Report

HW#1 Simple Calculator

Computer Architecture

Dpt. of Mobile System Engineering
32192530 양윤성

2023.04.07.

1. Introduction

컴퓨터구조 과목에서 진행한 첫 프로젝트인 Simple Calculator은 단순한 계산기가 아닌, ISA 방식으로 명령어를 문자열로 하나씩 받아 연산을 처리한다. 또한 전체적으로 폰 노이만 구조를 기반으로 하여 메모리를 사용하고, 명령 처리를 하는 파트가 존재한다. Simple Calculator는 레지스터의 연산에 초점이 맞춰져 있다. 레지스터는 r0부터 r9까지 총 10가지가 존재하며, 초기값은 0이고 명령어에 따라 다양한 16진수 값을 가질 수 있다. +, -, *, / 와 같은 이항 연산을 이용한 사칙연산은 물론 연산의 보조 역할을 하는 LW, MOV(Move), JMP(Jump), BEQ(Branch Equal), BNE(Branch Not Equal), SLT(Set Less Than)와 특정 레지스터를 출력(SW) 및 모든 레지스터를 초기화(SLT) 해주는 다양한 명령어들이 존재한다. 이 프로그램은 크게 read, decode, execute 3구간으로 나눌 수 있다. read에서 "input.txt"라는 명령 입력 파일을 읽어오고 한 줄씩 string으로 메모리에 저장한다. 메모리에 저장된 명령어를 모두 완료하기 전까지, 첫번째줄부터 decode로 명령을 분석하고 execute로 명령을 실행하는 과정을 반복한다.

이번 레포트에서는 프로그램을 이해하기 위한 배경지식을 먼저 소개하고 구현한 Simple Calculator의 상세한 과정과 원리를 설명한다. 추가로 프로젝트를 위한 개발 환경과 컴파일 과정에 대한 설명을 한 뒤 구현을 하며 생각했던 과정 및 개인적으로 배웠던 점에 대해 말할 것이다.

2. Background

앞서 말했듯이 Simple Calculator은 ISA 방식을 사용하고 폰 노이만 구조를 기반으로 한다고 했다. 여기서 ISA는 Instruction Set Architecture로 명령어 집합 구조를 말한다. 프로세서가 이해할 수 있는 명령어 집합과 그 명령어를 실행하는 방식을 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. ISA는 일반적으로 다양하고 복잡한 명령어를 가진 CISC 아키텍쳐와 종류는 적지만 실행 속도와 효율에 초점을 맞춘 RISC 아키텍쳐로 구분된다. 우리가 앞으로 컴퓨터 구조에서 배워나갈 MIPS는 RISC 계열의 ISA이다.

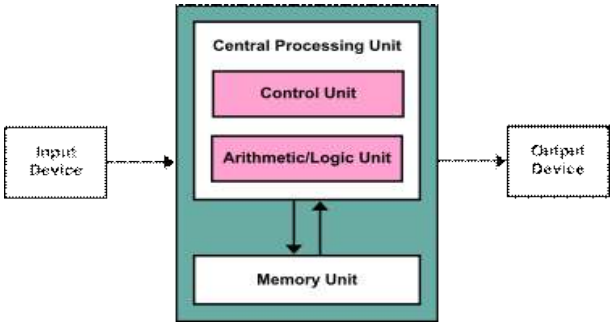

그림1. 폰 노이만 구조

폰 노이만 구조는 컴퓨터 구조의 한 종류로, 현재 사용하는 대부분 컴퓨터들의 가장 일반적인 구조 중 하나이다. 이 구조는 중앙 처리 장치인 CPU가 컴퓨터의 기본 구성 요소를 주앙 집중적으로 관리하는 방식으로 구성 요소에는 메모리와 입출력장치 등이 있다. 이 요소들은 서로 버스(bus)라는 통신선으로 연결되어 있어 데이터를 주고받을 수 있다. CPU는 프로그램에서 진행할 작업을 명령어로 정의하고 이러한 작업에 필요한 데이터들을 순차적으로 처리한다. CPU 내에는 PC(Program Counter)라는 레지스터가 존재하는데, PC가 다음에 실행할 명령어의 주소를 기억하고 있어 명령어들이 차례대로 실행 가능하게 한다. 메모리는 명령어와 데이터를 저장하는 역할을 하고, 입출력장치는 사용자와 시스템 간의 상호작용이 가능하도록 한다. 이전에는 컴퓨터에 다른 작업을 할당할 때 하드웨어를 재배치했어야 하지만, 폰 노이만 구조는 프로그램만 교체하면 끝이기에 범용성에서 큰 장점이 있다.

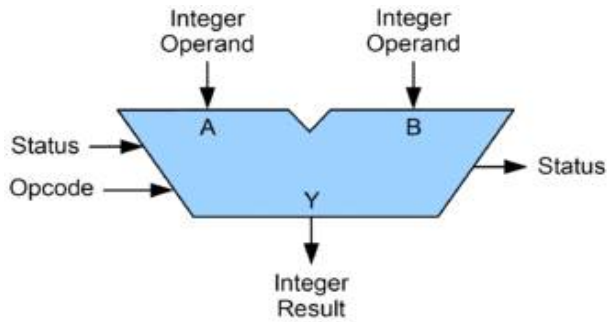

그림2. ALU (산술논리연산장치)

CPU 내에는 연산을 수행하는 장치인 ALU가 존재하는데, 주로 레지스터와 메모리의 데이터를 읽어서 명령어에 맞게 연산을 수행하고 결과를 레지스터나 메모리에 저장한다. 컴퓨터 시스템에서 연산 처리 속도를 높이는데 결정적인 역할을 하는 장치로, ALU의 연산 속도와 정확도는 CPU 성능에 큰 영향을 미친다. 이번에 구현한 Simple Calculator가 메모리에서 string 한 줄을 불러오고 명령어에 따라 연산을 한 뒤 레지스터에 저장하는 방식인데, 이 점이 ALU와 비슷하다.

