

32213763 임석민

Introduction

txt 파일에 문자로 쓰인 MIPS 코드를 읽어 온 뒤 해당 문자로 표현된 MIPS 코드의 내용을 판단하고 그 내용에 따라서 해당 코드를 실행하고 그 결과 각 라인마다 변경된 레지스터의 값을 출력해야 하는 프로그램이다.

Background

1. Program concept

a. MIPS code & how MIPS works

[illegible]

가지며 이는 폰노이만 구조를 따른다고 생각할 수 있다. 그리고 mips는 파이프라인을 이용할 수 있는데 이는 IF, ID, EX, MEM, WB의 5단계에 따라 명령어를 읽고 해독하고 연산한 뒤 데이터 메모리에 저장하게 되는 과정을 가지고 있다.

2. Considerations

a. File read.

파일을 읽어 오는 데에는 여러가지 방법이 있다. 그래서 나는 어떤 방식을 사용할 지 고민하다가 한 줄 마다 읽고 실행하는 것이 입력 format을 생각하였을 때 fgets를 통하여 한 줄씩 읽고 처리하는 것이 입력형식이 정해져 있기에 좋을 것이라고 생각하였다.

b. variables

이제 input되는 operands값은 숫자 또는 레지스터이다. 따라서 이를 구분하고 숫자인 경우이면 문자열로 표현된 16진수를 숫자로 변환시키기 위해 strtol이라는 함수를 사용하기로 하였다. 그리고 register의 경우 어떤 레지스터를 사용할 것인지를 이식하기위해 r뒤에 오는 숫자에 해당하는 ascii값에서 0의 ascii값을 빼서 바로 숫자로 만들어 몇 번째 register인지 빠르게 확인하기로 하였다. 이렇게 두가지 형식의 변수를 한 번에 관리를 어떻게 할지에 대하여 고민하였고 이를 배열과 구조체를 사용하여 해결하였다.

c. Branches

Opcod에 따른 다양한 명령어를 수행해야 하기 때문에 이를 어떻게 분류하여 처리할 지에 대한 고민을 enum을 이용하여 해결하였다.

Implementation

1. Design & C code

```
6 struct instruction
7 {
8     enum opcodes opcode;
9     int jump; //target line num
10    int present; //present line num
11    int r[3]; //register's name
12    int registers[13]; //0-9 normal register 10-12 number to register
13 };
14
```

먼저 한 줄씩 txt파일로부터 읽어올 때 그 줄에 해당하는 opcode를 enum 값으로 저장하는 opcode 변수가 있고 그 아래로는 만약

jmp, beq, bne와 같은 다른 라인으로 넘어가야 할 때 목표가 되는 line number까지 몇 번 건너뛰어야 하는지의 값을 jump에 저장하게 된다. 그리고 현재 line number를 저장하기 위해 present라는 변수를 가지고 있다. 그리고 12번째 줄의 register array는 입력된 명령어에서의 register들의 역할을 하기위해 만들어졌다. 그리고 크기가 13인 이유는 0-9레지스터를 등록하고 만약 16진수 숫자가 들어올 경우 이를 숫자 그대로 활용하는 것이 아닌 레지스터에 저장하여 사용함으로써 피연산자들을 통일하여 연산 시 추가적인 코드들이 필요 없도록 프로그램을 만들 수 있어 10-12까지의 레지스터를 추가하였다. 마지막으로 r array는 opcode마다 필요한 operand의 값과 위치들이 다른데 이들의 위치를 저장하여 연산 시 어떤 레지스터들이 활용되는지 검사

할 필요없이 해당 명령어에 필요한 operands의 순서에 따라 0번부터 저장하여 실행 시 0부터 순서대로 연산을 진행하면 되도록 작성하였다.

```
16 enum opcodes
17 {
18     ADD = 0,
19     SUB,
20     MUL,
21     DIV,
22     MOV,
23     LW,
24     SW,
25     RST,
26     SLT,
27     BEQ,
28     BNE,
29     JMP
30 };
31
```

