컴퓨터구조 및 모바일프로세서 HW1 결과 보고서

과목명 : 컴퓨터구조 및 모바일프로세서 2분반 (525060-2)
담당교수 : 남 재 현 교수님
학번 : 32212190
제출자 : 손보경
제출일 : 2023.04.06

---

1. 서론

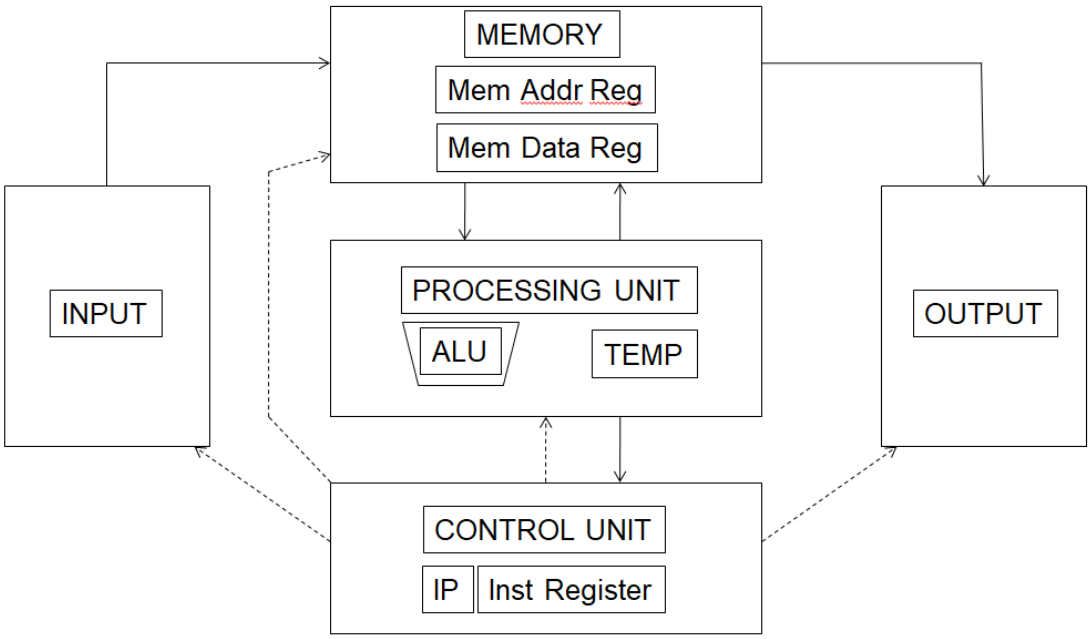
1.1. 과제에 대한 간략한 설명

해당 과제의 목표는 입력 문자열에 해당하는 계산을 수행하는 것이다. 10개의 레지스터(r0 ~ r9)를 지원하고, 기본으로 산술 이진 연산 (ADD, SUB, MUL, DIV) 및 MOV를 지원하여 레지스터 값을 이동하고, LW 및 SW를 지원하여 입력에서 데이터를 로드하고 결과를 출력한다. 입력 파일에서 문자열을 가져오고, 사용된 레지스터의 모든 상태 변경과 함께 디스플레이를 인쇄한다. 또한, 예외를 정상적으로 처리를 할 수 있어야 한다.

산술 이진 연산은 3개의 operand가 필요하고 3개 다 레지스터가 들어가게 된다. MOV는 2개의 operand가 필요하고 2개 다 레지스터가 들어가며, LW는 2개의 operand가 필요하며 1개의 레지스터와 1개의 16진수 string이 필요하다. SW은 우리가 알던 MIPS 문법에서 벗어나 SW [register] STDOUT 구조로 이루어져 있고, 출력 명령어이다. RST는 모든 레지스터 값을 초기화시키는 명령어이다. JMP는 16진수로 된 주솟값이 필요하다. BEQ, BNE는 2개의 operand와 1개의 분기 주소가 필요하다. 이 2개의 operand는 레지스터가 될 수도 있고, 16진수 형태의 정수가 될 수도 있다. 마지막으로 SLT는 1개의 레지스터와 2개의 operand가 필요하며, 2개의 operand는 레지스터가 될 수도 있고, 16진수 형태의 정수가 될 수도 있다.

위의 조건을 생각하며, 구조체 및 전역 변수, 함수, 메인 함수를 다음과 같이 짜게 되었고, 프로그램에 대해 설명하도록 하겠다.

1.2. 중요한 개념 - 폰노이만 구조



폰노이만 구조(Von Neumann Architecture)는 일반적인 컴퓨터 아키텍처 중 하나로, 중앙처리장치(CPU), 메모리, 입출력장치(IO) 등이 연결된 구조를 말한다.

이 구조는 CPU가 프로그램을 실행하는 동안 메모리에 저장된 데이터를 읽어와 연산하고, 결과를 다시 메모리에 저장한다. 이를 위해 CPU와 메모리 사이에는 데이터 버스와 주소 버스라는 통신 경로가 존재하며, 입출력장치는 CPU와 별도의 버스를 통해 데이터를 주고받는다.

폰노이만 구조는 기계어로 작성된 명령어를 순차적으로 처리하기 때문에, 순차적인 실행이 중요한 일련의 작업에 적합하다. 또한, 기억장치에 저장된 데이터를 읽고 쓰는 속도가 CPU 내부의 레지스터와 비교하여 매우 느리기 때문에, 대용량 데이터 처리에는 부적합하다. 이러한 한계를 극복하기 위해 캐시와 같은 고속 기억장치가 사용되어 왔다.

폰노이만 구조는 현재 대부분의 컴퓨터에서 사용되는 구조이다. 이 구조를 바탕으로 다양한 아키텍처가 발전해 나가고 있으며, 복수의 CPU와 메모리가 서로 다른 경로로 연결되어 있는 다중 프로세서 구조(Multiprocessor Architecture)나, GPU와 CPU를 병렬로 사용하는 구조 등도 발전하고 있다.