3. Implementation

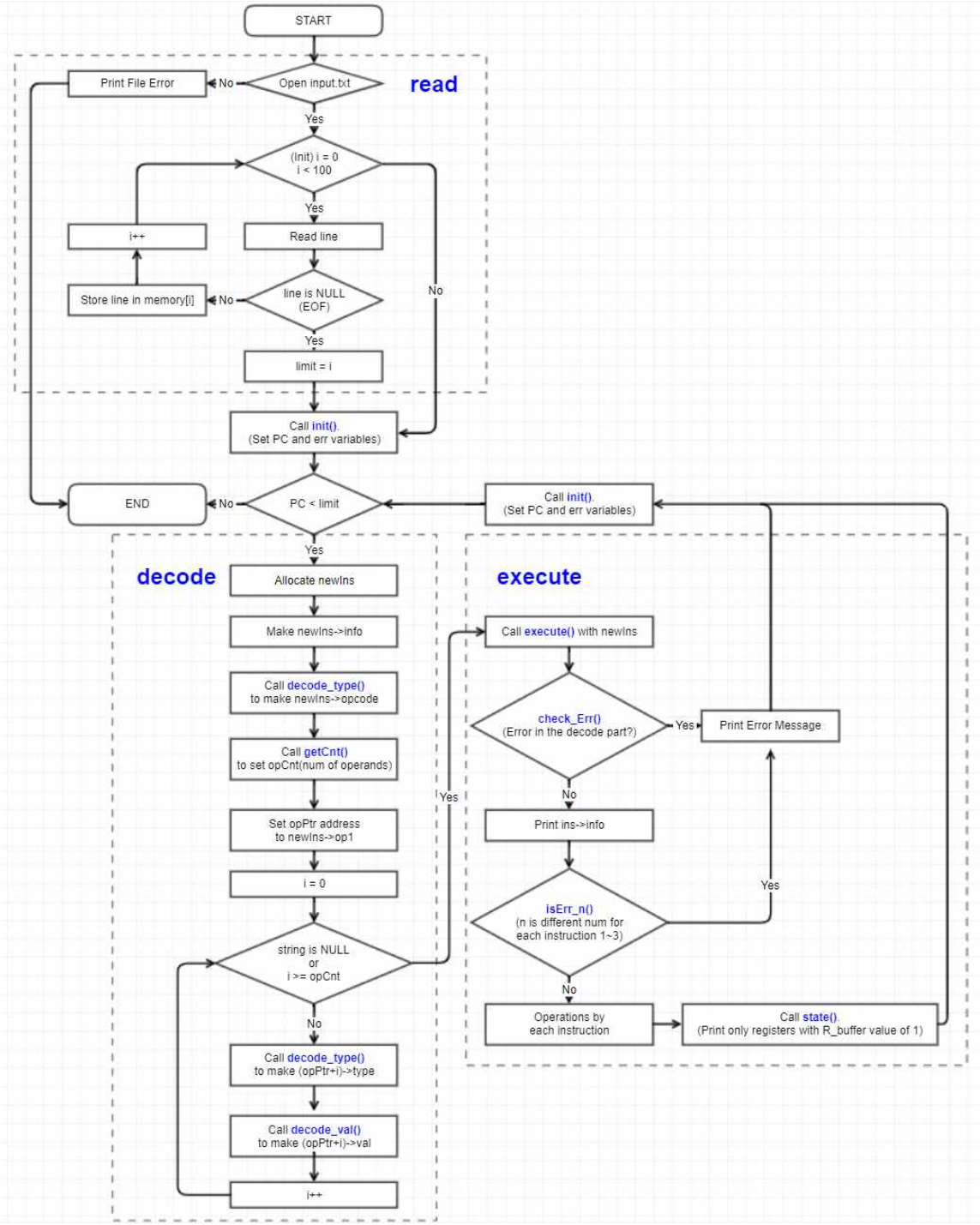

그림3. Simple Calculator 구조도 (Flow Chart)

```
printf("\n***** Start Program *****\n");
read();
init();
while (PC <= limit){
    ins = decode();
    execute(ins);
    init();
}
printf("\n***** End Program *****\n");
```

그림4. main function (구조체 할당/해제 및 print문 일부 제외)

