

# PC 어셈블리어

지은이 : Paul A. Carter  
번역자 : 이재범

December 8, 2009

Copyright © 2001, 2002, 2003, 2004 by Paul Carter  
This book is translated into Korean by Lee Jae Beom

이 문서 전체는 저자의 동의 없이도 자유롭게 수정되고(저자권, 저작권 및 권한 포함) 배포될 수 있으며 문서 자체에는 어떠한 가격도 매겨지지 않습니다. 이는 리뷰나 광고와 같은 “공정 사용(fair use)” 인용문과 번역과 같은 파생된 작업들을 모두 포함합니다.

참고로, 이 제한 조건은 문서를 복사 혹은 출력하는 것에 대해 값을 부과하는 것을 막기 위해서 만들어 진 것이 아닙니다.

지도 선생님들께서는 이 책을 학습 자료로 사용해도 되지만, 사용 시 저자에게 알려주었으면 합니다.

한국어에 관련한 번역 오류나 오탈자는 제 블로그 <http://kevin0960.tistory.com>이나 kev0960@gmail.com으로 메일을 보내주면 됩니다.

이 글은 이재범에 의해 번역되었습니다.

# Contents

머리말	v
<b>1 시작하기 앞서서</b>	<b>1</b>
1.1 기수법 . . . . .	1
1.1.1 십진법 . . . . .	1
1.1.2 이진법 . . . . .	1
1.1.3 16 진법 . . . . .	3
1.2 컴퓨터의 구조 . . . . .	4
1.2.1 메모리 . . . . .	4
1.2.2 CPU . . . . .	5
1.2.3 CPU 의 80x86 계열 . . . . .	6
1.2.4 8086 16비트 레지스터 . . . . .	7
1.2.5 80386 32비트 레지스터 . . . . .	8
1.2.6 실제 모드 . . . . .	8
1.2.7 16비트 보호 모드 . . . . .	9
1.2.8 32비트 보호 모드 . . . . .	10
1.2.9 인터럽트 . . . . .	10
1.3 어셈블리어 . . . . .	11
1.3.1 기계어 . . . . .	11
1.3.2 어셈블리어 . . . . .	12
1.3.3 피연산자 . . . . .	12
1.3.4 기초 명령 . . . . .	13
1.3.5 지시어 . . . . .	14
1.3.6 입출력 . . . . .	16
1.3.7 디버깅 . . . . .	18
1.4 프로그램 만들기 . . . . .	18
1.4.1 첫 프로그램 . . . . .	19
1.4.2 컴파일러 의존성 . . . . .	22
1.4.3 코드를 어셈블 하기 . . . . .	23
1.4.4 C 코드를 컴파일 하기 . . . . .	23
1.4.5 목적 파일 링크하기 . . . . .	24
1.4.6 어셈블리 리스트 파일 이해 . . . . .	24

1.5	빼대 파일 . . . . .	26
<b>2</b>	<b>어셈블리 언어의 기초</b>	<b>29</b>
2.1	정수들 다루기 . . . . .	29
2.1.1	정수의 표현 . . . . .	29
2.1.2	부호 확장 . . . . .	32
2.1.3	2의 보수의 산술 연산 . . . . .	35
2.1.4	예제 프로그램 . . . . .	37
2.1.5	확장 정밀도 산술 연산 . . . . .	38
2.2	제어 구조 . . . . .	39
2.2.1	비교 . . . . .	40
2.2.2	분기 명령 . . . . .	40
2.2.3	루프 명령 . . . . .	43
2.3	보통의 제어 구조를 번역하기 . . . . .	44
2.3.1	If 문 . . . . .	44
2.3.2	While 루프 . . . . .	45
2.3.3	Do while 루프 . . . . .	45
2.4	예제: 소수 찾는 프로그램 . . . . .	45
<b>3</b>	<b>비트 연산</b>	<b>49</b>
3.1	쉬프트 연산 . . . . .	49
3.1.1	논리 쉬프트 . . . . .	49
3.1.2	쉬프트의 이용 . . . . .	50
3.1.3	산술 쉬프트 . . . . .	50
3.1.4	회전 쉬프트 . . . . .	51
3.1.5	간단한 프로그램 . . . . .	51
3.2	불리언 비트 연산 . . . . .	52
3.2.1	AND 연산 . . . . .	52
3.2.2	OR 연산 . . . . .	52
3.2.3	XOR 연산 . . . . .	53
3.2.4	NOT 연산 . . . . .	53
3.2.5	TEST 명령 . . . . .	53
3.2.6	비트 연산의 사용 . . . . .	54
3.3	조건 분기 명령을 피하기 . . . . .	55
3.4	C에서 비트값 바꾸기 . . . . .	58
3.4.1	C에서의 비트 연산자 . . . . .	58
3.4.2	C에서 비트 연산자 사용하기 . . . . .	58
3.5	빅, 리틀 엔디안 표현 . . . . .	59
3.5.1	리틀 혹은 빅 엔디안인지 언제 신경 써야 하는가? . . . . .	60
3.6	비트 수 세기 . . . . .	62
3.6.1	첫 번째 방법 . . . . .	62
3.6.2	두 번째 방법 . . . . .	63
3.6.3	세 번째 방법 . . . . .	64

<b>4 서브프로그램</b>	<b>67</b>
4.1 간접 주소 지정 . . . . .	67
4.2 간단한 서브프로그램 예제 . . . . .	68
4.3 스택 . . . . .	70
4.4 CALL 과 RET 명령 . . . . .	71
4.5 호출 규약 . . . . .	72
4.5.1 스택에 인자들을 전달하기 . . . . .	72
4.5.2 스택에서의 지역변수 . . . . .	77
4.6 다중 모듈 프로그램 . . . . .	79
4.7 C 와 소통하기 . . . . .	82
4.7.1 레지스터 저장하기 . . . . .	83
4.7.2 함수들의 라벨 . . . . .	83
4.7.3 인자 전달하기 . . . . .	84
4.7.4 지역 변수의 주소 계산하기 . . . . .	85
4.7.5 리턴값 . . . . .	85
4.7.6 다른 호출 규약 . . . . .	85
4.7.7 예제 . . . . .	87
4.7.8 어셈블리에서 C 함수를 호출하기 . . . . .	90
4.8 재진입 및 재귀 서브프로그램 . . . . .	91
4.8.1 재귀 서브프로그램 . . . . .	91
4.8.2 C 변수 저장 형식에 대해서 . . . . .	93
<b>5 배열</b>	<b>97</b>
5.1 소개 . . . . .	97
5.1.1 배열 정의하기 . . . . .	97
5.1.2 배열의 원소에 접근하기 . . . . .	98
5.1.3 좀 더 향상된 간접 주소 지정 . . . . .	100
5.1.4 예제 . . . . .	101
5.1.5 다차원 배열 . . . . .	105
5.2 배열/문자열 명령 . . . . .	108
5.2.1 메모리에 쓰고 읽기 . . . . .	108
5.2.2 REP 명령 접두어 . . . . .	110
5.2.3 비교 문자열 명령 . . . . .	111
5.2.4 REPx 명령 접두어 . . . . .	111
5.2.5 예제 . . . . .	112
<b>6 부동 소수점</b>	<b>119</b>
6.1 부동 소수점 표현 . . . . .	119
6.1.1 정수가 아닌 이진수 . . . . .	119
6.1.2 IEEE 부동 소수점 표현 . . . . .	121
6.2 부동 소수점 수들의 산술 연산 . . . . .	124
6.2.1 덧셈 . . . . .	124
6.2.2 뺄셈 . . . . .	125

6.2.3	곱셈과 나눗셈	125
6.2.4	프로그래밍을 위한 조언	126
6.3	수치 부프로세서	126
6.3.1	하드웨어	126
6.3.2	명령	127
6.3.3	예제	133
6.3.4	2차 방정식의 근의 공식	133
6.3.5	파일로 부터 배열을 읽기	136
6.3.6	소수 찾기	139
<b>7</b>	<b>구조체와 C++</b>	<b>143</b>
7.1	구조체	143
7.1.1	서론	143
7.1.2	메모리 배열하기	145
7.1.3	비트필드	146
7.1.4	어셈블리에서 구조체 사용하기	150
7.2	어셈블리와 C++	150
7.2.1	오버로딩과 이름 맹글링(Name Mangling)	151
7.2.2	레퍼런스	153
7.2.3	인라인 함수	154
7.2.4	클래스	157
7.2.5	상속과 다형성	163
7.2.6	다른 C++ 기능들	171
<b>A</b>	<b>80x86 명령어 모음</b>	<b>173</b>
A.1	비-부동 소수점 명령들	173
A.2	부동 소수점 명령들	179
<b>Index</b>		<b>181</b>

# 머리말

## 이 책을 쓴 이유

이 책을 쓰게 된 동기는 바로 독자들에게 파스칼과 같은 언어들 보다 낫은 수준에서 컴퓨터가 실제로 어떻게 작동하는지에 대한 자세한 이해를 위해서이다. 컴퓨터가 실제로 어떻게 작동하는지에 대해 자세히 알게 된다면 독자 여러분은 C나 C++과 같은 고급 언어를 통해 프로그램을 개발하는 것을 훨씬 효율적으로 할 수 있을 것이다. 이 목적을 이루기 위해선 어셈블리 언어를 배우는 것이 가장 효과적이다. 다른 PC 어셈블리 언어 책들은 아직도 1981년 PC에 사용되었던 8086 프로세서에서 어떻게 프로그램을 만드는지에 대해 다루고 있다. 이 8086 프로세서는 오직 실제 모드만을 지원한다. 이 모드에선 어떠한 프로그램이라도 컴퓨터의 메모리나 다른 장치들에게 자유롭게 접근이 가능하다. 이 모드는 안정적인 멀티태스킹 운영체제에는 적합하지 않다. 이 책은 그 대신 보호 모드에서 작동하는 (현재의 윈도우즈나 리눅스가 실행되는) 80386이나 그 이후에 나온 프로세서들에서 어떻게 프로그램을 만드는지에 대해 다루고 있다. 이 모드는 현대의 운영체제들이 바라는 기능들을 가지고 있다. 예를 들자면 가상 메모리나 메모리 보호 등이다. 보호 모드를 이용하는 데에는 여러 이유들이 있다.

1. 다른 책이 다루는 8086 실제 모드에서 보다 프로그래밍 하기가 편하다
2. 현대 모든 PC 운영체제는 보호 모드에서 작동된다.
3. 보호 모드에서 작동되는 많은 수의 무료 프로그램들이 있다.

따라서 제가 이 책을 쓰게 된 가장 큰 동기는 바로 보호 모드 PC 어셈블리 프로그래밍에 대한 책의 부족 때문이다.

앞에서 살짝 말했듯이 이 책은 무료/공개 소스 소프트웨어만을 사용한다. 그 예로, NASM 어셈블리와 DJGPP C/C++ 컴파일러가 있다. 이들은 모두 인터넷을 통해서 다운로드가 가능하다. 또한 이 책은 앞으로 리눅스에서 NASM 어셈블리 코드를 어떻게 사용할 것인지와 Windows에서 불랜드사와 마이크로소프트사의 C/C++ 컴파일러를 어떻게 사용할 것인지에 대해 다룰 것이다. 앞서 말한 모든 프로그램들은 나의 웹사이트 <http://www.drpaulcarter.com/pcasm>에서 다운로드가 가능하다. 이 책에서

다를 예제들을 어셈블하고 실행시킬 것이라면 나의 사이트에서 프로그램들을 꼭 다운로드 해야 한다.

이 책이 어셈블리 프로그래밍의 모든 부분을 다루지 않았으리라 걱정 할 필요는 없다. 나는 모든 프로그래머가 보아야 할 어셈블리 언어의 중요한 주제들을 모두 다루려고 노력했다.

## 감사의 말

저는 무료/공개 소스 운동에 기여해 주신 많은 프로그래머들에게 감사의 말을 전합니다. 많은 프로그램, 심지어 이 책 자체도 무료 소프트웨어를 통해서 만들어졌습니다. 특히 저는 John S. Fine, Simon Tatham, Julian Hall, 그리고 NASM 어셈블러를 개발하신 다른 모든 분들께 감사의 말을 전합니다. 또한 DJGPP C/C++ 컴파일러를 개발한 DJ Delorie 와 DJGPP 가기반으로 한 GNU gcc 컴파일러를 개발한 많은 프로그래머들에게도, TeX와 L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>를 개발한 Donald Knuth, 마지막으로 자유 소프트웨어 재단을 창설한 Richard Stallman 과 리눅스 커널을 개발한 Linus Torvalds, 등에게도 감사의 말을 전합니다.

아래는 이 책의 오류를 정정해 주신 분들의 목록입니다.

- John S. Fine
- Marcelo Henrique Pinto de Almeida
- Sam Hopkins
- Nick D'Imperio
- Jeremiah Lawrence
- Ed Berozet
- Jerry Gembarowski
- Ziqiang Peng
- Eno Compton
- Josh I Cates
- Mik Mifflin
- Luke Wallis
- Gaku Ueda
- Brian Heward
- Chad Gorshing

- F. Gotti
- Bob Wilkinson
- Markus Koegel
- Louis Taber
- Dave Kiddell
- Eduardo Horowitz
- Sébastien Le Ray
- Nehal Mistry
- Jianyue Wang
- Jeremias Kleer
- Marc Janicki
- Trevor Hansen
- Giacomo Bruschi
- Leonardo Rodríguez Mújica
- Ulrich Bicheler
- Wu Xing

## 인터넷 레퍼런스

저자의 페이지	<a href="http://www.drpaulcarter.com/">http://www.drpaulcarter.com/</a>
NASM SourceForge 페이지	<a href="http://sourceforge.net/projects/nasm/">http://sourceforge.net/projects/nasm/</a>
DJGPP	<a href="http://www.delorie.com/djgpp">http://www.delorie.com/djgpp</a>
리눅스 어셈블리	<a href="http://www.linuxassembly.org/">http://www.linuxassembly.org/</a>
어셈블리 언어의 미학	<a href="http://webster.cs.ucr.edu/">http://webster.cs.ucr.edu/</a>
유즈넷	<a href="comp.lang.asm.x86">comp.lang.asm.x86</a>
인텔 문서	<a href="http://developer.intel.com/design/Pentium4/documentation.htm">http://developer.intel.com/design/Pentium4/documentation.htm</a>

## 피드백

저는 이 작업에 대한 피드백을 환영합니다.

E-mail: pacman128@gmail.com  
WWW: <http://www.drpaulcarter.com/pcasm>

머리말

viii

# Chapter 1

## 시작하기 앞서서

### 1.1 기수법

컴퓨터 내부의 메모리는 수들로 구성되어 있다. 컴퓨터 메모리는 그 수들을 10 개의 숫자를 이용하는 십진법(decimal)을 이용하여 저장하지 않는다. 그 대신 하드웨어를 매우 단순화 시키는 이진법(binary)을 이용하여 컴퓨터는 모든 정보를 저장한다. 먼저, 십진 체계에 대해 알아 보도록 하자.

#### 1.1.1 십진법

십진체계에서는 10 개의 숫자(0-9)들을 이용하여 수를 나타낸다. 십진법에서 수의 각 자리 숫자는 그 수에서의 위치에 대한 10 의 몇수로 나타나게 된다. 예를 들면

$$234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

#### 1.1.2 이진법

이진체계에서는 오직 2 개의 숫자(0,1) 만을 이용하여 십진법과는 달리 2 를 기준으로 나타난다. 이진수의 각 자리 숫자는 그 수에서의 위치에 대한 2 의 몇수로 나타나게 된다.(참고로, 이진수에서 각 자리 숫자를 비트(bit)라고 부른다) 예를 들자면

$$\begin{aligned} 11001_2 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 16 + 8 + 1 \\ &= 25 \end{aligned}$$

위 예를 통해 어떻게 이진수가 십진수로 바뀌어지는지 알 수 있다. 아래 표 1.1에서 일부 십진수들이 이진수로 어떻게 나타나는지 보여주고 있다.

십진수	이진수		십진수	이진수
0	0000		8	1000
1	0001		9	1001
2	0010		10	1010
3	0011		11	1011
4	0100		12	1100
5	0101		13	1101
6	0110		14	1110
7	0111		15	1111

Table 1.1: 0 부터 15 까지의 십진수들을 이진수로 나타냄

이전에 올림(carry) 없음				이전에 올림 있음			
0	0	1	1	0	0	1	1
+0	+1	+0	+1	+0	+1	+0	+1
0	1	1	0	1	0	0	1
	c			c	c	c	c

Figure 1.1: 이진수의 덧셈 (c 는 올림(carry)을 나타낸다 )

그림 1.1 는 어떻게 두 이진숫자가 더해지는지 (*i.e.*, 비트(bits)) 보여주고 있다. 여기, 그 예가 있다.

$$\begin{array}{r} 11011_2 \\ +10001_2 \\ \hline 101100_2 \end{array}$$

누군가가 아래와 같은 십진 나눗셈을 하였다고 하자.

$$1234 \div 10 = 123 r 4$$

그는 이 나눗셈이 피제수의 가장 오른쪽 숫자를 빼버리고 나머지 숫자를 오른쪽으로 한 자리 쉬프트 했다는 사실을 알 수 있다. 이진체계에서는 2로 나누는 연산을 통해 위와 동일한 작업을 발생시킬 수 있다. 예를 들어 아래와 같은 이진 나눗셈을 생각하자.<sup>1</sup>:

$$1101_2 \div 10_2 = 110_2 r 1$$

이 사실을 통해서 어떤 십진수를 값이 같은 이진수로 바꿀 수 있다는 것을 알 수 있다. 이는 그림 1.2 에서 나타난다. 이 방법은 가장 오른쪽의 비트

<sup>1</sup> 참고로 수 옆에 아래 첨자로 적힌 2 는 이 숫자가 십진법으로 나타낸 수가 아니라 이진법으로 나타낸 수임을 알려준다

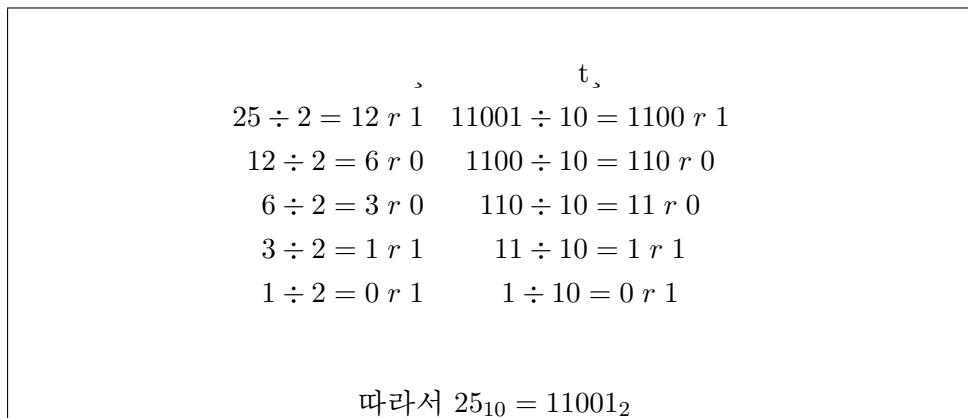


Figure 1.2: 십진수의 변환

부터 찾아 나가는데 그 비트를 흔히 최하위 비트(*least significant bit, lsb*) 라 부른다. 또한 가장 왼쪽의 비트는 최상위 비트(*most significant bit, msb*)라고 부른다. 메모리의 가장 기본적인 단위는 바이트(*byte*) 라 부르는 8 개의 비트들로 이루어져 있다.

### 1.1.3 16 진법

16진수들은 16 을 밑으로 하여 나타내어 진다. 16 진수들은 흔히 자리수가 큰 이진수들의 크기를 줄이기 위해 사용된다. 16 진수가 16 개의 숫자를 사용하기 때문에 6 개의 숫자가 모자르게 된다. 이 때문에 9 다음으로 나타나는 숫자들은 영어 대문자 A, B, C, D, E, F 를 사용하게 된다. 따라서 16 진수 A 는 10 진수 10 과 동일한 값을 가지며 B 는 11, 그리고 F 는 15 의 값을 가진다. 16 진수는 16 을 밑으로 하여 수를 나타내는데 아래 예를 보면

$$\begin{aligned} 2BD_{16} &= 2 \times 16^2 + 11 \times 16^1 + 13 \times 16^0 \\ &= 512 + 176 + 13 \\ &= 701 \end{aligned}$$

십진수를 16 진수로 변환하기 위해서는 우리가 앞서 이진수를 십진수로 변환하는데 사용했던 방법을 이용하면 된다. 다만, 이진수로 변환하는 것은 2 로 나누는 것이였지만 16 진수로 변환하는 것은 16 으로 나누어 주면 된다. 예시로 아래 그림 1.3 을 보면 알 수 있다.

16 진수가 유용한 이유는 바로 2진수를 손쉽게 16 진수로 바꿀 수 있기 때문이다. 이진수들은 그 자리수가 너무 커서 다루기 불편하지만 16 진수들은 그 길이가 훨씬 짧아지므로 표현하기가 매우 쉬워진다.

$\begin{array}{rcl} 589 \div 16 & = & 36 \ r\ 13 \\ 36 \div 16 & = & 2 \ r\ 4 \\ 2 \div 16 & = & 0 \ r\ 2 \end{array}$
결과적으로 $589 = 24D_{16}$

Figure 1.3:

16 진수를 2 진수로 바꾸는 방법은 단지 16 진수의 각 자리수를 4 비트의 이진수로 바꾸어 주면 된다. 예를 들어  $24D_{16}$  은  $0010\ 0100\ 1101_2$  으로 변환된다. 주의 해야 할 점은 16 진수의 모든 자리수는 반드시 4 비트로 바꾸어 주어야 하는데 (다만 16 진수의 첫 번째 자리수는 꼭 그렇게 할 필요는 없다) 왜냐하면 4 를 이진수로 바꾸었을 때  $4_{16} = 100_2$  라고 해서 그냥 100 으로 한다면 그 결과가 완전히 달라지게 된다. 마찬가지로 이진수를 16 진수로 바꾸는 것도 매우 쉬운데 오른쪽 끝에서부터 4 비트 씩 끊어 가면서 각각을 16 진수로 바꾸면 된다. 물론 이 때 반드시 오른쪽 끝에서부터 변환해야 한다.<sup>2</sup> 예를 들면

$$\begin{array}{ccccccc} 110 & 0000 & 0101 & 1010 & 0111 & 1110_2 \\ 6 & 0 & 5 & A & 7 & E_{16} \end{array}$$

4 비트의 이진수를 니블(nibble) 이라 하고, 따라서 16 진수의 각 자리수는 1 개의 니블에 대응된다. 또한 2 개의 니블은 1 바이트를 나타내며, 1 개의 바이트는 2 자리 16 진수에 대응된다. 1 바이트가 가질 수 있는 값의 범위는 이진수로 나타내면 0 부터  $11111111$ , 16 진수로 나타내면 0 부터 FF, 십진수로 나타내면 0 부터 255 까지 임을 알 수 있다.

## 1.2 컴퓨터의 구조

### 1.2.1 메모리

메모리의 크기를 나타내는 단위들은 킬로바이트 ( $2^{10} = 1,024$  바이트), 메가바이트( $2^{20} = 1,048,576$  바이트) 그리고 기가바이트( $2^{30} = 1,073,741,824$  바이트). 메모리의 기본적인 단위는 바이트이다. 만약 어떤 컴퓨터가 32 메가바이트의 메모리를 가진다면 그 컴퓨터는 대략 3200 만 바이트의 정보를 보관할 수 있다. 메모리의 각 바이트는 그림 1.4 에 잘 나와 있듯이 주소(address)라 불리는 고유의 값을 가지고 있다.

보통 메모리는 단일 바이트보다는 여러개의 바이트들의 묶음으로 많이 사용된다. PC 아키텍쳐에서는 그 묶음을 표 1.2 에서 나타나는 것 처럼 이름 붙인다.

<sup>2</sup>왜 그래야 하는지 이해가 잘 가지 않는다면 직접 어떤 이진수를 골라서 왼쪽 부터 16 진수로 변환해 보아라

주소	0	1	2	3	4	5	6	7
메모리	2A	45	B8	20	8F	CD	12	2E

Figure 1.4: 메모리 주소

워드(word)	2 바이트
더블워드(double word)	4 바이트
쿼드워드(quad word)	8 바이트
패러그래프(paragraph)	16 바이트

Table 1.2: 메모리의 단위

메모리에 있는 모든 데이터들은 수이다. 문자들은 문자 코드(*character code*)에 나타난 숫자들에 대응되어 기록된다. 문자 코드 중에서 가장 유명한 것으로는 아스키(*ASCII, American Standard Code for Information Interchange*) 코드를 들 수 있다. 현재 새롭게 아스키 코드를 대체해 나가고 있는 코드는 유니코드(*Unicode*)이다. 유니코드와 아스키코드의 가장 큰 차이점은 아스키 코드에서는 한 개의 문자를 나타내기 위해 1 바이트를 사용하지만 유니코드는 2 바이트(혹은 워드)를 사용한다. 예를 들어서 아스키코드에서는 대문자 A를 나타내기 위해 바이트  $41_{16}$ ( $65_{10}$ )를 사용하지만 유니코드에서는 워드  $0041_{16}$ 을 사용한다. 아스키코드가 1 바이트 만을 사용하기 때문에 오직 256 개의 서로 다른 문자들만 사용할 수 있으나<sup>3</sup> 반면에 유니코드는 아스키코드를 확장하여 훨씬 많은 문자들을 사용할 수 있게 하였다. 이를 통해 전 세계의 많은 언어들의 문자를 나타낼 수 있게 되었다.