1.3. 중요한 개념 - Stored program

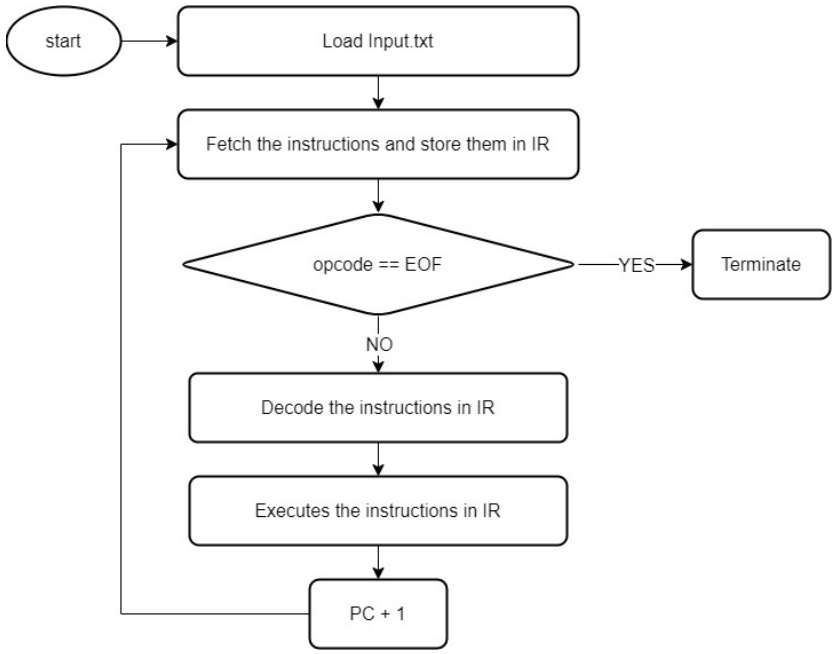
Stored program은 컴퓨터 아키텍처의 기본 개념 중 하나로, 컴퓨터 메모리에 저장된 프로그램 코드가 실행될 수 있는 형태를 한다.

기존의 컴퓨터는 연산을 수행할 때마다, 수행할 명령어와 연산 대상 데이터를 각각의 레지스터에서 가져와야 했다. 그러나 Stored program 구조를 갖는 컴퓨터는 프로그램 코드 또한 메모리에 저장되어 있으며, 명령어와 데이터가 모두 메모리에 저장되어 있으므로, 실행할 명령어를 차례대로 가져와서 실행할 수 있다.

Stored program 구조의 가장 큰 장점은 프로그램의 재사용성과 수정 용이성이다. 한 번 작성한 프로그램 코드를 메모리에 저장하고, 그것을 다시 실행할 수 있기 때문에 프로그램을 여러 번 사용할 수 있다. 또한, 프로그램 수정 시에도 메모리에 저장된 프로그램 코드를 수정하기만 하면 되므로, 수정이 용이하다.

이러한 장점들 때문에 대부분의 컴퓨터 아키텍처에서 Stored program 구조를 사용하고 있으며, 현대 컴퓨터의 운영체제, 애플리케이션 프로그램, 게임 등 모든 소프트웨어들은 Stored program 구조에서 작동하고 있다.

1.5. 프로그램 순서도



위의 그림은 프로그램을 어떻게 제작할지에 대한 간단한 순서도이다. 처음 시작하면, input.txt 파일을 로드한 뒤, instructions을 가져와 IR(구조체 instruction)에 저

장한다. 이후, op code가 EOF 라면 종료, 아니라면 다음 명령을 수행한다. instructions을 IR로 디코딩 즉, 읽어온 한 줄을 구조체에다가 저장한다. 그 후, opcode에 맞는 instruction을 실행한다.

2. 본론

2.1. 전반적인 프로그램 소개

본 프로그램은 MIPS 어셈블리 언어의 기능을 C언어로 구현하였다. 이 프로그램은 10개의 레지스터가 있고 기본적인 산술 및 논리 명령을 실행할 수 있는 컴퓨터 아키텍처를 위한 단순한 어셈블러를 구현하였다. 주어진 input.txt 파일을 읽어와서 명령어를 해석하고 실행하며, ADD, SUB, MUL, DIV 등의 산술 연산 및 MOV, LW, SW, RST 등의 데이터 전송 연산도 포함하고 있다. 또한, JMP, BEQ, BNE, SLT 등의 분기 명령어도 구현되어 있다. 코드에서는 전역 변수 및 구조체를 사용하며, 파일 입출력과 문자열 처리 함수 등을 사용하여 구현하였다. 메인 함수는 일부 변수와 구조를 초기화하고, 파일을 한 줄씩 읽고, 조건문을 사용하여 명령어 opcode에 해당하는 각 줄을 첫 번째 토큰을 기반으로 적절한 명령어 기능을 실행한다. 지금부터 각각의 부분을 자세히 소개하도록 하겠다.

2.2. 헤더 파일 및 전역 변수와 구조체

```
1     #pragma warning (disable : 4996)
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <string.h>
5
6                                   // 전역 변수 및 구조체
7   char LINE[100];                 // txt 파일 한 줄을 읽기 위한 배열
8   int R[10] = { 0, };             // 레지스터 0~9
9
10  typedef struct _instruction {   // instruction
11      char OPcode[3];
12      char op[3][10];
13  } Instruction;
```

해당 프로그램은 총 3개의 헤더 파일과 2개의 전역 변수, 1개의 구조체를 사용하고 있다.