이 장에서는 Introduction에서 간략하게 언급한 프로그램의 원리를 보다 자세하게 다룰 것이다. 코드의 흐름은 그림 3의 구조도와 같으며, input.txt의 존재를 확인하는 부분이 read이다. 파일이 존재하면 맨 윗줄부터 EOF까지의 모든 명령들을 메모리에 저장하고 init 함수로 PC와 에러 변수의 값을 설정해 준다. 에러 변수는 decode에서 감지하는 변수들로, false면 에러가 없고 true면 에러가 발생했다는 의미이다. init 후 PC가 limit(명령어 수)보다 크지 않을 때 까지 decode와 execute를 반복해준다. 명령어를 하나씩 실행 후에는 다음 명령어를 위해 init함수를 호출해주는데, 이때 업데이트되는 PC는 2가지 경우에 따라 값이 달라진다. 일반적인 경우에는 PC가 1씩 증가하고, JMP명령을 수행하거나 혹은 BEQ와 BNE 명령에서 조건을 만족할 경우엔 지정된 명령어 주소에서 1을 뺀 값이 PC에 우선 저장된 후에 PC가 1 증가한다. 원래 경우에는 명령어 4바이트가 1바이트씩 나눠서 저장되는 방식이라 PC가 4씩 증가하지만, 이번 프로젝트는 Simple Calculator이므로 이 부분은 고려하지 않고 1씩 증가하게 구현했다.

```
/* Operand and Instruction Formats */
typedef struct Operand{
    uint32_t type;
    int val;
}Operand;

typedef struct Instruction{
    uint32_t opcode;
    Operand op1;
    Operand op2;
    Operand op3;
    char info[MAX_LINE];
}Instruction;
```

```
/* Variables */
int limit; // 명령어 수
int PC = 0;
int R[10] = { 0, }; // r0 ~ r9 레지스터
int R_Buffer[10] = { 0, };
char memory[MAX_STORE][MAX_LINE];
bool isErr_NAME; // Name Error 신호
bool isErr_FEW; // Few Error 신호
bool isErr_MORE; // More Error 신호
FILE* fp; // 입력 파일
```

그림5. 명령어 구조체와 전역변수들

우선 전역변수는 명령어의 수를 저장하는 limit와 명령어 실행의 기준점이 되는 PC, 명령어를 저장할 char타입의 2차원 배열이 있다. 10개의 레지스터는 10의 index를 가진 int타입의 배열로 지정해 주었고, 레지스터 버퍼는 활성화된 레지스터들만 상태를 출력하기 위해 선언해 주었다. 가령 R_Buffer[0]이 1이고, R_buffer[2]가 0이라면, r0은 활성화된 상태이고, r2는 비활성화된 상태이다. 활성화의 기준은 연산결과가 1번이라도 저장된 적이 있다면 1, 없으면 0이다.

Instruction 구조체는 execute를 위해 명령어를 쪼개서 저장한 것으로, decode에서 명령어(opcode)와 명령어에 따른 피연산자를 나눈다. 이때 피연산자 또한 구조체인데, 피연산자에는 피연산자의 타입과 값에 대한 변수가 있다. opcode와 피연산자의 타입은 uint32_t타입으로 지정되어 있는데. 이는 바로 아래에서 설명한다.

```
/* Define : Opcode Macro Numbers */
#define ADD 0x414444
#define SUB 0x535542
#define MUL 0x4d554c
#define DIV 0x444956
#define MOV 0x4d4f56
#define LW 0x004c57
#define SW 0x005357
#define RST 0x525354
#define JMP 0x4a4d50
#define BEQ 0x424551
#define BNE 0x424e45
#define SLT 0x534c54
```

```
/* Define : Operand Type */
#define r0 0x7230
#define r1 0x7231
#define r2 0x7232
#define r3 0x7233
#define r4 0x7234
#define r5 0x7235
#define r6 0x7236
#define r7 0x7237
#define r8 0x7238
#define r9 0x7239
#define imm 0x7240
#define SWOp 0x7241
```

그림6. 명령어와 레지스터의 매크로

이번 과제에 가장 차별화를 둔 점으로, 모든 명령어 및 레지스터를 상수화하여 구현하였다. decode에서 문자열을 자른 다음 각각 잘라진 문자열을 아스키코드 연산을 통하여 변환 후에 구조체에 저장한다. 매크로로 지정해 준 상수의 기준과 변환 연산 방법은 decode_type 함수에서 후술한다. 이런 식으로 문자열이 아닌 숫자로 코드를 구현할 시 레지스터 관리와 에러 처리가 훨씬 수월해지며, 코드의 가시성도 증가한다. 명령어를 문자열 자체로 구분하면 switch-case문을 사용하지 못하기에 케이스 구분을 if-else문과 strcmp로 문자열을 비교한다. 그러나 strcmp는 내부에서 for문을 사용하기에 시간 복잡도가 증가하여 코드의 속도가 느려지고, 과다한 else if 사용으로 가시성이 떨어진다. 이러한 단점을 매크로가 단번에 해결하기에 채택하여 사용하였다.

```c
void read(){
    char tmp[MAX_LINE];
    char* write;
    fp = fopen("input.txt", "r");
    if (fp == NULL){
        printf("FILE ERROR : Input file not found.\n");
        return;
    }
    for (int i = 0; i < MAX_STORE; i++){
        write = fgets(tmp, sizeof(tmp), fp);
        if (write == NULL){
            limit = i;
            break;
        }
        if (write[strlen(write)-1] == '\n'){
            write[strlen(write)-1] = '\0';
        }
        strcpy(memory[i], write);
    }
    fclose(fp);
}
```

