마이크로프로세서 설계 무작정 따라하기 part-II (5)

부제: 명령어 시뮬레이터, 어셈블러, 컴파일러의 개발

KAIST 전자전산학과 박사과정 배영돈(donny@ics.kaist.ac.kr), 이종열(jylee@ics.kaist.ac.kr)

이 번 강좌에서는 GNU Assembler(GAS)를 새로운 프로세서에 포팅하여 새로운 어셈블러를 구현하는 과정에 대하여 다룬다. GCC 와 마찬가지로 GAS 는 새로운 프로세서에 포팅하는 과정을 용이하게 하기 위한 구조를 가지고 있으며, 프로세서에 따라 달라지는 부분 만을 다시 작성함으로써 새로운 어셈블러의 작성이 가능하다. 따라서 이 번 강좌는 이중 패스(two pass) 어셈블러의 일반적인 동작을 간단히 살펴보고, GAS 포팅에 필요한 간단한 기본 지식을 습득하는 것을 목표로 하고 있다. 본 강좌에서는 GNU 의 binutils-2.11.을 사용하고 있다.

1. 이중 패스 어셈블러의 동작

그림 1은 이중 패스 어셈블러의 동작을 보이고 있다. 이중 패스 어셈블러는 2 단계에 과정을 거쳐서 어셈블을 수행한다. 첫 번째 과정에서는 어셈블러의 디렉티브(directive)와 명령어를 어셈블하게 된다. 이 과정에서 정의된 곳 보다 앞서서 사용되는 심볼에 대한 정보는 처리할 수 없으므로 첫 번째 과정에서 생성되는 결과에서는 이러한 심볼의 값이 정해지지 않은 것으로 간주되어 처리된다. 두 번째 과정에서는 첫 번째 과정에서 처리되지 않은 심볼들을 심볼 테이블을 사용하여 처리하게 된다. “MOV LABEL, R1”과 같은 명령어에서 LABEL 이 뒤에서 정의되었다면, 첫 번째 과정을 거친 후의 어셈블리 코드는 LABEL 의 값을 0 으로 어셈블하고, 재배치 테이블 (relocation table)에 이 명령어는 두 번째 과정에서 LABEL 의 값을 이 명령어의 어셈블 결과에 더해야 함을 표시한다. 두 번째 과정에서는 재배치 테이블을 이용하여, 이 심볼 테이블에서 LABEL 의 값을 찾아서 그 값을 LABEL 이 어셈블된 위치에 더하게 된다.
이상과 같은 동작에서 어셈블러는 사용자가 정의하는 출력 파일의 형태에 따라서 어셈블 결과를 출력할 수 있어야 한다.

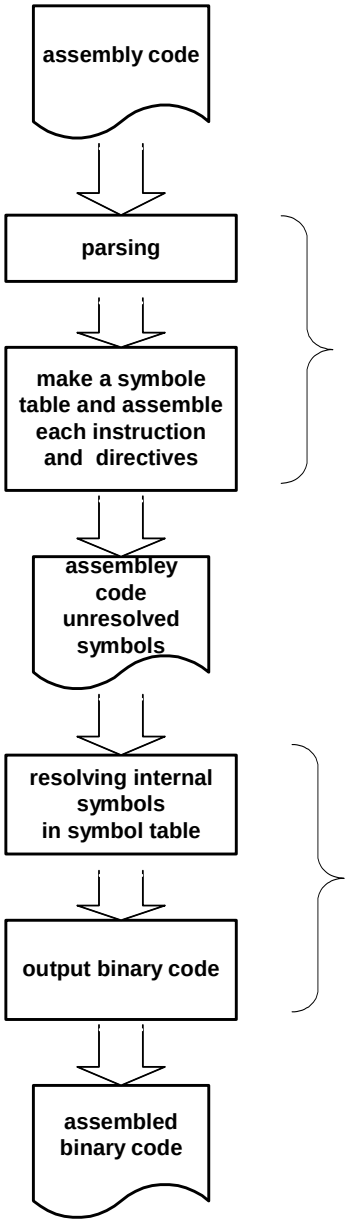

그림 1. 일반적인 이중 패스 어셈블러의 동작

현재 널리 사용되고 있는 출력 파일의 형태로는 ELF, COFF, a.out 등의 형태가 있다. 이러한 출력 형태에 따라서 심볼 등을 처리하는 방법이 달라지므로 어셈블러의 동작을 완전히 이해하기 위해서는 출력 파일 형태의 특징을 잘 알고 있어야 한다.