#pragma warning (disable : 4996)은 컴파일러가 경고하는 것 중에서 특정한 경고를 무시하도록 지시하는 프리 프로세서 지시어이다. 구체적으로, 4996 경고는

Visual Studio에서 사용되며, 보안상의 이유로 권장되지 않는 함수들을 사용했을 때 발생한다. scanf에서 %c, %s와 같은 입력을 사용하면 버퍼 오버플로우가 발생할 수 있다. 이를 무시하여 에러를 잘라버리기 위해 사용되었다.

stdlib.h는 C 언어의 표준 라이브러리로, 문자열 변환 및 구조체를 위해 사용되었다. 또한 string.h 헤더 파일 역시 문자열을 처리하기 위해 선언되었다.

char LINE[100]은 input.txt 파일을 읽어오기 위해 만든 전역 변수이다. input.txt 파일에 있는 문자열 한 줄을 읽어오고, 후에 main() 함수에서 공백을 기준으로 자르게 된다. 또한, 사용하기 전, null로 초기화하여 저장 값을 덮어쓰게 되었을 때 발생하는 오류를 잡았다.

int R[10]은 레지스터를 위한 전역 변수이다. r0 ~ r9으로 이루어져 있고, 총 10개의 레지스터가 있으며 값은 0으로 초기화되어 있다.

instruction 구조체는 input된 명령어를 opcode와 3개의 operand로 나누기 위해 설정되었다. typedef를 선언해준 이유는 메인 함수에서 따로 구조체를 선언할 필요가 없고, strcpy 라는 명령어를 사용하기 위해 선언하였다. 또한, 문자열을 자르고 저장할 때 쉽게 반복문으로 사용하기 위해 operand 부분은 2차원 배열로 선언하였다.

2.3. 메인 함수

```
34  int main(){
35      Instruction code[100];
36      int pc = 1;
37      int jumppc = 1;
38
39      freopen("input.txt", "r", stdin);
40
41      while (1) {
42          init();
43          gets(LINE);
44          if (pc == jumppc) printf("PC : %d₩tOPCODE : %s₩n", pc, LINE);
45
46          // instruction format
47          if (pc <= jumppc) {
48              int i = 0;
49
50              char* temp = strtok(LINE, " ");
51              strcpy(code[pc].OPcode, temp);
52              temp = strtok(NULL, " ");
53
54              while (temp != NULL) {
55                  strcpy(code[pc].op[i], temp);
56                  temp = strtok(NULL, " ");
57                  i++;
58              }
59          }
60          else if (pc > jumppc) {
61              pc = jumppc;
62          }
63
```

```
64          if (pc == jumppc) {
65              // run operator
66              if (strncmp(code[pc].OPcode, "EOF", 3) == 0) break;
67
68              else if (strncmp(code[pc].OPcode, "ADD", 3) == 0) { ADD(code[pc]); }
69              else if (strncmp(code[pc].OPcode, "SUB", 3) == 0) { SUB(code[pc]); }
70              else if (strncmp(code[pc].OPcode, "MUL", 3) == 0) { MUL(code[pc]); }
71              else if (strncmp(code[pc].OPcode, "DIV", 3) == 0) { DIV(code[pc]); }
72
73              else if (strncmp(code[pc].OPcode, "MOV", 3) == 0) { MOV(code[pc]); }
74              else if (strncmp(code[pc].OPcode, "LW", 2) == 0) { LW(code[pc]); }
75              else if (strncmp(code[pc].OPcode, "SW", 2) == 0) { SW(code[pc]); }
76              else if (strncmp(code[pc].OPcode, "RST", 3) == 0) { RST(); }
77
78              else if (strncmp(code[pc].OPcode, "JMP", 3) == 0) { jumppc = JMP(code[pc].op[0]); }
79
80              else if (strncmp(code[pc].OPcode, "BEQ", 3) == 0) { jumppc = BEQ(code[pc], pc); }
81              else if (strncmp(code[pc].OPcode, "BNE", 3) == 0) { jumppc = BNE(code[pc], pc); }
82              else if (strncmp(code[pc].OPcode, "SLT", 3) == 0) { SLT(code[pc]); }
83
84              if (jumppc == pc) { jumppc = jumppc + 1; }
85              pc = pc + 1;
86          }
```

```
87          else if (pc != jumppc) { pc = pc + 1; }
88      }
89
90      return 0;
91  }
```

main 함수는 34줄에서 91줄까지 총 57줄에 거쳐 만들어졌다. 35 - 37줄은 사용될 변수를 설정해 주는 구문이다. 사용될 instruction 구조체를 사이즈가 100인 배열로 선언해 줬다. 또한, 현재 명령어 위치인 pc와 이동될 위치인 jump pc를 각각 1로 초기화해주었다. 39줄에서 사용될 input.txt을 file open을 해주었다.

while 무한 루프 구문은 input.text 파일을 읽고, 그에 맞는 opcode의 명령어를 수행하는 형식으로 이루어져 있다. LINE을 초기화해주기 위해 init() 함수를 이용하여 초기화해주었고, 이후 한 줄씩 읽는 형식이다. scanf를 사용하여 %s 형식으로 읽게 되면, 텍스트 파일에 있는 공백 문자를 읽지 못한다는 문제점이 있다. 이를 해결하기 위해 gets() 함수를 사용하여 텍스트 파일을 읽어왔다. 이를 공백 문자를 기준으로 따로 저장하기 위해 48 - 57줄을 작성하였다. strtok() 함수를 이용하여 공백 문자를 기준으로 문자열을 자르고, 포인터를 반환하였다. 자른 문자열이 나오지 않을 때까지 즉 포인터가 null일 때까지 문자열을 잘라서 포인터를 반환하고 저장하였다. 이때 구조체에 저장하기 위해 strcpy() 함수를 사용하였다.

이후 code[pc].OPcode에 해당되는 명령어를 수행하는 구문을 만들었다. 총 66 - 85줄에 거쳐 작성하였으며, strncmp() 함수를 사용하여 문자열을 비교하고, 조건문을 이용하여 코드를 짰다. strncmp() 함수를 사용한 이유는 code[pc].op[i]의 문자열 크기와 비교하는 문자열의 크기가 달라서 strcmp() 함수를 사용하면 오류가 발생할 수 있어 strncmp() 함수를 사용하였다. 문자열이 “EOF”일 때 무한루프는 종료되고, 그 외 다른 명령어는 각각의 함수를 이용하여 실행된다. 분기 명령어일 경우는 해당 함

수의 return 값으로 jumppc를 변경해주어 분기할 수 있도록 설정하였다.