그림7. read function

Read Function에서는 File 타입 포인터를 사용해 input.txt를 읽기 모드로 연다. 파일이 없다면 명령어 자체를 읽어오지 못하므로 에러 출력 후에 프로그램이 종료된다. 문자열을 읽을 때는 write 변수에 임시로 저장하고, 개행 문자가 있다면 NULL 문자로 바꿔주며 memory 배열에 순차적으로 복사하여 저장한다. 더 이상 받아올 문자열이 없다면 limit 변수에 총 명령어의 수를 저장하고 읽어오는 작업을 마친다.

```c
Instruction* decode(){
    Instruction* newIns = (Instruction*)malloc(sizeof(Instruction));
    int i = 0;
    int opCnt;
    char* ptr;
    Operand* opPtr;
    strcpy(newIns->info, memory[PC-1]);
    // opcode
    ptr = strtok(memory[PC-1], " ");
    newIns->opcode = decode_type(ptr);
    opCnt = getCnt(newIns->opcode);
    // operand
    opPtr = &(newIns->op1);
    while (1){
        ptr = strtok(NULL, " ");
        if (ptr == NULL){
            if (i < opCnt) isErr_FEW = true;
            break;
        }
        if (i >= opCnt){
            isErr_MORE = true;
            break;
        }
        (opPtr + i)->type = decode_type(ptr);
        (opPtr + i)->val = decode_val((opPtr + i)->type, ptr);
        i++;
    }
    return newIns;
}
```

그림8. decode function

Decode Function은 명령어를 해석해서 실행을 위한 구조체를 만들어주는 단계로 Simple Calculator의 중심을 담당하고 있다. 각 명령 구조체에는 info라는 문자열 변수가 들어가 있는데, 이는 사용자가 적은 명령어를 출력하기 위함이다. 따라서 decode 전 알맞은 메모리 위치에서 문자열을 strcpy로 info에 복사한다. 이후에는 메모리에서 찾은 문자열을 strtok(공백 기준)으로 잘라가며 명령어와 피연산자를 구조체에 저장한다. 이때 명령어 opcode는 decode_type 함수 내에서 변환 후에 결과를 반환받아 사용한다. 피연산자를 변환하는 부분은 구조체 포인터를 사용하였는데 이는 명령어마다 피연산자의 수가 다르고 중복되는 코드 작성을 피하기 위함이다. 피연산자 필요량은 getCnt함수로 받아와 opCnt에 저장하고 이 함수에서 명령어도 올바르게 작성하였는지 확인한다. 만약 명령어가 틀린 경우거나 피연산자가 필요량보다 적거나 많다면, 이와 관련된 에러 변수를 true로 바꿔서 execute시에 실행하지 못하도록 방지한다. 피연산자의 타입도 opcode와 마찬가지로 decode_type 함수로 매크로 상수를 반환받고, decode_val 함수로 피연산자의 값도 저장해 준다.

```
int decode_val(uint32_t flag, char* str){
    if ((r0 <= flag) && (flag <= r9)){ // Case Register
        return R[flag - r0];
    }
    else if (flag == imm){ // Case Imm
        return strtol(str, NULL, 16);
    }
    else{
        return -1;
    }
}
```

그림9. decode_val function

Decode_Val Function은 피연산자의 타입과 문자열을 매개변수로 입력받는다. 피연산자의 타입에 따라 다른 값을 반환해 주는데, 타입이 레지스터일 경우 (r0 : 0x7230 ~ r9 : 0x7239) index에 맞는 레지스터 배열에 있는 값을 저장한다. index는 레지스터 타입 상수에서 r0(0x7230)을 빼주면 0부터 9까지의 범위가 된다. 만약 타입이 imm(0x7240)일 경우에는 매개변수로 받아온 문자열을 16진수로 바꿔서 반환한다. 여기서 strtol은 문자열을 정수로 변환하는 함수로 3번째 매개변수에 진법을 넣어 원하는 진수로 변환 가능하다. Simple Calculator에서는 16진수가 필요하여 16을 작성하였다. 레지스터도 상수도 아닌 타입엔 -1를 입력하여 없는 값임을 나타낸다. STDOUT는 값이 존재하지 않으므로 따로 구분할 필요가 없다.

```
uint32_t decode_type(char* str){
    uint32_t result = 0;
    char ch = str[0];
    int idx = 0;
    int STDOUT_flag = 0;
    while (ch != '\0'){
        if ((idx == 2) && (result == strImm)){
            return imm;
        }
        if ((idx == 4) && (result == STDO)){
            STDOUT_flag = 1;
        }
        result = result << 8;
        result = result | ch;
        ch = str[++idx];
    }
    if ((STDOUT_flag == 1) && (result == DOUT)){
        return SWOp;
    }
    return result;
}
```

그림10. decode_type function

Decode_Type Function은 문자열인 명령어를 매크로로 지정해 준 상수로 변환해 주고 피연산자의 타입 역시 피연산자의 형태를 보고 결정하여 매크로 상수로 변환해 준다. 매개변수로 받은 문자열을 앞에서부터 문자 하나씩 아스키코드로 변환하여 연산한다. 연산은 bit shift 방식으로, 문자를 변환할 때 마다 result를 8비트만큼 left shift 해준 후 or 연산을 해준다. 그렇게 나온 결과가 6쪽 그림 6에서의 매크로 상수가 된다. 예를 들어, ADD는 16진수 기준 아스키코드가 각각 0x41, 0x44, 0x44이다. ADD를 만약 decode_type 함수의 매개변수로 전달한다면, 최종적으로는 (0x410000 | 0x4400 | 0x44)를 연산하여 result가 0x414444가 되고 이 값을 반환한다. 레지스터 역시 같은 방식이다. r0은 아스키코드가 각각 0x72, 0x30이기에 변환 결과 0x7230이 된다. 여기서 명령어는 대문자로, 레지스터와 상수는 소문자인 r과 0x로, SW의 "stdout"은 대문자인 STDOUT으로 입력해줘야 한다. 입력 양식을 지키지 않으면 이후에 execute에서 에러를 출력한다. 한편 while문의 상위 2개 if문과 하단의 if문은 피연산자의 타입과 관련된 것이다. 1번째 if문은 문자열이 0x로 시작하는 경우, 즉 앞에서부터 문자를 2개 확인했을 시 0x에 대응하는 아스키코드인 0x3078이면 피연산자 타입을 imm로 지정해준다. 나머지 if문 2개는 SW명령어에서의 STDOUT 판별로, 앞의 4자리가 STDO(0x5354444f), 뒤의 4자리가 DOUT(0x444f5554)인지 확인하여 맞으면 피연산자 타입을 SWOp로 지정해 준다.

```
void execute(Instruction* ins){
    printf("\n0x%x : %s\n", PC, ins->info);
    if (check_Err()) return;
    int idx = (ins->opcode != RST) ? ins->op1.type - r0 : -1;
    switch(ins->opcode){
        case ADD: // op1 = op2 + op3
            if (!isErr_3(ins->opcode, ins->op1.type, ins->op2.type, ins->op3.type)){
                R[idx] = ins->op2.val + ins->op3.val;
                R_Buffer[idx] = 1;
                state();
            }
            break;
```

