0	0	0	0	0
ALUOp	01	01	x	x	x	x	10	10
MemSize	x	x	x	x	x	x	x	x

Control Signal Table (R-Type, J-Type, BEQ, BNE)

신호	SW	SH	SB	LW	LH	LHU	LB	LBU	AND /ORI	나머지 I-Type
Jump	0	0	0	0	0	0	0	0	0	0
JumpR	0	0	0	0	0	0	0	0	0	0
JumpAL	0	0	0	0	0	0	0	0	0	0
Branch	0	0	0	0	0	0	0	0	0	0
BranchNE	0	0	0	0	0	0	0	0	0	0
ZeroEx	0	0	0	0	0	0	0	0	1	0
Shift	0	0	0	0	0	0	0	0	0	0
RegDst	0	0	0	0	0	0	0	0	0	0
RegWrite	0	0	0	1	1	1	1	1	1	1
MemWrite	1	1	1	0	0	0	0	0	0	0
MemRead	0	0	0	1	1	1	1	1	0	0
MemtoReg	0	0	0	1	1	1	1	1	0	0
LoadU	0	0	0	0	0	1	0	1	0	0
ALUSrc	1	1	1	1	1	1	1	1	1	1
ALUOp	00	00	00	00	00	00	00	00	11	11
MemSize	00	01	10	00	01	01	10	10	x	x

Control Signal Table (I-Type)

3-6. ALUCU.c & ALU.c

```
uint8_t ALUCU_control(uint8_t ALUOp, uint8_t flagCode){
    switch(ALUOp){
        case 0b00:
            return 0b0010;
        case 0b01:
            return 0b0110;
        case 0b10:
            switch(flagCode){
                case AND:
                    return 0b0000;
                case OR:
                    return 0b0001;
                case NOR:
                    return 0b1100;
                case ADD:
                case ADDU:
                    return 0b0010;
                case SUB:
                case SUBU:
                    return 0b0110;
                case SLT:
                case SLTU:
                    return 0b0111;
                case SLL:
                    return 0b1010;
                case SRL:
                    return 0b1110;
            }
        case 0b11:
            switch(flagCode){
                case ANDI:
                    return 0b0000;
                case ORI:
                    return 0b0001;
                case ADDI:
                case ADDIU:
                    return 0b0010;
                case SLTI:
                case SLTIU:
                    return 0b0111;
                case LUI:
                    return 0b1000;
```

그림10. ALUCU.c

```
uint32_t ALU_calculate(uint8_t control, uint32_t op1, uint32_t op2){
    switch (control){
        case 0b0000:
            return op1 & op2;
        case 0b0001:
            return op1 | op2;
        case 0b1100:
            return ~(op1 | op2);
        case 0b0010:
            return op1 + op2;
        case 0b0110:
            return op1 + (~op2 + 1);
        case 0b0111:
            return op1 < op2;
        case 0b1000:
            return op2 << 16;
        case 0b1010:
            return op1 << op2;
        case 0b1110:
            return op1 >> op2;
```

그림11. ALU.c

ALUCU.c와 ALU.c는 CU에서 만들어낸 2bit의 ALUOp 신호로 ALU 연산할 결정하고 수행하는 코드이다. 우선 ALUCU.c에서 ALUOp와 flagCode를 매개변수로 받아온다. 이때 flagCode는 RegDst신호로 R타입인지 아닌지 확인하여, R타입이면 funct필드를, R타입이 아니면 opcode가 flagCode로 전송된다. ALUOp의 4가지 경우의 수에 따라 4bit의 ALU Control Signal를 만들어 ALU.c로 보낸다. ALU.c에서는 ALUCU.c로부터 받은 신호에 맞게 input으로 들어온 두 데이터를 연산하여 main.c의 ALUResult로 return해준다. CU에서 만들어내는 ALUOp 값과 ALUCU.c, ALU.c에서 사용되는 switch문을 정리하면 아래 표와 같다.

Type	명령어	ALUOp	ALU Control Signal	대응 ALU 연산
I	load or store 관련	00	0010	a + b
	branch 관련	01	0110	a - b
R	AND	10	0000	a & b
	OR		0001	a | b
	NOR		1100	~(a | b)
	ADD, ADDU		0010	a + b
	SUB, SUBU		0110	a - b
	SLT, SLTU		0111	a < b
	SLL		1010	a << b
	SRL		1110	a >> b
I	ANDI	11	0000	a & b
	ORI		0001	a | b
	ADDI, ADDIU		0010	a + b
	SLTI, SLTIU		0111	a < b
	LUI		1000	a << 16

(ALU Control Signal 구현 시 load/store관련, branch 관련, ADD, SUB, AND, OR, SLT는 교수님의 강의자료와 똑같이 구현했으며, 나머지 신호들은 타 자료들을 참고하거나 임의로 만들었습니다.)

.

3-7. DM.c (Data Memory)

```c
#include "base.h"

void DM_init(){
    for (int i = 0; i < MAX_SIZE; i++){
        DataMemory[i] = 0;
    }
}

void DM_operation(bool MemRead, bool MemWrite, uint32_t address, uint32_t writeData, int size){
    if (MemRead == 1){ // LW일 경우
        uint32_t tmp = 0;
        for (int i = 0; i < (size / 8); i++){
            tmp = tmp << 8;
            tmp = tmp | DataMemory[address + i];
        }
        ReadData = tmp;
    }
    else if (MemWrite == 1){ // SW일 경우
        for (int i = (size / 8) - 1; i >= 0; i--){
            DataMemory[address + i] = writeData & 0xff; // 오른쪽 1바이트만 저장
            writeData = writeData >> 8; // right bit shift
        }

    }
}
```

그림12. DM.c

Data Memory는 load나 store같이 메모리 접근 시에만 사용하는 부분으로 DM_init()과 DM_operation()이 있다. DM_init()은 DataMemory 배열을 0으로 초기화 해주는, 앞서 언급했던 초기화 함수들과 같은 역할이다. Data Memory 코드의 핵심은 DM_operation()이다. 우선 Control Signal 중 MemRead와 MemWrite에 따라 operation을 결정한다. Load 관련 명령어일 경우 MemRead가 1이 된다. 따라서 이에 맞게 매개변수로 받은 address를 이용하여 데이터 메모리에서 데이터를 찾는다. 불러오는 데이터의 크기는 매개변수로 받아온 size에 따라 달라진다. 이 값은 byte/halfword/word에 따라 각각 8/16/32 값을 가진다. 패딩을 완료한 데이터는 최종적으로 ReadData에 저장된다. 반대로 Store 관련 명령어일 경우 MemWrite가 1이 된다. 이 경우 가져온 writeData를 Big-Endian에 맞게 저장해야 하므로, writeData를 0xff와 &연산하고 8bit를 right shift하는 방식을 size만큼 반복하여 메모리의 주소 끝에서부터 저장한다.