다음은 pc와 jumppc를 이용한 조건문에 관해 설명하겠다. pc와 jumppc를 이용한 조건문은 모두 분기를 위한 조건문이다. 47줄에 있는 pc가 jumppc보다 작거나 같은 경우는 말 그대로, jumppc와 pc가 같으면 텍스트 파일에 있는 명령어를 읽고 구조체에 저장하여 명령어를 실행하라는 뜻이다. 또한, pc가 jumppc보다 작을 경우는 아직 분기 지점보다 pc address가 더 작은 것이므로 구조체에 텍스트 파일을 저장하나, 명령어 실행은 하지 않는다. 60줄에 있는 else if (pc > jumppc) 구문은 현재 명령어 이전으로 분기하기 때문에 pc 값을 jumppc로 바꿔주고, 해당 명령어를 수행해주는 역할이다. 84줄에 있는 if 절은 분기 명령어(JMP, BEQ, BNE)이 사용되지 않았을 때, 즉 현재 pc와 jumppc가 같을 때 jumppc에 +1을 해주는 구문이다. 마지막으로 87줄에 있는 else if (pc != jumppc) 구문은 pc가 jumppc 보다 작을 때, pc에 +1을 해주는 코드이다. 이를 통해 구조체에 명령어는 저장되지만, 명령어 실행은 되지 않게 해준다.

2.4. init() 함수

```
95  void init() {
96      for (int i = 0; i < 100; i++) LINE[i] = '₩0';
97  }
```

init() 함수는 input 텍스트 파일 한 줄을 읽어올 때 사용하는 LINE 배열을 초기화해주는 함수이다. 이를 통해 읽을 때 쓰레깃값이 저장되어있는 것을 방지할 수 있다. 예를 들어, 첫 번째에 읽은 텍스트의 길이가 14라고 가정하고, 두 번째에 읽은 텍스트의 길이가 10이라고 가정한다면, 뒤에 첫 번째에 읽은 텍스트가 저장되어있을 수 있다. 이를 방지하기 위해 미리 LINE 배열을 초기화해주는 것이다.

2.5. HextoDec() 함수

```
99  int HextoDec(char hex[10]) {
100     int dec = 0;
101
102     for (int i = 2; hex[i] != '₩0'; i++)
103     {
104         if (hex[i] >= 'A' && hex[i] <= 'F')              // hex값이 'A'(65)~'F'(70) 일때
105             dec = dec * 16 + hex[i] - 'A' + 10;
106         else if (hex[i] >= 'a' && hex[i] <= 'f')         // hex값이 'a'(97)~'f'(102) 일때
107             dec = dec * 16 + hex[i] - 'a' + 10;
108         else if (hex[i] >= '0' && hex[i] <= '9')         // hex값이 '0'(48)~'9'(57) 일때
109             dec = dec * 16 + hex[i] - '0';
110     }
111
112     return dec;
113 }
```

HextoDec() 함수는 16진수 형태의 문자열을 10진수로 변환하는 함수이다. 먼저 10

진수 값이 저장될 dec를 0으로 초기화한다. 그리고 for문을 이용하여 hex 문자열의 3번째부터 마지막 문자까지(마지막 문자는 널 문자 '\0') 반복한다. 16진수에서 맨 앞의 0x는 무시되기 때문에 i는 2부터 시작된다.

for문 안에서는 if-else 문을 이용하여 문자가 A~F, a~f, 0~9 범위 내에 있으면 각각의 조건에 맞게 16진수를 10진수로 변환한다. 만약 hex[i] 값이 'A'~'F' 범위 내에 있으면, 'A'를 뺀 후 10을 더해주어 10진수로 변환한다. 만약 hex[i] 값이 'a'~'f' 범위 내에 있으면, 'a'를 뺀 후 10을 더해주어 10진수로 변환한다. 마지막으로 만약 hex[i] 값이 '0'~'9' 범위 내에 있으면, '0'을 뺀 후 10진수로 변환한다.

16진수를 10진수로 변환할 때는 각 자리의 16진수 수치를 10진수로 변환하고 자릿수에 맞게 계산하여 전체 10진수 값을 구한다. 구한 10진수 값인 dec를 반환한다.

2.6. 산술 연산 함수

```
116  void ADD(Instruction a) {
117      int num[3] = { 0, };
118      for (int i = 0; i < 3; i++) {
119          num[i] = a.op[i][1] - '0';
120      }
121
122      R[num[0]] = R[num[1]] + R[num[2]];
123
124      printf("[ACTION] ADD : R[%d] = R[%d] + R[%d] (%d = %d + %d)₩n₩n", num[0], num[1], num[2], R[num[0]], R[num[1]], R[num[2]]);
125  }
126
127  void SUB(Instruction a) {
128      int num[3] = { 0, };
129      for (int i = 0; i < 3; i++) {
130          num[i] = a.op[i][1] - '0';
131      }
132
133      R[num[0]] = R[num[1]] - R[num[2]];
134
135      printf("[ACTION] SUB : R[%d] = R[%d] - R[%d] (%d = %d - %d)₩n₩n", num[0], num[1], num[2], R[num[0]], R[num[1]], R[num[2]]);
136  }
```