그 뒤로 나중에 switch state를 사용할 때 문자열을 case로 설정하지 못하기 때문에 이를 방지하고자 opcode를 enum type으로 변경하여 관리하기 용이함과 동시에 다른 곳에 사용하기 수월하도록 작성하였다.

```
54 int main() {
55     FILE* pFile = NULL;
56     const char* filename = "example.txt"; //filename
57     const char* mode = "r"; //file open type
58     char buffer[256];
59
60     //inst initialize
61     struct instruction* inst;
62     inst = malloc(sizeof(struct instruction)); //구조체 포인터
63     memset(inst->registers, -1, sizeof(int) * 13); //structure memory allocation
64     inst->jump = 0;
65     errno_t err = fopen_s(&pFile, filename, mode);
66
67     if (err != 0) {
68         // error handling
69     }
70
71     if (pFile == NULL) {
72         // error handling
73     }
74
75     while (fgetc(buffer, sizeof(buffer), pFile) != NULL) {
76
77         if (inst->jump != 0) {
78             inst->jump--;
79             printf("skip\n");
80             continue;
81         }
82
83         buffer[strlen(buffer) - 1] = '\0';
84         printf("%s", buffer);
85         inst->opcode = -1;
86         memset(inst->r, -1, sizeof(int) * 3);
87
88         printf("\n");
89         buffer2inst(inst, buffer);
90         process(inst);
91         printf("\n");
92     }
93
94     fclose(pFile);
95
96     return 0;
97 }
98
```

Main state은 파일 입출력을 위한 함수들과 파일이름과 파일 호출타입을 정의하고 txt파일과 읽을 때 이를 저장하기 위한 buffer array를 작성하였다. 그리고 그 아래에 if문은 만약 파일이 존재하지 않을 경우 오류를 걸러내기 위한 조건문들이다.

그 아래로는 txt파일에 있는 정보를 읽어 변수들로 변환하고 이 변수들을 저장할 구조체 instruction의 pointer로 변수를 설정하고 메모리에 할당해주었다.

while문이 실행되면서 본격적인 프로그램이 실행되는데 먼저 만약 jump가 필요한 경우 이를 건너뛰기 위한 조건문을 작성해놓았다.

그 뒤로 버퍼의 맨 마지막에는 개행문자가 있는 경우가 있어 출력 시 보기 불편하여 제거하였다. 이후 inst의 값들이 초기화 되는데 이는 새로운 명령어 set이 들어갈 경우 초기화 하지 않으면 잘못된 값을 연산할 수 있기에 매번 초기화를 시켰다. 그리고 buffer2inst와 process라는 함수를 거쳐 while문이 더 이상 읽어올 명령어가 없으면 끝나면 종료하게 된다.

```

100 void buffer2inst(struct instruction* inst, char* buffer) {
101
102     char* token = NULL, * saveptr = NULL;
103     char* inputArr[5];
104     char* opcodes[] = { "ADD", "SUB", "MUL", "DIV", "MOV", "LW", "SW", "RST", "SLT", "BEQ", "BNE", "JMP" };
105
106     int num_opcodes = sizeof(opcodes) / sizeof(opcodes[0]);
107     int seat = 0;
108
109     token = strtok_s(buffer, " ", &saveptr); //string passing
110
111     while (token != NULL) {
112         inputArr[seat] = token;
113         token = strtok_s(NULL, " ", &saveptr);
114         seat++;
115     }
116
117     inst->present = lineNum(inputArr[0]);
118
119     for (int i = 2; i < seat; i++)
120     {
121         if (inputArr[i][0] == '0') //if num
122         {
123             int number = (int)strtol(inputArr[i], NULL, 0);
124             inst->r[i - 2] = i - 2 + 10;
125             inst->registers[i - 2 + 10] = number;
126         }
127         else if (inputArr[i][0] == 'r') //if register
128             inst->r[i - 2] = inputArr[i][1] - 48;
129
130         else if (inputArr[i][0] == 's')
131             continue;
132
133         else // error
134         {
135             printf("error input is not correct");
136             return;
137         }
138     }
139
140     for (int i = 0; i < num_opcodes; i++) {
141         if (strcmp(inputArr[i], opcodes[i]) == 0)
142         {
143             inst->opcode = i;
144         }
145     }
146 }
147

```

Buffer2inst함수는 파일 입출력을 통해 버퍼에 받은 정보들을 연산하기 용이하게 instruction 구조에 맞게 설정하는 과정을 거치도록 하는 함수이다.