3-8. Other Logic Units

```
uint32_t signExtend(uint16_t imm){
    if (imm & 0x8000){
        return (0xffff0000 | imm);
    }
    else if ((imm & 0x0080) && !(imm & 0xff00)){
        return(0xffffff00 | imm);
    }
    else{
        if (!(imm & 0xff00)){
            return(0x000000ff & imm);
        }
        return (0x0000ffff & imm);
    }
}

uint32_t zeroExtend(uint16_t imm){
    if (!(imm & 0xff00)){
        return(0x000000ff & imm);
    }
    return (0x0000ffff & imm);
}

uint32_t PCAdder(uint32_t PC){
    return PC + MUX(JumpAL, 4, 8);
}

uint32_t branchAdder(uint32_t PC, uint32_t imm){
    return PC + 4 + (imm << 2);
}

uint32_t jumpMaker(uint32_t PC, uint32_t address){
    return ((PC + 4) & 0xf0000000) | (address << 2);
}
uint32_t MUX(bool sign, uint32_t input1, uint32_t input2){
    return (sign ? input2 : input1);
}

uint32_t MUX3(uint8_t sign, uint32_t input1, uint32_t input2, uint32_t input3){
    switch(sign){
        case 0b00:
            return input1;
        case 0b01:
            return input2;
        case 0b10:
            return input3;
    }
}
```

그림13. LU.c에 존재하는 function들

앞서 설명한 5가지 단계와 관련한 function 말고도 이 5가지를 완전히 수행하기 위해 보조 역할을 하는 function들이 LU.c에 존재한다. Extend function은 Zero와 Sign 2가지가 존재한다. Extend는 16비트의 immediate값을 32비트로 확장하거나, Byte/Halfword로 메모리에서 load한 데이터를 32비트로 확장할 때 사용한다. Zero Extend는 MIPS Green Sheet 기준으로 ANDI, ORI에서 사용하고 LBU, LHU처럼 Unsigned을 메모리에서 불러올 경우로 부호 상관 없이 0으로 확장해준다. 나머지는 대부분 Sign Extend로 진행하며 부호를 확인하고 1이나 0으로 확장해준다. 이 2가지 함수는 비트 연산을 활용한다. Sign Extend로 예시를 들자면, 우선 0x8000과 immediate를 &연산하면 최상위비트 MSB가 1인지 0인지 알 수 있다. 그리고 0xff00과 &연산을 하면 현재 데이터의 크기가 16bit인지 8bit인지 확인이 가능하다. 이렇게 2가지의 연산 결과에 맞게 크기 확장을 해주게 된다. PCAdder은 초기 PC값을 계산해주는 Circuit으로 JAL, JALR인 경우에만 PC에 8을 더해주고 나머지는 4를 더해준다. branchAdder는 Branch 관련 명령어 시 16bit의 immdiate값을 이용하여 PC값을 계산해준다. jumpMaker은 J Type 명령어일 때 26bit의 address값을 2만큼 left shift해주고, PC+4의 상위 4비트와 패딩하여 최종 Jump할 주소값을 계산해준다. 마지막으로 MUX는 Multiplexer로 2:1과 3:1이 있다. Single-Cycle 내에는 Control Signal에 따른 데이터 스위칭이 매우 중요하므로 MUX 사용은 필수이다.

3-9. main.c

```
int main(){
    initialize();
    // PC값이 0xffff:ffff될때까지 Simulator 구동
    while (PC != 0xffffffff){
        cycle++;
        IF();
        ID();
        EX();
        MEM();
        WB();
        if (ERROR == true) break;
        // Show current state.
        print_cycle();
    }
    print_result();
    return 0;
}
```

그림14. main()

3-2부터 3-8까지 각 단계별로 사용하는 변수나 function들의 원리를 설명했으므로 이제 main.c에서 전체적인 코드의 흐름을 소개한다. 1번의 반복이 한 Cycle이며 Cycle 내에서 Fetch → Decode → Execute → Memory Access → Write Back 과정을 모두 완료한다. 이 과정은 PC가 0xffff:ffff가 될때까지 반복한다. 앞으로 initialize부터 Loop의 마지막인 PC 업데이트까지 main.c의 단계별로 Instruction Stage가 어떻게 진행되는지 알아볼 것이다.

```
void initialize(){
    printf("\n=================================================================================\n");
    printf("=======================  Single Cycle MIPS Emulator  =======================\n");
    printf("=================================================================================\n");
    ERROR = false;
    cycle = 0;
    R_cnt = 0;
    I_cnt = 0;
    J_cnt = 0;
    Brc_cnt = 0;
    Mem_cnt = 0;
    PC = 0x00000000;
    IR = 0x00000000;
    ins = (Instruction*)malloc(sizeof(Instruction));
    IM_init();
    RF_init();
    DM_init();
    CU_init();
}
```

그림15. initialize()

에뮬레이터를 실행하기 전 모든 기본 값을 초기화해주고 각 코드 별 init()함수를 실행한다. 3-3에서 설명했듯이 IM_init()에서만 예외적으로 Component를 0으로 초기화하는 것이 아닌 binary파일에서 명령어를 읽어와서 InsMemory 배열에 저장한다. sp와 ra값 역시 0이 아닌 각자 고유의 값으로 초기화된다.

```
/* 1. Fetch : Instruction is moved from memory to CPU */
if (is_SIZE_INS_ERR()) ERROR = true; // SIZE_INS Error Check
IR = IM_fetch(PC);
print_fetch();
```

그림16. Stage1 : Fetch

초기화를 완료하면 본격적인 Loop가 시작된다. Loop에서 가장 먼저 실행되는 작업이 Fetch이다. Fetch전 cycle 계산을 위하여 1 증가시킨 다음, PC값과 IM_fetch() 함수로 Instruction Memory에서 명령어를 찾아 IR에 저장한다. Fetch가 완료되면 print_fetch()로 Fetch한 명령어 정보를 출력한다.

```
/* 2. Decode : Instruction decoded -> Identify the operation -> Registers usages */
ins->opcode = IR >> 26;
ins->rs = (IR & 0x03e00000) >> 21;
ins->rt = (IR & 0x001f0000) >> 16;
ins->rd = (IR & 0x0000f800) >> 11;
ins->shamt = (IR & 0x000007c0) >> 6;
ins->funct = IR & 0x0000003f;
ins->imm = IR & 0x0000ffff;
ins->address = IR & 0x03ffffff;
CU_setting(ins->opcode, ins->funct);
if (is_OPCODE_ERR()) ERROR = true; // OPCODE Error Check
RF_read(ins->rs, ins->rt);
print_decode();
```

