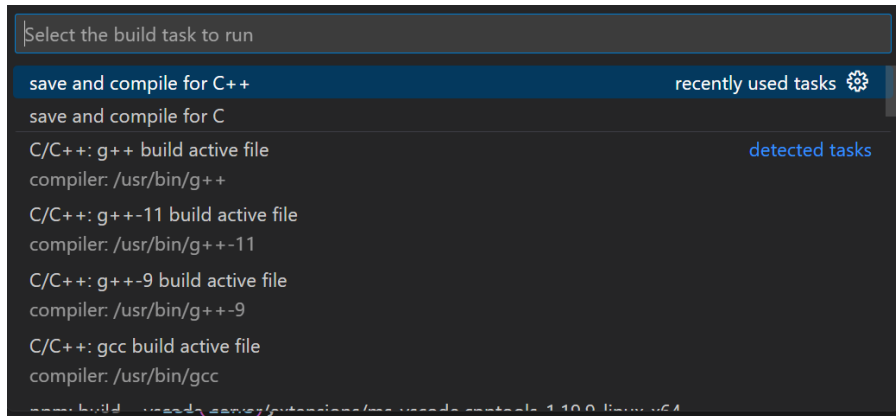


Project 1: Simple MIPS assembler

Submitter: Jiwon Jang (202211167)

1. 컴파일 및 실행 방법

언어는 C++를 사용하였고, 컴파일러는 g++ 9.5.0을 이용하였다. 코딩과 컴파일은 비주얼 라이징을 위해서 VSCode에 WSL을 연결하여 진행했다.



이 방법 외에도 우분투에 해당 명령어를 쳐서 컴파일할 수도 있다.

```
g++ -o homework1 homework1.cpp
```

이후 파일 실행은 다음과 같이 진행했다. `./<runfile> assembly file` 순서로 입력하면, 프로그램이 assembly file을 읽어와 같은 이름의 binary file을 생성해준다. 이후 생성된 파일은 vim editor를 통해서 확인해 보았다. 생성된 파일 결과는 밑에서 확인해 볼 것이다.

```
jang] iwon@DESKTOP-M4HGQUK:~/CompStruct$ ./homework1 sample.s
jang] iwon@DESKTOP-M4HGQUK:~/CompStruct$ vim sample.o
jang] iwon@DESKTOP-M4HGQUK:~/CompStruct$
```

2. 클래스 및 함수 설명

이번 과제에서 구현한 클래스와 함수에 대한 설명이다. 코드의 흐름을 설명하기 위한 배경 설명이다.

<library>

총 6개의 library를 이용하였다.

iostream: 표준 입출력을 지원한다.

fstream: 파일 생성 및 read/write를 지원한다.

string: 문자열 형식을 지원한다.

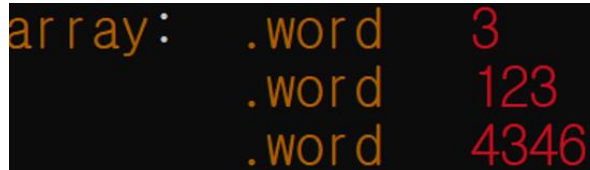
vector: vector 자료형을 지원한다.

bitset: 이진수 변환을 지원한다.

regex: 정규 표현식을 지원한다.

<class>

class는 각 label과 data, 그리고 text의 정보를 저장하기 위해 사용하였다. 예를 들어 코드가 밑 사진의 array: 라벨을 읽게 된다면, 'array' 라벨에 대한 DATA_LABEL class의 인스턴스를 만들게 된다. 나머지 클래스에 대해서도 마찬가지로 동작한다. 결론적으로 코드가 파일을 모두 읽고 나면, 모든 label과 data, 그리고 text에 대해 인스턴스가 생기게 된다. 클래스를 만든 이유는 label과 label에 속해있는 data, 그리고 text들을 더 효율적으로 관리하기 위함이다.



1) DATA_LABEL class definition

private:

int address: 해당 라벨의 주소가 담겨있다.

vector<DATA> data_list:* 해당 라벨에 속해 있는 data들이 담겨있는 리스트이다.

string line: 해당 라벨이 속해있는 '줄'이 담겨 있다.

public:

DATA_LABEL(string line): DATA_LABEL 클래스의 constructor이다. constructor에서는 주소를 설정을 진행한다. 주소는 '전 라벨의 주소 + 전 라벨의 data_list 크기*4'를 이용해서 구한다.

static vector<DATA_LABEL>data_label_list:* DATA_LABEL 클래스에 속해있는 data들이 담겨있는 리스트이다. static이다.