그림11. execute function

위의 과정에서 생성한 구조체로 Execute Function에서 실행한다. 본격적인 execute 전, decode 단계에서 감지한 에러를 check_Err 함수로 체크하고 is_Err 함수로 피연산자 오타를 한 번 더 체크한다. 만약 에러가 있다면 상황에 맞는 에러를 출력한 뒤, 실행하지 않고 바로 다음 명령어로 넘어간다. 또한 연산 전에는 DST의 레지스터 인덱스(idx)를 구해주는데, RST 명령은 op1이 존재하지 않으므로 삼항연산자를 사용해 준다. 인덱스 저장 후에 switch-case문으로 명령어별 연산을 수행한다. 연산 결과를 DST 레지스터에 저장하고 DST 레지스터의 버퍼를 1로 변경한다. 상태 출력은 명령어에 맞는 연산이 끝난 뒤에 출력한다. 명령어별 피연산자 종류와 명령어 설명은 다음과 같다.

명령어	op1	op2	op3	설명
ADD	r	r	r	R[idx] = op2.val + op3.val
SUB	r	r	r	R[idx] = op2.val - op3.val
MUL	r	r	r	R[idx] = op2.val * op3.val
DIV	r	r	r	R[idx] = op2.val / op3.val
MOV	r	r/imm	x	R[idx] = op2.val
LW	r	imm	x	R[idx] = op2.val
SW	r	STDOUT	x	Print "R[idx] = op1.val (STDOUT)"
RST	x	x	x	R[0~9] = 0, R_Buffer[0~9] = 0
JMP	imm	x	x	PC = op1.val - 1
BEQ	r/imm	r/imm	imm	(op1.val == op2.val) ? (PC = op1.val – 1) :
BNE	r/imm	r/imm	imm	(op1.val != op2.val) ? (PC = op1.val – 1) :
SLT	r	r/imm	r/imm	(op1.val < op2.val) ? R[idx] = 1 : R[idx] = 0

idx : op1.type – r0(0x7230)으로 0~9사이의 값 / **val** : 피연산자 구조체의 int val;

```
bool isErr_2(uint32_t opcode, uint32_t op1_type, uint32_t op2_type){
    switch(opcode){
        case MOV: // MOV r r/imm
            if ((isReg(op1_type)) && ((isReg(op2_type)) || (op2_type == imm))) return false;
            break;
        case LW: // LW r imm
            if ((isReg(op1_type)) && (op2_type == imm)) return false;
            break;
        case SW: // SW r STDOUT
            if ((isReg(op1_type)) && (op2_type == SWOp)) return false;
            break;
    }
    printf("INSTRUCTION ERROR - Some invalid operands\n");
    return true;
}
```

그림12. isErr_(n) function

is_Err Function은 decode 단계에서 감지하지 못한 피연산자의 오류를 찾아내는 함수로, 피연산자 개수를 기준으로 1, 2, 3으로 나뉘어져 매개변수의 수를 다르게 받으며 명령어별로 맞춰 호출한다. 명령어마다 들어가야 하는 피연산자의 종류가 다르기 때문에 is_Err 내부에서도 구분하여 판단한다. 레지스터 판단은 isReg 함수를 사용하며, 피연산자 타입이 r0~r9인지 확인하여 맞으면 true, 틀리면 false를 반환한다. 상수와 STDOUT은 각각 타입(상수)가 imm(0x7240), SWOp(0x7241)와 일치한지 확인한다. 예를 들어, MOV는 1번째 피연산자에 무조건 레지스터여야 하고, 2번째 피연산자는 레지스터 혹은 상수가 들어가므로 그림 12와 같은 식을 작성해 준다. 만약 타입이 맞지 않다면 에러 문구를 출력하고 에러가 있다는 표시인 true를 반환한다.

```
***** Start Program *****

0x1 : MOV r5 0x0
State : R[5] = 0x0

0x2 : LW r6 0x5
State : R[5] = 0x0  R[6] = 0x5

0x3 : DIV r7 r6 r5
DIV ERROR - Divisor is zero

0x4 : div r7 r5 r6
NAME ERROR - Instructions not found

0x5 : ADDI r0 r1 0x5
NAME ERROR - Instructions not found

0x6 : SLT r0 r3
FEW ERROR - Input has fewer operands

0x7 : BNE r2 0x2 0x3 0x4
MORE ERROR - Input has more operands

0x8 : SW R0 STDOUT
INSTRUCTION ERROR - Some invalid operands

0x9 : LW r0 0X10
INSTRUCTION ERROR - Some invalid operands

0xa : MOV a0 500
INSTRUCTION ERROR - Some invalid operands

***** End Program *****
```

```
MOV r5 0x0
LW r6 0x5
DIV r7 r6 r5
div r7 r5 r6
ADDI r0 r1 0x5
SLT r0 r3
BNE r2 0x2 0x3 0x4
SW R0 STDOUT
LW r0 0X10
MOV a0 500
```