그림17. Stage2 : Decode

이제 IR에 저장된 명령어를 해석한다. main.c에 내장되어있는 decode() 함수는 32비트의 명령어를 format에 맞게 명령어 구조체를 만들어준다. 이들은 모두 IR과 비트 연산으로 format에 알맞는 부분만 추출하여 저장한다. 명령어 구조체를 완성하면 이에 맞게 CU_setting()에서 Control Signal들을, RF_read()에서 ReadData을 설정해준다. decode가 끝나면 fetch와 비슷하게 print_decode()로 해석한 명령어의 Type 및 format 정보와 사용 예정인 레지스터들의 상태도 출력한다.

```
/* 3. Execute
ALU operation is performed and ALU result is produced */
ins->imm = MUX(ZeroEx, signExtend(ins->imm), zeroExtend(ins->imm));
ALUControl = ALUCU_control(ALUOp, MUX(RegDst, ins->opcode, ins->funct));
ALUResult = ALU_calculate(ALUControl, MUX(Shift, ReadData1, ReadData2),
                          MUX(Shift, MUX(ALUSrc, ReadData2, ins->imm), ins->shamt));
```

그림18. Stage3 : Execute

명령어 구조체의 component들로 ALU연산을 진행하는 단계이다. 우선 16bit의 immediate 값을 ZeroEx신호에 맞게 32bit로 Extend해주고, ALUControl 4bit 신호 역시 이번 단계에서 ALUCU_control()함수로 결정해준다. 모든 정보가 모였다면 ALU_calculate()로 ALU연산을 진행하여 나온 결과값을 ALUResult에 저장한다. 이때 여러 개의 MUX로 전송할 데이터를 스위칭해준다. ALU의 input 2개에 들어가는 값은 수많은 연산 상황에 따라 다르기 때문이다.

```
/* 4. Memory Access : Memory is accessed when load and store */
size = MUX3(MemSize, 32, 16, 8);
if (is_SIZE_DATA_ERR()) ERROR = true; // SIZE_DATA Error Check
DM_operation(MemRead, MemWrite, ALUResult, ReadData2, size);
```

그림19. Stage4 : Memory Access

load와 store 한정으로 필요한 단계이다. 3:1 MUX와 MemSize 신호를 활용하여 load나 store할 데이터의 크기를 정해주고, DM_operation()으로 Control Signal 상황별 Data Memory 접근을 해준다.

```
/* 5. Write Back : Destination registers are updated including PC */
outData = MUX(JumpAL, MUX(MemtoReg, ALUResult, ReadData), PCAdder(PC));
DataSrc = (MemSize == 0b00) || (MemSize == 0b11);
RF_write(RegWrite, MUX(JumpAL, MUX(RegDst, ins->rt, ins->rd), ra),
        MUX(DataSrc, MUX(LoadU, signExtend(outData), zeroExtend(outData)), outData));
```

그림20. Stage5 : Write Back

다음 Cycle로 넘어가기 전 마지막 단계이다. Control Signal에 따라 ALU 연산 결과나 Load한 데이터 중 필요한 것을 MUX로 스위칭하여 outData에 저장한다. 이 outData는 원본, sign-extend, zero-extend 세 갈래로 나뉘어 전송되는데, LoadU와 DataSrc 신호에 따라 셋 중 하나가 최종 선택되어 RF의 Write Data로 향한다. DataSrc 신호가 1이면 outData의 원본이 전송되고, 데이터의 크기가 32bit일때 MemSize가 00이고 DataSrc가 1이 된다. MemSize가 0b11인 경우는 없으며, 전체 회로도 기준으로 이 신호는 XNOR게이트로 결정되기 때문에 의미상 넣어주었다. Write Register 역시 Control Signal 상황 별로 MUX로 스위칭되어 결정되고 최종 선택된 Write Register에 Write Data를 저장해준다.

```
PCSrc = ((Branch && !ALUResult) || (BranchNE && ALUResult));
PC = MUX(Jump, MUX(PCSrc, MUX(JumpR, PCAdder(PC), ReadData1),
        branchAdder(PC, ins->imm)), jumpMaker(PC, ins->address));
```

그림21. PC 업데이트

원래 PC 업데이트는 Cycle이 시작되자마자 완료되는 것이 맞지만, 코드 구현 상 다음 PC값이 후반에 결정되야 하기 때문에 Loop의 가장 마지막에 위치하였다. Branch, BranchNE, ALUResult값에 따라 PCSrc가 정해지며, PCSrc, JumpR, Jump 신호에 따라 알맞는 다음 PC값이 정해진다. 앞의 신호들이 전부 0이라면 Jump를 하지 않는 상황이므로 정상적으로 PC+4가 다음 PC값으로 업데이트된다. PC값 업데이트까지 마치면 현재 Cycle의 실행 결과를 출력하고, 모든 Cycle이 종료되면 에뮬레이터의 최종 결과를 출력한다. (27p 4-3 참고)

3-10. Handle Exception

```c
if (f == NULL){ // 예외처리 : 파일이 없을 경우
    printf("FILE ERROR\n");
    return;
}
```

그림22. FILE ERROR Detection Part (in IM.c)

```c
int is_SIZE_INS_ERR(){
    if (PC > MAX_SIZE){
        printf("SIZE_INS ERROR\n");
        return 1;
    }
    return 0;
}

int is_SIZE_DATA_ERR(){
    if ((MemRead == 1 || MemWrite == 1) && (ALUResult > MAX_SIZE)){
        printf("SIZE_DATA ERROR\n");
        return 1;
    }
    return 0;
}

int is_OPCODE_ERR(){
    if ((ins->opcode) == 0){
        int check_R_arr[13] = {ADD, ADDU, AND, JR, JALR, NOR, OR,
                               SLT, SLTU, SLL, SRL, SUB, SUBU};
        for (int i = 0; i < 13; i++){
            if ((ins->funct) == check_R_arr[i]){
                return 0;
            }
        }
    }
    else{
        int check_I_J_arr[19] = {ADDI, ADDIU, ANDI, BEQ, BNE, LB, LBU, LH, LHU, LW,
                                 LUI, ORI, SLTI, SLTIU, SB, SH, SW, J, JAL};
        for (int i = 0; i < 19; i++){
            if ((ins->opcode) == check_I_J_arr[i]){
                return 0;
            }
        }
    }
    printf("OPCODE ERROR\n");
    return 1;
}
```