### 1.2.2 CPU

중앙처리장치(CPU)는 명령을 수행하는 장치이다. 보통 CPU들이 수행하는 명령들은 매우 단순하다. 명령들은 대부분 CPU 자체에 있는 특별한 데이타 저장소인 레지스터에 들어 있는 값들을 필요로 한다. CPU는 레지스터에 있는 데이터에 메모리에 있는 데이터 보다 훨씬 빠른 속도로 접근할 수 있다. 그러나 CPU 한 개에 있는 레지스터의 수는 제한적이기 때문에 프로그래머는 반드시 현재 사용되고 있는 데이터들만 레지스터에 보관하도록 해야 한다. CPU가 실행하는 명령들은 CPU의 기계어(*Machine language*)에 의해 작성되어 있다. 기계 프로그램은 고급 언어들 보다 훨씬 간단한 구조를 가지고 있다. 기계어의 명령들은 숫자들에 대응이 된다. CPU는 이러한 명령들을 빠른 속도로 해석하여 효율적으로 실행해야만 한다. 따라서 기계어는 이러한 점을 항상 염두에 두어 디자인 되기 때문에 인간이

<sup>3</sup> 사실 아스키코드는 오직 하위 7 비트만을 사용하므로 오직 128 개의 서로 다른 값들만 나타낼 수 있다.

해석하기에는 매우 난해한 면이 있다. 다른 언어로 쓰여진 프로그램들은 실행되기 위해서는 반드시 CPU 가 해석할 수 있는 기계어로 변환되어야만 한다. 따라서 특정한 컴퓨터 아키텍쳐에 대한 기계어 코드로 소스를 바꾸어 주는 프로그램을 컴파일러(*compiler*)라고 한다. 보통 각 CPU 마다 특정한 기계어를 가지고 있다. 이 때문에 Mac 에서 쓰여진 코드가 IBM PC 에서 잘 작동이 되지 않는다.

GHz 는 기가헤르츠의 약자로 초당 10억 번의 펄스가 발생했다는 뜻이다. 예로 1.5 GHz CPU 는 초당 15억번의 펄스가 발생한다.

컴퓨터는 클록(*clock*)을 이용하여 명령들의 실행을 동기화 한다. 클록은 특정한 진동수에 따라 펄스를 방출하는데 이 진동수를 클록 속도라고 한다. 당신이 만약 1.5GHz 의 컴퓨터를 구매 한다면 1.5Ghz 가 바로 이 컴퓨터의 클록의 진동수이다.<sup>4</sup> 우리가 메트로놈을 이용하여 음악을 정확한 리듬으로 연주할 수 있는 것 처럼 CPU 도 언제나 일정하게 클록에서 방출되는 펄스를 이용하여 명령을 정확하게 수행 할 수 있다. 각 명령이 필요로 하는 클록의 진동 횟수(보통 사이클이라 부른다)는 CPU 의 세대와 모델에 따라 차이가 난다.

### 1.2.3 CPU 의 80x86 계열

IBM 형의 PC 는 인텔의 80x86 계열의 CPU 를 장착하고 있다. 이 계열의 CPU 들은 모두 동일한 기반의 기계어를 사용한다. 최근에 출시된 것들의 경우 그 기능이 대폭 강화되었다.

**8088,8086:** 이 CPU 들은 초기의 PC 에서 사용되었던 것들이다. 이들은 AX, BX, CX, DX, SI, DI, BP, SP, CS, DS, SS, ES, IP, FLAGS 와 같은 16비트 레지스터들을 사용한다. 뿐만 아니라 최대 1 MB 의 메모리만 지원하며 오직 실제 모드에서만 실행된다. 이 모드에서는 하나의 프로그램이 메모리의 모든 부분에 접근할 수 있다. 심지어 다른 프로그램이 그 부분을 이미 사용하고 있다고 해도 말이다. 이 때문이 디버깅과 보안을 유지하기가 매우 힘들어 지게 된다. 뿐만 아니라 프로그램 메모리는 반드시 세그먼트(*segments*) 들로 나누어 져야 하는데 각각의 세그먼트들의 크기는 64KB 를 넘을 수 없다.

**80286:** 이 CPU 는 AT 계열의 PC 들에서 사용되었다. 이 CPU 는 몇 개의 새로운 명령들이 추가된 8088/86 기계어를 사용하였다. 그러나 이 CPU 의 가장 중요한 새기능은 바로 16 비트 보호 모드(*16-bits protected mode*)이다. 이 모드에서는 최대 16 MB 의 메모리에 접근 할 수 있고 특정한 프로그램이 다른 프로그램의 메모리에 함부로 접근하는 것을 방지해 준다. 하지만 아직도 각각의 프로그램들은 64 KB 보다 작아야 하는 세그먼트들에서만 실행 될 수 있었다.

**80386:** 이 CPU 는 80286 CPU 를 크게 발전시킨 것이다. 첫번째로, 기존의 16 비트 레지스터들을 32 비트 레지스터 (EAX, EBX, ECX, EDX,

---

<sup>4</sup>사실 클록 펄스는 컴퓨터의 여러 다른 부분들에서도 많이 사용되는데 그 부분들의 클록 속도는 CPU 와 다르다.



Figure 1.5: AX 레지스터

ESI, EDI, EBP, ESP, EIP)로 확장하였고 새로운 16 비트 레지스터인 FS 와 GS 를 추가하였다. 또한 이 CPU 는 32 비트 보호 모드(*32-bit protected mode*)를 지원한다. 이 모드에서는 최대 4 GB 의 메모리를 사용할 수 있다. 아직도 프로그램들은 세그먼트들에 나뉘어 들어가야 만 하지만 각 세그먼트의 크기는 최대 4 GB 까지 될 수 있다.

**80486/펜티엄/펜티엄 프로:** 이 CPU 들은 기존의 CPU 들과는 크게 달라진 것이 없다. 다만 명령들의 연산속도가 매우 빨라졌다.

**펜티엄 MMX:** 이 프로세서에는 MMX(멀티미디어 확장, MultiMedia eXtensions) 명령이 추가되었다. 이 명령은 자주 실행되는 그래픽 관련 명령들의 실행 속도를 향상 시켰다.

**펜티엄 II:** 이는 펜티엄 프로 프로세서에 MMX 명령들이 추가된 프로세서이다. (펜티엄 III 의 경우 단지 II 보다 빨라진 것이다.)

#### 1.2.4 8086 16비트 레지스터

기존의 8086 CPU 는 4 개의 16 비트 범용 프로세서들을 지원했는데 이는 AX, BX, CX, 그리고 DX 이다. 각각의 레지스터들은 2 개의 8 비트 레지스터들로 나뉘어 질 수 있다. 예를 들어 AX 레지스터는 AH, AL 레지스터로 나뉠 수 있는데 이는 그림 1.5 에 잘 나타나 있다. AH 레지스터는 AX 의 상위 8 비트를 나타내고, AL 은 AX 의 하위 8 비트를 나타낸다. AH 와 AL 을 종종 독립적인 1 바이트 레지스터 들로 사용된다. 그러나 두 레지스터들이 AX 와 다른 것이라고 생각하면 안된다. AX 레지스터의 값을 바꾸게 되면 AH 와 AL 의 값도 바뀌고 AH 와 AL 의 값을 바꾸면 AX 의 값이 바뀐다. 범용 레지스터들은 산술 연산과 데이터 이동 명령 들에서 잘 쓰인다.

CPU 에는 2 개의 SI 와 DI 라 불리는 16 비트 색인 레지스터가 있다. 이 들은 포인터로 자주 쓰이지만 범용 레지스터들과 같은 용도로 종종 쓰인다. 그러나 색인 레지스터들은 범용 레지스터와는 달리 2 개의 8 비트 레지스터들로 나누어 질 수 없다.

16 비트 BP 와 SP 레지스터들은 기계어 스택에 들어 있는 데이터의 위치를 가리키는데 쓰인다. 이 2 개의 레지스터를 각각 기준 포인터 와 스택 포인터 라고 부른다. 이 들에 대해선 나중에 다루기로 하자.

16 비트 CS, DS, SS, ES 레지스터들은 세그먼트 레지스터(*segment register*)라고 부른다. 이 프로그램의 각 부분에서 메모리의 어떤 부분이 사용되는지 가리킨다. CS 는 코드 세그먼트(Code Segment), DS 는 데이터

세그먼트(Data Segment), SS 는 스택 세그 먼트(Stack Segment), 그리고 ES 는 보조 세그먼트(Extra Segment) 의 약자 이다. ES 는 임시적인 세그먼트 레지스터로 사용된다. 이 레지스터 들에 대한 자세한 내용은 에서 다룰 것이다.

명령 포인터(IP) 레지스터는 CS 레지스터와 함께 사용되는데 이는 CPU에서 다음으로 실행될 명령의 주소를 저장한다. 보통 CPU에서 명령이 하나 실행되면 IP 는 메모리 상에서 그 다음으로 실행될 명령을 가리키게 된다.

플래그(FLAG) 레지스터들은 이전에 실행된 명령들의 중요한 결과값들을 저장하고 있다. 이 값들은 플래그 레지스터의 각각의 비트에 저장되는데 예를 들면 이전 연산의 결과가 0 이 였다면 플래그 레지스터의 Z 비트의 값은 1 이되고 0 이 아니라면 0 이 된다. 모든 명령이 플래그 레지스터의 비트 값들을 바꾸는 것은 아니며 이 책의 부록에 보면 어떠한 명령들이 플래그 레지스터의 값에 영향을 주는지 알 수 있다.

### 1.2.5 80386 32비트 레지스터

80386 과 그 이후에 출시된 프로세서들은 모두 확장 레지스터를 가지고 있다. 예를 들어 16 비트 AX 레지스터는 32 비트로 확장되었다. 이 때 하위 호환성을 위해 AX 는 계속 16 비트 레지스터로 남아 있고 그 대신 EAX 가 확장된 32 비트 레지스터를 가리키게 된다. AL 이 단지 AX 의 하위 8 비트인 것 처럼 AX 도 단지 EAX 의 하위 16 비트를 나타낸다. 이 때, EAX 의 상위 16 비트에 직접적으로 접근 할 수 있는 방법은 없다. 다른 확장된 레지스터들은 EBX, ECX, EDX, ESI, 그리고 EDI 가 있다.

다른 많은 수의 레지스터들도 확장되었다. BP 는 EBP로, SP 는 ESP로, FLAGS 는 EFLAGS로, 그리고 IP 는 EIP가 되었다. 그러나 32 비트 보호 모드에서는 범용 레지스터를 뺀 나머지 레지스터들은 모두 확장된 것으로만 사용해야 한다.

80386 에선 아직 세그먼트 레지스터들이 16 비트로 남아 있다. 하지만 새로운 2 개의 세그먼트 레지스터들이 추가되었는데 바로 FS 와 GS 이다. 이 레지스터들은 단지 ES 와 같이 임시적인 세그먼트 레지스터들로 사용된다.

워드(word) 라는 단어는 CPU 데이터 레지스터의 크기로 정의된다. 80x86 계열에서는 위 정의가 약간 혼란스럽다. 아래 표 1.2 에 나온 것 처럼 워드는 2 바이트 (혹은 16 비트) 로 정의됨을 알 수 있다. 이는 처음 8086 이 발표되었을 때 정해진 것이다. 하지만 80386 이 개발 되었을 때 레지스터의 크기가 바뀌었는데도 불구하고 기존의 워드의 정의를 그대로 남기기로 했다.

### 1.2.6 실제 모드

그렇다면 DOS 의 640KB 제한은 어디서 나타난 것 일까? BIOS 는 비디오 스 크린과 같은 하드웨어를 위해 1MB 의 일부분을 필요로 한다

실제 모드 예선 메모리 사용이 오직 1 MB ( $2^{20}$  바이트) 로 제한된다. 이는 00000 부터 FFFFF 에 해당하는 값이다. 따라서 메모리의 주소를 나타내기 위해선 20 비트의 숫자를 저장할 수 있는 레지스터가 필요한데 안타깝게도 8086 의 레지스터는 모두 16 비트 밖에 되지 않으므로 그럴 수 없다. 하지만

인텔은 하나의 메모리 주소를 정하기 위해 2 개의 16 비트 값을 사용하여 문제를 해결하였다. 처음의 16 비트 값을 선택터(selector)라고 부른다. 선택터의 값은 반드시 세그먼트 레지스터에 저장되어야만 한다. 두 번째 16 비트 값을 오프셋(offset)이라 부른다. 메모리의 물리 주소는 32 비트의 선택터: 오프셋 쌍으로 나타나며 그 값은

$$16 * \text{selector} + \text{offset}$$

이다. 16 을 곱하는 것은 16 진수에서는 단순히 오른쪽 끝에 0 을 하나 추가하는 것이므로 매우 쉬운 작업이다. 예를 들어 특정한 메모리 공간의 주소가 047C:0048 로 주어진다면 이는

$$\begin{array}{r} 047C0 \\ +0048 \\ \hline 04808 \end{array}$$

에 데이터가 저장되어 있다는 것이다. 사실상 선택터의 값은 언제나 16 의 배수(패러그램 수)가 된다. (표 1.2 참조)

그러나 위와 같은 방법도 몇 가지 단점이 있는데 :

- 하나의 선택터를 사용하면 최대 64 KB 의 메모리 까지 밖에 사용 할 수 없다. 따라서 만일 프로그램이 64KB 보다 더 큰 메모리를 필요로 한다면 하나의 선택터 만으로는 충분하지 못할 것이다. 이 때문에 프로그램이 64KB 이 상의 메모리를 사용시 프로그램은 반드시 64KB 이하의 조각들로 쪼개져야 한다 (이 조각들을 세그먼트 라 부른다) 만약 프로그램의 명령이 다른 세그먼트로 넘어갈 경우 CS 의 값도 반드시 달라져야 한다. 큰 데이터를 다룰 때 DS 레지스터에서도 비슷한 일이 발생하게 된다. 상당히 짜증나는 일이다!
- 위와 같이 주소 지정을 해준다면 메모리의 공간들은 유일한 주소값을 가지지 않는다. 예를 들어 물리 주소 04808 은 047C:0048, 047D:0038, 047E:0028, 047B:0058 등과 같이 나타날 수 있다. 이 때문에 두 개의 주소값이 같은지 비교시 복잡해진다.

### 1.2.7 16비트 보호 모드

80286 의 16비트 보호 모드에서 선택터의 값들은 실제 모드에서의 값과 다른 방식으로 해석된다. 실제 모드에서는 선택터의 값은 언제나 패러그램 수 이였다. 보호 모드에서는 선택터의 값은 디스크립터 테이블(descriptor table) 의 인덱스(index) 이다. 두 모드 모두 프로그램들이 세그먼트 들로 나뉘어 진다. 실제 모드에서는 이러한 세그먼트가 물리 메모리에 고정된 위치에 저장되며 선택터는 그 세그먼트의 시작 부분의 패러그램 값을 가리키게 된다. 그러나 보호 모드에서는 세그먼트 들이 물리 메모리에 고정된 위치에 저장되는 것이 아니다. 사실 메모리 상에 존재할 필요도 없다!

보호 모드는 가상 메모리(*virtual memory*)라는 기술을 이용한다. 가상 메모리 시스템의 기본적인 개념은 프로그램의 현재 사용하고 있는 데이터와 코드만 메모리 상에 저장하는 것이다. 다른 데이터나 코드들은 나중에 필요할 때 까지 임시적으로 디스크에 보관하게 된다. 16비트 보호 모드에선 세그먼트들이 메모리와 디스크 사이를 왔다 갔다 하게 되므로 세그먼트들이 언제나 메모리 상의 고정된 부분을 점유하고 있을 수가 없게 된다. 이러한 작업은 운영체系에 의해 제어되므로 프로그램이 가상 메모리에서의 작업을 위해 특별히 해 주어야 될 것은 없다.

보호 모드에서는 개개의 세그먼트가 디스크립터 테이블의 엔트리에 할당되는데 이 디스크립터 테이블에는 시스템이 그 세그먼트에 대해 알아야 할 모든 정보가 수록되어 있다. 예를 들면 현재 메모리 상에 있는지의 유무, 메모리 상에 있다면 어디에 있는지, 접근 가능한지(*e.g.*, 읽기 전용). 세그먼트 엔트리의 인덱스 값이 바로 세그먼트 레지스터에 저장되어 있는 실렉터 값이다.

16비트 보호 모드의 가장 큰 단점은 오프셋이 아직도 16비트이라는 점이다. 이 때문에 세그먼트의 크기는 아직도 64KB로 제한된다. 이는 큰 배열을 만들기 매우 골치아프게 한다.

한 유명한 PC 칼럼니스트  
는 286 CPU를 가리켜 '뇌  
사 상태'라고 말했다

### 1.2.8 32비트 보호 모드

80386 이 처음으로 32비트 보호 모드의 장을 열었다. 386 32비트와 286 16비트 보호 모드에는 2 개의 큰 차이점이 있는데

1. 오프셋이 32 비트로 확장되었다. 이는 오프셋의 값이 최대 40억 까지 가능하다는 뜻이다. 따라서, 하나의 세그먼트는 최대 4GB 까지 가질 수 있다.
2. 세그먼트들은 작은 4KB 크기의 페이지(*page*) 라 부르는 단위들로 나뉘어 질 수 있다. 가상 메모리 시스템도 세그먼트가 아닌 페이지를 기준으로 작업하게 된다. 이 말은 세그먼트의 일부분도 메모리에 존재할 수 있게 되었다는 뜻이다. 이전의 286 16 비트 모드에서는 전체 세그먼트가 메모리 상에 한꺼번에 존재하던지, 아니면 메모리 상에 아무 것도 존재 못하는 2 가지 경우 밖에 없었었다. 이를 거대한 32 비트 모드의 세그먼트에 그대로 적용하는 것은 이치에 맞지 않다.

윈도우즈 3.x 버전에서 표준 모드(*standard mode*)는 286 16비트 보호 모드를 가리키고 향상 모드(*enhanced mode*)는 32 비트 모드를 가리킨다. 윈도우즈 9X, 윈도우즈 NT/2000/XP, OS/2, 그리고 리눅스에선 모든 프로그램이 페이지 가능한 32비트 보호 모드에서 실행된다.

### 1.2.9 인터럽트

종종 보통의 프로그램의 흐름이 즉시 응답이 필요한 프로세스 이벤트(*event*)로부터 중지되는 경우가 있다. 컴퓨터 하드웨어는 인터럽트(*interrupt*)라는 메커니즘을 통해 이러한 이벤트를 처리한다. 예를 들어 마우스가

움직였을 때, 마우스 하드웨어는 마우스 움직임을 처리(마우스 커서를 움직이기 위해)하기 위해 현재 실행되는 일시적으로 프로그램을 중지시킨다. 인터럽트는 제어 흐름이 인터럽트 핸들러(*interrupt handler*)을 통과하게 한다. 인터럽트 핸들러는 인터럽트를 처리하는 루틴이다. 각각의 인터럽트는 하나의 정수에 대응이 되어 있다. 물리 메모리 상단에 있는 인터럽트 벡터 테이블(*interrupt vector table*)에 인터럽트 핸들러들의 주소값을 보관한다. 인터럽트의 번호는 본질적으로 테이블의 인덱스가 되는 것이다.

CPU 외부에서 오는 인터럽트를 외부 인터럽트(External interrupt)라 한다. 앞에서 예로 나온 마우스가 외부 인터럽트의 한 예이다. 대다수의 입출력(I/O) 장치들, 예를 들어 키보드, 타이머, 디스크 드라이브, CD-ROM, 그리고 사운드 카드들은 모두 외부 인터럽트를 발생시킨다. 내부 인터럽트는 CPU 내부에서 발생되는 인터럽트로 오류나 인터럽트 명령에 의해 발생된다. 오류 인터럽트는 트랩(*trap*)이라고도 불린다. 인터럽트 명령을 통해 발생되는 인터럽트는 소프트웨어 인터럽트(*software interrupt*)라 부른다. DOS는 자체 API(Application Programming Interface)를 구현하기 위해 이러한 인터럽트를 사용하였다. 많은 수의 현대 운영체계(윈도우즈, 유닉스 등등)에서는 C 기반의 인터페이스(interface)를 사용한다.<sup>5</sup>

많은 인터럽트 핸들러들은 자신의 역할이 끝나면 중단되었던 프로그램으로 제어 흐름이 돌려진다. 그들은 인터럽트 되기 전에 레지스터에 저장되어 있던 값을 모두 복원한다. 따라서 중단되었던 프로그램도 마치 아무 일이 없었던 것처럼 다시 원상태로 실행될 수 있다. (다만, 일부 CPU 사이클들을 잊어버렸다) 트랩의 경우 대부분 리턴되지 않으며 종종 프로그램을 종료 시킨다.

## 1.3 어셈블리어

### 1.3.1 기계어

CPU들은 자기들 만이 이해할 수 있는 기계어를 가지고 있다. 기계어의 명령들은 메모리에 바이트로 저장되어 있는 숫자들이다. 각 명령들은 자기만의 유일한 수 코드를 가지고 있고 이것을 연산 부호(*operation code*) 또는 줄여서 *opcode*라 부른다. 80x86 프로세서 명령의 크기는 각각 다르며 연산 부호는 언제나 명령 앞 부분에 위치한다. 많은 명령들은 명령에 의해 사용되는 데이터를 포함한다(*e.g.*, 상수나 주소)

기계어는 직접적으로 프로그래밍하기가 매우 힘들다. 수치화된 각각의 연산 부호들을 해독하는 일이 인간에게는 매우 힘든 일이기 때문이다. 예를 들어, EAX와 EBX 레지스터의 값을 더해 다시 EAX 레지스터에 대입하는 문장은 아래와 같이 기계어로 나타내진다.

---

<sup>5</sup>그러나 이들도 커널 수준에서는 저급 수준의 인터페이스를 사용할 것이다.

이는 정말로 이해하기 힘들다. 다행이도 어셈블러(assembler) 라 불리는 프로그램이 위와 같은 지루한 일을 프로그래머가 안해도 되게 해준다.

### 1.3.2 어셈블리어

어셈블리 언어 프로그램은 고급 언어 프로그램처럼 문자 형태로 저장된다. 각각의 어셈블리 명령들은 하나의 기계 명령에 대응된다. 예를 들어서 위에 예로 든 덧셈 연산은 어셈블리어로 아래와 같이 표현된다.

```
add eax, ebx
```

이를 통해 단순히 숫자로 나타낸 기계어 보다 훨씬 알기 쉽게 명령을 나타낼 수 있다. add 는 덧셈을 한다라는 명령을 인간이 알기 쉬운 문자열로 대응 시킨 연상 기호(mnemonic)이다. 어셈블리어의 일반적인 형식은 아래와 같다

*mnemonic operand(s)*

어셈블러(assembler)는 어셈블리 명령들로 작성된 파일을 읽어 들어서 기계어로 작성된 코드로 변환해 주는 프로그램이다. 컴파일러(Compiler)는 위와 비슷한 작업을 하지만 다만 고급 언어로 작성된 코드를 기계어로 변환해 준다. 모든 어셈블리 문장은 하나의 기계어 문장을 직접 적으로 나타낸다. 한편 고급 언어의 경우 하나의 문장이 훨씬 많고 복잡한 기계 명령을 나타낸다.

컴퓨터 과학자들은 컴파일러 자체를 어떻게 작성할지에 대해 수년간 연구하였다!

어셈블리어와 고급언어의 또 다른 차이점은 모든 CPU 가 각각의 고유한 기계어를 가지고 있고 또 고유의 어셈블리어를 가지고 있다는 점이다. 다른 컴퓨터 아키텍쳐에 어셈블리 프로그램을 포팅(porting) 하는 이는 고급 언어에서 처리하는 것 보다 훨씬 어려운 일이다.

이 책에서는 예제들은 Netwide Assembler 또는 NASM 이라 불리는 어셈블리어를 사용할 것이다. 이는 인터넷에서 무료로 받을 수 있다. (머리말 참조) 다른 어셈블러들로는 마이크로소프트 사의 MASM이나 볼랜드 사의 TASM 을 들 수 있다. NASM 과 MASM/TASM 의 문법은 약간 다르다.

### 1.3.3 피연산자

기계 코드 명령들에 따라 피연산자의 형태와 개수가 다르다. 그러나 대부분 각각의 명령들은 고정된 수의 피연산자 수(보통 0 - 3)를 갖는다. 피연산자로 올 수 있는 것들은 아래와 같다.

**레지스터:** 이 피연산자들은 CPU 레지스터들에 직접적으로 접근할 수 있다.

**메모리:** 이는 메모리에 저장된 데이터를 가리킨다. 이 때, 메모리의 주소값은 명령에 직접적으로 사용하거나, 레지스터에 저장하여 사용할 수도 있다. 언제나 세그먼트 최상단 부터의 오프셋 값으로 나타낸다.

**즉시 피연산자:** 즉시 피연산자(immediate)는 명령 자체에 있는 고정된 값들이다. 이들은 명령 자체에 저장된다(즉, 데이터 세그먼트가 아닌 코드 세그먼트에 저장됨)

**묵시적 피연산자:** 묵시적 피연산자(implied)는 정확히 나타나지 않는다. 예를 들어서 증가 명령은 레지스터나 메모리에 1을 더한다. 이 때 1을 묵시적 피연산자라고 부른다.

#### 1.3.4 기초 명령

어셈블리어에서 가장 기본이 되는 명령은 MOV 명령이다. 이는 특정한 지점의 데이터를 다른 지점으로 이동시킨다. (이는 고급 언어의 대입 연산자와 같다) 이 명령은 두 개의 피연산자를 필요로 한다

```
mov dest, src
```

src에 있는 데이터가 dest로 복사된다. 이 때, 두 피연산자가 모두 메모리이면 안된다. 이는 어셈블리어의 한 가지 흠이라 볼 수 있다. 어셈블리에서는 명령문에 따라 특정한 기준 없이 규칙이 달라지는 경우가 있다. 두 개의 피연산자의 크기가 같아야 한다. 예를 들어 AX(16비트)에 있는 값은 BL(8비트)로 저장될 수 없다.