2. GAS 의 버전

GAS 는 GNU 에서 제공하는 어셈블러이다. GAS 는 여러 가지 오브젝트 파일을 지원하고 있으며, 지원하는 오브젝트 파일의 형태와 섹션의 수 등에 따라 다음과 같은 버전으로 나뉜다.

- Original GAS version : 가장 먼저 구현된 버전으로 현재는 m68k 을 대상으로 하는 버전이 있다.
- MANY_SEGMENTS GAS version : 이 버전은 COFF 형태 만을 지원한다.
- BFD_ASSEMBLER GAS version : 현재 가장 선호되는 버전이다. 이 버전의 어셈블러는 BFD 라이브러리를 통하여 ELF, COFF, SOM 등의 형태를 지원한다. (BFD 는 바이너리 파일을 다루기 위하

여 GNU 에서 제공되는 라이브러리이다. 이 라이브러리는 거의 모든 형태의 오브젝트 파일을 지원하고 있으며 GNU 에서 제공되는 모든 툴은 이 라이브러리를 사용한다. 이 라이브러리에 대한 자세한 사항은 GNU 의 binutils 에 포함되어 있는 매뉴얼을 참조하기 바란다.) 본 강좌는 이 버전의 GAS 를 기준으로 진행된다.

3. GAS 의 동작

그림 2는 GAS 의 동작에 대한 flow chart 이다. 어셈블이 시작되면 GAS 는 여러 가지 초기화 함수를 호출하여 초기화한다. (1) Initialization). 초기화가 완료되면, 각 소스 파일을 “read_a_source_file” 함수를 통하여 읽은 후에 파싱하게 된다. 이때 전역 변수인 “input_line_pointer”는 현재 처리 중인 문자를 가리키고 있게 된다. 소스 코드는 줄 단위로 처리된다. GAS 는 각 줄에 대하여 label 을 “colon” 함수에 전달한다. 그리고 각 줄의 첫 단어를 검사하여 어셈블러 디렉티브를 가지고 있는 가를 검사한다 (4)). 만약 어셈블러 디렉티브를 가지고 있는 경우에는 해당 디렉티브를 “po_hash”에 전달하여 적절히 처리한다. (5) process the pseudo-op) 디렉티브를 포함하지 않은 경우에는 “md_assemble” 함수를 호출하여 명령어를 파싱하도록 한다. 처리된 줄이 데이터를 출력하는 경우에는 GAS 내부의 데이터 구조인 “frag”를 만들게 된다. 이 구조는 출력 데이터를 저장하고 처리하는 단위이다. (5), 6)). 입력 파일의 모든 줄이 처리 된 후에는 frag 들에 저장된 데이터를 출력 파일로 출력하게 된다. 이 과정에서 resolve 되지 않은 심볼들이 resolve 된다. 출력을 위해서는 BFD 라이브러리에서 제공되는 함수들이 사용된다.