그림23. 3 Types ERROR Detection Part (in error.c)

이번 과제에서 처리한 에러는 아래의 4가지밖에 없는데, 이는 c코드를 기반으로 MIPS-Cross Compiler를 사용해서 binary파일로 변경하여 실행하는 에뮬레이터이기 때문에 현실적으로 발생 가능하면서 꼭 예외처리를 해야하는 에러만을 관리했다. 만약 사용자가 직접 Assembly 언어를 작성하는 방식의 에뮬레이터라면 처리해야할 에러가 훨씬 많았을 것이다. 먼저 Instruction Memory나 Data Memory에서 찾을 데이터가 없는 경우, 어차피 처음 initialize()에서 각 메모리의 배열 요소들을 모두 0으로 초기화해줬기 때문에 0이 반환된다. 그래서 따로 에러 처리를 해줄 필요가 없었다. 이 외에도 레지스터의 번호가 r0~r31이 아닌 다른 번호가 온 경우나, 명령어의 format이 잘못된 경우 등과 같은 Assembly 코드 작성 부주의에 의해 발생하는 에러들은 처리하지 않았다. 또한 이번 프로젝트 요구사항에서 Stack의 시작점이 0x1000000이었다. 그래서 Data Memory의 크기를 0x1000000으로 설정했고, Instruction Memory 역시 크기를 맞춰 0x1000000으로 설정했기에 PC가 0xffff:ffff가 되는 것은 단지 main의 while loop의 종료조건으로만 해석하였다. 또한 OPCODE ERROR 감지에는 opcode나 funct를 체크하는 특수 배열을 비교하여 체크한다. 확장성을 고려하면 매우 아쉬운 방법이지만, 일단 에러 감지를 위하여 이렇게 구현했다. 에러가 발생하면 ERROR 전역변수가 true로 바뀌며 main loop이 break된다.

에러 이름	감지 시점	설명
FILE ERROR	Initialize	입력 파일이 같은 디렉터리에 존재하지 않는 경우
SIZE_INS ERROR	Fetch	PC값이 0x0100:000을 초과하는 경우 (0xffff:ffff는 종료시점이므로 제외)
OPCODE ERROR	Decode	opcode(명령어)가 존재하지 않는 경우
SIZE_DATA ERROR	Memory Access	찾으려는 데이터의 주소가 0x0100:000을 초과하는 경우

4. Environment

4-1. Build Environment

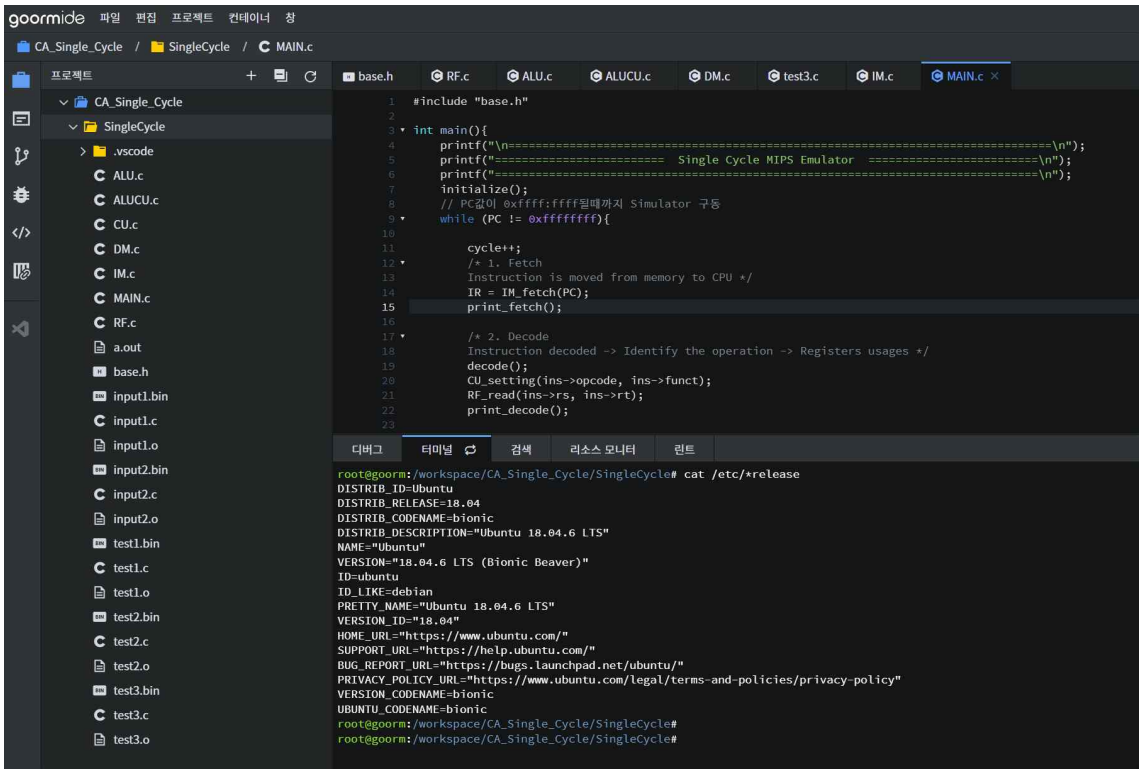


그림24. goormIDE 컨테이너 내의 Ubuntu(Linux) Environment

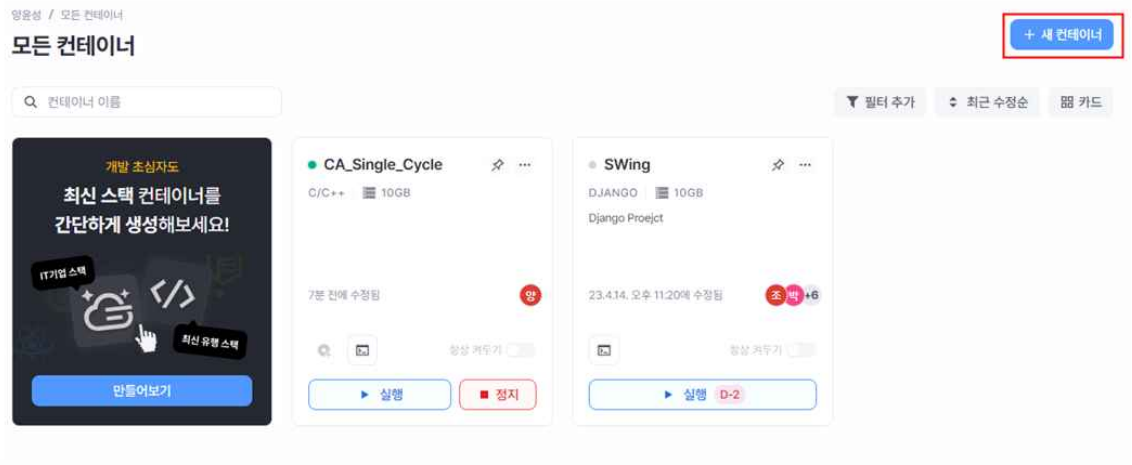
IDE : goorm Cloud IDE Service (https://www.goorm.io/dashboard)