*void PushData(DATA *DATA):* 해당 라벨에 속해 있는 data들을 data_list에 담기 위한 함수이다.

vector<DATA> GetDataList():* 해당 라벨에 속해 있는 data들이 담겨 있는 data_list를 반환하는 함수이다.

int GetAddress(): data의 주소를 얻기 위한 함수이다.

string GetLine(): data가 속해있는 '줄'을 얻기 위한 함수이다.

static int forward_address: 전에 나온 라벨의 주소가 담겨 있다.

2) TEXT_LABEL class definition

private:

int address: 해당 라벨의 주소가 담겨있다.

vector<TEXT> text_list:* 해당 라벨에 속해 있는 text들이 담겨있는 리스트이다.

string line: 해당 라벨이 속해있는 '줄'이 담겨 있다.

public:

TEXT_LABEL(string line): TEXT_LABEL 클래스의 constructor이다. constructor에서는 주소를 설정

정을 진행한다. 주소는 '전 라벨의 주소 + 전 라벨의 text_list 크기*4'를 이용해서 구한다.

static vector<TEXT_LABEL>text_label_list*: TEXT_LABEL 클래스의 인스턴스들이 담겨있는 리스트이다. static이다.

*void PushText(TEXT *TEXT)*: 해당 라벨에 속해 있는 text들을 TEXT_LABEL 클래스의 text_list에 담기 위한 함수이다.

vector<TEXT> GetTextList()*: 해당 라벨에 속해 있는 text들이 담겨있는 text_list를 반환하는 함수이다.

int GetAddress(): label의 주소를 얻기 위한 함수이다.

string GetLine(): label이 속해있는 '줄'을 얻기 위한 함수이다.

static int forward_address: 전에 나온 라벨의 주소가 담겨 있다.

3) DATA class definition

private:

int address: 해당 data의 주소가 담겨있다.

int data: 해당 data의 값이 담겨있다.

public:

DATA(int data): DATA 클래스의 constructor이다. constructor에서는 주소를 설정하고, section의 갯수를 count한다.

static int count_datasec: section의 수를 count하기 위한 변수이다. static이다.

int GetData(): DATA 클래스 인스턴스에 담겨있는 data를 가져오기 위한 함수이다.

int GetAddress: data의 주소를 가져오기 위한 함수이다.

4) TEXT class definition

private:

int address: 해당 text의 주소가 담겨있다.

int data: 해당 text의 값이 담겨있다.

public:

TEXT(int text): TEXT 클래스의 constructor이다. constructor에서는 주소를 설정하고, section의 갯수를 count한다.

static int count_datasec: section의 수를 count하기 위한 변수이다. static이다.

int GetText(): TEXT 클래스 인스턴스에 담겨있는 text를 가져오기 위한 함수이다.

int GetAddress: text의 주소를 가져오기 위한 함수이다.

<other functions>

main 함수를 제외한 함수의 종류는 크게 세 가지로 구분할 수 있다.

진수 변환을 하는 함수들: *Hexadecimaler(string num)*, *Binaryer(int num, int size)*

각 Instruction의 detail한 부분을 다루는 함수들: *addiu(string line)*, *addu(string line)...*etc...

Assembly 파일을 읽고, 최종 출력을 하는 함수들: *InduceInstance(string line)*, *Assembler()*

조금 더 자세히 알아보자.

1) 진수 변환을 하는 함수들

Hexadecimaler(string num): 2진수를 16진수로 변환하는 함수이다. 이 때 16진수의 자리를 고정하지 않는다.

Binaryer(int num, int size): 10진수를 2진수로 변환하는 함수이다. bitset 라이브러리를 사용하였다.

2) 각 Instruction의 detail한 부분을 다루는 함수들

이 함수들은 과제 pdf의 instruction detail을 바탕으로 구현하였다. 각 instruction 마다 detail에서 요구하는 형식으로 binary string을 리턴한다.

3) Assembly 파일을 읽고, 최종 출력을 하는 함수들

InduceInstance(string line): main 함수에서 파일을 읽는 동안 각 line을 한 줄씩 받아온다. 이후 각 line마다 타입에 맞는 클래스의 인스턴스를 생성한다. 또 각 label과 그 label의 하위 data나 text들을 관계 지어준다. 예를 들어 밑과 같은 assembly 코드가 있다고 생각해보자.

```
array2: .word    0x12345678
        .word    0xFFFFFFFF
```

그럼 *InduceInstance* 함수에서는 array2에 대한 DATA_LABEL class의 인스턴스와 0x12345678, 0xffffffff에 대한 DATA class의 인스턴스, 총 세 개의 인스턴스를 생성한다. 또한 위 코드는 array2라는 라벨에 0x12345678, 0xffffffff의 두 개의 데이터가 위치해 있는 형태이므로 array2 인스턴스의 data_list 배열에 위 데이터들을 추가시켜 줌으로써 세 데이터의 관계를 지어준다(array2의 하위 데이터로 0x12345678, 0xffffffff이 위치하도록). 위 함수에서는 위 과정을 모든 줄에 대해 수행한다.