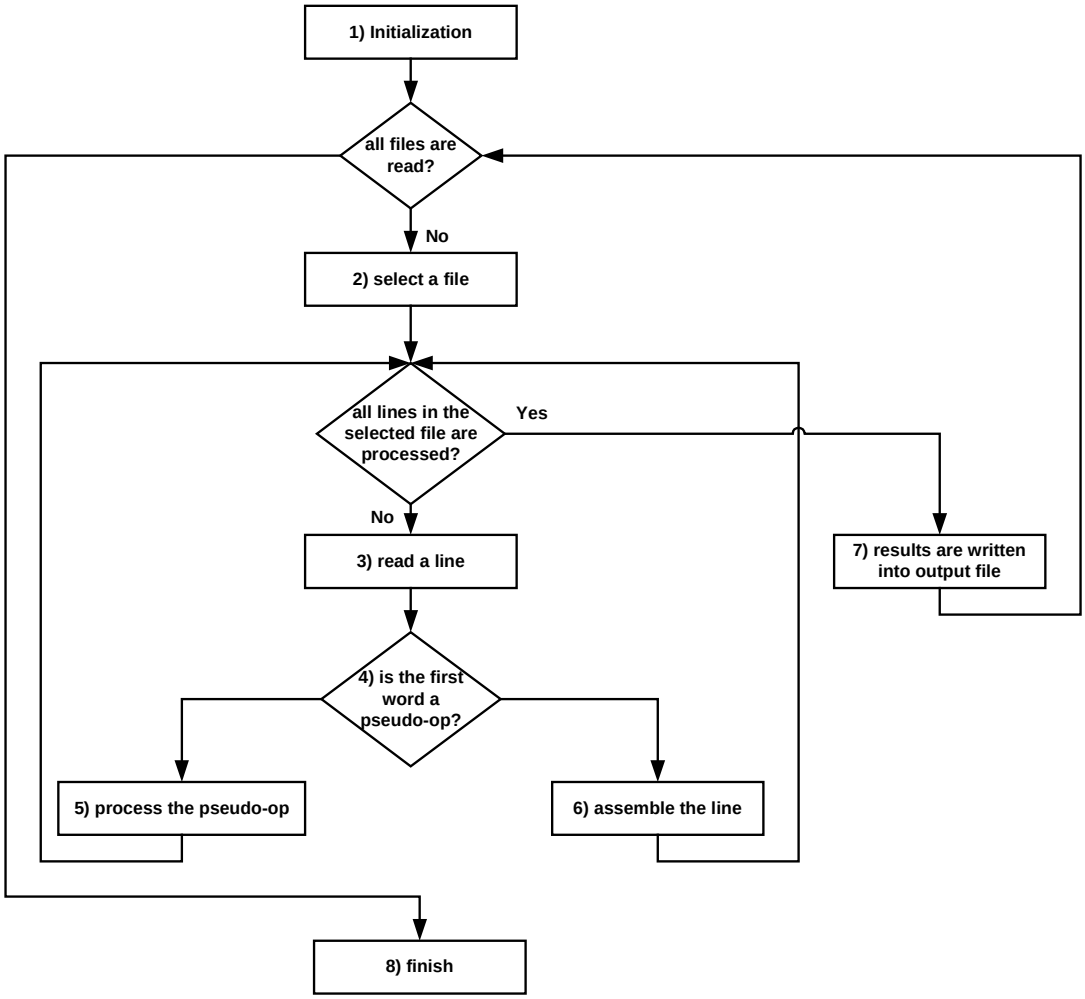

그림 2. GAS 의 동작

4. GAS 에서 사용되는 자료 구조

여기서는 GAS 내부에서 사용되는 자료 구조에 대하여 설명한다. 이들에 대한 정보는 포팅과정 내부 자료 구조를 수정해야 하는 경우에 유용하게 사용될 수 있다.

- Symbol : 이 것은 어셈블러 코드 내부에서 사용되는 심볼을 표현하는 방법으로 “struc-symbol.h” 에 structure 인 symbolS 로 정의되어 있다. 이 구조체에 포함된 필드는 심볼의 값(sy_value), 심볼이 resolve 되었는 가에 관한 정보 (sy_resolved, sy_resolving), 이전과 다음에 나오는 심볼에 대한 포인터 (sy_next, sy_previous)등이 포함된다. 그리고 symbolS 의 각 필드를 access 할 수 있는 함수들이 제공된다. (S_SET_VALUE, S_GET_VALUE, …)
- Expressions : 이 것은 어셈블리 코드 내의 expression 을 표현하기 위한 구조로 “expressionS”으로 정의된다. expressionS 는 하나의 오퍼레이션을 표현할 수 있다. 그리고 중첩된 expression 의 표현을 위하여 expression symbol 을 사용할 수 있다. expressionS 구조는 2 개의 symbolS 필드와 하나의

숫자 필드, 오퍼레이터 필드, 그리고 숫자가 unsigned 수인가를 나타내는 필드로 구성되었다. 오퍼레이터 필드는 “operatorT”로 표시된다.

- Fixups : 이 것은 어셈블러의 첫번째 단계에서 resolve 되지 않은 모든 것을 표현하는데 사용되는 자료 구조이다. Fixups 로 표시된 unresolved 심볼들은 두 번째 단계에서 resolve 되거나 (같은 파일 안에서 정의된 심볼인 경우) 오브젝트 파일의 relocation entry 에 남아 있게 된다. (전역 심볼의 경우)
- Frags : 이 것은 구조체 “fragS”로 정의되는 것으로 최종 오브젝트 파일의 부분을 표시한다. 따라서 어셈블이 완료된 줄은 fragS 에 저장된다. 어셈블러의 마지막 단계에서는 fragS 에 있는 데이터가 출력 파일에 출력된다.

5. GAS 의 포팅

GAS 를 포팅하기 위해서는 대상 프로세서와 오브젝트 파일 형태에 대하여 기술하여야 한다. 대상 프로세서에 대한 기술에서는 어셈블러의 syntax, directive 등을 어떻게 처리할 것인가를 기술 하여야 한다. 그리고 오브젝트 파일 형태에 따른 심볼 테이블, 각 섹션의 출력 방법에 대해서도 기술 해 주어야 한다.

1) 프로세서에 관한 기술

이 부분에서는 프로세서의 명령어 집합에 대하여 기술한다. 프로세서의 명령어 집합은 매크로와 C 함수를 기술함으로써 기술이 가능하다. 이러한 매크로와 함수들은 “binutils-2.11/gas/config”의 tc-cpu.c 와 tc-cpu.h 에 포함된다. 다음은 포팅을 위한 매크로와 함수들 중 주요한 것을 정리한 것이다.