OS : Ubuntu 18.04.6 LTS

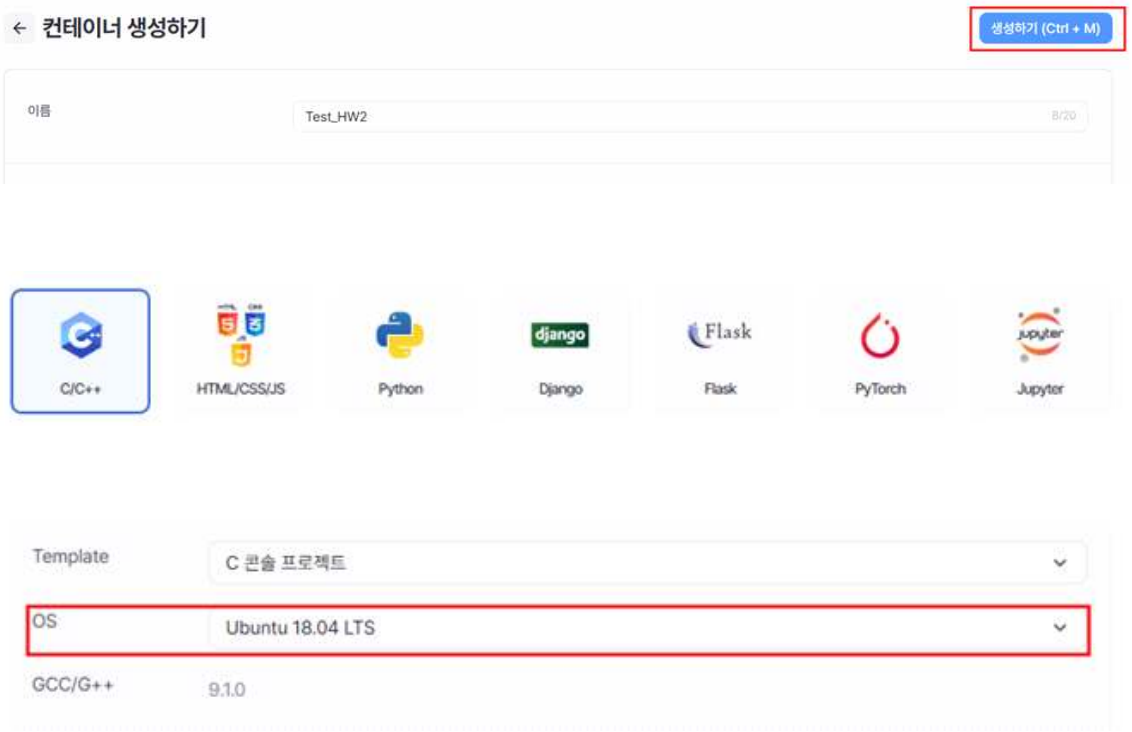
GCC Compiler (Ubuntu 9.1.0-2ubuntu2~18.04) 9.1.0

GoormIDE는 클라우드 기반의 통합 개발 환경이다. 웹 브라우저 상에서 작동하기 때문에 특정 운영 체제나 개발 환경에 구애받지 않고 자유로운 개발이 가능한 것이 특징이다. 컨테이너 기반의 work space를 제공하며, 각각의 컨테이너는 독립적인 가상 리눅스 환경 위에서 작동한다. 이 점이 이번 프로젝트를 GoormIDE에서 진행한 가장 큰 이유이다. 리눅스 환경을 기본 제공해 주므로 MIPS 크로스 컴파일러만 터미널에서 명령어로 따로 설치해 준다면, MIPS 컴파일과 구현한 Single-Cycle MIPS Emulator를 문제 없이 실행할 수 있다.

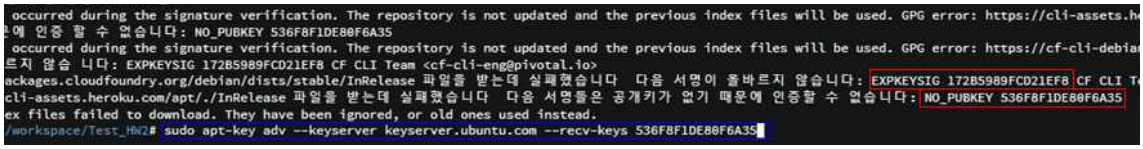
4-2. Compile



GoormIDE 홈페이지 (https://ide.goorm.io/) 에서 회원가입을 하고 접속하면 나의 대쉬보드로 자동 연결된다. 우선 오른쪽 위의 "새 컨테이너" 를 클릭한다.



컨테이너 생성하기 페이지가 나오면 프로젝트 이름을 입력한다. 생성하기 전에 스크롤을 아래로 내려서 소프트웨어 스택을 C/C++로, OS를 Ubuntu 18.04 LTS로 설정한 후에 생성하기를 누른다. 다른 옵션들은 기본 설정으로 해도 무방하다.