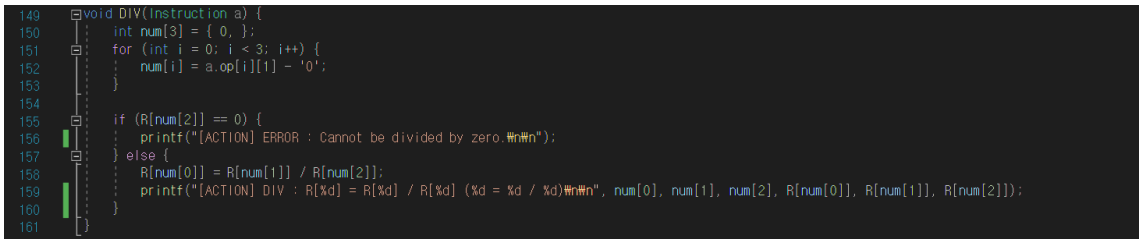
```
138  void MUL(Instruction a) {
139      int num[3] = { 0, };
140      for (int i = 0; i < 3; i++) {
141          num[i] = a.op[i][1] - '0';
142      }
143
144      R[num[0]] = R[num[1]] * R[num[2]];
145
146      printf("[ACTION] MUL : R[%d] = R[%d] * R[%d] (%d = %d * %d)₩n₩n", num[0], num[1], num[2], R[num[0]], R[num[1]], R[num[2]]);
147  }
```

산술 연산 함수는 ADD, SUB, MUL, DIV로 총 4개로 구성되어 있다. ADD, SUB, MUL, DIV 함수는 각각 덧셈, 뺄셈, 곱셈, 나눗셈을 수행하는 함수이다. 이 함수들은 모두 Instruction 타입의 인자를 받으며, 해당 인자는 명령어의 정보를 가지고 있다. 예를 들어, ADD 함수는 R[num[1]]과 R[num[2]]의 값을 더한 후, 그 결과를 R[num[0]]에 저장한다. 이때, R은 레지스터를 나타내는 배열이다. 각 함수는 연산 결과를 적절한 형식으로 출력하고, 레지스터의 값을 업데이트한다.

산술 연산 함수의 operand는 다 레지스터로 되어있기 때문에 따로 16진수일 경우

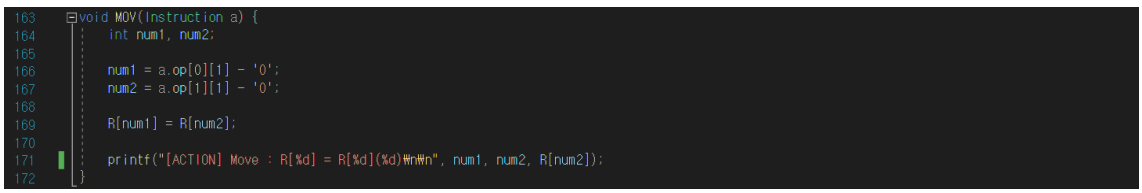
를 지정하지 않았다. 또한, 구조체의 type은 char이기 때문에 이를 int로 바꿔주기 위해 ASCII 코드를 활용해 '0'을 빼주어 정수형으로 형 변환을 해주었다.

각 함수는 연산에 필요한 레지스터 값을 a.op 배열에서 추출한다. a.op 배열은 명령어에서 인자로 전달된 레지스터의 정보를 담고 있다. 예를 들어, "ADD r1 r2 r3" 명령어에서는 a.op[0], a.op[1], a.op[2]이 각각 ['r', '1'], ['R', '2'], ['R', '3']을 저장하게 된다. 따라서 ADD 함수는 a.op[1][1]과 a.op[2][1]을 추출하여 num[1], num[2] 배열에 저장한다. 이렇게 추출된 값들을 이용하여 연산을 수행하고, 그 결과를 출력한다.



DIV 함수는 두 번째 인자로 전달된 값이 0인 경우, "[ACTION] ERROR : Can not be divided by zero" 메시지를 출력하고 연산을 수행하지 않는다. 그렇지 않은 경우, 명령어에서 전달된 레지스터의 값을 나눈 후, 그 결과를 레지스터에 저장하고 출력한다.

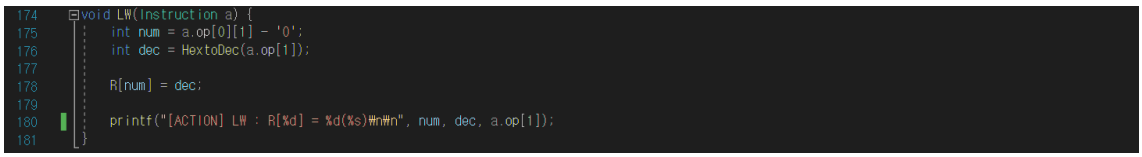
2.7. MOV() 함수



MOV 함수는 전달받은 인자 a의 두 번째 오퍼랜드 값을 첫 번째 오퍼랜드에 저장하는 함수이다. 이 함수는 인자 a에서 레지스터 번호를 추출하여 해당하는 레지스터 값을 불러와 저장하는 역할을 한다. 저장된 값을 출력하는 printf문을 통해 데이터 이동이 수행되었음을 알려준다.

위의 산술 연산 함수와 마찬가지로 MOV 함수도 똑같이 16진수 값을 처리할 필요가 없으므로 레지스터 번호만 추출하여 연산을 수행한다. 예를 들어 입력값이 MOV r0, r1였다면, r0에 r1의 값이 저장되는 것이다.

2.8. LW() 함수



```
174  void LW(Instruction a) {
175      int num = a.op[0][1] - '0';
176      int dec = HextoDec(a.op[1]);
177
178      R[num] = dec;
179
180      printf("[ACTION] LW : R[%d] = %d(%s)₩n₩n", num, dec, a.op[1]);
181  }
```

이 함수는 Load Word 명령어를 처리하는 함수로, a.op 배열의 두 번째 원소에 저장된 16진수 값을 가져와서 해당하는 레지스터에 저장한다. a.op[1] 배열에 저장된 16진수 값을 HextoDec 함수를 이용해 10진수로 변환한 후, 해당하는 레지스터(a.op[0][1]에서 추출한 레지스터 번호)에 저장한다.

2.9. SW() 함수

```
183  void SW(Instruction a) {
184      int num = a.op[0][1] - '0';
185
186      printf("[ACTION] SW : R[%d]is %d₩n₩n", num, R[num]);
187  }
```

이 함수는 Store Word 명령어를 처리하는 함수로, a.op 배열의 첫 번째 원소에 저장된 값을 출력하는 함수이다.