- TC_*CPU* : 이 매크로는 대상 프로세서를 알려주는 역할을 하는 매크로이다. 따라서 대상 프로세서가 ARM 인 경우, TC_ARM 을 정의한다.
- TARGET_FORMAT : 이 매크로를 대상 오브젝트 파일의 형태를 정의한다.
- TARGET_BYTES_BIG_ENDIAN : 이 것은 프로세서의 엔디안을 정의하는 매크로이므로 반드시 정의되어야 한다.
- md_*xxx* 함수들 : 이 함수들은 어셈블리 코드를 파싱하여 어셈블하는 등의 작업을 위하여 호추되는 함수 들이다. 이들 함수 중에서 특히 “md_assemble”의 경우에는 디렉티브를 포함하지 않는 하나의 소스 라인을 어셈블하는 함수이므로 어셈블러의 핵심이라고 할 수 있다. 이 외에도 디렉티브를 처리하는 함수와 배열 변수 (md_pseudo_table), 부동 소수점 수와 관련된 함수와 매크로 (FLT_CHARS, md_atof) , 정렬(alignment)을 수행하는 함수 (md_do_align), 재배치에 관련된 함수 (md_reloc_size) 등 여러 가지 함수를 기술하여야 한다. 자세한 사항은 소스 코드를 참조하기 바

란다.

2) 오브젝트 파일 형태에 관한 기술

대게의 경우, 오브젝트 파일의 포맷은 기존의 것을 사용함으로 실제 어셈블러를 포팅하는 과정에서 이 부분을 거의 수정할 필요가 없다. 그러나 새로운 오브젝트 파일 형태를 사용하거나, 기존의 오브젝트 파일 형태를 수정하여 사용하는 경우에는 오브젝트 파일 형태에 대한 기술이 필요하게 된다. 오브젝트 파일 형태에 대한 기술은 "binutils-2.11/gas/config"의 obj-cpu.c 와 obj-cpu.h 에 포함된다.

오브젝트 파일의 형태에 따른 처리 방법을 기술하는 함수는 "obj_xxx"라는 이름을 갖는다. 이들 함수의 예에는 "obj_begin, obj_adjust_symtab" 등이 있다. 이들 함수에 대한 자세한 사항은 소스 코드를 참조하기 바란다.

3) 포팅을 위하여 수정하여야 하는 파일 정리

다음은 포팅을 위하여 수정하여야 하는 파일을 정리한 것이다. 포팅을 위해서는 다음과 같은 파일을 기술해야 한다.

gas/config/tc-**CPU**.h	- 대상 프로세서에 대한 기술
gas/config/tc-**CPU**.c	- 대상 프로세서와 어셈블러 syntax 에 대한 기술
include/{**coff** | **elf**}/**CPU**.h	- 오브젝트 파일 포맷에 대한 헤더 파일
include/opcodes/**CPU**.h	- 프로세서의 명령어 집합의 encoding
opcodes/**CPU**-dis.c	- 디스어셈블러 루틴
opcodes/**CPU**-opc.c	- 프로세서의 명령어 집합의 encoding
bfd/{**elf32** | **coff**}-**CPU**.c	- 리로케이션 핸들러(relocation handlers)
bfd/cpu-**CPU**.c	- 해당 프로세서와 BFD 에 대한 기술
ld/scripttempl/**CPU**.sc	- 링커 템플릿 파일
ld/emulparam/**CPU**.sh	- 링커 스크립트

또 컴파일을 위하여 다음과 같은 configuration 파일을 수정한다.

gas/configure.in
opcodes/configure.in
bfd/configure.in
bfd/config.bfd
bfd/archures.c
bfd/reloc.c
ld/configure.in
ld/configure.tgt
include/dis-asm.h

[참고문헌]

- V. Aho, M. R. Setdi, and J. D. Ullman, *Compiler—Principles, Techniques, and Tools*, Addison-Wesly, Reading, MA, 1986.
- R. M. Stallman, *Using and Porting GNU CC for version 2.6*, Free Software Foundation Inc., 1996.
- 황승호, 이대현, 이종열, *컴파일러 개발—GNU C 컴파일러 포팅을 중심으로*, 시그마프레스, 2001

* 이번 회를 마지막으로 명령어 시뮬레이터, 어셈블러, 컴파일러의 개발 관한 연재를 마칩니다. 그 동안 연재를 지켜 보아 주신 독자 분들께 감사 드립니다.