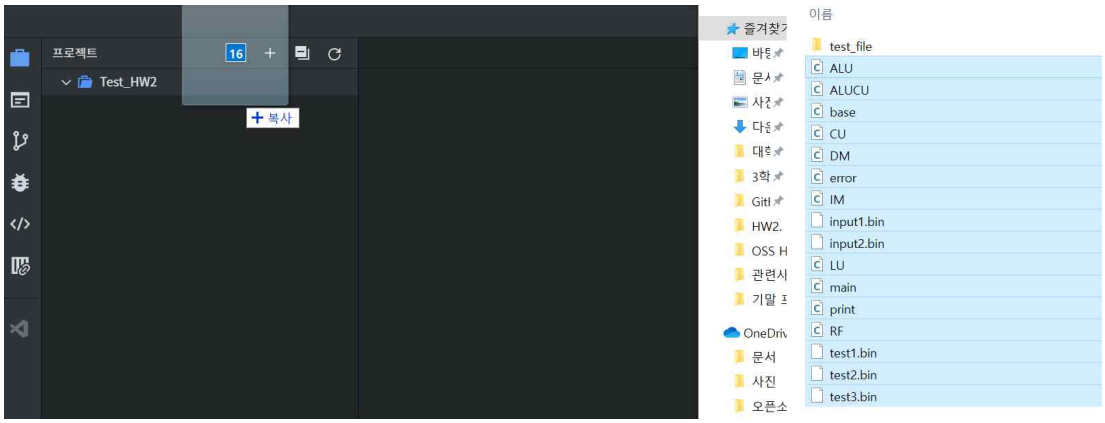


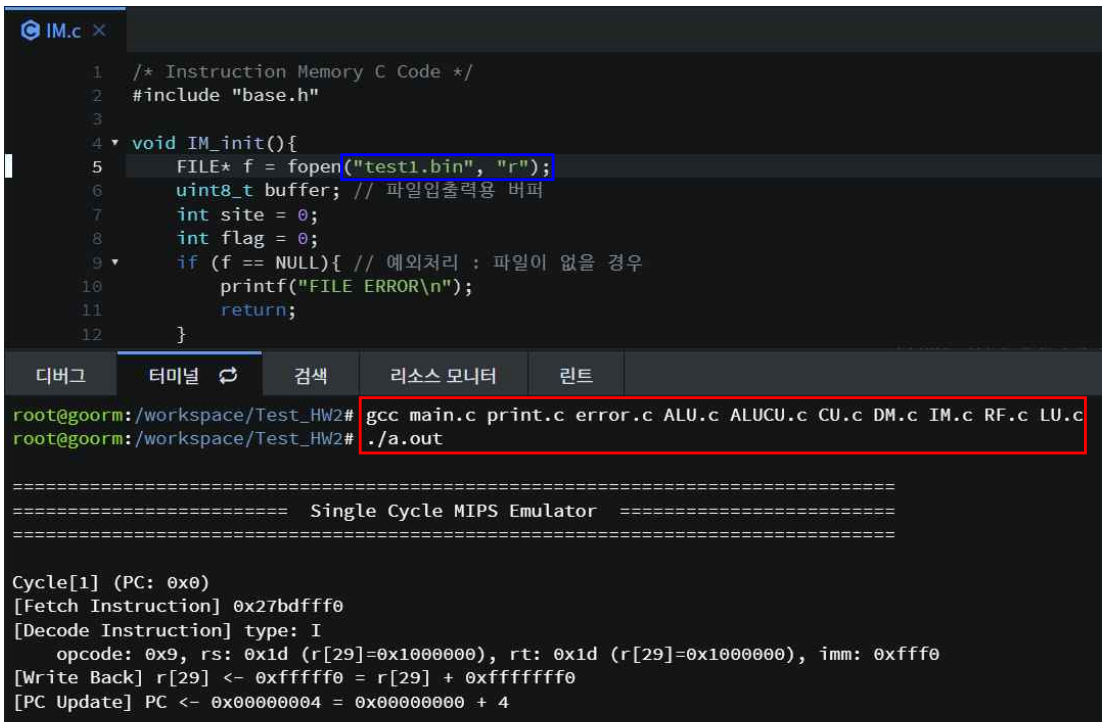
컨테이너가 생성되면 터미널을 열어서 **sudo apt-get update** 명령어를 입력한다. 그러나 위의 사진처럼 key 관련한 오류가 표시될 수 있다. 이는 사진 속 빨간색으로 강조한 부분처럼 공개키에 에러가 있는 경우이다. 아래의 명령어로 서버에서 2개의 key를 다시 가져오고 다시 update를 시도하면 정상적으로 실행된다.

sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys (key번호)

update 이후에는 아래의 명령어로 MIPS 크로스 컴파일러를 설치한다.