여기 예제가 있다. (세미 콜론은 주석의 시작을 나타낸다):

```
mov    eax, 3    ; eax 레지스터에 3을 대입한다 (이 때 3은 즉시 피연산자이다)
mov    bx, ax    ; ax 레지스터의 값을 bx 레지스터에 대입한다.
```

ADD 명령은 두 개의 정수를 더할 때 사용된다.

```
add    eax, 4    ; eax = eax + 4
add    al, ah    ; al = al + ah
```

SUB 명령은 한 정수에서 다른 정수를 뺄 때 사용된다.

```
sub    bx, 10    ; bx = bx - 10
sub    ebx, edi ; ebx = ebx - edi
```

INC와 DEC는 1을 증가 또는 감소시키는 명령이다. 1이 묵시적 피연산자이므로 INC와 DEC의 기계 코드는 동일한 작업을 수행하는 ADD와 SUB의 기계 코드의 크기 보다 작다.

```
inc    ecx        ; ecx++
dec    dl         ; dl--
```

### 1.3.5 지시어

지시어(*directive*)는 CPU가 아닌 어셈블러를 위해 만들어진 것이다. 이것들은 주로 어셈블러로 하여금 무언가를 하게 하거나 아니면 어셈블러에게 무언가를 알려주는 역할을 한다. 지시어는 기계 코드로 변환되지 않는다. 지시어들의 주 용도로는:

- 상수를 정의한다
- 데이터를 저장할 메모리를 정의한다
- 메모리를 세그먼트로 묶는다
- 조건적으로 소스 코드를 포함시킨다
- 다른 파일들을 포함시킨다.

NASM은 C와 비슷한 전처리기 명령들이 있다. 그러나 NASM의 전처리기 지시어는 C에서의 #이 아닌 %을 사용한다.

#### equ 지시어

equ 지시어는 심볼(*symbol*)을 정의할 때 사용된다. 심볼은 어셈블리 프로그래밍 시 사용되는 상수를 뜻한다. 이는 아래와 같이 사용한다

```
symbol equ value
```

한 번 정의된 심볼의 값은 절대로 재정의 될 수 없다.

#### The % 정의 지시어

이 지시어는 C에서의 `#define` 지시어와 비슷하다. 이는 C에서처럼 상수 매크로를 정의할 때 사용된다

```
%define SIZE 100
        mov    eax, SIZE
```

위의 나온 코드는 SIZE라는 이름의 매크로를 정의하고 이를 MOV 명령에 서 어떻게 사용하는지 보여주고 있다. 매크로는 심볼들 보다 좀 더 유연한다. 왜냐하면 매크로는 심볼들과는 다르게 재정의 될 수 있고 단순한 수가 아니여도 되기 때문이다.

#### 데이터 지시어

데이터 지시어들은 데이터 세그먼트에서 메모리 상의 공간을 정의하는데 사용된다. 메모리 공간이 정의되는데에는 2 가지 방법이 있다. 첫 번째 방법은 오직 데이터를 위한 공간들만 정의한다. 두 번째 방법은 초기의 값들을 위한 방들을 정의한다. 첫 번째 방법의 경우 RESX 지시어를 사용한다.

Unit	Letter
바이트	B
워드	W
더블워드	D
쿼드워드	Q
10 바이트	T

Table 1.3: RESX 와 DX 지시어를 위한 글자

*X*에는 저장될 데이터의 크기를 나타내는 문자가 들어간다. 이는 표 1.3를 참조하여라.

두 번째 방법(초기값도 정의하는)은 DX 지시어를 사용한다. *X*에는 역시 RESX 지시어 사용시 들어가는 문자가 들어가게 된다.

메모리 위치를 라벨(label)로 표시하는 것은 매우 흔한 일이다. 라벨은 코드 상에서 특정한 메모리 위치를 가리킬 수 있다. 아래 예제를 참조하면 :

```

L1 db 0 ; L1 토 라벨 불여진 바이트 가 0 으로 초기화 됨
L2 dw 1000 ; L2 토 라벨 불여진 워드가 1000 으로 초기화 됨
L3 db 110101b ; L3 토 라벨 불여진 바이트 가 이진수 110101(십진수로 53)로 초기화 됨
L4 db 12h ; L4 토 라벨 불여진 바이트 가 16진수 12 토 초기화 됨
L5 db 17o ; L5 토 라벨 불여진 바이트 가 8진수 17 토 초기화됨
L6 dd 1A92h ; L6 토 라벨 불여진 더블워드가 16진수 1A92 토 초기화 됨
L7 resb 1 ; L7 토 라벨 불여진 초기화 되지 않은 1 바이트
L8 db "A" ; L8 토 라벨 불여진 아스키 코드 A(65) 토 초기화 됨

```

큰따옴표와 작은따옴표는 모두 같은 것으로 취급된다. 데이터가 연속적으로 정의되면 그 데이터들은 메모리 상에 연속되게 존재하게 된다. 따라서, 메모리 상에서 L2는 L1 바로 다음으로 위치하게 된다. 메모리 상에서의 순서 또한 정의 될 수 있다.

```

L9 db 0, 1, 2, 3 ; 4 바이트를 정의한다
L10 db "w", "o", "r", 'd', 0 ; C 문자열 "word" 를 정의한다
L11 db 'word', 0 ; L10 과 같다

```

DD 지시어는 정수와 단일 정밀도 부동 소수점<sup>6</sup> 상수들을 정의할 수 있다. 그러나 DQ의 경우 오직 2배 정밀도 부동 소수점 상수만 정의할 수 있다.

크기가 큰 것들을 위해선 NASM의 TIMES 지시어를 이용하면 된다. 이 지시어는 피연산자를 특정한 횟수 만큼 반복한다. 예를 들면

```

L12 times 100 db 0 ; 이는 100 개의 (db 0) 을 나열하는 것과 같다
L13 resw 100 ; 100 개의 워드를 위한 공간을 정의한다.

```

<sup>6</sup>단일정밀도 부동 소수점은 C 언어의 float 와 같다

아까 라벨이 코드 상의 데이터를 가리키는데 쓰인다는 말을 상기해보자. 라벨이 사용되는 용도는 2 가지가 있다. 만약 라벨을 그냥 이용한다면 이는 데이터의 주소 (즉, 오프셋) 으로 생각된다. 만약 라벨이 대괄호([]) 속에 사용된다면 이는 그 주소에 위치한 데이터를 나타낸다. 다시 말해 라벨을 어떤 데이터에 대한 포인터(pointer) 라 하면 대괄호는 C 언어의 와 같은 역할을 하는 것이다. (MASM/TASM 에서는 다른 방법을 사용한다) 32비트 모드에서는 주소들의 크기는 32비트 이다. 아래 예제를 보면:

```

1    mov    al, [L1]      ; AL 에 위치한 데이터를 대입한다.
2    mov    eax, L1       ; EAX = L1 에 위치한 바이트의 주소
3    mov    [L1], ah      ; L1 에 위치한 바이트에 ah 를 대입한다
4    mov    eax, [L6]      ; L6 에 위치한 더블워드를 EAX 에 대입한다
5    add    eax, [L6]      ; EAX = EAX + L6 에 위치한 더블워드
6    add    [L6], eax      ; L6 에 위치한 더블워드 += EAX
7    mov    al, [L6]      ; L6 에 위치한 더블워드의 하위 비트를 AL 에 대입한다

```

7 번째 행을 보면 NASM 의 중요한 특징을 알 수 있다. 어셈블러는 라벨이 어떠한 데이터를 가리키고 있는지 전혀 신경쓰지 않는다. 이는 모두 프로그래머에 달려 있기에 언제나 라벨을 정확히 사용하는데 신중을 기해야 할 것이다. 나중에 데이터의 주소값을 레지스터에 저장하여 C 언어의 포인터처럼 쓰는 연산을 많이하게 된다. 이 때도 이 포인터가 정확하게 사용되고 있는지에 대해선 어셈블러가 전혀 신경쓰지 않는다. 이 때문에 어셈블리를 이용한 프로그래밍은 C 보다도 오류가 잣아지게 된다.

다음과 같은 명령을 고려하자.

```
mov    [L6], 1           ; L6 에 위치한 데이터에 1 을 저장한다
```

위 문장은 명령의 크기가 정의되지 않았습니다 라는 오류를 발생한다. 왜? 왜냐하면 어셈블러가 1 을 바이트로 저장할지, 워드로 저장할지, 더블워드로 저장할지 모르기 때문이다. 이 문제를 해결하기 위해선 아래와 같이 해주면 된다.

```
mov    dword [L6], 1      ; L6 에 1 을 저장한다
```

이를 통해 어셈블러는 1 을 메모리 상의 L6 에서 시작하는 부분부터 더블워드로 저장하라는 것을 알 수 있다. 그 외에도 BYTE, WORD, QWORD, TWORD<sup>7</sup> 가 있다.

### 1.3.6 입출력

입출력(I/O)은 매우 시스템 종속적인 작업들이다. 이는 컴퓨터 시스템의 하드웨어와의 소통을 필요로 한다. C 와 같은 고급 언어에선 I/O 를

---

<sup>7</sup>TWORD 는 메모리 상의 10 바이트를 정의한다 부동 소수점 보조 프로세서가 이 데이터 형식을 사용한다.

<b>print_int</b>	EAX에 저장된 값을 화면에 출력한다.
<b>print_char</b>	AL에 저장된 아스키 코드 값에 대응하는 문자를 화면에 표시한다
<b>print_string</b>	EAX에 저장된 문자열의 주소값에 해당하는 문자열을 화면에 출력한다. 문자열은 반드시 C 형식 이어야 한다( <i>i.e.</i> null로 종료).
<b>print_nl</b>	화면에 개행문자를 출력한다.
<b>read_int</b>	키보드로부터 정수를 입력받은 다음 EAX 레지스터에 저장한다.
<b>read_char</b>	키보드로부터 한 개의 문자를 입력받은 후 그 문자의 아스키코드값을 EAX 레지스터에 저장한다.

Table 1.4: 어셈블리 I/O 루틴

위한 표준 라이브러리를 제공한다. 어셈블리 언어에서는 C와 같은 라이브러리가 제공되지 않는다. 따라서, 어셈블리에선 직접적으로 하드웨어에 접근하거나 (이는 보호 모드에서 특별한 권한을 가지는 작업이기도 한다) 운영체제가 제공하는 저수준의 루틴들을 사용해야 한다.

어셈블리 루틴과 함께 C 언어를 사용하는 것은 매우 흔한 일이다. 왜냐하면 어셈블리 코드가 표준 C 입출력 라이브러리 루틴을 사용할 수 있기 때문이다. 그러나, C가 사용하는 루틴에게 정보를 전달하는데에는 규칙이 있다는 점을 알아야 한다. 그 규칙들을 여기서 직접 다루기에는 복잡하다 (나중에 다룰 것이다). 저자는 입출력을 단순화하기 위해 복잡한 C 규칙을 숨기고 훨씬 간단한 인터페이스를 이용하는 루틴을 제공한다. 표 1.4에 저자가 개발한 루틴들에 대해 설명하고 있다. 읽기 루틴만을 제외한 모든 루틴들은 모든 레지스터의 값을 저장한다. 이 루틴들을 이용하기 위해서는 이 루틴들에 대한 정보가 기록된 파일들을 소스에 포함시켜 어셈블러가 사용할 수 있게 해야 한다. NASM에서 파일을 포함하기 위해선 `%include` 전처리기 지시어를 사용하면 된다. 저자가 만든 I/O 루틴을<sup>8</sup> NASM에 포함시키기 위해서는 아래와 같이 하면 된다.

```
%include "asm_io.inc"
```

위 I/O 루틴 중 출력 루틴을 사용하고 싶다면 EAX 레지스터에 적당한 값을 대입하고 CALL 명령을 사용해 호출한다. CALL 명령은 고급 언어에서의 함수를 호출하는 것과 동일하다. 이 명령 사용 시 코드의 다른 부분으로 실행을 분기했다가 그 루틴이 종료된다면 다시 원래 호출했던 부분으로 돌아오게(리턴) 된다. 아래의 예제 프로그램은 이러한 I/O 루틴의 호출을 보여준다.

---

<sup>8</sup> `asm_io.inc` (그리고 `asm_io.inc`가 필요로 하는 `asm_io` 파일)은 이 책의 웹 페이지인 <http://www.drpaulcarter.com/pcasm>에서 다운로드가 가능하다

### 1.3.7 디버깅

저자의 라이브러리는 프로그램을 디버깅하는데 유용한 루틴들을 포함하고 있다. 이 디버깅 루틴은 컴퓨터의 상태를 변경하지 않고서 화면에 그대로 상태를 출력하게 해준다. 이 루틴들을 CPU의 현 상태를 저장하고 서브루틴을 호출하는 매크로이다. 이 매크로들은 `asm.io.inc` 파일에 정의되어 있다. 매크로들은 보통 명령들처럼 사용된다. 이 때 매크로들의 피연산자들은 반점(,)을 통해 구분한다.

저자의 라이브러리에는 4개의 디버깅 루틴이 있는데 이는 각각 `dump_regs`, `dump_mem`, `dump_stack`, `dump_math`이다. 이 들은 레지스터, 메모리, 스택, 그리고 수학 보조프로세서에 들어있는 값들을 모두 보여준다

`dump_regs` 이 매크로는 레지스터에 들어있는 값들을 화면에(`stdout`) 16진수로 출력한다. 또한 플래그 레지스터의 비트 세트<sup>9</sup>를 화면에 출력한다. 예를 들어서 제로 플래그가 1이라면 ZF가 화면에 출력된다. 0이라면 화면에 출력되지 않는다. 이 매크로는 한 개의 정수를 인자로 가지는데 이를 통해 서로 다른 `dump_regs` 명령의 출력들을 구분 할 수 있게 한다.

`dump_mem` 이 매크로는 메모리의 일부 영역의 값들을 16진수나 아스키 문자로 출력한다. 이 매크로는 반점으로 구분되는 3개의 매개 변수를 가진다. 첫 번째는 `dump_regs` 매크로처럼 출력을 구분하기 위한 정수, 두 번째는 출력하기 위한 메모리의 주소, 마지막은 주소 뒤에 출력될 16바이트 패러그래프의 수이다. 요청된 메모리 주소 이전의 패러그래프 경계 부터 메모리 값들이 출력 될 것이다.

`dump_stack` 이 매크로는 CPU의 스택에 저장된 값을 출력한다. (스택에 대해서는 Chapter 4에서 다룰 것이다) 스택은 더블워드로 구성되어 있으며 이 매크로를 통해 스택에 들어 있는 값을 출력할 수 있다. 3개의 반점으로 구분되는 인자를 가지며 첫 번째는 `dump_regs`에서처럼 출력을 구분하기 위한 정수, 두 번째는 EBP 레지스터에 저장된 주소 아래부터 출력할 더블워드의 수, 세 번째는 EBP 레지스터 주소 위로 출력할 더블워드의 수이다.

`dump_math` 이는 수치 부프로세서의 레지스터에 저장되어 있는 값을 출력한다. `dump_regs`에서처럼 결과를 구분하기 위한 정수를 매개 변수로 입력 받는다.

## 1.4 프로그램 만들기

요즘에는 어셈블리어 언어만을 이용하여 프로그램을 만드는 것은 흔치 않은 일이다. 어셈블리는 오직 중요한 일부 루틴에서만 이용하게 된다.

---

<sup>9</sup>Chapter 2에서 다룰 것이다

```

1 int main()
2 {
3     int ret_status ;
4     ret_status = asm_main();
5     return ret_status ;
6 }
```

Figure 1.6: driver.c 코드

왜냐하면, 어셈블리 언어 보다 고급 언어로 작성하는 일이 훨씬 편하기 때문이다. 또한 어셈블리 언어를 이용시 프로그램을 다른 플랫폼에 프로그램을 포팅하는 작업이 매우 어렵게 된다. 사실상, 어셈블리어를 사용하는 일은 드물다.

하지만 우리는 왜 어셈블리 언어를 배워야 하는가?

1. 때때로 컴파일러가 생성한 코드 보다 어셈블리어로 직접 쓴 코드가 훨씬 빠르고 작을 수 있다.
2. 어셈블리어를 통해 고급언어 사용시 힘들거나 불가능한 시스템 하드웨어의 직접적인 접근을 할 수 있다.
3. 어셈블리 언어를 공부함을 통해서 컴퓨터의 작동 원리를 깊게 파악할 수 있다.
4. 어셈블리 언어를 공부함을 통해 컴파일러와 C 와 같은 고급 언어가 어떻게 동작하는지 파악하는데 도움이 된다.

특히 마지막 두 가지 이유를 보면 우리는 어셈블리로 직접 프로그래밍을 하지 않아도 많은 도움을 받을 수 있다는 사실을 알 수 있다. 사실 저자인 나도 어셈블리어 만으로는 거의 프로그램을 짜지 않지만 어셈블리에서 얻은 아이디어를 매일 사용하고 있다.

#### 1.4.1 첫 프로그램

우리가 처음에 작성할 프로그램들은 그림 1.6 와 같은 단순한 C 드라이버 프로그램 들이다. 이는 단순히 `asm_main` 함수 만을 호출한다. 이 루틴은 우리가 나중에 어셈블리로 쓸 것이다. C 드라이버 루틴을 이용하면 몇 가지 장점들이 있다. 첫 번째로 C 시스템은 보호 모드에서 프로그램이 잘 돌아가도록 모든 것을 설정해 준다. C 에 의해 모든 세그먼트와 이에 대응 되는 세그먼트 레지스터들은 초기화 된다. 어셈블리에서도 걱정할 필요는 없다. 두 번째로 어셈블리 코드를 사용하여 C 라이브러리를 사용할 수 있다. 저자의 I/O 루틴이 이 사실을 이용한다. 이들은 C 의 I/O 함수를 사용한다 (`printf` 등등) 아래에는 간단한 어셈블리 프로그램을 소개했다.

---

first.asm

---

```

1 ; file: first.asm
2 ; 최초의 어셈블리 프로그램. 이 프로그램은 2 개의 정수를 입력 받아
3 ; 그 합을 출력한다.
4 ;
5 ; 실행 가능한 프로그램을 만들려면 DJGPP 를 이용 해라 :
6 ; nasm -f coff first.asm
7 ; gcc -o first first.o driver.c asm_io.o
8
9 %include "asm_io.inc"
10 ;
11 ; 초기화 된 데이터는 .data 세그먼트에 들어간다.
12
13 segment .data
14 ;
15 ; 아래 라벨들은 출력을 위한 문자열들을 가리킨다.
16 ;
17 prompt1 db      "Enter a number: ", 0          ; 널 종료 문자임을 잊지 말라!
18 prompt2 db      "Enter another number: ", 0
19 outmsg1 db      "You entered ", 0
20 outmsg2 db      " and ", 0
21 outmsg3 db      ", the sum of these is ", 0
22
23 ;
24 ; 초기화 되지 않은 데이터는 .bss 세그먼트에 들어간다.
25 ;
26 segment .bss
27 ;
28 ; 이 라벨들은 입력 값을 저장하기 위한 더블워드를 가리킨다.
29 input1 resd 1
30 input2 resd 1
31
32 ;
33 ; 코드는 .text 세그먼트에 들어간다.
34 ;
35 segment .text
36     global _asm_main
37 _asm_main:
38         enter 0,0           ; 셋업(set up) 루틴
39         pusha
40
41         mov    eax, prompt1 ; prompt 를 출력

```

```

42      call    print_string
43
44      call    read_int      ; 정수를 읽는다.
45      mov     [input1], eax ; input1에 저장.
46
47      mov     eax, prompt2 ; prompt를 출력
48      call    print_string
49
50      call    read_int      ; 정수를 읽는다.
51      mov     [input2], eax ; input2에 저장.
52
53      mov     eax, [input1]  ; eax = input1에 위치한 dword
54      add     eax, [input2]  ; eax += input2에 위치한 dword
55      mov     ebx, eax       ; ebx = eax
56
57      dump_regs 1          ; 레지스터의 값을 출력
58      dump_mem  2, outmsg1, 1 ; 메모리를 출력
59 ;
60 ; 아래 단계별로 메세지를 출력한다.
61 ;
62      mov     eax, outmsg1
63      call    print_string  ; 첫 번째 메세지를 출력
64      mov     eax, [input1]
65      call    print_int     ; input1을 출력
66      mov     eax, outmsg2
67      call    print_string  ; 두 번째 메세지를 출력
68      mov     eax, [input2]
69      call    print_int     ; input2를 출력
70      mov     eax, outmsg3
71      call    print_string  ; 세 번째 메세지를 출력
72      mov     eax, ebx
73      call    print_int     ; 합을 출력 (ebx)
74      call    print_nl      ; 개행문자를 출력
75
76      popa
77      mov     eax, 0          ; C로 리턴된다.
78      leave
79      ret

```

---

first.asm

13 행에서 프로그램의 데이터 세그먼트(.data)에 저장될 메모리 공간을 정의하였다. 이 세그먼트에는 오직 초기화된 데이터 만이 올 수 있다. 17에서 21 행에서는 몇 개의 문자열들이 선언되었다. 이들은 C 라이브러리를

통해 출력되기 때문에 반드시 널(NULL) 문자로 끝마쳐져야 한다 (아스키 코드 값이 0). 참고로 0 과 '0'에는 매우 큰 차이가 있다는 사실을 알아두라.

초기화 되지 않은 데이터는 반드시 bss 세그먼트(.bss)에서 선언되어야 한다. 이 세그먼트는 초기 유닉스 기반의 어셈블러에서 사용된 “심볼을 통해 시작된 블록(block started by symbol).” 연산자의 이름에서 따온 것이다. 이것에 대해선 나중에 다루겠다.

코드 세그먼트는 역사적으로 .text 라 불려왔다. 이는 명령어들이 저장되는 곳이다. (38 행)의 메인 루틴의 라벨에 접두어 \_ (under score)가 사용되었음을 유의하자. 이는 C 함수 호출 규약(C calling convention)의 일부분이다. 이 규약은 코드 컴파일 시에 C 가 사용하는 규칙들을 명시해 준다. 이는 어셈블리어와 C 를 같이 사용시에 이 호출 규약을 아는 것이 매우 중요하다. 나중에 이 호출 규약에 대해 속속 살펴 볼 것이다. 그러나, 지금은 오직 모든 C 심볼들 (i.e., 함수와 전역 변수)에는 C 컴파일러에 의해 \_ 가 맨 앞에 붙는다는 사실만 알면 된다. (이 규칙은 오직 DOS/Windows 에만 해당하며 리눅스 C 컴파일러에서는 C 심볼 이름에 아무것도 붙이지 않는다)

37 행의 global 지시어는 어셈블러로 하여금 \_asm\_main 라벨을 전역으로 생성하라고 알려준다. C에서와는 달리 라벨들은 내부 지역(internal scope)이 기본으로 된다. 이 뜻은 오직 같은 모듈에서만 그 라벨을 사용할 수 있다는 뜻이다. 하지만 전역(global) 지시어를 이용하면 라벨을 외부 지역(external scope)에서도 사용할 수 있게 된다. 이러한 형식의 라벨은 프로그램의 어떠한 다른 모듈에서도 접근이 가능하다. asm\_io 모듈은 print\_int 등을 모두 전역으로 선언한다. 이 때문에 우리가 first.asm 모듈에서 이를 사용할 수 있었다.

#### 1.4.2 컴파일러 의존성

위에 나온 어셈블리 코드는 오직 무료 GNU<sup>10</sup> 의 DJGPP C/C++ 컴파일러에서만 돌아간다<sup>11</sup> 이 컴파일러는 인터넷을 통해 무료로 다운로드 할 수 있다. 이 컴파일러는 386 이상의 PC 를 필요로 하며, DOS 나 Windows 95/98, NT 등에서 잘 실행된다. 이 컴파일러는 COFF(Common Object File Format) 형식의 목적 파일(Object File) 을 이용한다. 이 형식으로 어셈블하기 위해서는 nasm 과 함께 -f coff 스위치를 사용해야 한다. (위 소스코드 주석이 잘 나와 있다) 생성되는 목적 파일의 확장자는 o 가 된다.

리눅스 C 컴파일러 또한 GNU 컴파일러이다. 위 코드를 리눅스에서 실행되게 바꾸려면 단순히 37에서 38 행의 \_ 를 지워주기만 하면 된다. 리눅스에서 목적파일은 ELF(링크 및 실행 가능한 포맷, Executable and Linkable Format) 형식을 이용한다. 리눅스에선 -f elf 스위치를 이용하면 된다. 또한 목적파일의 확장자는 o 이다.

이 컴파일러를 위한 예제  
파일들은 저자의 홈페이지  
에서 다운 받을 수 있다.