2.10. RST() 함수

```
189  void RST() {
190      for (int i = 0; i < 10; i++) { R[i] = 0; }
191
192      printf("[ACTION] Reset all registers₩n₩n");
193  }
```

이 함수는 Reset all registers를 수행하는 함수로, r0 - r9의 모든 레지스터를 0으로 초기화시켜주는 함수다.

2.11. JMP() 함수

```
195  int JMP(char hex[10]) {
196      int dec = HextoDec(hex);
197
198      printf("[ACTION] Jump to %d₩n₩n", dec);
199      return dec;
200  }
```

이 함수는 jump 할 주소를 지정하는 함수로, 문자열 형태의 16진수를 받으면 HexoDec() 함수를 이용하여 10진수로 바꿔주고, dec를 return하는 함수이다. 반환 값은 main에서 jumppc에 저장이 된다.

2.12. BEQ() 함수

```
202  int BEQ(Instruction a, int pc) {
203      int op[2] = { 0, };
204      int num = HextoDec(a.op[2]);
205
206      for (int i = 0; i < 2; i++) {
207          if (a.op[i][0] == 'r') {
208              int temp = a.op[i][1] - '0';
209              op[i] = R[temp];
210          }
211          else if (a.op[i][0] == '0') {
212              int temp = HextoDec(a.op[i]);
213              op[i] = temp;
214          }
215      }
216
217      int jump = 0;
218      if (op[0] == op[1]) { jump = num; }
219      else if (op[0] != op[1]) { jump = pc; }
220
221      printf("[ACTION] Jump to %d₩n₩n", jump);
222
223      return jump;
224  }
```

이 코드는 BEQ(분기 등가) 명령어를 실행하는 함수이다. 이 함수는 현재 실행 중인 명령어의 주소인 pc와 함께 Instruction 구조체를 인자로 받아온다.

함수의 첫 번째 부분에서는 명령어에서 비교할 값과 분기할 위치를 구한다. 첫 번째와 두 번째 연산자 중 'r'로 시작하는 것은 레지스터의 값을 가리키므로 해당 레지스터의 값으로 대체한다. 그렇지 않은 경우는 16진수 문자열로 된 값으로 가정하고 HextoDec 함수를 이용하여 10진수 값으로 변환한다.

다음으로, 첫 번째와 두 번째 연산자의 값이 같으면 jump 변수에 분기할 위치를 저장하고, 그렇지 않은 경우에는 현재 주소인 pc를 저장한다. 마지막으로 jump 변수를 출력하고 반환한다.

2.13. BNE() 함수

```
226  int BNE(Instruction a, int pc) {
227      int op[2] = { 0, };
228      int num = HextoDec(a.op[2]);
229
230      for (int i = 0; i < 2; i++) {
231          if (a.op[i][0] == 'r') {
232              int temp = a.op[i][1] - '0';
233              op[i] = R[temp];
234          }
235          else if (a.op[i][0] == '0') {
236              int temp = HextoDec(a.op[i]);
237              op[i] = temp;
238          }
239      }
240
241      int jump = 0;
242      if (op[0] != op[1]) { jump = num; }
243      else if (op[0] == op[1]) { jump = pc; }
244
245      printf("[ACTION] Jump to %d₩n₩n", jump);
246
247      return jump;
248  }
```

이 코드는 BNE(Branch if Not Equal) 명령어를 구현한 함수이다. BNE 명령어는 두 개의 레지스터나 값이 다른 경우에 지정된 주소로 분기하는 명령어입니다.

함수의 입력으로는 현재 명령어와 다음에 실행될 명령어의 위치를 나타내는 pc 값

이 주어지며, 함수의 반환값으로는 분기할 명령어의 위치를 나타내는 jump 값이 반환된다.

함수의 구현 방식은 BEQ 함수와 유사하다. 입력된 명령어에서 비교할 레지스터나 값들을 가져와 op 배열에 저장한 뒤, 두 개의 값이 다른 경우에는 jump 변수에 지정된 주소를 할당하고, 두 개의 값이 같은 경우에는 현재 명령어의 다음 위치인 pc 값을 할당한다. 마지막으로, jump 변수의 값을 출력하고 반환한다.

2.14. SLT() 함수

```
250  void SLT(Instruction a) {
251      int op[2] = { 0, };
252
253      for (int i = 1; i < 3; i++) {
254          if (a.op[i][0] == 'r') {
255              int temp = a.op[i][1] - '0';
256              op[i-1] = R[temp];
257          }
258          else if (a.op[i][0] == '0') {
259              int temp = HextoDec(a.op[i]);
260              op[i-1] = temp;
261          }
262      }
263
264      int num = a.op[0][1] - '0';
265
266      if (op[0] < op[1]) { R[num] = 1; }
267      else { R[num] = 0; }
268
269      printf("[ACTION] SLT : R[%d] = %d₩n₩n", num, R[num]);
270  }
```

해당 코드는 SLT(Set on Less Than) 명령어를 구현하는 함수다. 함수의 기능은 두 개의 레지스터 혹은 레지스터와 상수의 값을 비교하여, 첫 번째 인자로 받은 레지스터에 비교 결과를 저장하는 것이다. 비교 결과가 참이면 1, 거짓이면 0을 저장합니다.

코드는 먼저 명령어에서 비교할 레지스터 혹은 상수 값을 가져와서 op 배열에 저장합니다. 그리고 첫 번째 인자로 받은 레지스터에, op[0] < op[1] 이면 1, 그렇지 않으면 0을 저장합니다. 마지막으로 결과를 출력합니다.

예를 들어, "slt $r1, $r2, 10"이라는 명령어는 $r2의 값이 10보다 작으면 $r1에 1을 저장하고, 그렇지 않으면 0을 저장합니다.