sudo apt-get -y install gcc-mips-linux-gnu

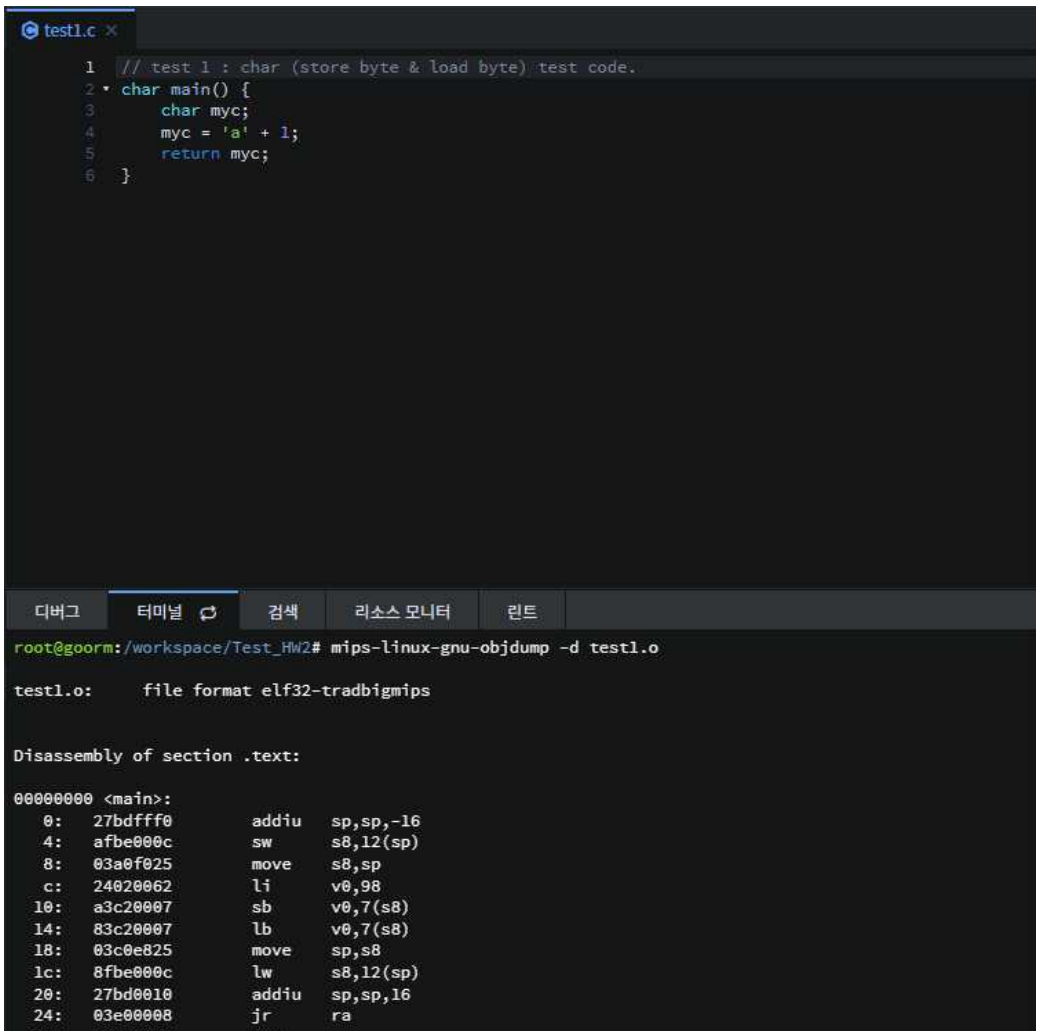




셋팅이 모두 완료되면 **HW2_32192530_양윤성_code**의 압축을 풀고 모든 파일들을 프로젝트 내에 드래그하여 추가한다. 파일에는 총 5가지의 기본 테스트 코드(input1, input2, test1, test2, test3)가 존재한다. 이 코드들 중 bin 파일로 하나를 선택하여 사진 속 파란색 부분처럼 IM.c의 5번째 줄에 fopen의 인수로 작성해준다. 이 5가지의 bin 파일들은 HxD로 Jump Address를 수정한 상태이므로 Jump 명령어가 정상 실행된다. 이제 다음 2줄의 명령어를 입력하면 에뮬레이터가 실행된다. 필요한 테스트 파일의 c코드와 object 파일은 test_file 폴더에서 찾아 사용하면 된다.

gcc main.c print.c error.c ALU.c ALUCU.c CU.c DM.c IM.c RF.c LU.c

./a.out



크로스 컴파일러가 정상적으로 설치되었기에 위와 같이 test_file 폴더에서 object 파일을 가져와 MIPS Assembly 코드도 확인할 수도 있다. 이 외의 다른 MIPS 관련 명령어 입력도 모두 가능하다.

4-3. Working Proofs

테스트 파일은 다음과 같으며, test1.bin 결과를 제외한 나머지 4가지 bin 파일의 테스트 결과는 너무 길어 Result만 남기고 모두 생략했다.

파일 이름	원본 코드	테스트 주 목적
test1.bin	test1.c	간단한 char 자료형 사용으로 load/store byte를 테스트
test2.bin	test2.c	다른 함수로 jump 여부 테스트
test3.bin	test3.c	fibonacci 수열 재귀 테스트
input1.bin	input1.c	0부터 9까지 더하는 for 반복문 테스트 (과제 테스트 코드)
input2.bin	input2.c	다른 함수로 jump 여부 테스트 (과제 테스트 코드)

```
================================================================================
=======================  Single Cycle MIPS Emulator  ===========================
================================================================================

Cycle[1] (PC: 0x0)
[Fetch Instruction] 0x27bdfff0
[Decode Instruction] type: I
    opcode: 0x9, rs: 0x1d (r[29]=0x1000000), rt: 0x1d (r[29]=0x1000000), imm: 0xfff0
[Write Back] r[29] <- 0xffffff0 = r[29] + 0xfffffff0
[PC Update] PC <- 0x00000004 = 0x00000000 + 4

--------------------------------------------------------------------------------

Cycle[2] (PC: 0x4)
[Fetch Instruction] 0xafbe000c
[Decode Instruction] type: I
    opcode: 0x2b, rs: 0x1d (r[29]=0xffffff0), rt: 0x1e (r[30]=0x0), imm: 0xc
[Store Word] Mem[0x00ffffffc] <- r[30] = 0x00000000
[PC Update] PC <- 0x00000008 = 0x00000004 + 4

--------------------------------------------------------------------------------

Cycle[3] (PC: 0x8)
[Fetch Instruction] 0x03a0f025
[Decode Instruction] type: R
    opcode: 0x0, rs: 0x1d (r[29]=0xffffff0), rt: 0x0 (r[0]=0x0), rd: 0x1e (r[30]=0x0), shamt: 0x0, funct: 0x25
[Write Back] r[30] <- 0xffffff0 = r[29] | r[0]
[PC Update] PC <- 0x0000000c = 0x00000008 + 4

--------------------------------------------------------------------------------

Cycle[4] (PC: 0xc)
[Fetch Instruction] 0x24020062
[Decode Instruction] type: I
    opcode: 0x9, rs: 0x0 (r[0]=0x0), rt: 0x2 (r[2]=0x0), imm: 0x62
[Write Back] r[2] <- 0x62 = r[0] + 0x00000062
[PC Update] PC <- 0x00000010 = 0x0000000c + 4

--------------------------------------------------------------------------------

Cycle[5] (PC: 0x10)
[Fetch Instruction] 0xa3c20007
[Decode Instruction] type: I
    opcode: 0x28, rs: 0x1e (r[30]=0xffffff0), rt: 0x2 (r[2]=0x62), imm: 0x7
[Store Byte] Mem[0x00fffff7] <- r[2] = 0x62
[PC Update] PC <- 0x00000014 = 0x00000010 + 4

--------------------------------------------------------------------------------
```