<sup>10</sup>무료 소프트웨어 재단(Free Software Foundation)

<sup>11</sup><http://www.delorie.com/djgpp>

볼랜드 C/C++ 는 또 다른 유명한 컴파일러이다. 이는 목적파일을 마이크로소프트의 OMF 형식을 이용한다. 볼랜드사 컴파일러를 위해선 -f obj 스위치를 이용하면 된다. OMF 형식은 다른 목적 파일 포맷들과는 달리 다른 세그먼트 지시어를 사용한다. 13 행의 데이터 세그먼트는 아래와 같이 바꿔어야 한다. 목적 파일의 확장자는 obj 이다.

```
segment _DATA public align=4 class=DATA use32
```

26 행의 bss 세그먼트는 아래와 같이 바뀐다:

```
segment _BSS public align=4 class=BSS use32
```

36행의 텍스트 세그먼트는 아래와 같이 바뀐다.

```
segment _TEXT public align=1 class=CODE use32
```

뿐만 아니라 36 행에 아래와 같은 새로운 행이 덧붙여져야 한다.

```
group DGROUP _BSS _DATA
```

마이크로소프트 C/C++ 컴파일러는 목적 파일으로 OMF 나 Win32 형식을 이용할 수 있다. (OMF 형식을 이용한다 해도 그 정보를 내부적으로는 Win32 로 바꾸어서 처리한다) Win32 형식을 이용하면 DJGPP 나 리눅스에서처럼 세그먼트를 정의할 수 있다. -f win32 스위치를 통해 위 모드로 출력할 수 있다. 목적 파일의 확장자는 obj 가 된다.

### 1.4.3 코드를 어셈블 하기

커맨드 라인에 아래와 같이 입력한다.

```
nasm -f object-format first.asm
```

이 때, *object-format* 부분에는 *coff*, *elf*, *obj*, *win32* 중 하나가 올 수 있으며 사용할 C 컴파일러에 따라 달라진다. (물론 리눅스나 볼랜드나에 따라서도 소스파일의 내용이 바뀌어야 함을 잊지 말아야한다)

### 1.4.4 C 코드를 컴파일 하기

DJGPP 를 사용한다면:

```
gcc -c driver.c
```

-c 스위치는 단지 컴파일 만을 하란 뜻이다. 아직 링크는 하지 않는다. 리눅스나 볼랜드, 마이크로소프트 컴파일러도 동일한 스위치를 사용한다.

### 1.4.5 목적 파일 링크하기

링킹은 기계어 코드와 목적 파일과 라이브러리 파일에 들어있는 데이터를 하나로 합쳐서 실행 가능한 파일을 만드는 과정이다. 아래에 나타나있듯이 이 과정은 복잡한다.

C 코드는 표준 C 라이브러리와 특별한 개시 코드(*startup code*)를 필요로 한다. 이는 C 컴파일러가 정확한 매개 변수와 함께 링커를 호출하고 직접적으로 링커를 호출하는 작업을 훨씬 쉽게 하게 해준다. 예를 들어서 위 프로그램을 DJGPP 으로 링크 할 때,

```
gcc -o first driver.o first.o asm_io.o
```

를 쓰면 `first.exe` (리눅스에선 그냥 `first`)라는 실행 가능한 파일이 만들어 진다.

볼랜드 컴파일러는

```
bcc32 first.obj driver.obj asm_io.obj
```

와 같이 하면 된다. 볼랜드는 실행 가능한 파일의 이름을 정할 때 나열된 첫 번째 파일의 이름을 따오게 된다. 따라서, 위 경우 프로그램의 이름은 `first.exe` 가 된다.

컴파일과 링크 과정을 하나로 묶어서 할 수도 있다. 예를 들어서

```
gcc -o first driver.c first.o asm_io.o
```

그러면 `gcc` 는 `driver.c` 를 컴파일 한 후 링크할 것이다.

### 1.4.6 어셈블리 리스트 파일 이해

`-l listing-file` 스위치는 `nasm` 으로 하여금 주어진 이름의 리스트 파일을 생성하라고 명령한다. 이 파일은 코드가 어떻게 어셈블 되었는지 보여준다. 아래 17에서 18 행 (데이터 세그먼트) 이 리스트 파일에 어떻게 나타났는지 보여준다. (리스트 파일의 행 번호를 잘 보면 실제 소스 파일의 행 번호와 다름을 알 수 있다)

```
48 00000000 456E7465722061206E-      prompt1 db      "Enter a number: ", 0
49 00000009 756D6265723A2000
50 00000011 456E74657220616E6F-      prompt2 db      "Enter another number: ", 0
51 0000001A 74686572206E756D62-
52 00000023 65723A2000
```

첫 번째 열은 각 행의 번호를 표시하고 두 번째 열은 세그먼트의 데이터의 오프셋을 16 진수로 나타낸다. 세 번째 열은 저장될 16 진수 값을 보여준다. 위 경우 그 값은 아스키코드 값과 대응이 된다. 마지막으로 소스 파일의 코드가 나타나게 된다. 두 번째 열에 나타나는 오프셋은 실제 프로그램에서의

데이터의 오프셋과 다를 가능성이 매우 높다. 각각의 모듈은 데이터 세그먼트(를 포함한 다른 세그먼트들에서도)에서의 자신만의 라벨을 가지고 있다. 링크 과정에선 ( 1.4.5 참조 ) 에선 모든 데이터 세그먼트의 라벨들의 정의가 하나의 데이터 세그먼트로 통합된다. 그리고 마지막으로 정리된 오프셋들은 링커에 의해 계산된다.

아래 소스코드의 54에서 56 행의 텍스트 세그먼트의 리스트 파일에서 해당되는 부분을 보여주고 있다.

```

94 0000002C A1[00000000]      mov     eax, [input1]
95 00000031 0305[04000000]    add     eax, [input2]
96 00000037 89C3              mov     ebx, eax

```

3 번째 열은 어셈블리에 의해 만들어진 기계어 코드를 보여준다. 보통 명령어들의 완전한 코드는 아직 계산되지 못한다. 예를 들어서 94 행에서 input1의 오프셋(혹은 주소)은 코드가 링크되기 전 까지는 확실히 알 수 없다. 어셈블러는 mov 명령의 연산 부호(Opcode)는 계산할 수 있지만 (이는 A1에 나타남) 대괄호 안의 정확한 값은 알 수 없다. 위 경우 어셈블러는 위 파일에서 정의된 bss 세그먼트의 시작을 임시적으로 오프셋 0으로 하였다. 이것이 프로그램의 마지막 bss 세그먼트의 시작을 의미한다는 것이 아니라는 사실을 명심하여라. 코드가 링크가 된다면 링커는 정확한 오프셋 값을 그 자리에 집어 넣을 것이다. 다른 명령들의 경우, 예를 들어 96 행을 본다면 어떠한 라벨도 참조하지 않음을 알 수 있다. 여기에선 어셈블러가 정확한 기계어 코드를 계산할 수 있다.

### 빅, 리틀 엔디안 표현

95 행을 자세히 본다면 기계어 코드의 대괄호 속에 있는 오프셋 값이 매우 이상하다는 것을 알 수 있다. input2 라벨은 오프셋 4 (파일에 정의되어 있는 대로)이다. 그러나, 메모리에서의 오프셋은 00000004가 아니라 04000000이다! 왜냐하면 프로세서가 정수를 메모리 상에 다른 순서로 저장하기 때문이다. 정수를 저장하는 방법에는 대표적으로 2 가지 방법이 있다. 빅 엔디안(Big endian)과 리틀 엔디안(Little endian)이 그것이다. 빅 엔디안은 우리가 생각하는 그대로 정수를 저장한다. 가장 큰 바이트(*i.e.* 가장 상위 바이트)가 먼저 저장되고 가장 하위 바이트가 나중에 저장된다. 예를 들어서 dword 00000004는 4 개의 바이트 00 00 00 04로 저장된다. IBM 메인 프레임, 대부분의 RISC 프로세서, 그리고 모토롤라 프로세서들은 빅 엔디안 방법을 이용한다. 그러나 인텔 기반의 프로세서는 모두 리틀 엔디안 방식을 이용한다. 여기서는 하위 바이트가 먼저 저장된다. 따라서 00000004는 메모리 상에 04 00 00 00으로 저장된다. 이 형식은 CPU에 따라 정해져 있어서 바뀔 수 없다. 보통 프로그래머는 어떤 방식을 사용하느냐를 구별한 필요가 없다. 그러나 이 사실을 아는 것이 중요한 이유가 몇 가지 있는데

1. 이진 데이터가 서로 다른 컴퓨터로 전송될 때 (파일 또는 네트워크를 통해서 ).

```

1  %include "asm_io.inc" _____ skel.asm _____
2  segment .data
3  ;
4  ; 초기화된 데이터는 여기 데이터 세그먼트에 들어간다.
5  ;
6
7  segment .bss
8  ;
9  ; 초기화되지 않은 데이터는 여기 bss 세그먼트에 들어간다.
10 ;
11
12 segment .text
13     global _asm_main
14 _asm_main:
15     enter 0,0           ; 셋업 루틴
16     pusha
17
18 ;
19 ; 코드는 텍스트 세그먼트에 들어간다. 이 주석 앞 뒤의 코드를
20 ; 수정하지 마세요
21 ;
22
23     popa
24     mov    eax, 0         ; C로 리턴된다.
25     leave
26     ret

```

Figure 1.7: 뼈대 프로그램

2. 멀티바이트(multibyte) 정수 데이터가 메모리에 쓰이고 개개의 바이트로 읽어 질 때(역도 같음)

엔디안 표현은 배열의 원소들의 순서에 대해서는 영향을 주지 않는다. 배열의 첫 번째 원소는 언제나 최하위의 주소를 갖는다. 이는 문자열(단지 문자들의 배열인)에서도 해당된다. 엔디안 표현은 단지 배열의 개개의 원소들에 대해서만 영향을 줄 뿐이다.

## 1.5 뼈대 파일

그림 1.7은 뼈대 파일(Skeleton file)을 보여준다. 어셈블리 프로그램 코딩시 매번 같은 내용을 작성하기보단 위 코드를 기준으로 시작하면 편리

하다.



## Chapter 2

# 어셈블리 언어의 기초

### 2.1 정수들 다루기

#### 2.1.1 정수의 표현

정수는 두 가지 종류로 나눌 수 있다 : 부호 없는 정수(unsigned) 와 부호 있는 정수(signed) 이다. 부호 없는 정수 (즉, 음이 아닌) 는 이진수로 직관적으로 표현된다. 예를 들어 200 은 1 바이트 부호 없는 정수로, 11001000 (또는 16 진수로 C8) 로 나타난다.

부호 있는 정수 (이는 양수 또는 음수가 될 수 있다) 는 좀 더 복잡한 방법으로 표현된다. 예를 들어, -56 과 +56 은 바이트로 똑같이 00111000 으로 표현될 것이다. 종이에다가는 우리가 -56 을 -111000 이라 쓸 수 있지만 컴퓨터 메모리 상에선 어떻게 할 것인가? 어떻게 마이너스 기호가 저장될 수 있는가?

컴퓨터 메모리에 부호 있는 정수를 나타내는 방법은 대표적으로 3 가지가 있다. 이 모든 방법들은 공통적으로 정수의 최상위 비트를 부호 비트(sign bit) 로 사용한다. 이 비트는 만약 이 정수가 양수이면 0, 음수이면 1의 값을 가진다.

#### 부호 있는 크기 표현법

첫 번째 방법은 가장 단순한 방법인데, 부호 있는 크기(signed magnitude) 표현법 이라 부른다. 이는 정수를 두 부분으로 나누는데, 첫 번째 부분은 부호 비트이고 나머지 부분 그 정수의 절대값(크기)을 나타낸다. 따라서 56 은 바이트로 00111000 으로 나타내지고 (부호 비트에 밑줄 쳐짐) -56 은 바이트로 110111000 이 된다. 가장 큰 바이트 값은 01111111, 혹은 +127 이 되고 가장 작은 바이트 값은 11111111 혹은 -127 이 된다. 어떠한 값의 부호를 바꾸려면 단순히 부호 비트의 값만 바꾸어 주면 된다. 이 표현법은 상당히 직관적이지만 몇 가지 단점이 있다. 0 은 양수도, 음수도 아니기 때문에 0 을 두 가지로 표현할 수 있게 된다. +0(00000000) 과 -0(10000000) 이다. 이 때문에 CPU 의 산술 연산 로직이 복잡하게 된다. 두 번째로 보통

의 산술 연산 로직도 복잡해 지는데 예를 들어 10 에  $-56$  이 더해진다면 이는 다시 10에서 56 을 빼었다고 생각되어야 하기 때문이다. 결과적으로 이 표현법을 사용하면 CPU 의 로직이 복잡해지게 된다.

### 1 의 보수 표현법

두 번째 방법은 1의 보수(*one's complement*) 표현법이다. 어떠한 수의 1의 보수는 단순히 각각의 비트를 반전(1 을 0, 0 을 1로)시키면 얻을 수 있다. (이는 새 비트의 값을  $(1 - \text{이전의 비트})$  로 하는 것으로도 생각할 수 있다.) 예를 들어 00111000 ( $+56$ ) 의 1의 보수는 11000111 이 된다. 1의 보수 표현법이서, 어떤 수의 1의 보수를 계산하는 것은 단순히 - 를 붙이는 것과 같다. 따라서 11000111 은  $-56$  을 표현한 것과 같다. 1의 보수 표현에서 부호 비트의 값이 자연스럽게 바뀌었다. 첫 번째 방법에서처럼 0 을 나타내는 방법은 두 가지가 있는데 00000000 ( $+0$ ) 과 11111111 ( $-0$ ) 이 된다. 16 진수를 이진수로 바꾸지 않고도 보수를 계산하는 방법은 단순히 각 자리를 F (또는 십진수로 15) 에서 빼는 것이다. 예를 들어  $+56$  의 경우 16 진수로 38 로 나타나는데 1의 보수를 찾기위해 각 자리를 F 에서 빼면 C7 이 된다. 이는 위에서 나타난 결과와 일치한다.

### 2의 보수 표현법

위 두 방법들은 모두 초기의 컴퓨터에서만 사용되었던 방법들이다. 현대의 컴퓨터는 이 세 번째 방법을 사용하는데 이를 2의 보수 표현법 이라 한다. 어떤 수의 2의 보수는 아래 두 단계를 통해 찾을 수 있다 :

1. 어떤 수의 1의 보수를 계산한다.
2. 위 단계에서 나온 결과에 1 을 더한다.

아래는 00111000(56) 을 이용한 방법이다. 먼저, 56 의 1의 보수를 계산하면: 11000111. 그리고 1 을 더하면:

$$\begin{array}{r} \underline{11000111} \\ + \quad \quad \quad 1 \\ \hline \underline{11001000} \end{array}$$

2 의 보수 표현법에서 2 의 보수를 계산하는 것은 숫자에 - 를 붙이는 것과 같다. 따라서 11001000 은  $-56$  를 2 의 보수 표현법으로 나타낸 것 이다. 어떤 수에 음을 두 번 취하면 원래 수로 돌아와야 하는데 놀랍게도 2 의 보수 표현법에서는 이를 잘 만족한다. 한 번 11001000 의 2의 보수 표현을 취해 보자.

$$\begin{array}{r} \underline{00110111} \\ + \quad \quad \quad 1 \\ \hline \underline{00111000} \end{array}$$

수	16 진수 표현 n
0	00
1	01
127	7F
-128	80
-127	81
-2	FE
-1	FF

Table 2.1: 2의 보수 표현

2의 보수 표현에서 1을 더할 때, 이 덧셈에서 최상위 비트에서 캐리(carry)가 있을 것이다. 이 캐리는 쓰이지 않는다. 컴퓨터의 모든 데이터들은 정해진 크기(비트의 수)로 보관된다. 두 개의 바이트를 더하면 언제나 결과는 바이트로 나온다. (또한 워드에 워드를 더하면 워드가 나오고, 다른 것들도 매한가지) 이는 2의 보수 표현에서 가장 중요한 부분이다. 예를 들어서 1 바이트의 0을 생각하자. (00000000). 이것의 2의 보수를 계산하면

$$\begin{array}{r}
 & \underline{11111111} \\
 + & \quad \quad \quad 1 \\
 \hline
 c & \underline{00000000}
 \end{array}$$

이 때,  $c$ 는 캐리(carry)를 나타낸다. (이 장 뒷부분에서, 결과에 저장되지 않는 올림이 발생되었음을 어떻게 알아채는지에 대해 이야기 하겠다.) 따라서 0을 나타내는 2의 보수 표현은 오직 한 가지 방법밖에 없다. 이 때문에 2의 보수 표현을 이용한 산술 연산을 이전의 방법을 이용하였을 때 보다 훨씬 쉽게 수행할 수 있게 된다.

2의 보수 표현을 이용하여 한 개의 바이트를 통해 -128에서 +127 까지 나타낼 수 있게 되었다. 표 2.1은 2의 보수 표현으로 나타낸 몇 개의 수를 보여준다. 만약 16 비트가 이용된다면 -32,768에서 +32,767 까지 표현할 수 있게 된다. +32,767은 7FFF로 나타나고, -32,768은 8000으로, -128은 FF80, -1은 FFFF로 나타나게 된다. 32 비트를 이용한다면 대략 -20 억에서 +20 억의 수를 모두 나타낼 수 있게 된다.

CPU는 어떤 바이트 (혹은 워드나 더블워드)가 어떠한 형식으로 표현되어 있는지 알 수 없다. 어셈블리어는 고급 언어가 가지고 있는 형(type)에 대한 개념이 없다. 데이터가 어떻게 생각되느냐는 그 데이터에 어떠한 명령이 사용되는지에 달려 있다. 16 진수 값 FF가 -1로 생각되든지 (부호 있는 경우) 아니면 +255로 생각되든지 (부호 없는 경우) 오직 프로그래머에 달려 있다. 반면에 C 언어는 부호 있는 정수(signed)와 부호 없는 정수(unsigned)를 각각 정의한다. 이는 C 컴파일러로 하여금 그 데이터에 정확한 명령들을 내리게 해준다.

### 2.1.2 부호 확장

어셈블리에서 모든 데이터는 정해진 크기가 있다. 다른 데이터와 사용하기 위해 데이터의 크기를 줄이는 작업을 자주 하게 된다. 크기를 감소시키는 것이 가장 쉽다.

#### 데이터의 크기 줄이기

데이터의 크기를 줄이기 위해 단순히 데이터의 가장자리 비트를 줄이면 된다. 아래는 그 예이다.

```
mov ax, 0034h      ; ax = 52 (16 비트로 저장됨)
mov cl, al         ; cl = ax 의 하위 8 바이트
```

당연하게도, 크기를 줄였는데 수의 값이 정확하게 표현이 되지 않으면 크기를 줄이는 방법은 소용이 없을 것이다. 예를 들어서 AX에 0134h(혹은 10 진수로 308, 이 때 끝에 붙은 h는 이 수가 16 진수(hex)로 나타내져 있음을 알려준다.)이 있었는데 위 코드를 실행하면 CL에 34h가 된다. 이 방법은 부호 있는 정수와 부호 없는 정수 모두 사용 가능하다. 부호 있는 정수를 고려해보면 AX에 FFFFh(워드로 -1)가 있었다면 CL은 FFh(바이트로 -1)이 된다. 그러나, AX의 값이 부호가 없는 경우라면 맞지 않음을 유의해라.

부호 없는 정수의 경우 변환이 올바르게 일어나기 위해선 제거되는 비트들이 반드시 0이어야 한다. 반면에 부호 있는 정수들의 경우 제거되는 비트들은 모두 0이거나 모두 1이어야 한다. 또한, 제거되지 않은 첫 번째 비트는 반드시 제거된 비트들과 같은 값을 가져야한다. 이 비트는 나중에 작아진 값의 새로운 부호 비트가 될 것이다. 이 비트가 이전의 부호 비트와 같아야 한다는 점이 중요하다.

#### 데이터의 크기 늘리기

데이터의 크기를 늘리는 작업은 줄이는 작업보다 좀 더 복잡하다. 예를 들어서 1 바이트 FF를 생각해보자. 만약 이것이 워드로 늘려진다면, 그 워드가 가질 수 있는 값은 무엇이 있을까? 이는 FF가 어떻게 해석되기에 따라 달라진다. 만약 FF가 부호 없는 정수 였다면(십진수로 255), 워드로 확장 시, 그 값은 00FF가 된다. 그러나, 만약 이 FF가 부호 있는 정수(십진수로 -1) 였다면 워드로 확장시 FFFF가 되어야만 한다.

보통 부호 없는 정수를 확장한다면 확장된 새 비트가 모두 0이 된다. 따라서 FF는 00FF가 된다. 그러나, 부호 있는 정수를 확장하려면 반드시 부호 비트를 확장하게 된다. 이 말은 새로운 비트가 이전의 부호 비트의 값과 같다는 뜻이다. 따라서, FF의 부호 비트의 값이 1이였으므로 새롭게 확장된 수에 추가된 비트들은 모두 1이 되어 FFFF가 된다. 만약 부호 있는 정수의 값이 5A(십진수로 90)이였다면 확장시 005A가 될 것이다.

80386 은 몇 가지 수를 확장하는 명령들을 제공한다. 컴퓨터는 그 수가 부호가 있는 정수 인지, 부호 없는 정수인지 알 수 없다는 사실을 명심하라. 이 모든 것은 프로그래머가 정확한 명령을 사용하는데 달려 있다.

부호 없는 정수들에 한해서 MOV 연산을 통해 상위 비트에 0 을 집어 넣을 수 있다. 예를 들어서 바이트 AL 을 워드 AX 로 확장시키기 위해선:

```
mov ah, 0 ; 상위 8 비트들을 모두 0 으로 만들
```

그러나 MOV 명령을 통해 AX 에 저장된 부호 없는 워드를 부호 없는 더블워드인 EAX 에 저장할 순 없다. 왜냐하면, EAX 의 상위 16 비트를 가리키는 말이 MOV 명령에선 없기 때문이다. 80386 은 이 문제를 MOVZX 명령을 추가함으로써 해결하였다. 이 명령은 2 개의 피연산자를 가진다. 값이 들어가는 피연산자(데스티네이션, destination) 는 첫 번째 피연산자로, 16 비트나 32 비트 레지스터여야 한다. 값을 주는 피연산자(소스, source) 는 두 번째 피연산자로 8 또는 16 비트 레지스터, 혹은 메모리상의 바이트나 워드 이어야만 한다. 다른 조건으로는 데스티네이션이 반드시 소스의 크기보다 커야 한다는 점이다. (대부분의 명령들은 모두 소스의 크기와 데스티네이션의 크기가 같아야 한다) 아래 그 예가 있다:

```
movzx eax, ax      ; ax 를 eax 를 확장
movzx eax, al      ; al 을 eax 를 확장
movzx ax, al       ; al 을 ax 를 확장
movzx ebx, ax      ; ax 를 ebx 를 확장
```

부호 있는 정수들의 경우 MOV 연산을 적용하리란 쉽지 않다. 8086 은 부호 있는 정수들의 확장을 위해 몇 가지 명령들을 지원하였다. CBW (바이트를 워드로 변환) 명령은 AL 레지스터를 AX 로 확장해 준다. 이 때, 피연산자는 쓸 필요가 없다. CWD (워드를 더블 워드로 변환) 명령은 AX 를 DX:AX 로 확장해 준다. 이 때, DX:AX 의 뜻은 DX 와 AX 레지스터를 묶어서 하나의 32 비트 레지스터로 생각하란 뜻이다. 이 때 상위 16 비트는 DX 레지스터가, 하위 16 비트는 AX 레지스터가 갖게 된다. (8086은 32 비트 레지스터가 없었음을 상기하자!) 80386 에서는 몇 가지 새로운 명령들을 사용할 수 있다. CWDE (워드를 더블워드 확장으로 변환) 명령은 AX 를 EAX 로 확장한다. CDQ 명령은 EAX 를 EDX:EAX ( 64 비트!) 로 확장해준다. 마지막으로 MOVSX 명령은 부호 있는 정수에 대해서 MOVZX 와 똑같은 작업을 한다.

## C 프로그래밍에 응용하기

C 에서도 부호 있는/없는 정수들의 확장이 사용된다. C 의 변수들은 부호가 있다/없다로 정의된다. (int 는 부호가 있다). 그럼 2.1 의 코드를 생각해 보자. 행 3 에서 변수 a 는 부호 없는 정수들을 위한 확장규칙에 따라 (MOVZX 를 사용하여) 확장되었지만 행 4 를 보면 부호 있는 정수들을 위한 확장규칙에 따라 b 가 확장되었음을 알 수 있다. (MOVSX 를 사용하여)

ANSI C 는 char 가 부호가 있는지 없는지 정의하지 않고 개개의 컴파일러에게 정하라고 한다. 이는 타입이 그림 2.1 에서 명확히 정의 되는 이유이다.

```

1 unsigned char uchar = 0xFF;
2 signed char schar = 0xFF;
3 int a = (int) uchar; /* a = 255 (0x000000FF) */
4 int b = (int) schar; /* b = -1 (0xFFFFFFFF) */

```

Figure 2.1:

```

char ch;
while( (ch = fgetc(fp)) != EOF ) {
    /* ch 로 무언가 작업을 한다 */
}

```

Figure 2.2:

이 주제와 관련하여 대표적인 C 프로그래밍 버그가 있다. 그림 2.2 의 코드를 보자. `fgetc()` 함수의 원형은 :

```
int fgetc( FILE * );
```

이다. 아마 여러분은 이 함수가 문자를 읽어들이는데 왜 `int` 를 반환 하는지 궁금해 할 것이다. 사실 이 함수는 `char` 형태로 반환한다. (다만, `int` 로 앞에 0 이 붙는 방법으로 확장되어서) 그러나 `char` 형태로 반환할 수 없는 것이 딱 하나 있는데 바로 `EOF` 이다. 이 매크로는 보통 `-1` 로 정의된다. 따라서, `fgetc()` 는 `char` 이 `int` 형태로 확장된 값을 반환하거나 (이는 `000000xx` 와 같은 형태로 16 진수로 나타날 것이다) `EOF` (이는 16 진수로 `FFFFFF` 로 나타난다) 를 반환할 것이다.