Assembler(): *InduceInstance* 함수를 실행하고 나면, 데이터, 텍스트 라벨이 담겨있는 리스트를 얻을 수 있다. *Assembler()* 함수에서는 이 중 텍스트 라벨이 담겨 있는 리스트를 iterate 하며 각 텍스트 라벨의 하위 명령에 접근하고, 그 명령들을 수행한다. 예를 들어 밑과 같은 assembly 코드가 있다고 생각해보자.

```

main:
    la $8, var
    lw $9, 0($8)
    addu $2, $0, $9
    jal sum
    j exit

sum: sltiu $1, $2, 1
    bne $1, $0, sum_exit
    addu $3, $3, $2
    addiu $2, $2, -1
    j sum
sum_exit:
    addu $4, $3, $0
    jr $31

```

InduceInstance 함수가 실행되고 나면 [main, sum, sum_exit]와 같은 text label이 담겨있는 리스트가 생성된다 (실제로는 *TEXT_LABEL 타입의 인스턴스들이 리스트의 원소로써 자리하고 있을 것이다). 그럼 Assembler 함수에서는 이 리스트를 iterate하며 각 라벨에 속해있는 명령들에 접근한다. 첫 번째로 Assembler 함수는 main label의 인스턴스에 접근할 것이다. main label의 인스턴스에는 la, lw.. 등의 명령이 담겨있는 text_list가 있을 것이고, Assembler 함수에서는 이 text_list 리스트를 iterate하며 각 명령에 모두 접근한다. 다른 라벨에 대해서도 마찬가지로 과정을 거치며 최종적으로는 모든 명령에 대한 Hex output을 생성한다.

<main function>

main 함수에서는 파일을 읽고, 파일을 쓰는 작업만을 진행한다.

3. 코드 흐름

코드는

- 1) main 함수를 통한 파일 read
- 2) induceinstance 함수를 통한 label, data, text 인스턴스 생성 및 text, data 별 label 리스트 생성, 각 label의 하위 data, text에 대한 리스트 생성
- 3) induceinstance 함수를 통해 생성한 리스트를 바탕으로, assembler 함수를 통한 각 instruction 접근 및 최종 출력될 문자열 반환

이 과정으로 실행된다.

4. 결과

결과는 다음과 같다.

- 1) sample.s

```

jlangjiwon@DESKTOP-M4HGQUK: ~/CompStruct
-
        .data
array:   .word    3
        .word    123
        .word    4346
array2:  .word    0x12345678
        .word    0xFFFFFFFF
        .text
main:
        addiu    $2, $0, 1024
        addu     $3, $2, $2
        or       $4, $3, $2
        addiu    $5, $0, 1234
        sll      $6, $5, 16
        addiu    $7, $6, 9999
        subu     $8, $7, $2
        nor      $9, $4, $3
        ori      $10, $2, 255
        srl      $11, $6, 5
        srl      $12, $6, 4
        la       $4, array2
        lb       $2, 1($4)
        sb       $2, 6($4)
        and      $13, $11, $5
        andi     $14, $4, 100
        subu     $15, $0, $10

```

```

jlangjiwon@DESKTOP-M4HGQUK: ~/CompStruct
0x50
0x14
0x24020400
0x421821
0x622025
0x240504d2
0x53400
0x24c7270f
0xe24023
0x834827
0x344a00ff
0x65942
0x66102
0x3c041000
0x3484000c
0x80820001
0xa0820006
0x1656824
0x308e0064
0xa7823
0x3c110064
0x2402000a
0x3
0x7b
0x10fa
0x12345678
0xffffffff
~

```

2) sample2.s

```

jlangjiwon@DESKTOP-M4HGQUK: ~/CompStruct
-
        .data
var:     .word    5
        .text
main:
        la       $3, var
        lw       $9, 0($8)
        addu     $2, $0, $9
        jal      sum
        j        exit

sum:     sltiu    $1, $2, 1
        bne     $1, $0, sum_exit
        addu     $3, $3, $2
        addiu    $2, $2, -1
        j        sum
sum_exit:
        addu     $4, $3, $0
        jr      $31
exit:

```

```

jlangjiwon@DESKTOP-M4HGQUK: ~/CompStruct
0x30
0x4
0x3c081000
0x8d090000
0x91021
0xc100005
0x810000c
0x2c410001
0x14200003
0x621821
0x2442ffff
0x8100005
0x602021
0x3e00008
0x5
~

```