그림25. test1.bin 테스트 결과 (Cycle1~5)

```
Cycle[6] (PC: 0x14)
[Fetch Instruction] 0x83c20007
[Decode Instruction] type: I
    opcode: 0x20, rs: 0x1e (r[30]=0xfffff0), rt: 0x2 (r[2]=0x62), imm: 0x7
[Load Byte] r[2] <- Mem[0x00fffff7] = 0x62
[PC Update] PC <- 0x00000018 = 0x00000014 + 4

--------------------------------------------------------------------------------

Cycle[7] (PC: 0x18)
[Fetch Instruction] 0x03c0e825
[Decode Instruction] type: R
    opcode: 0x0, rs: 0x1e (r[30]=0xfffff0), rt: 0x0 (r[0]=0x0), rd: 0x1d (r[29]=0xfffff0), shamt: 0x0, funct: 0x25
[Write Back] r[29] <- 0xfffff0 = r[30] | r[0]
[PC Update] PC <- 0x0000001c = 0x00000018 + 4

--------------------------------------------------------------------------------

Cycle[8] (PC: 0x1c)
[Fetch Instruction] 0x8fbe000c
[Decode Instruction] type: I
    opcode: 0x23, rs: 0x1d (r[29]=0xfffff0), rt: 0x1e (r[30]=0xfffff0), imm: 0xc
[Load Word] r[30] <- Mem[0x00fffffc] = 0x00000000
[PC Update] PC <- 0x00000020 = 0x0000001c + 4

--------------------------------------------------------------------------------

Cycle[9] (PC: 0x20)
[Fetch Instruction] 0x27bd0010
[Decode Instruction] type: I
    opcode: 0x9, rs: 0x1d (r[29]=0xfffff0), rt: 0x1d (r[29]=0xfffff0), imm: 0x10
[Write Back] r[29] <- 0x1000000 = r[29] + 0x00000010
[PC Update] PC <- 0x00000024 = 0x00000020 + 4

--------------------------------------------------------------------------------

Cycle[10] (PC: 0x24)
[Fetch Instruction] 0x03e00008
[Decode Instruction] type: R
    opcode: 0x0, rs: 0x1f (r[31]=0xffffffff), rt: 0x0 (r[0]=0x0), rd: 0x0 (r[0]=0x0), shamt: 0x0, funct: 0x8
[PC Update] (Jump) PC <- 0xffffffff = r[31]

--------------------------------------------------------------------------------

*******************************   Result   *******************************

Return Value (r2): 98 (0x62)
Total Cycle: 10
Executed 'R' Instruction: 3
Executed 'I' Instruction: 7
Executed 'J' Instruction: 0
Number of Branch Taken: 0
Number of Memory Access Instruction: 4

***************************  End of Emulator!  ***************************
```

그림26. test1.bin 테스트 결과 (Cycle6~10, Result)

```
***********************************   Result   ***********************************

Return Value (r2): 201 (0xc9)
Total Cycle: 33
Executed 'R' Instruction: 7
Executed 'I' Instruction: 23
Executed 'J' Instruction: 1
Number of Branch Taken: 0
Number of Memory Access Instruction: 17

*******************************  End of Emulator!  *******************************
```

그림27. test2.bin 테스트 결과 (Result)

```
********************************   Result   ********************************

Return Value (r2): 55 (0x37)
Total Cycle: 2572
Executed 'R' Instruction: 546
Executed 'I' Instruction: 1645
Executed 'J' Instruction: 109
Number of Branch Taken: 109
Number of Memory Access Instruction: 988

*****************************  End of Emulator!  *****************************
```

그림28. test3.bin 테스트 결과 (Result)

```
********************************   Result   ********************************

Return Value (r2): 45 (0x2d)
Total Cycle: 146
Executed 'R' Instruction: 13
Executed 'I' Instruction: 101
Executed 'J' Instruction: 0
Number of Branch Taken: 11
Number of Memory Access Instruction: 66

*****************************  End of Emulator!  *****************************
```

그림29. input1.bin 테스트 결과 (Result)

```
********************************   Result   ********************************

Return Value (r2): 10 (0xa)
Total Cycle: 95
Executed 'R' Instruction: 24
Executed 'I' Instruction: 60
Executed 'J' Instruction: 4
Number of Branch Taken: 4
Number of Memory Access Instruction: 36

*****************************  End of Emulator!  *****************************
```

그림30. input2.bin 테스트 결과 (Result)

각 단계별 사이클 출력문에서는 레지스터들이 어떻게 변화하는지 확인할 수 있다. 1번째 줄에는 현재 Cycle 턴수와 PC값, 2번째 줄에는 어떤 명령어를 Fetch했는지를 보여준다. 3~4번째 줄에는 Fetch한 명령어의 포맷에 맞춰 정보를 표시하는데, 이때 사용할 예정인 레지스터들의 상태도 보여준다. 5번째줄 부터는 명령어에 따른 실행 결과와 PC값의 업데이트를 확인할 수 있다. 모든 사이클이 종료되면 가장 아래에 Return Value인 r2(v0)값과 함께 결과 창이 나온다. 여기서는 전체 명령어를 수행하는데 소요된 Cycle의 수와 각 명령어 포맷별 실행 횟수 (R, I, J), Branch Taken 수와 메모리 접근 횟수를 알려준다.

5. Lesson

첫번째 프로젝트인 Simple Calculator은 단순히 문자열을 나누어 레지스터와 상수를 연산하는 프로그램이었지만, 이번 프로젝트는 Simple Calculator의 개념을 확장한 프로그램이다. 자신만의 Single-Cycle Data Path를 설계하고 이를 바탕으로 문자열이 아닌, 실제 binary 파일을 사용하는 MIPS Emulator을 구현하는 것이었다. 처음에는 주요 명령어들부터 실행이 가능하도록 간단하게 프로그램을 만든 뒤, 남은 명령어들을 순차적으로 추가하는 방식으로 에뮬레이터를 완성했다. 많은 내용을 한번에 구현했으면 복잡한 코드로 인해 힘들었겠지만, 명령어 별 Data Path를 분석하며 차근차근 구현을 하다 보니 Data Path에 대한 깊은 이해를 바탕으로 성공적인 프로젝트 완성을 할 수 있었다. 또한 저번에 비해 프로젝트 규모가 훨씬 커지면서, 코드 아키텍쳐의 중요성 역시 다시 한번 체감할 수 있었다. 이번에는 바로 코딩을 시작한 것이 아닌, Single-Cycle에 대한 어느정도의 연구를 하고 Data Path 초안 회로도까지 설계한 후 코딩을 시작했기에 수월한 프로젝트 진행이 가능했다. 다만 Control Signal의 Case를 직접 생각하며 Table을 작성한 점과, ALU Control Signal 역시 임의로 신호값을 설정한 다음 ALU에서 모든 연산을 다 하게끔 구현을 한 점이 매우 아쉬웠다. Control Signal에 대한 카르노맵을 직접 그려 보다 정확한 Case를 도출하고, 실제 ALU의 회로도를 바탕으로 Emulator을 구현했으면 논리적인 허점이 없는 좋은 프로그램이 됐을 것 같다. 또한 너무 많은 Control Signal이 들어갔다고 생각한다. 다음에는 최대한 신호 수를 줄여서 복잡하지 않은 구조를 만들어보고 싶다. Single-Cycle을 공부하며 한 번에 한 개의 명령어밖에 처리를 못한다는 단점을 알았는데, 이를 Pipelining 방식으로 어떻게 해결하는지에 대한 궁금해 졌다. 이에 다음 프로젝트 구현을 통해 알아보고 싶다는 생각이 들었다.