

# Computer Architecture Project 1 : Simple MIPS assembler

201911106 유제이

## 1. Files

Assembler Program의 소스파일은 다음과 같다.

### 1) assembler.cpp

: assembler.cpp는 main function이 있는 소스파일로, 프로그램 작동을 위해서는 해당 파일을 컴파일 및 실행해야 한다.

### 2) Directive.h

: Directive.h는 assembler.cpp가 include하는 헤더파일로, 읽어들이는 file의 정보를 Directive, Words, Instruction class의 instance로 저장한다.

## 2. Compile and Execution

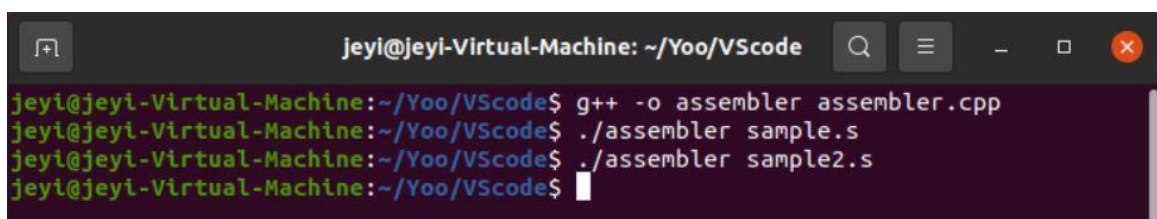
해당 코드는 Window hyper-V를 이용한 가상 컴퓨터 내 Ubuntu 20.04 환경에서 실행하였다. 컴파일 및 실행 방법은 아래와 같다.

### 1) Compile

assembler.cpp, Directive.h, sample.s, sample2.s 등 실행 및 읽기에 관련된 파일이 있는 Directory에서 Open in terminal -> `g++ -o assembler assembler.cpp` 입력

### 2) Execution

Compile 후 `./assembler sample.s` 또는 `./assembler sample2.s` 입력 (runfile\_name : assembler로 지정)



```
jeyi@jeyi-Virtual-Machine: ~/Yoo/VScode
jeyi@jeyi-Virtual-Machine:~/Yoo/VScode$ g++ -o assembler assembler.cpp
jeyi@jeyi-Virtual-Machine:~/Yoo/VScode$ ./assembler sample.s
jeyi@jeyi-Virtual-Machine:~/Yoo/VScode$ ./assembler sample2.s
jeyi@jeyi-Virtual-Machine:~/Yoo/VScode$
```

## 3. Functions and Flow

각 파일에 들어있는 function들은 아래와 같은 역할을 수행한다. (1은 assembler.cpp, 2는 Directive.h에 들어있는 function임을 의미, 초록색으로 표기된 함수는 다른 function에서 이용하기 위한 함수다.

### 1-a) int main (int argc, char\* argv[])

: 해당 main function은 int argc와 char\* argv[]를 인자로 받는다. 리눅스 terminal에서 실행할 file명을 받으면 argv[]에 저장되고, 이를 string filename으로 받는다.

string filename은 ① `vector<Directive> readFile(string file)`로 전달, ② filename에서 "s"를 지우고 "o"를 추가

해 새롭게 assembler를 거친 데이터를 저장할 파일의 이름으로 바뀐다.

이후 ①에서 return 된 값을 `vector<Directive> datas`에 저장, 이를 ③ `void writeFile(vector<Directive> datas, string filename)`에 parameter로 넘겨주며 출력 파일을 만든다.

### 1-b) `vector<Directive> readFile(string file)`

- ① file name (확장자 포함)을 가져와 파일을 한 줄씩 읽어 (ifstream, getline() 이용) vector에 저장
- ② 저장된 line들에 접근해 이를 단어 단위로 쪼개어 (stringstream 이용) #lines \* 5 array에 저장
- ③ 이때 각 line에 적힌 contents들의 directive (word, type, just label or null line etc.)에 따라 각각에 맞는 instance를 만든다 (Directive.h 활용). "la"의 변환 등도 이 단계에서 이루어지며, 생성된 instance들은 `vector<Directive>`에 저장한다. 이 벡터는 이후 return된다.

(stringstream을 이용하여 2D array에 단어들을 저장할 때, label:의 유무 등에 따라 값이 당겨져서 저장되는 경우가 있어 이를 구분하며 다루었다. Directive, 빈 줄 혹은 label만 있는 line의 경우에도 따로 처리를 해주어 return하는 vector에 저장되지 않게 하였다.)

### 1-c) `void writeFile(vector<Directive> datas, string filename)`

- ① file에 저장되어있던 data, text들의 정보가 담긴 Directive type instance들을 가져와 읽는다.
- ② Word의 section size 및 각 value를 읽어온다.
- ③ Instruction의 Format (R, I, J)에 따라서 opcode, rs, rt, rd, shamt, funct, immediate, jump target 각각의 변수에 값을 저장 – binary로 변환 – Hexadecimal로 변환한다. 변환된 값은 다시 `vector<string>`에 저장한다. 이때 Instruction section size의 계산도 이루어진다. ("la"변환의 경우는 변환 이전의 instruction size, 즉 la instruction 한 개에 대해서만 고려하여 size를 계산했다.)
- ④ text size, word size, instructions, value 등을 출력 파일에 저장한다.