2.15. 프로그램 실행 결과

왼쪽 그림은 input.text 파일의 내용이고 우측 그림은 실행 화면이다. 순서대로 LW, SLT가 잘 작동하고 있음을 알 수 있고, BNE를 통한 분기도 잘 되는 것을 알 수 있다. 간단하게만 살펴보자면, 1-2번째 줄에서 r0에 2를, r1에 1을 저장하고 3번

째 줄에서 r0와 r1을 비교한 결과 r0가 더 크므로 r2에는 0이 저장된다. 4-6번째 줄도 마찬가지로 잘 작동하고 있음을 볼 수 있다. 7번째 줄에서 r2와 r3를 비교했을 때 r2는 0이고 r3는 1이므로 같지 않아 0xA번째 즉 10번째 줄로 분기한다. 아래의 실행결과를 보면 10번째 명령어인 RST로 잘 분기되었음을 알 수 있다. 이후 11번째에서 22번째로 분기하고, 22번째 명령어인 LW로 인해 r1에 0xFFFE의 값인 65534가 잘 저장되었음을 확인할 수 있다. 25번째 줄에서 r0 = r1 + r2가 수행되고 26번째 줄에서 r0가 출력되는 모습도 확인할 수 있다. input 파일의 내용을 여러 번 수정하여 실행시켜 본 결과, 오류 없이 잘 실행되는 것을 볼 수 있다.

```
1  LW r0 0x2
2  LW r1 0x1
3  SLT r2 r0 r1
4  SLT r3 r0 0x5
5  SLT r4 0xF r1
6  SLT r5 0xF 0xFF
7  BNE r2 r3 0xa
8  LW r0 0x1
9  LW r1 0x2
10 RST
11 JMP 0x16
12 ADD r2 r0 r1
13 LW r3 0x5
14 LW r4 0xa
15 MUL r5 r3 r4
16 SW r2 STDOUT
17 LW r0 0x10
18 LW r1 0x3
19 DIV r2 r0 r1
20 MOV r3 r2
21 LW r0 0xFF
22 LW r1 0xFFFe
23 LW r2 0xFFd
24 LW r9 0x1
25 ADD r0 r1 r2
26 SW r0 STDOUT
27 MOV r2 r0
28 MOV r4 r2
29 MOV r5 r4
30 MOV r7 r5
31 SW r7 STDOUT
32 EOF
```

```
Microsoft Visual Studio 디버그 콘솔
PC : 1  OPCODE : LW r0 0x2
[ACTION] LW : R[0] = 2(0x2)

PC : 2  OPCODE : LW r1 0x1
[ACTION] LW : R[1] = 1(0x1)

PC : 3  OPCODE : SLT r2 r0 r1
[ACTION] SLT : R[2] = 0

PC : 4  OPCODE : SLT r3 r0 0x5
[ACTION] SLT : R[3] = 1

PC : 5  OPCODE : SLT r4 0xF r1
[ACTION] SLT : R[4] = 0

PC : 6  OPCODE : SLT r5 0xF 0xFF
[ACTION] SLT : R[5] = 1

PC : 7  OPCODE : BNE r2 r3 0xa
[ACTION] Jump to 10

PC : 10 OPCODE : RST
[ACTION] Reset all registers

PC : 11 OPCODE : JMP 0x16
[ACTION] Jump to 22

PC : 22 OPCODE : LW r1 0xFFFe
[ACTION] LW : R[1] = 65534(0xFFFe)

PC : 23 OPCODE : LW r2 0xFFd
[ACTION] LW : R[2] = 4093(0xFFd)

PC : 24 OPCODE : LW r9 0x1
[ACTION] LW : R[9] = 1(0x1)

PC : 25 OPCODE : ADD r0 r1 r2
[ACTION] ADD : R[0] = R[1] + R[2] (69627 = 65534 + 4093)

PC : 26 OPCODE : SW r0 STDOUT
[ACTION] SW : R[0]is 69627

PC : 27 OPCODE : MOV r2 r0
[ACTION] Move : R[2] = R[0](69627)

PC : 28 OPCODE : MOV r4 r2
[ACTION] Move : R[4] = R[2](69627)

PC : 29 OPCODE : MOV r5 r4
[ACTION] Move : R[5] = R[4](69627)

PC : 30 OPCODE : MOV r7 r5
[ACTION] Move : R[7] = R[5](69627)

PC : 31 OPCODE : SW r7 STDOUT
[ACTION] SW : R[7]is 69627

PC : 32 OPCODE : EOF

C:₩Users₩User₩Desktop₩Simple Calculator₩x64₩Debug₩Simple Calculator.exe(프로세스 27428개)이(가) 종료되었습니다(코드: 0개
).
디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [옵션] -> [디버깅] > [디버깅이 중지되면 자동으로 콘솔 닫기]를 사용
하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...
```

2.16. 오류 및 수정사항

제일 처음 텍스트 파일 내용을 읽어올 때 scanf로 읽어와 공백 문자를 읽지 못해 오류가 발생했다. 이를 고치기 위해, scanf로 읽어오는 방법이 아닌, gets로 읽어오는 방법을 택했다.

맨 처음 파일을 읽어올 때 pc value를 증가시키는 알고리즘을 택했는데, 이후 pc와 jumppc가 같을 때 명령어를 실행한다는 조건과 충돌이 있어서 명령어를 다 실행시킨 후에 pc value를 증가시키는 방안으로 수정했다.