그림 2.2 의 기본적인 문제는 `fgetc()` 가 `int` 를 반환하지만 그 값은 `char` 에 저장되어 있다는 것이다. C 는 `char` 에 값을 맞추기 위해 `int` 값의 상위 비트들을 잘라낼 것이다. 문제는 (16 진수로) `000000FF` 와 `FFFFFF` 모두 `FF` 로 알려진다는 것이다. 따라서 while 루프는 파일의 바이트 `FF` 를 읽어들이고 있는지, 아니면 파일의 마지막을 읽어들이고 있는지 알 수 없게 된다.

위 상황에서 코드가 어떤 일을 하느냐는 `char` 이 부호가 있느냐, 없느냐에 달려있다. 왜냐하면 2 행에서 `ch` 는 `EOF` 와 비교되기 때문이다. `EOF` 가 `int` 의 값을 가지고 있으므로 <sup>1</sup> `ch` 의 값은 `int` 로 확장되어 두 값이 같은 크기로 비교될 수 있게 한다 <sup>2</sup> 그림 2.1 에서 나타나듯이, 변수에 부호가 있는지 없는지에 대한 유무는 매우 중요하다.

만약 `char` 이 부호가 없다면 `FF` 는 `000000FF` 로 확장될 것이다. 이는 `EOF` (`FFFFFF`) 와 비교되어 값이 다르다고 나타날 것이다. 따라서, 루프에서 절대로 빠져 나갈 수 없다!

<sup>1</sup> 파일의 끝에 `EOF` 문자가 있다는 것은 흔한 잘못된 생각이다. 이는 사실이 아니다

<sup>2</sup> 이에 대한 이유는 나중에 다루도록 하자

만약 `char` 부호가 있다면 FF 는 FFFFFFFF 로 확장될 것이다. 이는 EOF 문자를 입력된다면 성공적으로 루프를 빠져 나갈 수 있게 된다. 그러나, FF 는 파일의 중간에서 입력 될 수도 있다. 따라서, 루프가 파일을 다 읽어들 이지 못한 채 끝날 수도 있다.

위 문제에 대한 해결책은 `ch` 변수를 `char` 이 아닌 `int`로 선언하는 것이다. 이렇게 된다면 2 행에서 값의 확장이나 잘리는 일이 없게 된다. 루프 내부에선 값을 잘라도 아무런 문제가 없는데 왜냐하면 `ch` 가 반드시 단순한 바이트로 행동하기 때문이다.

### 2.1.3 2의 보수의 산술 연산

이미 앞에서 보았듯이 `add` 명령은 덧셈을 수행하고 `sub` 명령은 뺄셈을 수행한다. 이 두 개의 명령들은 플래그 레지스터의 2 개의 비트를 변경하는데 이 비트들은 각각 오버플로우(overflow)와 캐리 플래그(carry flag)이다. 오버플로우 플래그는 부호 있는 정수 연산시, 결과값이 너무나 커서 데스티네이션에 들어갈 수 없을 때 세트(set) 된다. 캐리 플래그는 부호 없는 정수 연산시, 덧셈이나 뺄셈에서 받아 올림이나 받아 내림이 최상위/하위 (MSB, Most Significant Bit) 비트에 있을 때 세트 된다. 부호 있는 연산에서의 캐리 플래그의 역할은 조금 있다가 살펴 볼 것이다. 2의 보수의 가장 큰 장점은 부호 없는 정수에 대해서 덧셈과 뺄셈이 정확히 일치한다는 것이다. 따라서 `add` 와 `sub` 는 부호 있는/없는 정수들에게서 사용 될 수 있다.

$$\begin{array}{r} 002C & 44 \\ + \quad FFFF & + \quad (-1) \\ \hline 002B & 43 \end{array}$$

여기에선 캐리가 발생하지만 결과엔 포함되지 않는다.

곱셈과 나눗셈 연산에는 각각 2 개의 명령들이 있다. 첫째로, 곱셈에선 `MUL`이나 `IMUL` 연산을 사용할 수 있다. `MUL` 연산은 두 개의 부호 없는 정수를 곱할 때 사용하고 `IMUL` 연산은 부호 있는 정수를 곱할 때 사용한다. 왜 두 개의 다른 명령들이 필요하냐면, 곱셈을 하는 방법이 부호 없는 정수와 2의 보수의 부호 있는 정수들과 다르기 때문이다. 어떻게 다르느냐면, 바이트 FF 를 제곱해 워드로 결과를 얻어보자. 일단, 부호 없는 경우 255 곱하기 255 로 그 값은 65025 (혹은 16 진수로 FE01) 이 된다. 만약 부호 있는 곱셈을 하였다면 -1 곱하기 -1 가 되어 1 (혹은 16 진수로 0001) 이 된다.

곱셈 명령에는 여러 형태가 있다. 가장 오래된 형태는 아래와 같다.

`mul source`

`source` 는 레지스터나 메모리의 한 값이 될 수 있다. 이 값은 즉시값(immediate value) 이 될 수 없다. 피연산자(source)의 크기에 따라 곱셈 연산이 어떻게 실행될지는 달라진다. 만약 피연산자가 바이트 크기라면 이는 AL 레지스터의 바이트와 곱해져 그 값은 16 비트 AX 에 저장된다. 만약 16 비트라면 이는 AX 의 워드와 곱해져 32 비트 결과값으로 DX:AX 에 저장된다.

destination	source1	source2	작업
	reg/mem8		$AX = AL * source1$
	reg/mem16		$DX:AX = AX * source1$
	reg/mem32		$EDX:EAX = EAX * source1$
reg16	reg/mem16		$dest *= source1$
reg32	reg/mem32		$dest *= source1$
reg16	immed8		$dest *= immed8$
reg32	immed8		$dest *= immed8$
reg16	immed16		$dest *= immed16$
reg32	immed32		$dest *= immed32$
reg16	reg/mem16	immed8	$dest = source1 * source2$
reg32	reg/mem32	immed8	$dest = source1 * source2$
reg16	reg/mem16	immed16	$dest = source1 * source2$
reg32	reg/mem32	immed32	$dest = source1 * source2$

Table 2.2: `imul` 명령들

만약 소스가 32 비트라면 그 값은 EAX에 곱해져서 64 비트 결과값으로 EDX:EAX에 저장된다.

`IMUL` 명령은 `MUL` 명령과 형식이 같지만 몇 개의 다른 명령 형식들을 추가하였다. 2 또한 3 개의 피연산자를 가질 수 있는데:

```
imul dest, source1
imul dest, source1, source2
```

표 2.2 이 가능한 모든 조합을 보여준다.

나눗셈 명령으로는 `DIV` 와 `IDIV`로 2 개가 있다. 이 들은 부호 없는/있는 정수들의 나눗셈을 각각 수행한다. 통상적인 형식은:

```
div source
```

만약 소스가 8비트라면 AX는 피연산자로 나뉘어 진다. 그 뒷은 AL에 저장되고 나머지는 AH에 저장된다. 만약 소스가 16비트라면 DX:AX는 소스에 의해 나누어 지고 뒷은 AX에, 나머지는 DX에 저장된다. 만약 소스가 32비트라면 EDX:EAX는 소스로 나누어 지고 뒷이 EAX에, 나머지가 EDX에 저장된다. `IDIV` 명령도 같은 방식으로 작동한다. `IDIV`의 경우 `IMUL`과는 달리 특별한 명령들이 추가된 것은 없다. 만약 뒷이 레지스터에 저장되기에 너무 크거나, 제수가 0이라면 프로그램은 인터럽트 되고 종료된다. 가장 흔한 애러는 DX와 EDX를 나눗셈 하기 전에 초기화 하지 않는 것이다.

`NEG` 명령은 한 개의 피연산자를 가지며 피연산자의 2의 보수를 계산함으로써 부호를 바꾸어 버린다. 이 때 피연산자는 8비트, 16비트, 32비트 레지스터거나 메모리 값이여야 한다.

### 2.1.4 예제 프로그램

---

math.asm

```

1 %include "asm_io.inc"
2 segment .data          ; 출력되는 문자열
3 prompt      db      "Enter a number: ", 0
4 square_msg   db      "Square of input is ", 0
5 cube_msg     db      "Cube of input is ", 0
6 cube25_msg   db      "Cube of input times 25 is ", 0
7 quot_msg     db      "Quotient of cube/100 is ", 0
8 rem_msg      db      "Remainder of cube/100 is ", 0
9 neg_msg      db      "The negation of the remainder is ", 0
10
11 segment .bss
12 input    resd 1
13
14 segment .text
15     global _asm_main
16 _asm_main:
17         enter 0,0           ; 세입 루틴
18         pusha
19
20         mov    eax, prompt
21         call   print_string
22
23         call   read_int
24         mov    [input], eax
25
26         imul  eax             ; edx:eax = eax * eax
27         mov    ebx, eax         ; 그 결과를 ebx에 저장
28         mov    eax, square_msg
29         call   print_string
30         mov    eax, ebx
31         call   print_int
32         call   print_nl
33
34         mov    ebx, eax
35         imul  ebx, [input]     ; ebx *= [input]
36         mov    eax, cube_msg
37         call   print_string
38         mov    eax, ebx
39         call   print_int
40         call   print_nl

```

---

```

41
42     imul    ecx, ebx, 25      ; ecx = ebx*25
43     mov     eax, cube25_msg
44     call    print_string
45     mov     eax, ecx
46     call    print_int
47     call    print_nl

48
49     mov     eax, ebx
50     cdq
51     mov     ecx, 100          ; 부호 확장을 위해 edx 를 초기화
52     idiv   ecx              ; 즉 시값으로 나눌 수 없다.
53     mov     ecx, eax          ; edx:eax / ecx
54     mov     eax, quot_msg    ; 몫을 ecx 에 저장
55     call    print_string
56     mov     eax, ecx
57     call    print_int
58     call    print_nl
59     mov     eax, rem_msg
60     call    print_string
61     mov     eax, edx
62     call    print_int
63     call    print_nl

64
65     neg     edx              ; 나머지에 음을 취한다
66     mov     eax, neg_msg
67     call    print_string
68     mov     eax, edx
69     call    print_int
70     call    print_nl

71
72     popa
73     mov     eax, 0            ; C 를 리턴
74     leave
75     ret

```

---

### 2.1.5 확장 정밀도 산술 연산

어셈블리어는 또한 더블워드 보다 큰 수들을 위한 덧/뺄셈 명령을 지원한다. 이 명령들을 캐리 플래그를 이용한다. 위에서 설명했던 대로 ADD 와 SUB 명령은 모두 만약 받아 올림/내림이 발생했을 경우 캐리 플래그의

값을 변경한다. 이 캐리 플래그에 저장된 정보를 통해 큰 수를 작은 더블워드 조각(혹은 더 작게)들로 조개에서 연산을 수행해 덧/뺄셈을 수행할 수 있게 한다.

ADC 와 SBB 명령은 이 캐리 플래그의 저장된 정보를 이용한다. ADC 명령은 아래의 작업을 수행한다 :

$$\text{operand1} = \text{operand1} + \text{carry flag} + \text{operand2}$$

SBB 명령은 아래의 작업을 수행한다:

$$\text{operand1} = \text{operand1} - \text{carry flag} - \text{operand2}$$

이들이 어떻게 사용될까? EDX:EAX 와 EBX:ECX 레지스터에 저장된 64 비트 정수들의 합을 계산한다고 하자. 아래의 코드는 그 합을 EDX:EAX 레지스터에 넣을 것이다.

```

1 add    eax, ecx      ; 하위 32 비트를 더한다
2 adc    edx, ebx      ; 상위 32 비트와 이전의 덧셈에서의 캐리를 더한다.

```

뺄셈도 매우 비슷하다. 아래의 코드는 EBX:ECX 를 EDX:EAX 에서 뺀다.

```

1 sub    eax, ecx      ; 하위 32 비트를 뺀다.
2 sbb    edx, ebx      ; 상위 32 비트를 빼고 받아 내림을 뺀다.

```

정말로 큰 수들의 경우, 루프를 사용할 수 있다. ( 2.2 참조) 덧셈 루프에서, 되풀이 되는 부분마다 ADC 명령을 사용한다면 편리할 것이다. 그 대신 첫 번째 루프에는 ADD 를 이용해야 되는데 이는 CLC (캐리 클리어, Clear Carry) 명령을 통해 통해 루프 시작하기 직전에 캐리 플래그를 0 으로 만들어버림으로써 시행될 수 있다. 만약 캐리 플래그가 0 이라면 ADD 와 ADC 명령 사이에는 아무 런 차이가 없게 된다. 같은 방식으로 뺄셈도 할 수 있다.

## 2.2 제어 구조

고급 언어들은 실행의 흐름을 제어하기 위해 높은 수준의 제어 구조 (e.g. if 문이나 while 문) 를 지원한다. 어셈블리는 그와 같은 높은 수준의 제어 구조를 제공하지 않는다. 그 대신에 악명높은 goto 와 잘못 사용시 난잡한 코드를 만들 수 있는 것들을 지원한다. 그러나 어셈블리를 통해서도 잘 짜인 프로그램을 만드는 것은 가능 하다. 이를 위해 먼저 친숙한 고급 언어를 이용하여 프로그램의 구조를 디자인 한 뒤에 동일한 어셈블리로 바꾸어 주면 된다. (이는 컴파일러가 하는 일과 비슷하다)

### 2.2.1 비교

제어 구조는 데이터의 비교를 통해 무엇을 할지 결정한다. 어셈블리어에서 그 비교의 결과를 나중에 사용하기 위해 플래그 레지스터에 저장된다. 80x86은 비교를 위해 CMP 명령을 지원한다. 플래그 레지스터는 CMP 연산의 두 개의 피연산자의 차이에 의해 설정된다. 피연산자가 서로 뺄셈을 수행하고 그 결과에 따라 플래그 레지스터가 설정 되는데 그 결과는 플래그 레지스터의 정해진 부분에 저장된다. 만약 뺄셈의 결과를 얻고 싶다면 CMP 명령 대신 SUB 을 수행해야 한다.

부호 없는 정수들의 경우 2 개의 플래그(플래그 레지스터의 개개의 비트)가 중요하다: 제로(ZF) 와 캐리(CF) 플래그 이다. 제로 플래그는 두 피연산자의 차이가 0 일 때 세트<sup>3</sup> 된다. 캐리 플래그는 뺄셈에서의 받아내림 플래그와 같이 사용된다. 아래와 같은 비교문을 고려하자.

```
cmp    vleft, vright
```

vleft - vright 의 차이가 계산되어 이에 따라 플래그들이 세팅된다. 만약 CMP 연산 결과가 0 이라면 vleft = vright, ZF 플래그가 세트(*i.e.* 1) 되고 CF 는 언세트(받아내림 없음) 된다. 만일 vleft > vright 이라면, ZF 는 언세트 되고 CF 도 언세트(받아내림 없음) 된다. 만약 vleft < vright 인 경우 ZF 는 언세트되고, CF 는 세트(받아내림 있음) 된다.

부호 있는 정수들의 경우, 3 개의 플래그들이 중요한 역할을 한다: 제로 (ZF) 플래그, 오버플로우(OF) 플래그, 부호 (SF) 플래그 이다. 오버플로우 플래그는 연산의 결과에 오버플로우(또는 언더플로우(underflow) 발생 시) 세트 된다. 부호 플래그는 연산의 결과가 음수이면 세트 된다. 만약 vleft = vright 라면 ZF 가 세트 된다. (부호 없는 정수들의 경우와 마찬가지) 만약 vleft > vright 라면 ZF 가 언세트 되고 SF = OF 가 된다. 만약 vleft < vright 라면 ZF 는 언세트 되고 SF ≠ OF 이다.

CMP 만이 아닌 다른 연산들 역시 플래그 레지스터의 값을 바꿀 수 있다는 사실을 명심하라.

왜 vleft > vright 이면 SF = OF 일까? 만약 오버플로우가 없다면 그 차이는 양수이고 정확한 값을 가질 것이다. 따라서 SF = OF = 0 이다. 그러나 오버플로우가 있다면 그 차이는 정확한 값을 아닐 것이다. (그리고 사실 음수이다.) 따라서, SF = OF = 1 이 된다.

### 2.2.2 분기 명령

분기 명령(Branch instruction)은 프로그램의 임의의 부분으로 실행의 흐름을 옮길 수 있다. 다시 말해, 이들은 *goto* 문과 같다. 2 가지 종류의 분기 명령들이 있는데 : 무조건 분기 명령과 조건 분기 명령이다. 무조건 분기 명령은 *goto* 와 같이 반드시 분기를 한다. 조건 분기 명령은 플래그 레지스터의 플래그의 상태에 따라서 분기를 할지 안할지 결정한다. 만약 조건 분기 명령에서 분기가 일어나지 않으면 그 다음 명령이 실행되게 된다.

JMP (*jump* 의 준말) 명령은 무조건 분기 명령을 실행한다. 이는 하나의 인자를 가지며 보통 분기할 코드의 라벨(code label)을 가진다. 어셈블리,

---

<sup>3</sup>역자 주) 여기서 세트 된다는 의미는 1로 만든다는 것이다. 뒤에 나타날 언세트 된다는 의미는 0으로 만든다는 것이다

JZ	ZF 가 세트 될 때 예만 분기
JNZ	ZF 가 언세트 될 때 예만 분기
JO	OF 가 세트 될 때 예만 분기
JNO	OF 가 언세트 될 때 예만 분기
JS	SF 가 세트 될 때 예만 분기
JNS	SF 가 언세트 될 때 예만 분기
JC	CF 가 세트 될 때 예만 분기
JNC	CF 가 언세트 될 때 예만 분기
JP	PF 가 세트 될 때 예만 분기
JNP	PF 가 언세트 될 때 예만 분기

Table 2.3: 단순한 조건 분기들

혹은 링커는 이 라벨을 정확한 주소로 대체하게 된다. 이는 어셈블러가 프로그래밍을 편리하게 해주는 부분이기도 한다. 한 가지 명심할 점은 JMP 명령의 바로 다음 명령으로 분기하지 않는다면 그 명령은 수행되지 않는다는 것이다.

분기 명령에 몇 가지 형태들이 있다.

**SHORT** 이는 매우 한정된 범위에서 분기를 한다. 이는 메모리 상의 최대 128 바이트 까지 분기 할 수 있다. 이 분기에 장점은 다른 것들에 비해 메모리를 적게 차지한다는 점이다. 이는 점프 할 범위(*displacement*)를 저장하기 위해 1 부호 있는 바이트를 사용한다. 이 범위는 얼마나 많은 바이트를 분기할지 나타낸다. (EIP에 이 범위가 더해진다) short 분기를 명확히 지정하기 위해선 SHORT 키워드를 JMP 명령에서의 라벨 바로 앞에 써 주면 된다.

**NEAR** 이 분기는 무조건/조건 분기의 기본으로 지정된 형태로, 세그먼트의 어떠한 부분으로도 점프를 할 수 있다. 사실, 80386은 near 분기의 두 가지 형태를 지원한다. 첫 번째는 범위를 위해 2 바이트를 사용한다. 이는 대략 32,000 바이트 위, 아래로 분기할 수 있게 한다. 다른 형태는 범위를 위해 4 바이트를 사용하는데 이를 통해 코드 세그먼트의 어떠한 부분으로도 분기할 수 있게 해준다. 386 보호 모드에선 4 바이트 유형이 기본값으로 설정되어 있다. 만약 2 바이트 만 사용하는 것으로 하려면 JMP 명령에서의 라벨 바로 앞에 WORD를 써주면 된다.

**FAR** 이 분기는 다른 코드 세그먼트로의 분기를 허용한다. 이를 이용하는 것은 386 보호 모드에선 매우 드문 일이다.

올바른 코드 라벨들은 데이터 라벨들과 같은 규칙이 적용된다. 코드 라벨은 코드 세그먼트에서 그들이 가리키는(label) 문장 바로 앞에 놓임으로써 정의된다. 라벨 뒤에는 콜론(:)이 오게 되는데 이는 라벨 이름에 포함되지 않는다.

많은 수에 조건 분기 명령들이 있다. 이들은 코드 라벨 만을 인자로 갖는다. 단순한 것들은 단지 플래그 레지스터의 하나의 플래그 값만 보고 분기를 할지 안할지 결정 한다. 표 2.3 은 이러한 명령들을 정리했다. (PF 는 패리티 플래그(parity flag)로 결과값의 하위 8 비트의 수의 홀짝성을 나타낸다.)

다음 의사코드 (pseudo code, 알고리즘을 이해하기 쉽게 프로그래밍 문법으로 표현한 것) 는:

```
if ( EAX == 0 )
    EBX = 1;
else
    EBX = 2;
```

어셈블리로 아래와 같이 쓰일 수 있다.

```
1      cmp    eax, 0           ; 플래그를 세팅한다 (eax - 0 = 0 이면 ZF 가 세트)
2      jz     thenblock       ; ZF 가 세트되면 thenblock 분기한다.
3      mov    ebx, 2           ; if 문의 else 부분
4      jmp    next            ; if 문의 then 부분. 그냥 분기한다
5  thenblock:
6      mov    ebx, 1           ; if 문의 then 부분.
7  next:
```

다른 명령들의 경우 표 2.3 에 있는 조건 분기를 사용하기에는 너무 어렵다. 이를 설명하기 위해 아래 의사코드를 보자

```
if ( EAX >= 5 )
    EBX = 1;
else
    EBX = 2;
```

만약 EAX 가 5 이상이라면, ZF 는 세트 되거나 언세트 되고 SF 는 OF 와 같을 것이다. 아래는 이를 확인하는 어셈블리 코드이다. (이 때, EAX 가 부호 있는 정수라고 생각하자)

```
1      cmp    eax, 5
2      js    signon          ; SF 가 1 이면 signon 으로 분기
3      jo    elseblock        ; OF = 1, SF = 0 면 elseblock 으로 분기
4      jmp    thenblock       ; SF = 0, OF = 0 이면 thenblock 으로 분기
5  signon:
6      jo    thenblock        ; SF = 1, OF = 1 이면 thenblock 으로 분기
7  elseblock:
8      mov    ebx, 2
9      jmp    next
10 thenblock:
11     mov    ebx, 1
12 next:
```

부호 있음	부호 없음
JE $vleft = vright$ 이면 분기	JE $vleft = vright$ 이면 분기
JNE $vleft \neq vright$ 이면 분기	JNE $vleft \neq vright$ 이면 분기
JL, JNGE $vleft < vright$ 이면 분기	JB, JNAE $vleft < vright$ 이면 분기
JLE, JNG $vleft \leq vright$ 이면 분기	JBE, JNA $vleft \leq vright$ 이면 분기
JG, JNLE $vleft > vright$ 이면 분기	JA, JNBE $vleft > vright$ 이면 분기
JGE, JNL $vleft \geq vright$ 이면 분기	JAE, JNB $vleft \geq vright$ 이면 분기

Table 2.4: 부호 있는/없는 경우의 비교 명령들

위의 코드들은 매우 난해하다. 다행이도, 80x86 은 위와 같은 작업들을 훨씬 쉽게 할 수 있도록 여러 종류의 분기 명령들을 더 지원한다. 이들은 각각 부호 있는/없는 경우가 각각 나뉘어 있다. 표 2.4에 이 명령들이 잘 나타나 있다. 같다/다르다 분기 명령은 (JE, JNE) 는 부호가 있는/없는 정수에 모두 동일하다. (사실, JE 와 JNE 는 JZ 와 JNZ 와 각각 동일하다) 각각의 다른 분기 명령들은 2 개의 동의어를 갖고 있다. 예를 들어서 JL (미만이면 분기, jump less than) 과 JNGE(같거나 초과가 아니면 분기, jump not greater than or equal to) 이다. 이들은 모두 동일한 명령들인데 왜냐하면

$$x < y \implies \text{not}(x \geq y)$$

부호 없는 정수의 분기 명령의 경우 초과를 A(above), 미만을 B(below)로 대응 시켰고 부호 있는 정수의 분기 명령의 경우 G 와 L 에 대응시켰다.

위 새로운 분기 명령들을 이용하여 위의 의사 코드는 다음과 같이 좀 더 쉽게 어셈블리어로 바꿀 수 있다.

```

1      cmp    eax, 5
2      jge   thenblock
3      mov    ebx, 2
4      jmp   next
5 thenblock:
6      mov    ebx, 1
7 next:

```

### 2.2.3 루프 명령

80x86 은 *for* 과 같은 명령을 위해 몇 가지 명령들을 지원한다. 이 명령들은 한 개의 인자만 가지며 이는 코드 라벨이다.

**LOOP** ECX 의 값을 감소시키고, ECX  $\neq 0$  이면 라벨로 분기한다.

**LOOPE, LOOPZ** ECX 의 값을 감소시키고 (플래그 레지스터의 값은 바뀌지 않는다) 만일 ECX  $\neq 0$  이고 ZF = 1 이면 분기한다.

**LOOPNE, LOOPNZ** ECX 의 값을 감소시키고 (플래그 레지스터의 값은 바꾸지 않는다) 만일 ECX ≠ 0 이고 ZF = 0, 이면 분기한다.

아래 두 개의 루프들은 순차적인 검색 루프에 매우 효과적이다. 아래는 그 의사 코드이다.

```
sum = 0;
for( i=10; i >0; i-- )
    sum += i;
```

어셈블리로 다음과 같이 바뀐다:

```
1      mov     eax, 0          ; eax 는 합
2      mov     ecx, 10         ; ecx 는 i
3  loop_start:
4      add     eax, ecx
5      loop   loop_start
```

## 2.3 보통의 제어 구조를 번역하기

이 부분에서는 고급 언어의 제어 구조가 어셈블리 언어로 어떻게 바뀌는지 살펴 볼 것이다.