먼저 토큰화를 통해 ' '를 기준으로 나누고 이를 inputArr에 저장하게 된다. 이후 이를 input format에 맞게 가공하여 opcode는 enum형으로 숫자면 16진수를 표현한 문자열을 숫자로 변경시킨 뒤 10-12사이의 레지스터에 저장하고 그 위치를 연산 순서에 알맞은 자리에 해당하는 r array에 숫자를 저장한 register의 위치를

저장한다. 그리고 register의 경우 register가 연산 될 위치의 r값에 register의 번호를 저장하게 된다. 그리고 입력된 값이 숫자나 레지스터 이외의 값이라면 오류를 출력한다.

```

148 void process(struct instruction* inst)
149 {
150     int orig = inst->registers[inst->r[0]];
151
152     //문자로 실행
153     switch (inst->opcode)
154     {
155         case 0:
156             add(inst);
157             printf("ADD r[%d]: %d -> %d\n", inst->r[0], orig, inst->registers[inst->r[0]]);
158             break;
159         case 1:
160             sub(inst);
161             printf("SUB r[%d]: %d -> %d\n", inst->r[0], orig, inst->registers[inst->r[0]]);
162             break;
163         case 2:
164             mul(inst);
165             printf("MUL r[%d]: %d -> %d\n", inst->r[0], orig, inst->registers[inst->r[0]]);
166             break;
167         case 3:
168             div(inst);
169             printf("DIV r[%d]: %d -> %d\n", inst->r[0], orig, inst->registers[inst->r[0]]);
170             break;
171         case 4:
172             mov(inst);
173             printf("MOV r[%d]: %d -> %d\n", inst->r[0], orig, inst->registers[inst->r[0]]);
174             break;
175         case 5:
176             lw(inst);
177             printf("LW r[%d]: %d -> %d\n", inst->r[0], orig, inst->registers[inst->r[0]]);
178             break;
179         case 6:
180             sw(inst);
181             printf("SW r[%d]: %d -> %d\n", inst->r[0], inst->registers[inst->r[0]]);
182             break;
183         case 7:
184             for (int i = 0; i < 10; i++)
185             {
186                 if (inst->registers[i] != -1) {
187                     printf("r[%d]: %d -> %d\n", i, inst->registers[i], -1);
188                 }
189             }
190             printf("\n");
191             rst(inst);
192             break;
193         case 8:
194             slt(inst);
195             printf("SLT r[%d]: %d -> %d\n", inst->r[0], orig, inst->registers[inst->r[0]]);
196             break;
197         case 9:
198             beq(inst);
199             break;
200         case 10:
201             bne(inst);
202             break;
203         case 11:
204             jmp(inst);
205             break;
206         default:
207             printf("invalid opcode");
208             break;
209     }
210 }
211
212

```

instruction structure을 pointer로 가져와 opcode로 분기하여 opcode에 해당하는 함수를 실행시키는 함수이다. 이때 앞서말한 enum형이 사용되어 간편하게 분류할 수 있다. 그리고 변하는 레지스터 값을 출력할 수 있도록 연산이 저장될 레지스터의 값을 origin에 저장하여 레지스터의 연산 전의 값과 연산 후의 값을 출력이 가능하도록 하였다. 그리고 만약 잘못된 opcode가 입력 된 경우 -1로 유지되어 dealt로 분기되어 오류임을 확인 할 수 있도록 하였다.

```

215 int lineNum(char* input)
216 {
217     int line = 0;
218     int i = 0;
219
220     while (input[i] != '\0')
221     {
222         i++;
223     }
224
225     i--;
226
227     for (int j = 0; i >= 0; i--, j++)
228     {
229         line += (input[i] - 48) * (int)pow(10, j);
230     }
231
232     return line;
233 }
234

