Project 1: Simple MIPS assembler Report

201811118 이 구

1. Code Flow

1.1. 구현

MIPS assembler는 크게 4번의 순차적인 loop를 통해 동작하도록 구현되었다. Data, Label struct를 이용하여 각각 data section의 label 정보(이름, 주소, 값)와 text section의 label 정보(이름, 주소)를 관리하였다.

첫 번째 loop에서 수행하는 기능은 아래와 같다.

1) 입력받은 input 파일을 한 줄씩 읽은 후, '\n', '\t'를 기준으로 word 단위로 parsing 한다.
2) word에서 불필요한 정보를 나타내는 문자 (ex. '$', ',' ':')를 제거한다.
3) 만약 'offset($register)' 형태의 string이 존재한다면, 'offset'과 'register'로 나누어 각각 저장한다.
4) Data/Text section의 정보를 각각 Data, Label struct로 만들어 저장한다.
5) text section을 parsing한 정보들을 vector<vector<string>> 형태로 저장한다.
6) data section에 존재하는 전체 data의 수를 카운팅 한다.

두 번째 loop에서 수행하는 기능은 아래와 같다.

1) 위에서 저장한 data, text section의 label address를 기준으로, la가 'lui'로 바뀔 것인지, 'lui + ori'로 바뀔 것인지를 고려한 program counter값을 계산한다. 이때, 실제 저장되어 있는 'la'가 포함된 instruction은 바뀌지 않는다.
2) 위에서 계산한 program counter 값을 기반으로, text section에 위치하는 label의 주소를 update 한다.
3) 주어진 instruction 중 data section에 위치한 label이 존재한다면, 그 값은 la와 관계없이 변하지 않는 값이므로, 첫 번째 loop에서 저장한 data의 address로 변경한다.

세 번째 루프에서 수행하는 기능은 아래와 같다.

1) 주어진 instruction 중, text section에 위치한 label이 존재한다면, 그 값을 label의 address로 변경한다.
2) Label의 주소를 고려하여, la 명령어를 'lui' 또는 'lui + ori' instruction으로 바꾸어 준다. 이때 실제로 la 명령어가 포함된 명령어를 지우고, 새롭게 lui, ori instruction을 추가한다.
3) 위의 경우를 고려하여, text section에 존재하는 전체 instruction의 수를 카운팅한다.

네 번째 루프에서 수행하는 기능은 아래와 같다.

1) text section에 존재하는 전체 instruction의 수, data section에 존재하는 전체 data의 수를 이용하여 각각의 영역(text, data section)의 크기를 계산하여 output file에 출력한다.
2) 위 세 개의 루프를 통해 얻은 정보를 이용하여, R, I, J format의 32-bit instruction을 계산 및 output file에 출력한다.
3) text section에 존재하는 data들을 output file에 출력한다.

이때, R, I, J format의 instruction을 만드는 과정은 shift operator와 plus operation 을 이용하여 구현하였다. I-format의 경우 imm에 음수 값이 들어올 수 있다. 사용하고자 하는 것은 2의 보수 형태로 표현된 하위 16-bit이지만, 입력받는 값의 자료형이 int 이므로 상위 16-bit까지 모두 1로 채워져 있는 형태이고, 이를 그대로 더해 줄 경우 의도와 다르게 표현된다. 따라서, explicit casting 후 shift 연산을 취해 상위 16-bit를 0으로 채워준 후 계산하였다.

이외에도, vector 형태로 저장되어 있는 struct들을 탐색하여 이름이 같은 struct의 index를 반환하는 함수, decimal 또는 hexadecimal 형식의 string을 int값으로 변환해주는 str2int 함수, string을 공백을 기준으로 나눈 후, vector<string> 형태로 return하는 split_line 함수 등을 구현하여 사용하였다.

1.2. 디버깅

각 단계별로 수행하는 연산을 자세하게, 순차적으로 확인할 수 있도록 **verbose flag**를 만들어 두었다. main function 내부에 존재하는 verbose 값을 1로 설정한 후 컴파일 및 실행하면 각 loop 별로 수행하는 동작들을 터미널에 출력하도록 한다. 자세한 내용은 아래와 같다.

첫 번째 loop에서는 아래의 내용들을 터미널에 출력한다.

1) 첫 번째 loop에서 수행하는 기능들에 대한 요약
2) parsing을 완료한 text section의 내용. text section에 존재하는 label을 포함한다.
3) 저장된 data, label struct. data의 경우 이름, 주소(dec), 주소(hex), 값(dec), 값(hex), label의 경우 이름, 주소(dec), 주소(hex)

두 번째 loop에서는 아래의 내용들을 터미널에 출력한다.

1) 두 번째 loop에서 수행하는 기능들에 대한 요약
2) 현재 수행중인 instruction의 위치 (program counter) 및 내용.
3) la 명령어가 존재하는 경우, 주어진 주소를 이용하여 계산한 상위 16bit 주소, 하위 16-bit 주소 계산 결과.
4) 2)를 기반으로 la 명령어가 lui가 될지, lui + ori가 될 지에 대한 결과.
5) Data section에 위치한 label이 존재하는 경우, 해당하는 주소값으로 변환하는 과정.
6) 최종적으로 저장된 label struct의 이름, 주소(dec), 주소(hex). la의 연산 결과에 의해 첫 번째 loop가 종료된 후의 결과와 다른 주소값을 가질 수 있다.

세 번째 loop에서는 아래의 내용들을 터미널에 출력한다.

1) 세 번째 loop에서 수행하는 기능들에 대한 요약
2) Text section에 위치한 label이 존재하는 경우, 해당 주소값으로 변환하는 과정.
3) la 명령어가 존재하는 경우, 주어진 주소를 이용하여 계산한 상위 16bit 주소, 하위 16-bit 주소 계산 결과.
4) 2)를 기반으로 la 명령어가 lui가 될지, lui + ori가 될 지에 대한 결과.

네 번째 loop에서는 아래의 내용들을 터미널에 출력한다.

1) 네 번째 loop에서 수행하는 기능들에 대한 요약
2) 전체 data 수, 전체 instruction 수, text section의 크기, data section의 크기를 각각 dec, hex 형식의 나타낸 값.
3) 현재 수행중인 instruction의 위치 (program counter) 및 내용.

4) R/I/J-format 명령어를 계산하기 위해 필요한 값 (각 register에 담긴 값 등)
5) 각 instruction을 해당하는 R/I/J-format으로 표현한 결과를 hex, bin 형태로 나타낸 값.

2. Compile & Execution

제출한 압축 파일은 MIPS_assembler.cpp, Makefile, Report로 구성되어있다. 압축을 해제한 폴더에서 아래의 명령어를 통해 컴파일 및 실행할 수 있다.

```
$ (sudo) make
$ (sudo) ./runfile <assembly file>
```

3. Environment

코드는 Ubuntu 22.04.3 LTS 환경에서 c++로 작성되었으며, g++ 9.5.0을 사용하여 컴파일하였다. 테스트는 Windows Subsystem for Linux (Ubuntu 20.04) 환경에서 g++ 9.4.0을 사용하여 진행하였다.