### 2.3.1 If 문

아래는 의사 코드는 :

```
if ( condition )
    then_block;
else
    else_block;
```

와 같이 바뀐다 :

```
1      ; code to set FLAGS
2      jxx    else_block    ; xx 를 적절히 선택하여 조건이 거짓이면 분기하게 한다.
3      ; then block 을 위한 부분
4      jmp    endif
5  else_block:
6      ; else block 을 위한 부분
7  endif:
```

만약 else 가 없다면 else\_block 분기는 endif 로의 분기로 바뀔 수 있다.

```
1      ; code to set FLAGS
2      jxx    endif        ; xx 를 적절히 선택하여 조건이 거짓이면 분기하게 한다.
3      ; then block 을 위한 부분
4  endif:
```

### 2.3.2 While 루프

*while* 루프는 조건을 상단에서 검사하는 루프이다 :

```
while( condition ) {
    body of loop;
}
```

이는 어셈블리어로 다음과 같이 바뀐다:

```
1 while:
2     ; code to set FLAGS based on condition
3     jxx endwhile      ; xx 를 적절히 선택해 거짓이면 분기하게 함
4     ; body of loop
5     jmp while
6 endwhile:
```

### 2.3.3 Do while 루프

*do while* 루프는 조건을 하단에서 검사한다 :

```
do {
    body of loop;
} while( condition );
```

이는 어셈블리어로 다음과 같이 바뀐다:

```
1 do:
2     ; body of loop
3     ; 조건에 따라 플래그 레지스터를 세트 한다.
4     jxx do          ; xx 를 적절히 선택해 거짓이면 분기하게 함
```

## 2.4 예제: 소수 찾는 프로그램

이 부분에서는 소수를 찾는 프로그램에 대해 살펴 볼 것이다. 소수는 1과 자기 자신만을 약수로 갖는 수임을 기억해라. 이 소수를 찾는 공식은 알려져 있지 않다. 이 프로그램이 사용하는 기본적인 방법은 특정한 값 이하의 모든 홀수들의 소인수를 찾는 것이다<sup>4</sup> 만약 어떤 홀수의 소인수가 없다면 그 홀수는 소수가 된다. 그럼 2.3 은 위 내용을 C 로 쓴 기본적인 알고리즘을 나타낸다.

여기 어셈블리어 버전이 있다:

---

<sup>4</sup>2 는 유일한 짝수 소수이다

```

1  unsigned guess; /*현재 소수라 추측되는 것 */
2  unsigned factor; /* guess 의 가능한 소인수 */
3  unsigned limit; /* 이 값까지 소수를 찾는다 */
4
5  printf ("Find primes up to: ");
6  scanf ("%u", &limit);
7  printf ("2\n"); /* 처음 두 소수를 특별한 경우로*/
8  printf ("3\n"); /* 취급한다 */
9  guess = 5; /*guess 의 초기값*/
10 while ( guess <= limit ) {
11     /* guess 의 소인수를 살펴본다*/
12     factor = 3;
13     while ( factor*factor < guess &&
14             guess % factor != 0 )
15         factor += 2;
16     if ( guess % factor != 0 )
17         printf ("%d\n", guess);
18     guess += 2; /*홀수만 살펴본다*/
19 }
```

Figure 2.3:

---

```

1  %include "asm_io.inc"           prime.asm
2  segment .data
3  Message      db      "Find primes up to: ", 0
4
5  segment .bss
6  Limit        resd    1          ; 최대 이 수 까지 소수를 찾는다.
7  Guess        resd    1          ; 현재 소수라 추측되고 있는 것
8
9  segment .text
10    global _asm_main
11  _asm_main:
12      enter 0,0          ; 셋업 루틴
13      pusha
14
15      mov eax, Message
16      call print_string
17      call read_int       ; scanf("%u", & limit );
18      mov [Limit], eax
```

---

```

20      mov    eax, 2          ; printf("2\n");
21      call   print_int
22      call   print_nl
23      mov    eax, 3          ; printf("3\n");
24      call   print_int
25      call   print_nl
26
27      mov    dword [Guess], 5 ; Guess = 5;
28 while_limit:
29      mov    eax,[Guess]
30      cmp    eax, [Limit]
31      jnbe  end_while_limit ; 숫자들이 부호 없는 정수 이므로 jnbe 이용
32
33      mov    ebx, 3          ; ebx 은 factor = 3;
34 while_factor:
35      mov    eax,ebx
36      mul    eax
37      jo    end_while_factor ; 위 계산서 오버플로우 발생시 점프.
38      cmp    eax, [Guess]
39      jnb   end_while_factor ; if !(factor*factor < guess)
40      mov    eax,[Guess]
41      mov    edx,0
42      div    ebx
43      cmp    edx, 0
44      je    end_while_factor ; if !(guess % factor != 0)
45
46      add    ebx,2
47      jmp    while_factor
48 end_while_factor:
49      je    end_if
50      mov    eax,[Guess]
51      call   print_int
52      call   print_nl
53 end_if:
54      add    dword [Guess], 2 ; guess += 2
55      jmp    while_limit
56 end_while_limit:
57
58      popa
59      mov    eax, 0          ; return back to C
60      leave
61      ret

```

---

prime.asm



# Chapter 3

## 비트 연산

### 3.1 쉬프트 연산

어셈블리 언어는 프로그래머가 데이터의 개개의 비트를 바꿀 수 있게 해준다. 이 중 가장 흔한 비트 연산은 바로 쉬프트(shift) 연산이다. 쉬프트 연산은 데이터의 비트의 위치를 옮겨준다. 쉬프트는 왼쪽(*i.e* 최상위비트 쪽으로)이나 오른쪽(최하위비트 쪽으로) 일어 날 수 있다.

#### 3.1.1 논리 쉬프트

논리 쉬프트는 쉬프트 중 가장 단순한 형태이다. 이는 매우 단순하게 쉬프트 연산을 한다. 그림 3.1 는 단일 바이트의 쉬프트 연산을 보여 준다.

원래 값	1   1   1   0   1   0   1   0
왼쪽 쉬프트	1   1   0   1   0   1   0   0
오른쪽 쉬프트	0   1   1   1   0   1   0   1

Figure 3.1: 논리 쉬프트

한가지 주목할 점은 새롭게 나오는 비트는 언제나 0 이라는 사실이다. SHL 과 SHR 명령은 각각 왼쪽, 오른쪽 쉬프트 연산을 수행한다. 이 명령들을 통해 몇 자리든지 쉬프트 할 수 있게 된다. 쉬프트 할 자리수는 상수이거나, CL 레지스터에 저장된 값이여야 한다. 데이터로 부터 쉬프트 된 마지막 비트는 캐리 플래그에 저장된다. 아래는 예제 코드이다.

```
1    mov    ax, 0C123H
2    shl    ax, 1           ; 1 비트 왼쪽으로 쉬프트,   ax = 8246H, CF = 1
3    shr    ax, 1           ; 1 비트 오른쪽으로 쉬프트,  ax = 4123H, CF = 0
4    shr    ax, 1           ; 1 비트 오른쪽으로 쉬프트,  ax = 2091H, CF = 1
5    mov    ax, 0C123H
6    shl    ax, 2           ; 2 비트 왼쪽으로 쉬프트,  ax = 048CH, CF = 1
```

```

7      mov    cl, 3
8      shr    ax, cl           ; 3 비트 오른쪽으로 쉬프트, ax = 0091H, CF = 1

```

### 3.1.2 쉬프트의 이용

쉬프트 연산은 빠른 곱셈과 나눗셈 연산에 가장 많이 사용된다. 십진 체계에서는 10의 몇수를 곱하거나 나누는 것이 매우 단순했다. 단지 숫자들만 쉬프트 해주면 되기 때문이다. 이는 이진수에서 2의 몇수의 경우도 마찬가지이다. 예를 들어서 이진수  $1011_2$  (십진수로 11)에 2를 곱하는 것은 단지 왼쪽으로 한 자리 쉬프트 하여  $10110_2$  (십진수로 22)를 얻는 것과 같다. 또한 2로 나눈 것의 몫도 또한 오른쪽으로 쉬프트 한 결과이다. 2로 나누기 위해선 한 자리를 오른쪽으로 쉬프트; 4( $2^2$ )로 나누기 위해선, 두 자리를 오른쪽으로 쉬프트; 8( $2^3$ )로 나누기 위해선 3자리를 오른쪽으로 쉬프트 하면 된다. 쉬프트 연산은 MUL나 DIV 명령에 비해 매우 빠르다.

사실, 부호 없는 정수들에서만 곱셈과 나눗셈을 위해 쉬프트 연산이 사용될 수 있다. 부호가 있는 정수들에게선 적용되지 않는다. 예를 들어서 2바이트 FFFF(-1)를 생각해 보면 오른쪽으로 한 번 쉬프트 되면 그 결과는 7FFF로 +32,767이 된다. 따라서 부호 있는 값에선 다른 형태의 쉬프트 연산이 사용되어야 한다.

### 3.1.3 산술 쉬프트

이 쉬프트들은 부호 있는 정수에 2의 몇수의 곱셈과 나눗셈을 빠르게 수행하게 위해서 만들어졌다. 이들은 부호 비트가 올바르게 설정되어 있다고 생각한다.

**SAL** 산술 왼쪽 쉬프트 - 이는 SHL과 동일하다. 이는 SHL과 정확히 같은 기계어 코드로 번역된다. 부호 비트가 쉬프트 연산에 의해 바뀌지 않는 이상, 결과값은 정확하다.

**SAR** 산술 오른쪽 쉬프트 - 이 새로운 명령은 피연산자의 부호 비트(*i.e.* 최상위비트)를 쉬프트 하지 않는다. 다른 비트들은 보통의 비트 연산들처럼 오른쪽으로 쉬프트 되며 왼쪽에서 추가되는 새 비트들은 이전의 부호 비트들과 값이 같다. (즉, 부호 비트가 1이였다면 새롭게 추가되는 비트들도 1이 된다) 따라서, 바이트가 이 연산에 따라 쉬프트 된다면 하위 7비트들만 쉬프트 되는 것이다. 다른 쉬프트 연산들처럼 마지막으로 쉬프트 되는 비트는 캐리 플래그에 저장된다.

```

1      mov    ax, 0C123H
2      sal    ax, 1           ; ax = 8246H, CF = 1
3      sal    ax, 1           ; ax = 048CH, CF = 1
4      sar    ax, 2           ; ax = 0123H, CF = 0

```

### 3.1.4 회전 쉬프트

회전 쉬프트는 논리 쉬프트처럼 작동하지만 다른 점은 쉬프트를 통해 사라진 끄트머리의 데이터가 새롭게 추가되는 데이터와 같다는 점이다. 따라서, 데이터는 하나의 순환 고리로 볼 수 있다. 두 개의 가장 단순한 회전 쉬프트 명령으로는 ROL 과 ROR 이 있으며 이는 각각 왼쪽과 오른쪽 회전을 가리킨다. 다른 쉬프트 연산들처럼 이 명령은 마지막으로 쉬프트된 비트를 캐리 플래그에 저장한다.

```

1    mov    ax, 0C123H
2    rol    ax, 1          ; ax = 8247H, CF = 1
3    rol    ax, 1          ; ax = 048FH, CF = 1
4    rol    ax, 1          ; ax = 091EH, CF = 0
5    ror    ax, 2          ; ax = 8247H, CF = 1
6    ror    ax, 1          ; ax = C123H, CF = 1

```

데이터의 비트들과 함께 캐리 플래그의 값을 회전시키는 명령들도 있는데, 이는 RCL 과 RCR 이다. 예를 들어서 AX 레지스터가 이 명령을 통해 회전된다면 AX 와 캐리 플래그와 함께 총 17 비트가 회전되게 된다.

```

1    mov    ax, 0C123H
2    clc                ; clear the carry flag (CF = 0)
3    rcl    ax, 1          ; ax = 8246H, CF = 1
4    rcl    ax, 1          ; ax = 048DH, CF = 1
5    rcl    ax, 1          ; ax = 091BH, CF = 0
6    rcr    ax, 2          ; ax = 8246H, CF = 1
7    rcr    ax, 1          ; ax = C123H, CF = 0

```

### 3.1.5 간단한 프로그램

아래는 EAX 레지스터에 얼마나 많은 수의 비트들이 켜져 있는지 (*i.e.* 1인지) 세는 소스이다.

---

```

1    mov    bl, 0          ; bl에 켜진 비트들의 수를 보관
2    mov    ecx, 32         ; ecx는 루프 카운터
3    count_loop:
4        shl    eax, 1      ; 쉬프트 된 비트는 캐리 플래그에 들어감.
5        jnc    skip_inc    ; 만일 CF == 0 면 skip_inc로 분기
6        inc    bl
7    skip_inc:
8        loop   count_loop

```

---

위의 코드는 원래 EAX 에 저장되어 있던 값을 파괴시킨다. (EAX 는 루프 끝에선 0 이 되어 버린다) 만약 EAX 의 값을 저장하고 싶으면 4 행을 rol eax, 1 로 바꾸어 주면 된다.

$X$	$Y$	$X \text{ AND } Y$
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.1: AND 연산

$$\begin{array}{ccccccccc}
 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
 \text{AND} & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
 \hline
 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0
 \end{array}$$

Figure 3.2: 바이트에 AND 연산 취하기

## 3.2 불리언 비트 연산

4 개의 중요한 불리언(Boolean) 연산자가 있는데 :  $AND$ ,  $OR$ ,  $XOR$ ,  $NOT$  이다. 진리표에는 이들의 가능한 모든 피연산자들에 대한 연산 수행 시 나타나는 결과값을 정리해놓았다.

### 3.2.1 AND 연산

오직 두 비트의 값이 1 일 때 예만  $AND$  연산의 결과값이 1 이 된다. 나머지 경우에는 모두 표 3.1 의 진리표에 잘 나와있듯이 0 이 된다.

프로세서는 이러한 연산들을 데이터의 각 비트에 대해 일일히 적용 시킨다. 예를 들어서 AL 과 BL 의 값에  $AND$  연산을 취했을 때, 기본적인  $AND$  연산은 두 개의 레지스터의 각각 대응되는 8 쌍의 비트들에 대해 각각 적용된다. 이는 그림 3.2 에 잘 나타나 있다. 아래는 예제 코드이다.

```

1      mov     ax, 0C123H
2      and     ax, 82F6H           ; ax = 8022H

```

### 3.2.2 OR 연산

두 비트가 모두 0 일 때 예만 포함적(inclusive)  $OR$  연산<sup>1</sup> 의 결과가 0 이 된다. 나머지 경우는 모두 결과가 1 이 되며 이는 진리표 3.2 에 잘 나타나 있다. 아래는 예제 코드이다 :

```

1      mov     ax, 0C123H
2      or      ax, 0E831H         ; ax = E933H

```

$X$	$Y$	$X \text{ OR } Y$
0	0	0
0	1	1
1	0	1
1	1	1

Table 3.2: OR 연산

$X$	$Y$	$X \text{ XOR } Y$
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.3: XOR 연산

### 3.2.3 $XOR$ 연산

두 비트가 같을 때 예만 배타적  $OR$ 의 연산 결과가 0이 된다. 두 비트가 다르면 연산의 결과는 1이 되며 이는 진리표 3.3에 잘 나타나 있다. 아래는 예제 코드이다:

```

1    mov     ax, 0C123H
2    xor     ax, 0E831H           ; ax = 2912H

```

### 3.2.4 $NOT$ 연산

$NOT$  연산은 단항(unary) 연산이다. (*i.e.* 이는 오직 한 개의 피연산자에 대해 연산한다.  $AND$  연산 같이 두 개의 피연산자를 가지는 연산이 아니다) 어떠한 비트의  $NOT$  연산을 하게 되면 그 비트의 정반대 되는 값이 되며 이는 진리표 3.4에 잘 나타나 있다. 아래는 예제 코드이다:

```

1    mov     ax, 0C123H
2    not     ax                 ; ax = 3EDCH

```

$NOT$  연산이 1의 보수를 찾는다는 것을 주목해라. 다른 비트 연산(Bitwise operation)들과는 달리  $NOT$  명령은 FLAGS 레지스터의 어떠한 값도 바꾸지 않는다.

### 3.2.5 TEST 명령

$TEST$  명령은  $AND$  연산을 수행하지만 결과는 보관하지 않는다. 이는 오직 연산 결과의 따라 FLAGS 레지스터의 값만을 바꿀 뿐이다. (*i.e.* CMP

---

<sup>1</sup>(역자 주) 이는 우리가 보통 OR 연산이라 부르는 것이다

$X$	$\text{NOT } X$
0	1
1	0

Table 3.4: NOT 연산

$i$ 번째 비트를 켠다	$2^i$ 와 $OR$ 연산을 한다. (이는 $i$ 번째 비트만 켜진(1인) 이진수이다.)
$i$ 번째 비트를 끈다	오직 $i$ 번째 비트만 꺼진(0인) 이진수와 $AND$ 연산을 한다. 이러한 피연산자를 마스크(mask)라고 부른다.
$i$ 번째 비트에 보수 취하기	$2^i$ 와 $XOR$ 연산을 한다.

Table 3.5: 불리언 연산의 사용

명령이 빨리 수행하지만 오직 FLAGS 레지스터의 값만을 바꾸었던 것과 일맥상통하다) 예를 들어, 연산의 결과가 0이라면 ZF가 세트된다.

### 3.2.6 비트 연산의 사용

비트 연산은 개개의 값을 다른 비트에는 전혀 영향을 주지 않고도 수정하는 경우에 매우 편리하다. 표 3.5에는 이러한 연산들의 3 가지 자주 쓰이는 경우를 보여주고 있다. 아래는 위 아이디어를 적용한 예제 코드이다.

```

1    mov    ax, 0C123H
2    or     ax, 8          ; 3 번째 비트를 켠다,    ax = C12BH
3    and    ax, OFFDFH   ; 5 번째 비트를 끈다,    ax = C10BH
4    xor    ax, 8000H    ; 31 비트를 반전,    ax = 410BH
5    or     ax, 0F00H    ; 니블을 켠다,    ax = 4F0BH
6    and    ax, OFFFOH   ; 니블을 끈다,    ax = 4F00H
7    xor    ax, 0F00FH   ; 니블을 반전,    ax = BF0FH
8    xor    ax, OFFFFH   ; 1의 보수,    ax = 40F0H

```

$AND$  연산은 2의 몇수로 나눈 나머지를 찾는데 사용할 수 있다.  $2^i$ 로 나눈 수의 나머지를 찾기 위해선 값이  $2^i - 1$ 인 마스크와  $AND$  연산을 하면 된다. 이 마스크는  $i - 1$  번째 비트 까지 모두 1이다. 이 마스크에 나머지가 저장된다.  $AND$  연산의 결과는 이러한 비트들을 저장하고 나머지는 모두 0이 되게 한다. 아래는 100을 16으로 나눈 나머지와 몫을 저장하는 코드이다.

```

1    mov    eax, 100        ; 100 = 64H
2    mov    ebx, 0000000FH  ; mask = 16 - 1 = 15 or F
3    and    ebx, eax       ; ebx = remainder = 4

```

---

```

1    mov    bl, 0          ; bl 은 값이 1 인 비트의 수를 갖는다.
2    mov    ecx, 32         ; ecx 는 투프 카운터이다.
3 count_loop:
4    shl    eax, 1          ; 캐리 플래그에 비트를 쉬프트
5    adc    bl, 0          ; bl 에 캐리 플래그를 더한다.
6    loop   count_loop

```

---

Figure 3.3: ADC 를 이용해 비트세기

CL 레지스터를 이용하여 데이터의 임의의 비트를 수정하는 것이 가능하다. 아래의 코드는 EAX 의 임의의 비트를 세트 (1 로 만듦) 하는 코드이다. BH 에 세트가 될 비트의 위치가 저장되어 있다.

```

1    mov    cl, bh          ; OR 연산을 할 수를 만듦
2    mov    ebx, 1
3    shl    ebx, cl          ; CL 번 쉬프트
4    or     eax, ebx          ; 비트를 세트

```

비트를 오프(off, 0 으로 만듦) 하는 일은 조금 더 복잡하다.

```

1    mov    cl, bh          ; AND 연산을 할 수를 만듦
2    mov    ebx, 1
3    shl    ebx, cl          ; cl 번 쉬프트
4    not   ebx              ; 비트 반전
5    and   eax, ebx          ; 비트를 끔

```

임의의 비트를 반전시키는 코드는 여러분의 몫으로 남기겠다.

80x86 프로그램에서 아래와 같은 난해한 명령을 보는 일이 종종 있다:

```
xor    eax, eax          ; eax = 0
```

어떠한 수가 자기 자신과 XOR 연산을 하면 0 이 된다. 이 명령은 동일한 MOV 명령 보다 기계어 코드 크기가 작기 때문에 자주 사용된다.

### 3.3 조건 분기 명령을 피하기

현대의 프로세서들은 코드를 빠르게 실행하기 위해 여러가지 복잡한 방법들을 사용한다. 가장 흔한 방법으로는 추론적 실행(*speculative execution*)이다. 이 기술은 CPU 의 병렬 처리 기술을 이용하여 여러개의 명령들을 동시에 실행한다. 그런데 조건 분기 명령에는 이 방법을 적용할 수 없다. 왜냐하면 대다수의 경우 프로세서는 분기를 할지 안할지 알 수 없기 때문이다. 만약 분기를 하게 되면 분기를 안할 때와 다른 명령들을 실행하게

된다. 프로세서는 분기를 할지 안할지 예측을 하려고 노력한다. 만약 예측이 틀리다면 프로세서는 쓸데 없는 코드를 실행하는데 시간을 빼앗기게 된다.

위 문제를 피하는 방법으로는 조건 분기 명령을 되도록 사용하지 않는 것이다. 그림 3.1.5의 예제 코드는 어떻게 조건 분기 명령 사용 회수를 줄이는지 잘 나타내고 있다. 이전의 예제에서는 EAX 레지스터의 켜진 비트들의 수를 세었다. 이는 INC 명령을 중단하기 위해 조건 분기 명령을 사용했다. 그림 3.3은 ADC 명령을 사용해 캐리 플래그에 직접적으로 더해 분기를 사용하지 않는 방법을 보여주고 있다.

SET $xx$  명령은 특정한 경우에서 분기를 이용하지 않을 수 있는 방법을 제공한다. 이 명령들은 플래그 레지스터의 상태에 따라 바이트 레지스터나 메모리의 값을 0 또는 1로 바꾸어 준다. SET 다음에 오는 문자들은 조건 분기 명령에서의 것들과 같다. 만약 SET $xx$ 의 조건이 참이라면 그 결과는 1, 거짓이면 0이 저장된다. 예를 들어

```
setz    al      ; Z 플래그가 세트되었으면 1, 아니면 0이 al에 저장
```

이 명령들을 이용하여 여러분은 분기를 사용하지 않고도 값을 계산하는 놀라운 방법들을 만들 수 있을 것이다.

예를 들어서 두 값을 중 최대값을 찾는 프로그램을 만든다고 하자. 이 문제를 해결하는 간단한 방법으로는 CMP와 조건 분기 명령을 사용해 어떤 한 값이 더 큰지 결정하는 것이다. 하지만 아래의 코드는 어떻게 조건 분기 명령을 사용하지 않고도 최대값을 찾을 수 있는지 보여준다.

---

```

1 ; file: max.asm
2 %include "asm_io.inc"
3 segment .data
4
5 message1 db "Enter a number: ",0
6 message2 db "Enter another number: ", 0
7 message3 db "The larger number is: ", 0
8
9 segment .bss
10
11 input1 resd    1      ; 첫 번째 숫자 입력
12 segment .text
13     global _asm_main
14 _asm_main:
15     enter 0,0          ; 셋업 루틴
16     pusha
17
18     mov eax, message1 ; 첫 번째 메시지 출력

```

```

19      call    print_string
20      call    read_int      ; 첫 번째 수 입력
21      mov     [input1], eax
22
23      mov     eax, message2 ; 두 번째 메세지 출력
24      call    print_string
25      call    read_int      ; eax에 두 번째 수 입력
26
27      xor     ebx, ebx      ; ebx = 0
28      cmp     eax, [input1] ; 두 번째와 첫 번째 숫자를 비교
29      setg   bl             ; ebx = (input2 > input1) ? 1 : 0
30      neg     ebx           ; ebx = (input2 > input1) ? 0xFFFFFFFF : 0
31      mov     ecx, ebx       ; ecx = (input2 > input1) ? 0xFFFFFFFF : 0
32      and     ecx, eax       ; ecx = (input2 > input1) ? input2 : 0
33      not     ebx           ; ebx = (input2 > input1) ? 0 : 0xFFFFFFFF
34      and     ebx, [input1] ; ebx = (input2 > input1) ? 0 : input1
35      or      ecx, ebx       ; ecx = (input2 > input1) ? input2 : input1
36
37      mov     eax, message3 ; 결과를 출력
38      call    print_string
39      mov     eax, ecx
40      call    print_int
41      call    print_nl
42
43      popa
44      mov     eax, 0          ; C로 리턴
45      leave
46      ret

```

---

위 코드에서 최대값을 올바르게 선택하기 위해 비트 마스크를 이용할 수 있다는 사실을 이용했다. 30 행에서 사용된 SETG 명령은 두 번째 입력값이 최대값이면 1, 아니면 0으로 BL 레지스터를 설정한다. 이는 우리가 원하던 비트 마스크가 아니다. 따라서 올바른 비트 마스크를 만들기 위해 31 행에서 EBX 레지스터 전체에 NEG 명령을 사용하였다. (이전의 EBX 레지스터의 값은 0이 되었음을 주목해라) 만약 EBX 가 0이라면 아무것도 하지 않는다. 그러나 EBX 가 1이라면 그 결과는 -1의 2의 보수, 즉 0xFFFFFFFF 가 된다. 이는 우리가 필요했던 비트 마스크이다. 코드의 나머지 부분은 비트 마스크를 이용하여 입력값의 정확한 최대값을 계산하게 된다.