**1-d) `bool isln(string str1, string r1)`** : 저장된 단어들에 특정 문자열이 있는지 확인한다. ① "."가 있는 경우는 label로, "\$"가 있는 경우 register로, ","가 있는 경우는 뒤에 더 variable이 온다는 것을 확인함으로써 정보 저장의 case 분류 용도로 사용한 함수이다.

**1-e) `bool isInt(string str0)`** : 이는 string type으로 표기된 수의 표현이 10진법인지 16진법인지를 구분하는 함수이다. 이를 통해 hexadecimal과 decimal을 구분하고 각 표현법을 지원하는 assembler를 구축할 수 있다.

**1-f) `string pureWord(string str2, string r2)`** : 앞서 isln() 함수를 보면 알 수 있듯, 저장된 데이터값에는 다른 문자열이 섞여있다. 따라서 이를 다른 특수 기호 등이 없는 순수 형태로 변환해주어야 했다. 이 함수는 문자열에서 특수 기호를 제거하여 데이터를 정리하는 데에 사용하였다.

**1-g) `long long setToDec(string v)`** : string type의 수를 10진법으로 나타내도록 한다. 프로그램 내에서 데이터를 다룰 때 (ex. Address calculation 등) 10진법으로 일괄 데이터를 통일한 후 저장시에만 16진법으로 표현되게끔 하였다.

**1-h) `string dToh(int decimal)`** : 10진법의 수를 16진법의 string으로 변환하는 함수로, 16진법 수로써 다루어

저야하나 10진법으로 오해받는 데이터들의 처리를 위해 사용하였다. 특히 "la" operation이 상/하위 address에 따라 변환이 달라지게끔 처리하는 과정에서 la가 가리키는 label의 address를 가져올 때 이를 사용하였다. (코드에서는 수 데이터를 10진법으로 통일하여 활용하였기 때문에 16진법의 address를 확인해야하는 위의 경우에는 따로 이와 같은 처리를 해주어야했다.)

**1-i) string btoH(string bits)** : instruction을 표현하는 각 bit들을 16진법 표기로 변환시켜준다. 이를 활용하여 instruction의 정보를 최종적인 형태로 변환하였다.

**1-j) string intoBit(long op, int rs, int rt, int rd, int sh, long f, long long imm, long long target, string type)** : Instruction의 detail들을 참조하여 bit로 만든다. (bitset 이용)

**1-k) int registerNum(string res)** : "\$n"과 같이 표기된 register에 대한 정보를 해당 정수로 반환해주는 함수로, rs, rt, rd 등을 처리할 때 사용하였다.

## 2-a) Class Directive (Base)

Words, Instruction class의 Base class로 label, value, address 등 전반적인 data의 요소들을 인자로 가지고 있다. 이들은 private으로 직접 접근은 불가하고, public 함수 중 get()함수들을 통해 읽어올 수 있다.

파일의 Word, Instruction 등 각기 다른 정보들을 한 번에 간추려 가져오기 위해 Upcasting을 하는 용도로 사용한다.

## 2-b) Words (Child 1)

label, value, address 3개를 인자로 받아 construct되며, .word의 directive를 가진 데이터들이 해당 class instance를 만든다.

## 2-c) Instruction (Child 2)

label, instruction, address 3개를 인자로 받아 construct되며, .text의 directive를 가진 데이터들이 해당 class instance를 만든다. 이 때 instruction (vector<string> type)의 0번 index를 가진 값 (operation)을 가져와 이를 mapping하여 op code를 통해 instruction format이 어느 것인지를 내부적으로 update한다.

각 instruction의 이름과 op code (R format의 경우 func도 함께 묶어야 하므로 vector 형태로 value값을 할당)를 확인할 수 있는 map은 Directive.h 내 global scope로 선언해두었으며, Instruction class의 경우 이 map을 불러올 수 있는 getMap() 함수를 통해 assembler.cpp에서도 일반 instruction instance를 임의로 선언하게 되면 이를 통해 해당 map에 접근할 수 있게 하였다.

**2-d) isR(string fstElmt, map<string, vector<string>> ma)** : map에 저장된 vector<string>의 size를 체크함으로써 먼저 R format인지를 확인하는 함수이다.

전체적인 Flow를 정리하면 다음과 같다. (편의상 이 파트에서는 func\_name() 만 기재한다.)

① **readFile()** 통해 sample.s 내 각 data 정보 저장 (label, directive, value / instruction, address 등)

② directive 따른 각 class instance 생성 및 vector 구조체에 저장 (upcasting)

③ **writeFile()**에서 vector 내 저장 정보를 참조해 sample.o에 저장될 형태로 데이터 변환 및 출력파일 생성