```

이 함수는 입출력시 line number를 int type으로 변환시켜 return해주는 함수이다. 이를 이용하여 jmp, beq, bne같은 jump하는 경우가 발생하면 현재 프로그램의 실행되는 위치를 인지하여 target과 나의 위치를 연산하여 jump하는 코드를 적성하기 편해지도록 하였다.

```

236 void add(struct instruction* inst)
237 {
238     inst->registers[inst->r[0]] = inst->registers[inst->r[1]] + inst->registers[inst->r[2]];
239 }
240 void sub(struct instruction* inst) {
241     inst->registers[inst->r[0]] = inst->registers[inst->r[1]] - inst->registers[inst->r[2]];
242 }
243 void mul(struct instruction* inst) {
244     inst->registers[inst->r[0]] = inst->registers[inst->r[1]] * inst->registers[inst->r[2]];
245 }
246 void div(struct instruction* inst) {
247     inst->registers[inst->r[0]] = inst->registers[inst->r[1]] / inst->registers[inst->r[2]];
248 }
249 void mov(struct instruction* inst) {
250     inst->registers[inst->r[0]] = inst->registers[inst->r[1]];
251 }
252 void lw(struct instruction* inst) {
253     inst->registers[inst->r[0]] = inst->registers[inst->r[1]];
254 }
255 void sw(struct instruction* inst) {
256 }
257 void rst(struct instruction* inst) {
258     memset(inst->registers, -1, sizeof(int) * 13);
259 }
260 void slt(struct instruction* inst) {
261     if (inst->registers[inst->r[1]] < inst->registers[inst->r[2]])
262         inst->registers[inst->r[0]] = 1;
263     else
264         inst->registers[inst->r[0]] = 0;
265 }
266 void beq(struct instruction* inst) {
267     if (inst->registers[inst->r[0]] == inst->registers[inst->r[1]]) {
268         inst->jump = inst->registers[inst->r[2]] - inst->present - 1;
269         printf("BEQ\n");
270     }
271     else
272         printf("skip BEQ\n");
273 }
274 void bne(struct instruction* inst) {
275     if (inst->registers[inst->r[0]] != inst->registers[inst->r[1]]) {
276         inst->jump = inst->registers[inst->r[2]] - inst->present - 1;
277         printf("BNE\n");
278     }
279     else
280         printf("skip BNE\n");
281 }
282 void jmp(struct instruction* inst) {
283     printf("JMP\n");
284     inst->jump = inst->registers[inst->r[0]] - inst->present - 1;
285 }

```

이 코드는 instruction pointer를 통해 instruction structure 내에 있는 레지스터들을 역참조하여 연산을 시행하고 다시 지정된 레지스터에 저장을 하는 함수들이다. 앞에서 말했듯이 모든 피연산자를 배열에 저장하여 같은 방식으로 연산 할 수 있게 되어서 사칙연산과 LW, SW, MOV, RST는 뜻에 직관적으로 실행 가능하도록 하였다. 그런데 여기서 rst를 -1로 초기화 하는 경우는 값이 바뀐 것을 인지해야 하는데 0으로 초기화 할 경우 입력 값이 0이면 값이 들어왔는지 아닌지 인지할 수 없기에 -1로 초기화 하였다. 그리고 jmp, beq, bne와 같은 함수 들은 jump 값을 저장하도록 작성했고 SLT도 pdf에 나온 대로 작성하였다.

Environment

1. Build environment.

Window 11, visual studio 2022 17.4.0에서 작성됨.