그림13. 상황별 에러 출력 (FILE 에러 제외)

에러는 케이스별로 다른 문구를 출력하며, read 단계에서 감지하는 FILE ERROR는 에러 출력 중 가장 우선순위가 높다. decode 단계에서 isErr_NAME, isErr_FEW, isErr_MORE 변수로 감지하는 에러는 각각 NAME, FEW, MORE ERROR와 대응하며 우선순위는 FILE ERROR 다음이다. 만약 중복 에러가 발생 시 우선 출력하는 에러는 NAMR > FEW > MORE 순이다. execute 단계에서는 is_Err 함수로 INSTRUCTION ERROR를, DIV명령어의 경우에만 연산 시 분모를 확인하여 DIV ERROR를 감지한다. 이 2가지 에러는 위의 4가지 에러를 모두 통과한 상황에서만 출력한다. 에러의 자세한 내용과 발생 케이스는 아래와 같다.

에러 이름	설명
FILE ERROR	입력 파일(input.txt)이 같은 디렉터리에 존재하지 않는 경우
NAME ERROR	12가지 명령어 외 다른 문자열을 입력한 경우
	명령어를 소문자로 입력하거나 오타가 난 경우
FEW ERROR	필요로 하는 피연산자가 부족할 경우
MORE ERROR	필요로 하는 피연산자를 초과할 경우
INSTRUCTION ERROR	피연산자(레지스터)에 r이 아닌 R을 입력한 경우
	피연산자(레지스터)에 다른 이름의 레지스터를 입력한 경우
	피연산자(상수)에 16진수 표현을 0x가 아닌 0X로 입력한 경우
	피연산자(상수)에 16진수가 아닌 수를 입력한 경우
	이 외의 모든 오타가 있는 경우
DIV ERROR	DIV에서만 발생하는 에러로, 분모가 0이라 나눌 수 없는 경우

에러 우선순위 : FILE > NAME > FEW > MORE > INSTRUCTION > DIV

4. Environment

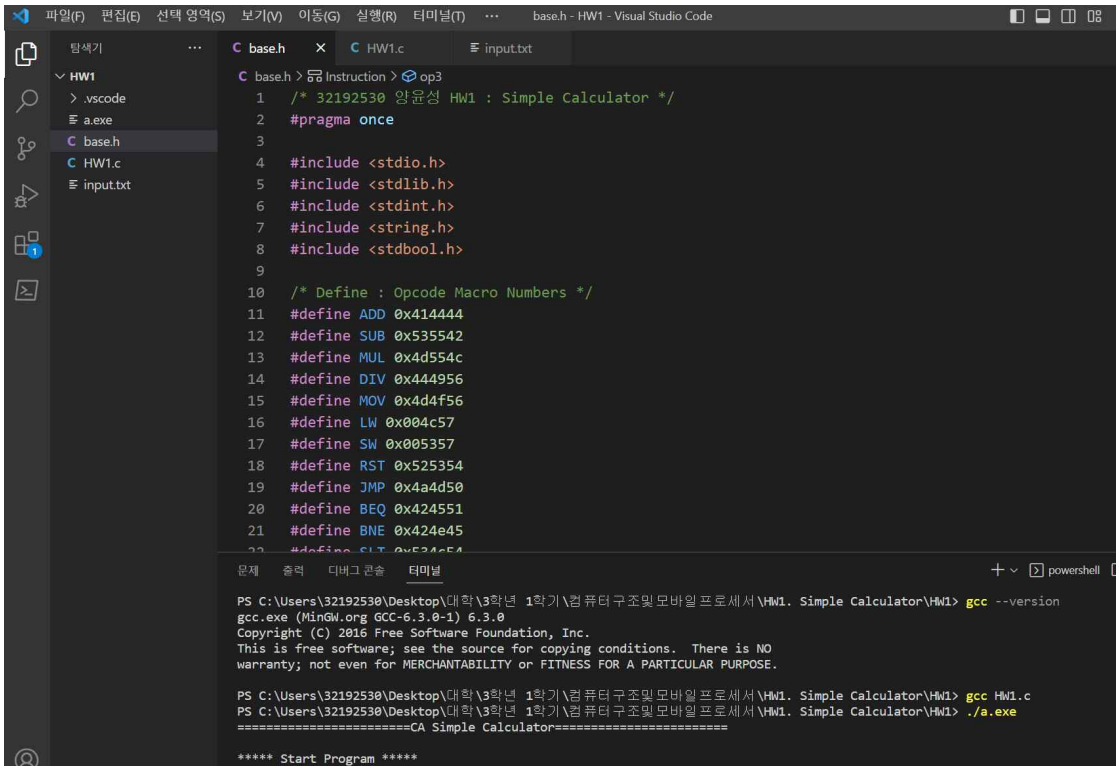

그림14. VS Code와 GCC Compiler

- **Build Environment**

Windows 10 Education x64

Visual Studio Code 1.76.2

GCC Compiler (MinGW.org GCC-6.3.0-1)