다른 방법으로는 DEC 문장을 이용하는 것이다. 위 코드에선 NEG 를 DEC 로 바꾸어도 그 결과는 0이나 0xFFFFFFFF 가 된다. 하지만 NEG 명령을 사용할 때와 달리 경우가 뒤바뀐다.

## 3.4 C에서 비트값 바꾸기

### 3.4.1 C에서의 비트 연산자

일부 고급 언어들과는 달리 C는 비트 연산을 위한 연산자들을 지원한다. *AND* 연산은 이진 **&** 연산자<sup>2</sup>로 나타난다. *OR* 연산은 이진 **|** 연산자로 나타난다. *XOR* 연산은 이진 **^** 연산자로 나타난다. 그리고 *NOT* 연산은 단항 **~** 연산자로 나타난다.

C에서의 쉬프트 연산은 << 와 >> 이진 연산자가 수행한다. << 연산자는 왼쪽 쉬프트, >> 연산자는 오른쪽 쉬프트를 수행한다. 이들은 2개의 피연산자를 갖는다. 왼쪽의 피연산자는 쉬프트 할 값이고 오른쪽의 피연산자는 쉬프트 할 비트 수를 나타낸다. 만약 부호 없는 형식의 값을 쉬프트한다면 논리 쉬프트가 이루어진다. 만약 부호 있는 형식(예를 들면 int)의 쉬프트를 하면 산술 쉬프트가 이루어 진다. 아래는 이러한 연산자들을 이용한 예제 코드이다.

```

1 short int s;           /* short int 가 16비트라고 생각 */
2 short unsigned u;
3 s = -1;                /* s = 0xFFFF (2의 보수) */
4 u = 100;                /* u = 0x0064 */
5 u = u | 0x0100;         /* u = 0x0164 */
6 s = s & 0xFF00;         /* s = 0xFFFF0 */
7 s = s ^ u;              /* s = 0xFE94 */
8 u = u << 3;            /* u = 0xB20 (논리 쉬프트) */
9 s = s >> 2;            /* s = 0xFFA5 (산술 쉬프트) */

```

### 3.4.2 C에서 비트 연산자 사용하기

C에서 사용되는 비트 연산자들은 어셈블리에서 사용되는 경우와 동일하게 사용할 수 있다. 이를 통해 빠른 곱셈과 나눗셈을 수행하고 개개의 비트를 조작할 수 있다. 사실, 똑똑한 C 컴파일러들은 **x\*=2**와 같은 문장에선 쉬프트 연산을 자동적으로 사용한다.

많은 수의 운영체계 API<sup>3</sup>들은 (예를 들어서 *POSIX*<sup>4</sup>와 Win32가 대표적이다.) 들은 비트로 데이터가 인코드(encode)된 피연산자를 사용하는 함수를 사용한다. 예를 들어서 POSIX 시스템에서 파일 시스템 접근 권한에는 3 가지 유형의 유저들이 있다: 유저(*user*) (정확히 말하자면 오너(*owner*)), 그룹(*group*), 아더(*others*)이다. 각각의 유형의 사용자들은 파일을 읽기, 쓰기, 실행 수 있는 권한을 가질 수 있다. 파일의 권한을 변경하기 위해선 C 프로그래머가 각각의 비트들을 수정할 수 있어야만 한다. POSIX는 이를

<sup>2</sup>이 연산자는 **&&** 와 단항 **&** 연산자와는 다르다!

<sup>3</sup>응용 프로그램 프로그래밍 인터페이스(Application Programming Interface)

<sup>4</sup>이식 가능 운영체계 인터페이스 (Portable Operating System Interface for Computer Environment)의 준말로, UNIX를 기반으로 IEEE에 의해 개발됨

매크로	의미
S_IRUSR	유저가 읽기 가능
S_IWUSR	유저가 쓰기 가능
S_IXUSR	유저가 실행 가능
S_IRGRP	그룹이 읽기 가능
S_IWGRP	그룹이 쓰기 가능
S_IXGRP	그룹이 실행 가능
S_IROTH	아더(other) 읽기 가능
S_IWOTH	아더가 쓰기 가능
S_IXOTH	아더가 실행 가능

Table 3.6: POSIX 파일 실행 권한 명령들

돕기 위해 몇 가지의 매크로를 정의했다 (표 3.6 참조) chmod 함수는 파일의 권한을 설정하기 위해 사용된다. 이 함수는 두 개의 인자를 가지는데, 하나는 권한을 설정할 파일의 이름 문자열이고, 또 하나는 어떻게 권한을 설정할지에 대한 정수(Integer) 값<sup>5</sup>이다. 예를 들어서 아래의 코드는 파일의 권한을 오너는 파일을 읽고 쓰기 가능, 그룹은 읽기 가능, 그리고 아더는 어떠한 접근도 불가하게 설정하는 코드이다.

```
chmod("foo", S_IRUSR | S_IWUSR | S_IRGRP );
```

POSIX stat 함수는 현재 파일의 접근 권한 비트값을 알아내는데 사용할 수 있다. chmod 함수와 함께 사용하면 다른 값을 바꾸지 않고도 현재 파일의 권한들을 변경할 수 있다. 아래는 아더의 쓰기 권한을 지우고, 오너에 읽기 권한을 추가하는 명령이다. 다른 권한들은 변경되지 않았다.

```

1 struct stat file_stats; /* stat() 이 사용할 구조체 */
2 stat("foo", &file_stats); /* 파일 정보를 읽는다
3 file_stats.st_mode 에는 권한 비트가 들어간다. */
4 chmod("foo", (file_stats.st_mode & S_IWOTH) | S_IRUSR);
```

### 3.5 빅, 리틀 엔디안 표현

1장에서 단일 바이트 이상의 데이터에 대한 빅, 리틀 엔디안 표현에 대해 다루었었다. 그러나, 저자는 많은 사람들이 아직도 이 개념에 대해 잘 이해하지 못하므로 이 단원에서 좀 더 심화되게 다루고자 한다.

여러분은 엔디안이 멀티바이트 데이터의 원소들의 개개의 바이트(비트가 아님)를 메모리에 저장하는 순서를 가리키는 말이라는 사실을 기억할 것이다. 빅 엔디안은 직관적인 방법이다. 이는 최상위 바이트를 먼저 저장하고 그 순서로 쭉 저장한다. 다시 말해, 가장 높은 자리의 바이트들이 먼저

<sup>5</sup> 사실, 이 정수형은 int 형으로 typedef 에 의해 mode\_t 형으로 정의되어 있다

```

unsigned short word = 0x1234; /* sizeof(short) == 2 라 생각 */
unsigned char * p = (unsigned char *) &word;

if ( p[0] == 0x12 )
    printf ("Big Endian Machine\n");
else
    printf (" Little Endian Machine\n");

```

Figure 3.4: 무슨 엔디안인지 어떻게 결정할까?

저장된다. 리틀 엔디안은 바이트를 역순으로 (낮은 순서대로) 저장한다. x86 계열의 프로세서들은 리틀 엔디안 표현을 사용한다.

예를 들어서 더블워드  $12345678_{16}$  을 생각해 보자. 빅 엔디안 표현에선 바이트 들은 12 34 56 78 순으로 저장된다. 하지만 리틀 엔디안 표현에선 바이트 들이 78 56 34 12 순으로 저장된다.

아마 독자 여러분은 머리속에 다음과 같은 생각이 떠오를 것이다. 왜 칩 디자이너들이 리틀 엔디안 표현을 사용할까? 사악한 인텔 엔지니어들이 프로그래머들을 혼란스럽게 하기 위해 그런 것이였을까? 아마 여러분은 CPU 가 메모리에 이러한 방식으로 데이터를 저장하기 위해 부수적인 작업을 해야 한다고 생각한다. (또한 메모리에 서도 역순으로 읽어 들일 때도) 하지만, 그 답은 '아니오' 이다. 리틀 엔디안 표현을 사용할 때에도 CPU 는 어떠한 부수적인 작업을 하지 않는다. 여러분은 CPU 가 단지 비트값에만 작업하는 수 많은 전기 회로들로 구성되어 있다는 것일 명심해야 한다. 그리고 비트(또한 바이트)들의 순서는 CPU 에서 정렬되어 있을 필요가 없다.

2 바이트 AX 레지스터를 생각해 보자. 이는 두 개의 바이트 레지스터 AH 와 AL 로 나뉜다. CPU 에선 AH 와 AL 레지스터의 값들을 지정하는 회로들이 있다. CPU 에선 이 회로들이 특정한 순서로 놓여 있지 않다. 따라서, AH 를 위한 회로들은 AL 을 위한 회로들 앞에나 뒤에 놓여 있다고 말할 수 없다. AX 레지스터의 값을 메모리로 복사하는 mov 명령은 AL 의 값을 복사한 다음 AH 의 값을 복사 하지만 이 순서를 바꿔서 해도 CPU 에선 아무런 상관이 없다.

같은 논리가 바이트의 개개의 비트들에도 적용이 된다. CPU 의 회로에선 이 개개의 비트들이 특정한 순서로 놓여 있지는 않다. 그러나, 개개의 비트 들이 CPU 나 메모리에서 주소 지정이 될 수 없기 때문에 우리는 CPU 내부적으로 어떠한 순서로 이들이 놓여있는지 알 방법(알 필요도) 이 없다.

그럼 3.4 에선 CPU 가 무슨 엔디안 형식을 사용 하는지 알 수 있다. p 포인터는 word 를 두 개의 원소를 가지는 문자 배열로 취급한다. 따라서 p[0] 는 word 의 메모리에 저장된 값의 첫 번째 바이트가 된다. 이는 CPU 가 어떤 엔디안 형식을 사용하는지에 따라 달라지게 된다.

### 3.5.1 리틀 혹은 빅 엔디안인지 언제 신경 써야 하는가?

많은 경우 프로그래밍 시에 CPU 가 무슨 엔디안 형식을 사용하는지는

```

1  unsigned invert_endian( unsigned x )
2  {
3      unsigned invert;
4      const unsigned char * xp = (const unsigned char *) &x;
5      unsigned char * ip = (unsigned char *) & invert;
6
7      ip [0] = xp[3]; /* 개개의 바이트를 역으로 재배치 */
8      ip [1] = xp[2];
9      ip [2] = xp[1];
10     ip [3] = xp[0];
11
12     return invert; /*뒤집어진 바이트를 리턴*/
13 }
```

Figure 3.5: \_엔디안 변환 함수

큰 상관이 없다. 그러나 바로 두 개의 서로 다른 컴퓨터 시스템에서 이진 데이터가 송신될 때 이를 심각하게 고려해야 한다. 이렇게 송신되는 경우는 주로 특정한 형식의 물리 데이터 미디어(예를 들어 디스크)나 네트워크이다. 아스키 데이터는 단일 바이트 이므로 엔디안은 큰 문제가 아니다.

모든 TCP/IP 헤더들은 데이터를 빅 엔디안 형식(네트워크 바이트 순서 (*network byte order*) 라 부른다)으로 정수들을 저장한다. TCP/IP 라이브러리는 C로 하여금 엔디안 관련 문제들을 해결하기 위한 함수들을 지원한다. 예를 들어서 `htonl()` 함수는 더블워드(혹은 long integer)를 호스트(*host*)의 엔디안 형식에서 네트워크(*network*)의 엔디안 형식으로 변환한다. `ntohl()` 함수는 이와 정반대인 작업을 수행한다.<sup>6</sup> 빅 엔디안 시스템에서, 두 함수는 두 함수들은 입력값을 변화시키지 않고 리턴한다. 이를 통해 프로그래머들은 네트워크 프로그램들을 엔디안의 형식에 상관하지 않고 어떠한 시스템에서도 올바르게 동작할 수 있게 한다. 엔디안 형식과 네트워크 프로그래밍에 대한 자세한 정보는 W. Richard Steven의 놀라운 책인 *UNIX Network Programming*을 읽어 보면 된다.

그림 3.5은 더블워드의 엔디안 형식을 변환하는 C 함수를 보여주고 있다. 486 프로세서는 32비트 레지스터의 바이트를 역순으로 재배치하는 명령인 `BSWAP`를 추가하였다. 예를 들면

`bswap edx ; edx의 바이트를 역으로 재배치`

이 명령은 16비트 레지스터에서는 사용될 수 없다. 그러나 `XCHG` 명령은 두 개의 8비트 레지스터로 조개어 질 수 있는 16비트 레지스터의 바이트를

`UNICODE` 와 같은 멀티바이트 문자 세트의 출현에 따라 텍스트 데이터에서 조차도 무슨 엔디안을 사용하는지 중요하게 되었다. `UNICODE`는 두 엔디안 형식을 모두 지원하며 데이터를 표현하는 엔디안이 무슨 형식이니 알아내는 메카니즘이 있다

<sup>6</sup> 사실, 정수의 엔디안 형식을 변환하는 일은 단순히 바이트들을 역순으로 재배치 하는 것이다: 따라서, 빅에서 리틀 엔디안 형식으로 바꾸는 작업은 리틀에서 빅 엔디안 형식으로 바꾸는 것과 동일하다. 따라서 이 두 함수는 같은 작업을 한다.

```

1 int count_bits( unsigned int data )
2 {
3     int cnt = 0;
4
5     while( data != 0 ) {
6         data = data & (data - 1);
7         cnt++;
8     }
9     return cnt;
10 }
```

Figure 3.6: 비트 수 세기: 첫 번째 방법

역순으로 재배치 할 수 있다. 예를 들면

xchg ah,al ; ax 의 바이트를 역순으로 재배치

## 3.6 비트 수 세기

이전에 더블워드에서 직관적인 방법으로 켜진 비트들의 수를 세는 프로그램을 만들었던 적이 있었다. 이 섹션에서는 덜 직관적이지만 이 장에서 논의된 비트 연산을 이용하여 세는 방법에 대해 이야기 하고자 한다.

### 3.6.1 첫 번째 방법

첫 번째 방법은 매우 단순하지만 명료하다. 그림 3.6 이 그 코드를 낸다.

이 방법은, 루프의 매 단계에서 data 의 한 개의 비트는 꺼진다. 만일 데이터의 모든 비트들이 꺼진다면(*i.e.* data 가 0 일 때) 루프가 멈춘다. data 를 0 으로 만들기 위해 필요한 작업의 회수는 data 에 들어 있는 켜진 비트들의 수와 동일하게 된다.

6 행에서 data 의 한 개의 비트가 꺼진다. 왜 그럴까? 예를 들어 data 값에서 오른쪽에서 부터 최초로 1 이 나타나는 지점을 생각하자. 정의에 따라, 그 1 이 나타나기 이전 비트들은 모두 0 이 된다. 그렇다면 data - 1 의 이진 표현은 어떻게 될까? 아까 최초로 1 이 나타나는 부분 부터 마지막 자리까지는 모두 이전 데이터의 보수가 되고 나머지 부분은 그대로 일 것이다. 예를 들면

data = xxxxx10000  
data - 1 = xxxxx01111

```

1 static unsigned char byte_bit_count [256]; /* 룩업 테이블 */
2
3 void initialize_count_bits ()
4 {
5     int cnt, i, data;
6
7     for( i = 0; i < 256; i++ ) {
8         cnt = 0;
9         data = i;
10        while( data != 0 ) { /* 첫 번째 방법 */
11            data = data & (data - 1);
12            cnt++;
13        }
14        byte_bit_count [i] = cnt;
15    }
16}
17
18 int count_bits( unsigned int data )
19 {
20     const unsigned char * byte = ( unsigned char * ) & data;
21
22     return byte_bit_count [byte [0]] + byte_bit_count [byte [1]] +
23            byte_bit_count [byte [2]] + byte_bit_count [byte [3]];
24 }
```

Figure 3.7: 두 번째 방법

이 때,  $x$  들은 이전의 숫자와 동일하다. 만약  $data$  가  $data - 1$ 과 AND 연산 하였다면 나머지 비트들은 바뀌지 않지만 오른쪽에서 처음으로 나타났던 1 은 0 이 되어 있을 것이다.

### 3.6.2 두 번째 방법

임의의 더블워드에서의 켜진 비트 수를 세는 방법으로 룩업 테이블 (lookup table)<sup>7</sup> 을 이용하는 방법도 있다. 직관적으로 모든 더블워드들의 값마다 켜진 비트들의 수를 계산해 놓으면 될 것 같다. 그러나, 이 방법엔 2 가지 문제점이 있다. 첫 째로, 대략 40억 개의 가능한 더블워드의 값이 존재한다. 이 말은, 그 룩업 테이블의 크기는 매우 크고 이를 초기화 하는 작업도 매우 오래 걸릴 것이다. (사실상, 배열의 크기가 40억이 넘어간다면

<sup>7</sup>(역자 주)참고로 룩업 테이블이란 컴퓨터의 계산 속도를 향상 시키기 위하여 컴퓨터가 미리 계산 결과를 표로 정리해 놓은 것이다. 대표적으로 sine 값은 계산하기 위한 룩업 테이블이 있다.

그냥 켜진 비트의 수를 계산하는 것 보다도 이를 초기화 하는 것이 시간이 훨씬 오래 걸린다)

좀 더 현실적인 방법으로는 모든 바이트 값들의 켜진 비트 수를 계산하여 배열에 저장해 놓는 것이다. 그리고 더블워드를 4 개의 바이트를 쪼개어서 각각의 바이트를 배열에서 찾아 각 켜진 비트들의 합을 더하면 된다. 그럼 3.7 는 이 방법을 적용한 코드를 보여주고 있다.

`initialize_count_bits` 함수는 `count_bits` 함수를 호출하기 전에 반드시 호출되어야만 한다. 왜냐하면 이 함수는 전역 `byte_bit_count` 배열을 초기화 하기 때문이다. `count_bits` 함수는 `data` 를 크기가 더블워드인 변수로 보지 않고 4개의 바이트의 배열로 생각한다. `dword` 포인터는 이 4 바이트 배열의 포인터 역할을 한다. 따라서 `dword[0]` 은 `data` 의 한 개의 바이트(하드웨어가 빅 엔디안이냐, 리틀 엔디안이냐의 따라서 이 바이트가 최상위 바이트거나 최하위 바이트가 된다) 를 가리킨다. 당연하게도 여러분은 아래와 같은 방법을 사용하여

```
(data >> 24) & 0x000000FF
```

최상위 바이트 값을 찾을 수 있고, 다른 바이트 들에게도 비슷한 방법을 적용 할 수 있다. 그러나, 이러한 방법은 위 배열을 이용한 방법보다 속도가 느린다.

마지막으로 볼 것은 `for` 루프를 통해 22 와 23 행의 수 들의 합을 손쉽게 계산할 수 있다. 그러나 `for` 루프는 루프 상단에 인덱스의 초기화를 필요로 하고, 루프마다 조건을 검사하고, 인덱스를 증가 시킬 것이다. 하지만 위 처럼 그냥 4 개의 수를 직접 더하면 훨씬 빠르게 된다. 사실, 똑똑한 컴파일러는 `for` 루프를 이용한 형태를 위처럼 4 개의 합으로 변경해 줄 것이다. 이렇게 되풀이 되는 루프를 줄여주는 것은 루프 언롤링(*loop unrolling*)이라는 컴파일러 최적화 기술이다.

### 3.6.3 세 번째 방법

이는 데이터의 켜진 비트수를 세는 또다른 놀라운 방법이다. 이 방법은 말그대로 데이터의 1 들과 0 들을 더하게 된다. 이 합은 당연히 이 데이터의 1 의 개수와 동일하게 된다. 예를 들어 `data` 변수에 들어있는 1 바이트 데이터의 1 의 개수를 센다고 하자. 첫 번째로 알의 명령을 수행하게 된다.

```
data = (data & 0x55) + ((data >> 1) & 0x55);
```

위 작업은 무엇을 할까? 16진수 0x55 는 이진수로 01010101 로 나타낸다. 위 덧셈의 첫 번째 항에서, `data` 는 0x55 와 AND 연산을 하게 되어 홀수번째 비트들만 남게 된다. 두 번째 부분  $((data >> 1) \& 0x55)$  는 먼저 짹수번째 비트들을 모두 홀수 번째로 쉬프트 하고 다시 같은 마스크를 이용하여 홀수 번째 비트들만 남긴다. 따라서, 첫 번째 부분은 `data` 의 홀수 번째 비트들을, 두 번째 부분은 짹수 번째 비트들을 가지게 된다. 이 두 개의 부분이 더해진다면 `data` 의 홀수 번째 비트들과 짹수 번째 비트들이 서로

```

1 int count_bits(unsigned int x)
2 {
3     static unsigned int mask[] = { 0x55555555,
4                                     0x33333333,
5                                     0x0F0F0F0F,
6                                     0x00FF00FF,
7                                     0x0000FFFF };
8     int i;
9     int shift; /* 오른쪽으로 쉬프트 연산 할 횟수 */
10
11    for( i=0, shift=1; i < 5; i++, shift *= 2 )
12        x = (x & mask[i]) + ( (x >> shift) & mask[i] );
13    return x;
14 }
```

Figure 3.8: 세 번째 방법

더해진다. 예를 들어  $\text{data}$  가  $10110011_2$  였다면

$$\begin{array}{r} \text{data} \& 01010101_2 \\ + (\text{data} \gg 1) \& 01010101_2 \\ \hline \end{array} \text{ or } + \boxed{\begin{array}{|c|c|c|c|} \hline 00 & 01 & 00 & 01 \\ \hline 01 & 01 & 00 & 01 \\ \hline 01 & 10 & 00 & 10 \\ \hline \end{array}}$$

오른쪽에 나타난 덧셈은 실제로 더해지는 비트들을 보여준다. 바이트의 비트들은 사실상 4 개의 2 비트 부분으로 나뉘어서 4 개 각각이 독립적인 덧셈을 수행하는 것처럼 보인다. 2 비트 부분들의 합이 최대 2 이므로 오버플로우가 일어나 다른 부분들 의 덧셈의 결과를 엉망으로 만드는 경우는 발생하지 않는다.

당연하게도, 켜진 비트들의 총 수는 아직 계산되지 않았다. 그러나, 위와 같은 방법을 몇 번 적용하게 되면 켜진 비트들의 총 수를 계산할 수 있게 된다. 두 번째 단계는 다음과 같다:

$\text{data} = (\text{data} \& 0x33) + ((\text{data} \gg 2) \& 0x33);$

계속 적용하면, ( $\text{data}$  는 이제  $01100010_2$  이다)

$$\begin{array}{r} \text{data} \& 00110011_2 \\ + (\text{data} \gg 2) \& 00110011_2 \\ \hline \end{array} \text{ or } + \boxed{\begin{array}{|c|c|} \hline 0010 & 0010 \\ \hline 0001 & 0000 \\ \hline 0011 & 0010 \\ \hline \end{array}}$$

이제, 2 개의 4 비트 부분들이 독립적으로 더해졌다.

다음 단계는 이 2 비트 합들을 더해서 결과를 얻는 것이다.

$\text{data} = (\text{data} \& 0x0F) + ((\text{data} \gg 4) \& 0x0F);$

data (00110010<sub>2</sub> 와 값이 같다) 를 이용한 결과는

$$\begin{array}{rcl} \text{data} & \& 00001111_2 & 00000010 \\ + (\text{data} \gg 4) & \& \text{or} & + 00000011 \\ \hline & & & 00000101 \end{array}$$

이제 data 는 5 이고 이는 정확한 결과이다. 그림 3.8 은 더블워드의 경우 어떻게 적용하는지 나타내고 있다. 이는 for 루프를 이용하여 합을 계산한다. 루프를 풀어 내면 좀 더 빠르게 수행할 것이다. 그러나, 루프를 통해 데이터의 크기가 다른 경우에도 어떻게 적용하는지 보기 쉽다.

# Chapter 4

## 서브프로그램

이 장에서는 서브프로그램(subprogram)을 이용하여 모듈화 된 프로그램(modular program) 및 C와 같은 고급 언어들과 함께 작업하는 방법에 대해서 다루어 보고자 한다. 고급언어에서의 서브프로그램으로는 함수와 프로시저들이 있다.

서브프로그램을 호출하는 코드와 서브프로그램 그 자체는 반드시 이들 사이에 데이터를 주고 받는 방법이 같아야 한다. 이렇게, 이들 사이에 어떠한 방식으로 데이터를 주고받는 방법을 정해 놓은 것을 호출 규약(*calling convention*)이라고 부른다. 이 장은 C 프로그램들과 소통(interface)할 수 있는 어셈블리 서브프로그램 제작을 위해서, C 호출 규약에 대해 많은 부분을 할애할 것이다. 이 호출 규약은 (그 외 많은 호출 규약들도) 데이터의 주소(*i.e.* 포인터)를 전달해 서브프로그램이 메모리의 데이터에 접근 할 수 있게 하는 방식으로 이루어진다.