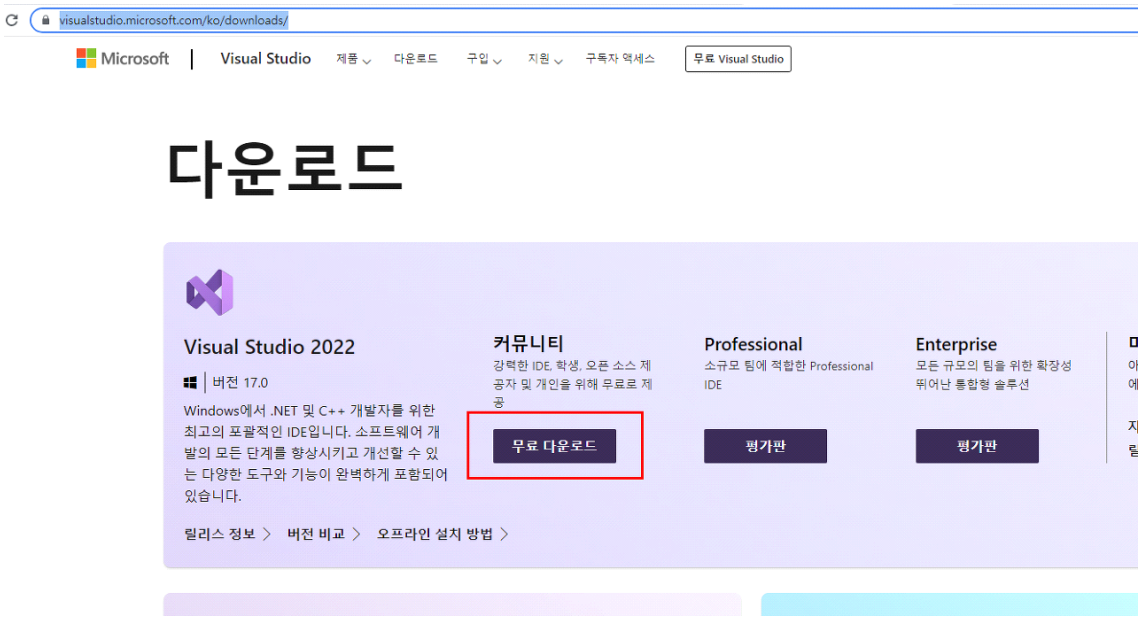
문자열을 비교할 때 '=='기호를 사용하여 비교하였으나, 이는 틀린 문법임을 깨닫고 문자열을 비교할 때 사용하는 함수인 strcmp를 사용했다. 그러나 오류가 계속 고쳐지지 않았고, 이 이유는 구조체의 op 크기는 10인 반면에 비교 문자의 크기는 3 또는 2라서 애초에 문자열이 같을 수 없음을 깨달았고, 지정된 크기만큼 문자열을 비교할 수 있는 strncmp 함수로 전체적으로 수정하였다.

16진수를 10진수로 변환하는 함수에서 그냥 숫자일 경우에는 dec = dec * 16 + hex[i]로 코드를 짰으나, 원하는 답이랑 다르게 나오는 것을 보고 숫자가 아니라 문자임을 깨닫게 되었다. 그 후 ASCII 코드를 이용하여 '0'의 값인 48을 빼서 정수형으로 변환시키는 데 성공했다.

2.17. 개발환경

본 노트북의 운영체제는 윈도우 10이고, 시스템은 64비트 운영체제이다. 코드는 Visual Studio 2022를 사용하였으며, 설치방법은 다음과 같다.

1. 다운로드 사이트 접속 후 다운로드



2. 설치할 항목 선택 후 설치 진행 : 내려받은 파일을 실행하면 설치할 항목들이 나타난다. 그 중, 맞는 항목을 선택 후 설치한다.
3. 이후 설치가 완료되면, 로그인하면 된다. 로그인 계정을 바로 사용하지 않을 때는 나중에 로그인 항목을 선택한다.

다음은 Visual Studio에서 프로그램을 실행하는 방법에 관해서 설명하겠다.

1. 해당 압축 폴더를 압축을 풀고, Visual Studio를 열어서 [새프로젝트 만들기] - [빈프로젝트]를 클릭한다.
2. Windows에서 프로젝트 폴더를 열어 c 파일과 txt 파일을 넣어준다. txt 파일과 c 파일이 같은 위치에 있어야 함을 유의한다.
3. 다시 생성한 프로젝트에 들어가 상단에 있는 [로컬 Windows 디버거]를 클릭하여 프로그램을 실행한다.

위와 같은 방법으로 파일을 실행하면, 디버그 콘솔에서 출력물을 볼 수 있다.

3. 결론

이렇게 총 270줄에 거쳐 mips simple calculator를 제작하게 되었다. 텍스트 파일의 내용을 읽을 전역 변수, 레지스터 값을 저장할 전역변수, instruction을 위한 구조체를 중심으로 메인 함수에서는 파일 입력 및 조건문, 그 밖의 함수에서는 각 명령어를 위한 코드가 있다.

init() 함수는 파일 입력을 담당하는 LINE 배열의 초기화, HextoDec() 함수에서는 16진수를 10진수로 변경해주는 역할을 수행하며, ADD, SUB, MUL, DIV는 각각의 사칙연산을 수행한다. MOV에서는 레지스터 값의 이동, LW에서는 16진수 값을 레지스터에 저장, SW는 출력, RST에서는 레지스터 초기화를 담당한다. JMP, BEQ, BNE에서는 각각의 조건이 참이라면 jump pc value를 반환하고 SLT에서는 조건에 맞게 레지스터에 값을 저장하는 역할을 했다.

처음 과제를 수행할 때 우리가 배웠던 MIPS 구조에 살짝씩 다른 점을 이해하지 못하여 헤매던 부분도 많았고, 분기를 어떻게 해야 하는지 몰라 코드를 짜는데 많은 시간을 보냈다. 마음을 잡고 MIPS 구조가 어떻게 되었는지, 어셈블리어가 어떻게 흘러가는지 고민하며 코드를 짜니 한층 수월해진 것 같다. 이번 과제를 하면서 단순 MIPS에 대해 잘 알게 된 것뿐만 아니라, 흐름이 어떻게 되는지도 많이 알아갔던 것 같다. 또한, 간만에 C언어로 프로그래밍을 하면서 파일 입출력, 문자열 변환 및 구조체에 대해 다시 복습할 좋은 기회였던 것 같다. 다만, 구조체를 배열로 선언하여 메모리를 낭비한 점이 아쉬웠고, 다음번엔 동적 할당과 node를 활용하여 메모리를 낭비하지 않는 쪽으로 구현하고 싶다.