- **Compile**

1. HW1_32192530_양윤성_code.zip 압축풀기
2. Visual Studio Code 실행
3. 확장 → C/C++ Extension Pack 검색하여 설치
4. ①번에서 압축푼 폴더를 열기 (HW1.c, base.h, input.txt가 같이 들어있는 폴더)
5. 터미널 → 새 터미널 열기
6. 터미널에 **gcc HW1.c** 입력하여 a.exe 실행파일 생성
7. 터미널에 **./a.exe** 입력하여 a.exe 실행

- **Working Proofs**

input - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
LW r0 0xF
LW r1 0x4
ADD r2 r0 r1
MOV r0 r1
MOV r1 0x2
MUL r3 r0 r1
DIV r4 r0 r1
SW r0 STDOUT
RST
LW r0 0x5
LW r1 0xA
BEQ r0 0x7 0x10
BNE r0 r1 0xF
RST
BNE 0xA r1 0x13
BEQ 0x1 0x1 0x12
RST
JMP 0x14
RST
SLT r2 r0 r1
SLT r3 r0 0x3
SLT r4 0x9 r1
SLT r5 0xA 0xA
RST
```

그림15. test에 사용된 input.txt

```
PS C:\Users\32192530\Desktop\대학\3학년 1학기\컴퓨터구조및모바일프로세서\HW1. Simple Calculator\HW1> gcc HW1.c
PS C:\Users\32192530\Desktop\대학\3학년 1학기\컴퓨터구조및모바일프로세서\HW1. Simple Calculator\HW1> ./a.exe
=======================CA Simple Calculator=======================

***** Start Program *****

0x1 : LW r0 0xF
State : R[0] = 0xf

0x2 : LW r1 0x4
State : R[0] = 0xf  R[1] = 0x4

0x3 : ADD r2 r0 r1
State : R[0] = 0xf  R[1] = 0x4  R[2] = 0x13

0x4 : MOV r0 r1
State : R[0] = 0x4  R[1] = 0x4  R[2] = 0x13

0x5 : MOV r1 0x2
State : R[0] = 0x4  R[1] = 0x2  R[2] = 0x13

0x6 : MUL r3 r0 r1
State : R[0] = 0x4  R[1] = 0x2  R[2] = 0x13  R[3] = 0x8

0x7 : DIV r4 r0 r1
State : R[0] = 0x4  R[1] = 0x2  R[2] = 0x13  R[3] = 0x8  R[4] = 0x2

0x8 : SW r0 STDOUT
R[0] = 0x4 (STDOUT)
State : R[0] = 0x4  R[1] = 0x2  R[2] = 0x13  R[3] = 0x8  R[4] = 0x2

0x9 : RST
Reset all registers and buffers to zero.
```

그림16. compile 후 input.txt 테스트 결과 (0x0~0x9)

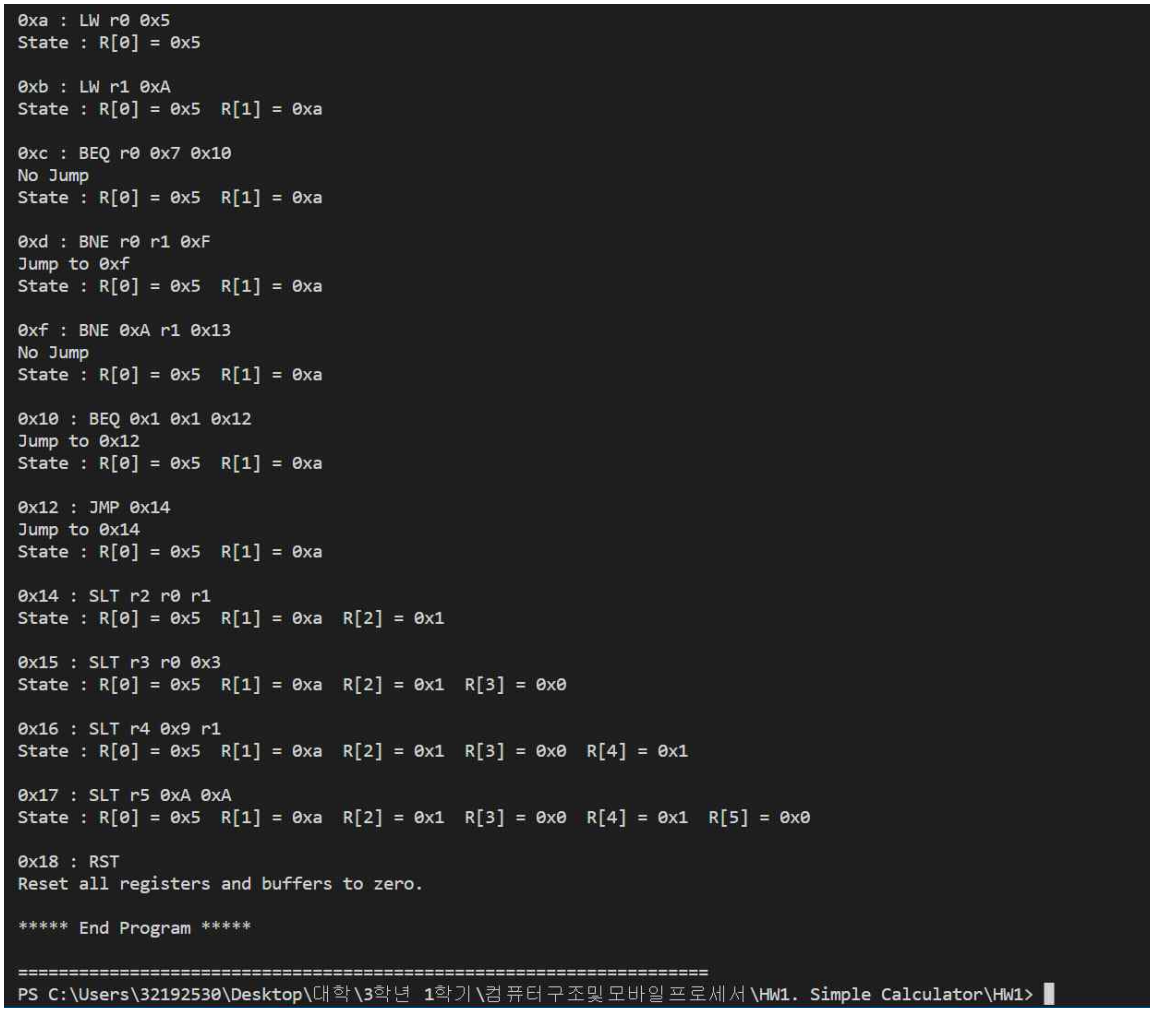

그림17. input.txt 테스트 결과 (0xa~0x18)

gcc로 컴파일할 시 "base.h"는 메인 코드에 include 되어 있으므로 같이 컴파일하지 않아도 된다. 컴파일 완료 후 실행파일인 a.exe를 실행하면 가장 위에서부터 순차적으로 명령어를 실행한 결과를 보여준다. 각 명령어 주소에는 어떤 명령어와 피연산자들이 있는지 보여준다. 모든 레지스터의 초깃값은 0인데, 연산 결과가 한번이라도 저장되면 Buffer 값이 1로 변경되어 레지스터 상태가 출력된다. 연산 결과값이 0이더라도 Buffer는 1이 된다. 결과가 1줄 더 적힌 경우가 있는데, SW는 피연산자에 적은 레지스터를 STDOUT으로 출력해 주고, JMP/BEQ/BNE의 경우 점프 정보를 출력해 준다. 또한 위 3가지 명령어는 특정 주소로 점프하게 되면 그 사이의 명령어들은 모두 무시한다. FILE 에러의 경우 파일 자체가 존재하지 않아 저장한 명령어가 없으므로 바로 프로그램을 중단한다. 그러나 나머지 에러들은 에러가 발생한 명령어만 실행하지 않고, 올바르게 작성한 다른 명령어들은 정상적으로 실행한다.

5. Lesson