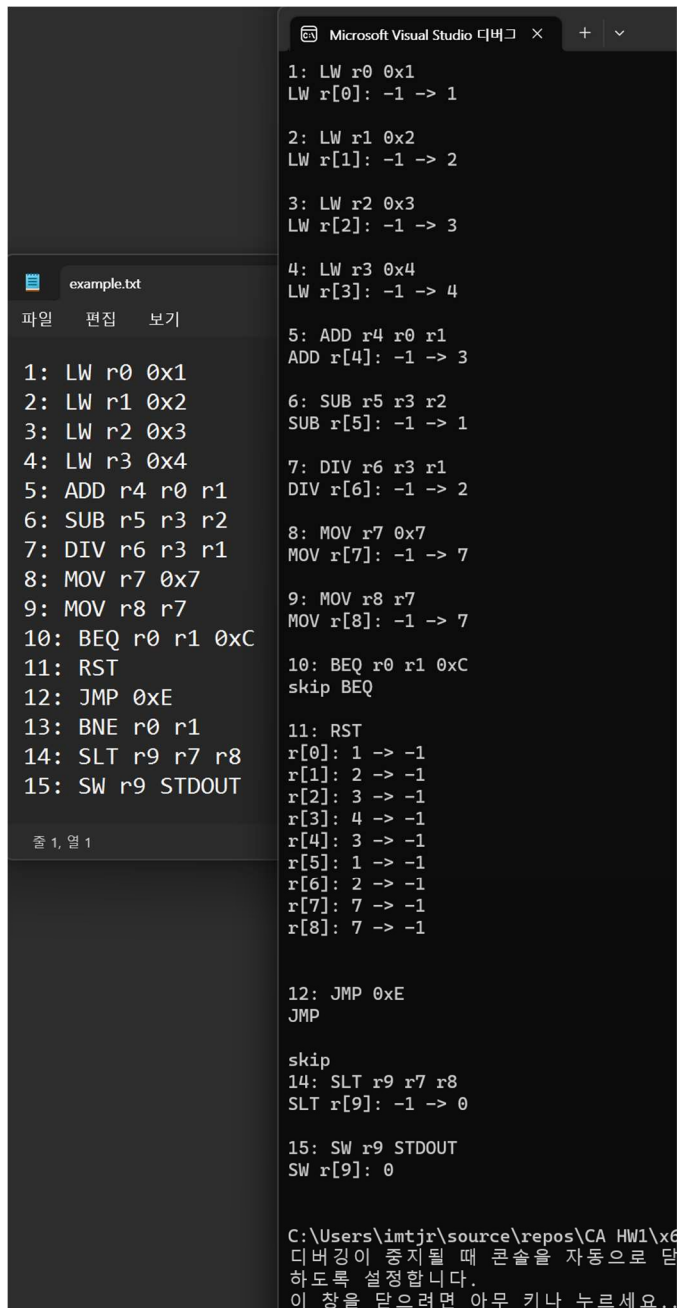
2. How to compile and run

CA HW1 file에 있는 example.txt file에 본인이 실행시키고자 하는 mips code를

Line number: opcode oprd1 oprd2 oprd3

위와 같은 형식으로 작성한뒤 저장하고 visual studio 2022에서 f5를 눌러 실행시키면 결과가 출력된다.

3. Working Program Screen shots



```
Microsoft Visual Studio 디버그 x + v

1: LW r0 0x1
LW r[0]: -1 -> 1

2: LW r1 0x2
LW r[1]: -1 -> 2

3: LW r2 0x3
LW r[2]: -1 -> 3

4: LW r3 0x4
LW r[3]: -1 -> 4

5: ADD r4 r0 r1
ADD r[4]: -1 -> 3

6: SUB r5 r3 r2
SUB r[5]: -1 -> 1

7: DIV r6 r3 r1
DIV r[6]: -1 -> 2

8: MOV r7 0x7
MOV r[7]: -1 -> 7

9: MOV r8 r7
MOV r[8]: -1 -> 7

10: BEQ r0 r1 0xC
10: BEQ r0 r1 0xC
skip BEQ

11: RST
r[0]: 1 -> -1
r[1]: 2 -> -1
r[2]: 3 -> -1
r[3]: 4 -> -1
r[4]: 3 -> -1
r[5]: 1 -> -1
r[6]: 2 -> -1
r[7]: 7 -> -1
r[8]: 7 -> -1

12: JMP 0xE
JMP

skip
14: SLT r9 r7 r8
SLT r[9]: -1 -> 0

15: SW r9 STDOUT
SW r[9]: 0

C:\Users\imtjr\source\repos\CA HW1\src\main.c
디버깅이 중지될 때 콘솔을 자동으로 닫
하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...
```

프로그램을 실행시키면 다음과같이 실행이 되는데 먼저 프로그램이 읽어드린 명령어를 출력하고 그 다음에 변화를 출력하게 된다.

옆의 예시를 설명하면 1-4까지는 lw로 읽어드린 명령어를 잘 숫자로 변환하여 해당하는 명령어를 실행하였다.

5번째 줄은 r0와 r1의 합을 r4에 저장하는 것으로 r1, r0는 1과 2임으로 올바르게 실행되었다. 그리고 나머지 사칙연산들도 정확하게 작동하였다.

Beq는 값이 같아야 점프를 실행하는데 값이 달라서 skip되었다.

Rst은 다시 값들을 -1로 초기화 하였다.

Jmp도 14번째 줄로 잘 점프했다.

slt도 두값을 비교하여 값을 잘 바꿔내는 것을 확인할 수 있다.

마지막으로 SW는 출력하는 것으로 해당 레지스터 값을 잘 출력해 낸을 확인할 수 있다.

Lesson

이번 과제를 통해 미리 프로그램이 어떻게 작동되고 동작할지 생각을 한 뒤에 프로그램을 작성해보았는데 생각하는 과정을 통해 좀 더 필요한 기능들을 구체적으로 떠올리게 되고 진짜 필요한 기능들을 찾으려고 하니 효율적으로 코드를 작성할 수 있었다. 그리고 작년에 배웠던 mips코드를 다시 보고 이에 대한 코드를 작성 해봄으로써 mips를 복습해보는 좋은 기회였던 것 같다.