```
else if (!strncmp(inst->opcode, "SLT", 3)){
    idx1 = atoi(inst->op1 + 1);
    R_Buffer[idx1] = 1;
    if (inst->op2[0] == 'r'){ // op1이 R일 경우
        idx2 = atoi(inst->op2 + 1);
        R_Buffer[idx2] = 1;
        val1 = R[idx2];
        op1 = inst->op2;
    }
    else{ // op1이 Imm일 경우
        val1 = strtol(inst->op2, NULL, 16);
        op1 = inst->op2;
    }
    if (inst->op3[0] == 'r'){ // op2가 R일 경우
        idx3 = atoi(inst->op3 + 1);
        R_Buffer[idx3] = 1;
        val2 = R[idx3];
        op2 = inst->op3;
    }
    else{ // op2가 Imm일 경우
        val2 = strtol(inst->op3, NULL, 16);
        op2 = inst->op3;
    }
    printf("SLT r%d, %s, %s\n", idx1, op1, op2);
    if (val1 < val2){
        R[idx1] = 1;
        printf("r%d = 1 (%s < %s) \n", idx1, op1, op2);
    }
    else{
        R[idx1] = 0;
        printf("r%d = 0 (%s >= %s) \n", idx1, op1, op2);
    }
    state();
}
```

그림18. 초기 HW1.c의 SLT 구현 부분

처음 과제를 읽고 당일에 바로 과제를 완성했었다. 그저 빠르게 끝내겠다는 생각으로, 코드를 설계하지 않고 생각나는 그대로 코드를 작성했다. 그러나 동일한 주 금요일 수업 시간에 MOV, BEQ, BNE, SLT의 피연산자 일부에 상수를 넣어도 된다는 정정 공지를 받고 코드를 다시 고치려니 막막했다. 확장성을 고려하지 않고 과제를 완료하다 보니 수정 시에 억지로 코드를 덧붙인 느낌이었고, 결국 복잡한 스파게티 코드가 되어버렸다. 딱 정해진 작업만큼은 문제없이 실행하지만, 기능 추가 난이도가 어려웠고, 코드가 너무 난해하여 알아보기가 힘들었다. 그래서 설계부터 다시 시작하여 프로그램을 재구현했다. 완성 후 두 코드를 비교해 보니 코드 설계가 얼마나 중요한지를 깨달았다. 다만 확장성을 더 고려하여, Instruction 구조체에 Operand 타입 변수(피연산자)를 배열로 한 뒤 포인터를 사용하거나, 사용되는 변수를 줄여 메모리를 절약했으면 좋았을 것 같은 아쉬움이 든다. 그래도 레포트를 작성하면서 폰 노이만 구조와 ISA에 대해 확실히 공부하였고, 이 구조들이 어떤 원리로 작동하는지를 코드를 직접 작성하며 알 수 있었다. 이번 프로젝트가 후에 있을 Single-Cycle MIPS와 Pipelined MIPS를 구현할 시 확실한 밑거름이 될 거라 생각한다.