### 4.1 간접 주소 지정

간접적인 주소 지정은 레지스터를 포인터 변수들처럼 사용할 수 있게 한다. 레지스터가 포인터로 사용됨을 알려주기 위해서 대괄호([])로 묶어 주어야 한다. 예를 들어

```
1 mov ax, [Data]      ; 워드의 직접 메모리 주소 지정
2 mov ebx, Data        ; ebx = & Data
3 mov ax, [ebx]        ; ax = *ebx
```

AX가 워드를 보관하고 있기 때문에 3행은 EBX에 저장된 주소에서 시작되는 워드를 읽어 들이게 된다. 만약 AX 대신 AL을 이용한다면 오직 한 개의 바이트만 읽어 들이게 될 것이다. C에서의 변수들과는 달리 레지스터들은 형(type)이 없다는 사실을 명심해야 한다. EBX에 무엇을 가리키느냐는 어떠한 명령이 사용되느냐에 따라 완전히 달라진다. 심지어 명령에 의해서도 EBX가 포인터라는 사실도 정해진다. 만약 EBX가 잘못

사용 되어도 어셈블러는 아무런 오류를 내놓지 않는다. 그러나, 프로그램은 올바르게 작동하지 않는다. 이 때문에 어셈블리 프로그래밍은 고급 언어 프로그래밍 보다 훨씬 오류가 많게 된다.

모든 32 비트 범용 레지스터 (EAX, EBX, EDX, EDX) 와 인덱스 (ESI, EDI) 레지스터들은 모두 간접 주소 지정으로 사용될 수 있다. 통상적으로 16 비트와 8 비트는 사용될 수 없다,

## 4.2 간단한 서브프로그램 예제

서브프로그램은 프로그램의 다른 영역에서 쓰이는 독립적인 코드라고 볼 수 있다. 즉, 서브프로그램은 C 언어에서의 함수라고 생각하면 된다. 분기 명령을 통해 서브프로그램을 호출 할 수 있지만 리턴하는 부분에서 문제가 생긴다. 만약 서브프로그램이 프로그램의 각기 다른 영역에서 호출되었다면 서브프로그램은 호출된 각각의 위치로 돌아가기 위해 엄청난 수의 라벨을 사용하게 되므로 매우 복잡해 질 것이다. 반면에 아래의 코드는 이를 간접 형태의 JMP 명령을 사용하여 구현한 것을 볼 수 있다. 이 형태의 명령은 레지스터에 저장된 값을 이용하여 어디로 분기할 것인지 알 게 된다. (따라서, 레지스터들은 C에서의 함수 포인터(function pointer)과 같이 쓰인다) 아래는 1 장의 첫 번째 프로그램을 서브프로그램을 이용한 형태로 다시 작성한 것이다.

---

sub1.asm

```

1 ; 파일: sub1.asm
2 ; 서브프로그램 예제 프로그램
3 %include "asm_io.inc"
4
5 segment .data
6 prompt1 db      "Enter a number: ", 0          ; 널 종료 문자를 잊지 마시오
7 prompt2 db      "Enter another number: ", 0
8 outmsg1 db      "You entered ", 0
9 outmsg2 db      " and ", 0
10 outmsg3 db     " , the sum of these is ", 0
11
12 segment .bss
13 input1 resd 1
14 input2 resd 1
15
16 segment .text
17     global _asm_main
18 _asm_main:
19         enter 0,0                      ; 셋업 루틴
20         pusha
21

```

```

22      mov     eax, prompt1      ; prompt 를 출력
23      call    print_string
24
25      mov     ebx, input1       ; input1 의 주소를 ebx 에 저장
26      mov     ecx, ret1        ; ecx 에 리턴 주소를 저장
27      jmp     short get_int   ; 정수를 읽는다
28 ret1:
29      mov     eax, prompt2      ; prompt 출력
30      call    print_string
31
32      mov     ebx, input2       ; ebx = 현재 주소 + 7
33      mov     ecx, $ + 7
34      jmp     short get_int
35
36      mov     eax, [input1]      ; eax = input1 의 dword
37      add     eax, [input2]      ; eax += input2 의 dword
38      mov     ebx, eax          ; ebx = eax
39
40      mov     eax, outmsg1      ; 첫 번째 메세지 출력
41      call    print_string
42      mov     eax, [input1]
43      call    print_int         ; input1 을 출력
44      mov     eax, outmsg2      ; 두 번째 메세지 출력
45      call    print_string
46      mov     eax, [input2]
47      call    print_int         ; input2 를 출력
48      mov     eax, outmsg3      ; 세 번째 메세지 출력
49      call    print_string
50      mov     eax, ebx
51      call    print_int         ; 합 (ebx) 출력
52      call    print_nl          ; 개행 문자 출력
53
54      popa
55      mov     eax, 0            ; C 로 리턴
56      leave
57      ret
58 ; 서브프로그램 get_int
59 ; 인자:
60 ;   ebx - 정수를 저장한 dword 의 주소
61 ;   ecx - 리턴할 명령의 주소
62 ; 참고 사항:
63 ;   eax 에 저장된 값은 사라진다.

```

```

64    get_int:
65        call    read_int
66        mov     [ebx], eax      ; input 을 메모리에 저장
67        jmp     ecx           ; 호출한 곳으로 리턴


---



```

`get_int` 서브프로그램은 단순한 레지스터 기반 호출 규약을 사용한다. 이는 EBX 레지스터가 입력값을 저장할 DWORD의 주소를 갖고 있고, ECX 레지스터가 되돌아갈 코드 주소값을 가지고 있다고 생각한다. 25에서 28 행까지 `ret1` 라벨은 리턴 주소를 계산하기 위해 사용되었다. 32에서 34 행까지 \$ 연산자는 리턴 주소를 계산하기 위해 사용되었다. \$는 현재 행의 주소를 리턴한다. \$ + 7은 36 행의 MOV 명령의 주소를 계산하기 위해 사용되었다.

위 두 가지 방식의 리턴 주소 계산은 매우 복잡하다. 첫 번째 방법은 매 서브프로그램 호출마다 라벨을 필요로 한다. 두 번째 방법은 라벨을 필요로 하진 않지만 오류가 나지 않기 위해 심사숙고 할 필요가 있다. 예를 들어 `near` 분기 대신에 `short` 분기가 사용되었다면 \$에 더해진 값은 7이 아니게 된다. 다행이도, 서브프로그램을 호출하기 위한 더 쉬운 방법이 있다. 이 방법은 스택(stack)을 이용한다.

### 4.3 스택

대다수의 CPU는 스택의 사용을 지원한다. 스택은 후입 선출(Last-In First-Out, LIFO)라고 부르며, 나중에 들어간 데이터가 먼저 나오게 된다) 리스트이다. 스택은 이와 같은 규칙으로 구성된 메모리의 일부분이다. `PUSH` 명령을 통해 스택에 데이터를 추가하고 `POP` 명령을 통해 데이터를 빼낼 수 있다. 언제나 빼내진 데이터는 마지막으로 추가된 데이터이다. (이 때문에 스택을 후입 선출 리스트라 부른다)

SS 세그먼트 레지스터는 스택을 보관한 세그먼트를 정의한다. (보통 이는 데이터가 저장된 세그먼트와 동일하다) `ESP` 레지스터는 스택으로부터 빼내질 데이터의 주소를 보관한다. 이 데이터는 흔히 스택의 최상위(*top*)에 있다고 말한다. 데이터는 오직 더블워드의 형태로만 저장된다. 따라서, 스택에 단일 바이트를 집어 넣을 수 없다.

`PUSH` 명령은 `ESP` 레지스터의 값을 4 감소 시킨 후, 더블워드<sup>1</sup>를 `[ESP]`에 위치한 더블워드에 집어 넣는다. `POP` 명령은 `[ESP]`에 위치한 더블워드를 읽어 들이고, `ESP`에 4를 더한다. 아래의 코드는 `ESP`가 `1000H`이였다고 할 때 명령에 따라 값이 어떻게 변화하는지 보여준다.

```

1    push    dword 1    ; 1이 OFFCh에 저장됨, ESP = OFFCh
2    push    dword 2    ; 2가 OFF8h에 저장됨, ESP = OFF8h
3    push    dword 3    ; 3이 OFF4h에 저장됨, ESP = OFF4h

```

<sup>1</sup> 사실, 워드들도 스택에 들어갈 수 있지만 32비트 보호 모드에서는 오직 더블워드들만 가지고 다루는 것이 낫다.

```

4      pop    eax          ; EAX = 3, ESP = OFF8h
5      pop    ebx          ; EBX = 2, ESP = OFFCh
6      pop    ecx          ; ECX = 1, ESP = 1000h

```

스택은 임시적으로 데이터를 저장하는데 요긴하게 사용할 수 있다. 이를 통해서 인자(parameter)들과 지역 변수(local variable)을 전달하고 서브프로그램 호출을 할 수 있다.

80x86은 또한 PUSHA 명령을 이용하여 EAX, EBX, ECX, EDX, ESI, EDI, EBP 레지스터의 값을 모두 스택에 집어 넣을(푸시) 수 있게 한다. (이 순서로 집어 넣는 것은 아니다) POPA 명령을 통해 이들을 다시 빼낼(팝) 수 있게 한다.

## 4.4 CALL 과 RET 명령

80x86은 서브프로그램을 빼고 간편하게 호출하기 위해서 스택을 이용한 2 가지의 명령을 지원한다. CALL 명령은 서브프로그램으로의 무조건 분기를 한 후, 그 다음에 실행될 명령의 주소를 스택에 푸시(push)한다. RET 명령은 그 주소를 팝(pop) 한 후 그 주소로 점프를 한다. 이 명령을 이용할 때, 스택을 정확하게 관리하여 RET 명령에 의해 정확한 주소 값이 팝 될 수 있도록 해야 한다.

따라서 이전의 프로그램은 새로운 명령을 이용하여 아래와 같이 25에서 34 행을 바꾸어 쓸 수 있다.

---

```

mov    ebx, input1
call   get_int

mov    ebx, input2
call   get_int

```

---

또한 서브프로그램 `get_int`를 아래와 같이 바꾼다:

---

```

get_int:
    call   read_int
    mov    [ebx], eax
    ret

```

---

CALL과 RET에는 몇 가지 장점들이 있다.

- 이는 단순하다!

- 서브프로그램 호출이 쉽게 이루어지게 한다. `get_int` 가 `read_int` 를 호출 하는 부분을 보아라. 이 호출은 스택에 또 다른 주소를 집어 넣는다. `read_int` 코드의 마지막 부분에는 RET 가 리턴 주소를 빼내고, `get_int` 의 코드로 다시 분기 된다. 그리고 `get_int` 의 RET 이 이루어진다면 리턴 주소를 빼내어 `asm_main` 으로 분기된다. 이는 스택이 LIFO 구조 이기에 올바르게 일어난다.

스택에 푸시 된 모든 데이터는 나중에 반드시 다시 팝 해야 한다. 예를 들어, 아래와 같은 소스 코드를 생각해 보자:

```

1 get_int:
2     call    read_int
3     mov     [ebx], eax
4     push   eax
5     ret          ; 리턴 주소 대신 EAX 값을 팝!

```

이 코드는 올바르게 리턴되지 않는다.

## 4.5 호출 규약

서브프로그램이 호출 되었을 때, 호출하는 코드와 서브프로그램 (이를 피호출자(callee) 은 서로의 데이터 전송 방식이 같아야 한다. 고급 언어에서는 호출 규약(calling conventions) 이라는 기본적인 데이터 전송 방법이 있다. 고급 언어 코드와 어셈블리가 함께 작업하기 위해서는 어셈블리 코드들도 반드시 고급 언어에서의 규약과 일치해야 한다. 이 호출 규약은 컴파일러마다 다를 수 있으며, 코드가 어떠한 방식으로 컴파일 되느냐에 따라서 (e.g 최적화의 유무)도 다를 수 있다. 하나의 공통된 점이라 하면 코드는 CALL 을 통해 호출하고 RET 을 통해 리턴된다는 점이다.

모든 PC C 컴파일러는 이 장의 나머지 부분에서 다루게 될 호출 규약을 지원한다. 이 규약을 통해 재진입(reentrant) 서브프로그램을 만들 수 있다. 재진입 서브프로그램은 프로그램의 어떤 부분에서도 자유롭게 호출 될 수 있다. (심지어 서브프로그램 그 자체 내부에서도)

### 4.5.1 스택에 인자들을 전달하기

서브프로그램으로 스택을 통해 인자들이 전달 될 수 있다. 이 인자들은 CALL 명령을 하기 전에 스택으로 푸시되어진다. C 에서처럼 서브프로그램에서 인자의 값이 바뀌기 위해서는 인자의 값이 아닌, 인자의 주소가 반드시 전달 되어야만 한다. 만약 인자의 크기가 더블워드보다 작다면, 반드시 푸시되기 전에 더블워드로 변환되어야만 한다.

스택에 저장된 인자들은 서브프로그램에 의해 빼내어(Pop) 지지 않는다. 그 대신에, 이 들은 스택 자체에서 접근이 가능하다. 왜냐하면,

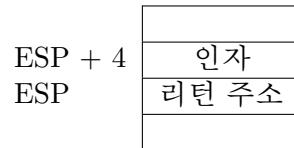


Figure 4.1:

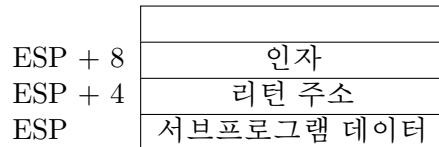


Figure 4.2:

- 이들은 CALL 명령 전에 스택에 푸시 되어야만 하기 때문에 리턴 주소는 가장 먼저 팝 되어야만 한다
- 인자들은 종종 서브프로그램의 여러 부분에서 사용된다. 보통, 서브프로그램 실행 내내 레지스터에 인자들을 저장할 수 없으므로, 메모리에 저장되어야 한다. 따라서, 인자들을 스택에 보관함으로써 서브프로그램의 어떠한 부분에서도 인자에 접근 할 수 있게 된다.

스택에 하나의 인자를 저장한 서브프로그램을 생각하자 서브프로그램이 호출되었을 때, 스택은 4.1 과 같이 나타난다. 간접 주소 지정을 사용하여 인자에 접근 ( $[ESP+4]$ <sup>2</sup>) 할 수 있다.

만일, 서브프로그램 내에서도 데이터를 저장하기 위해 스택을 사용한다면,  $ESP$  에 더해져야 할 값이 달라지게 된다. 예를 들어 그림 4.2에서 더블워드가 스택에 푸시되었을 때 어떻게 나타나는지 보여준다. 이 때, 인자는  $ESP + 4$  가 아닌  $ESP + 8$ 에 위치하게 된다. 따라서,  $ESP$  를 이용하여 인자에 접근시 오류를 낼 확률이 크다. 이 문제를 해결하기 위해 80386은  $EBP$  라는 또 다른 레지스터를 사용 가능하게 했다. 이 레지스터의 목적은 스택의 데이터를 가리키는 것 이다. C 호출 규약에선 반드시 서브프로그램이 먼저  $EBP$  의 값을 스택에 저장하고  $EBP$  가  $ESP$  와 같게 만들게 해야 한다. 이를 통해  $EBP$  의 값을 변경하지 않고도 스택에 데이터가 푸시되고 팝됨에 따라  $ESP$  가 변경될 수 있게 한다. 서브프로그램의 끝 부분에선 원래의  $EBP$  값이 반드시 불러와져야 한다(이 때문에 우리가 서브프로그램 처음에  $EBP$  값을 저장했다) 그림 4.3 은 위 규약들을 만족하는 서브프로그램의 일반적인 모습을 보여준다.

그림 4.3의 2 와 3 행에서 보통의 서브프로그램의 시작 부분을 보여주고 있다. 5 와 6 행에서는 마지막 부분을 보여준다. 그림 4.4에서 시작 부분을 막 통과한 후의 스택의 모습을 보여준다. 이제, 인자들은  $[EBP + 8]$  을 통

간접 주소지정을 사용시, 80x86 프로세서는 간접 주소지정 표현시 어떠한 레지스터가 사용되느냐에 따라서 각기 다른 세그먼트에 접근한다.  $ESP$  (또한  $EBP$ ) 는 스택 세그먼트를 사용하며,  $EAX$ ,  $EBX$ ,  $ECX$ ,  $EDX$  는 데이터 세그먼트를 사용한다. 그러나, 보호 모드 프로그램에서는 별로 중요한 문제는 아니다. 왜냐하면 데이터 세그먼트와 스택 세그먼트는 동일하기 때문이다.

<sup>2</sup>간접 주소지정을 사용시 레지스터에 상수를 더하는 것이 가능하다. 뿐만 아니라 더 복잡한 표현들도 가능하다. 이것에 대한 자세한 내용은 다음 장에서 다룬다

```

1 subprogram_label:
2     push    ebp          ; 원래의 EBP 값을 스택에 저장
3     mov     ebp, esp    ; 새로운 EBP = ESP
4 ; subprogram code
5     pop    ebp          ; 이전의 EBP 값을 불러온다.
6     ret

```

Figure 4.3: 통상적인 서브프로그램의 형태

ESP + 8	EBP + 8	인자
ESP + 4	EBP + 4	리턴 주소
ESP	EBP	저장된 EBP

Figure 4.4:

해 서브프로그램에 의해 스택에 데이터가 푸시되고 팝 되는 것에 무관하게 어떠한 부분에서도 인자에 접근할 수 있게 되었다.

서브프로그램이 종료되면, 스택에 푸시되었던 인자들은 반드시 제거되어야만 한다. C 호출 규약은 호출자의 코드가 이를 하게 명시했다. 모든 언어의 호출 규약이 동일한 것이 아니다. 예를 들어 파스칼의 호출 규약은 서브프로그램이 이를 제거하라 명시했다. (이를 쉽게 하기 위한 RET 명령의 다른 형태가 있다) 일부 C 컴파일러들은 이 규약을 지원하기도 한다. pascal 키워드를 함수의 원형과 정의 부분에서 사용하여 컴파일러로 하여금 파스칼의 호출 규약을 이용하게 만들 수 있다. 사실, 마이크로소프트 윈도우즈 API C 함수들이 이용하는 stdcall 규약도 파스칼의 경우와 같다. 이 방법을 이용하면 어떠한 장점이 있을까? 이는 C 호출 규약보다 조금 더 능률적이다. 그렇다면 왜 C 함수들은 이 호출규약을 이용하지 않는 것일까? 보통 C는 함수가 가변 개수의 인자를 가짐을 지원한다 (e.g., printf 와 scanf 가 대표적). 이러한 형식의 함수에선 스택에서 인자들을 지우는 명령들의 횟수가 달라지게 된다. 이러한 형식의 함수들은, 스택으로부터 인자들을 제거하는 명령은 함수의 호출마다 달라지게 된다. C 호출 규약에선 이러한 연산들을 손쉽게 할 수 있는 명령들을 지원한다. 그러나 파스칼이나 stdcall 호출 규약에선 이러한 연산들은 매우 복잡하다. 따라서 파스칼 호출 규약은 이러한 형식의 함수를 지원하지 않는다. 마이크로소프트 윈도우즈의 경우 어떠한 API 함수도 가변 인자를 가지지 않기 때문에 파스칼 형태의 호출 규약을 이용할 수 있었다.

그림 4.5는 C 호출 규약을 이용하는 서브프로그램을 어떻게 호출하는지 보여준다. 3행은 스택 포인터의 값을 조정함으로써 인자를 스택에서 지워버린다. POP 명령을 통해 이를 할 수도 있으나, 필요 없는 값이 레지스터에 저장되어어야 한다. 사실, 이 특별한 경우에 대해선 많은 컴파일러들은

```

1      push    dword 1          ; 1 을 인자로 전달
2      call    fun
3      add     esp, 4          ; 스택에서 인자를 제거

```

Figure 4.5: 서브프로그램 호출 예제

POP ECX 명령을 이용하여 인자를 제거한다. 왜냐하면 ADD 명령이 더 많은 바이트를 사용하기 때문이다. 그러나 POP 또한 ECX의 값을 바꾼다. 다음의 또 다른 예제는 위에서 설명한 C 호출 규약을 사용한 프로그램들이다. 54 행 (그리고 다른 행들에서)에서 하나의 소스파일에 여러개의 데이터와 텍스트 세그먼트들이 정의되었음을 볼 수 있다. 이들은 링크 과정에서 하나의 데이터와 텍스트 세그먼트들로 합쳐질 것이다. 데이터와 코드를 여러개의 세그먼트로 쪼갬을 통해 서브프로그램이 사용하는 데이터가 서브프로그램 근처에 정의되게 할 수 있다.

---

sub3.asm

```

1  %include "asm_io.inc"
2
3  segment .data
4  sum    dd    0
5
6  segment .bss
7  input   resd 1
8
9  ;
10 ; 의사 코드 알고리즘
11 ; i = 1;
12 ; sum = 0;
13 ; while( get_int(i, &input), input != 0 ) {
14 ;     sum += input;
15 ;     i++;
16 ; }
17 ; print_sum(num);
18 segment .text
19     global _asm_main
20 _asm_main:
21         enter  0,0          ; 셋업 루틴
22         pusha
23
24         mov     edx, 1          ; 의사 코드에서의 i 가 edx 이다.
25 while_loop:

```

```

26      push    edx          ; i 를 스택에 저장
27      push    dword input   ; 입력값의 주소를 스택에 저장 % 정확한가?
28      call    get_int
29      add     esp, 8        ; i 와 &input 을 스택에서 제거
30
31      mov     eax, [input]
32      cmp     eax, 0
33      je     end_while
34
35      add     [sum], eax    ; sum += input
36
37      inc     edx
38      jmp     short while_loop
39
40  end_while:
41      push    dword [sum]    ; 합을 스택에 푸시
42      call    print_sum
43      pop     ecx          ; 스택에서 [sum] 을 제거
44
45      popa
46      leave
47      ret
48
49 ; 서브프로그램 get_int
50 ; 인자들 (스택에 푸시된 순서로 나탁남)
51 ;   입력값의 수 ([ebp + 12] 에 위치)
52 ;   입력값을 저장할 워드의 주소 ([ebp + 8] 에 위치)
53 ; 참고 :
54 ;   eax 와 ebx 의 값들은 짜고된다.
55 segment .data
56 prompt db      ") Enter an integer number (0 to quit): ", 0
57
58 segment .text
59 get_int:
60      push    ebp
61      mov     ebp, esp
62
63      mov     eax, [ebp + 12]
64      call    print_int
65
66      mov     eax, prompt
67      call    print_string

```

```

68
69      call    read_int
70      mov     ebx, [ebp + 8]
71      mov     [ebx], eax          ; 입력 값을 메모리에 저장
72
73      pop     ebp
74      ret                 ; 호출자로 분기
75
76 ; 서브프로그램 print_sum
77 ; 합을 출력한다
78 ; 인자:
79 ;   출력하기 위한 합([ebp+8]에 위치)
80 ; 참고: eax의 값이 짜고된다.
81 ;
82 segment .data
83 result db      "The sum is ", 0
84
85 segment .text
86 print_sum:
87     push   ebp
88     mov    ebp, esp
89
90     mov    eax, result
91     call   print_string
92
93     mov    eax, [ebp+8]
94     call   print_int
95     call   print_nl
96
97     pop    ebp
98     ret

```

---

#### 4.5.2 스택에서의 지역변수

스택은 지역변수들의 편리한 저장 창고로 사용될 수 있다. 이는 C가 보통의 변수들(또는 C 용어로 *automatic*)을 저장하는 곳이다. 변수들을 위해 스택을 사용하는 것은 재진입 서브프로그램을 만드는데 유용하다. 재진입 서브프로그램은 서브프로그램 그 자체를 포함한 어떠한 곳에서도 호출되어도 잘 작동하는 서브프로그램을 말한다. 다시 말해, 재진입 서브프로그램은 재귀적(*recursively*)으로 호출 가능하다. 변수들을 위해 스택을 사용하는 것을 통해 메모리도 아낄 수 있다. 스택에 저장되지 않는 데이터들은 프로그램의 시작부터 종료 될 때 까지 메모리 상에 상주하게 된다. (C는 이러한

```

1 subprogram_label:
2     push    ebp          ; 기존의 EBP 값을 스택에 저장한다.
3     mov     ebp, esp    ; 새로운 EBP = ESP
4     sub     esp, LOCAL_BYTES ; 지역 변수에 의해 # 바이트 필요
5 ; 서브프로그램 코드
6     mov     esp, ebp    ; 지역 변수를 해제(deallocate)
7     pop     ebp          ; 기존의 EBP 값을 불러온다.
8     ret

```

Figure 4.6: 지역 변수를 이용한 일반적인 서브프로그램 모습

```

1 void calc_sum( int n, int * sump )
2 {
3     int i, sum = 0;
4
5     for( i=1; i <= n; i++ )
6         sum += i;
7     *sump = sum;
8 }

```

Figure 4.7: 합 프로그램의 C 언어 버전

형식의 변수들을 전역(*global*)이나 정적(*static*) 변수라 부른다) 스택에 저장된 데이터들의 경우 오직 서브프로그램이 실행될 때에만 메모리 상에 상주한다.

지역 변수들은 스택에 저장된 EBP 값 바로 다음부터 저장된다. 이들은 프로그램 서두에서, ESP로 부터 필요한 만큼의 바이트를 뺌으로써 할당될 수 있다. 그림 4.6은 새로운 서브프로그램의 구조를 보여준다. EBP 레지스터는 지역 변수에 접근하기 위해 사용된다. 그림 4.7의 C 함수를 보자. 그림 4.8은 이와 동일한 서브프로그램이 어셈블리어로 어떻게 쓰일 수 있는지 보여준다.

그림 4.9은 그림 4.8의 프로그램 상단 부분을 실행한 후의 스택의 모습을 보여준다. 인자, 리턴 정보, 지역 변수를 포함한 위와 같은 스택의 영역을 스택 프레임(*stack frame*)이라 부른다. C 함수를 호출할 때마다 스택에 새로운 스택 프레임을 생성하게 된다.

프로그램의 시작 부분과 끝 부분은 오직 이러한 목적을 위해 만들어진 두 명령 ENTER 와 LEAVE 명령이 사용되고 있다. ENTER 명령은 시작 부분의 명령을 처리하고 LEAVE 명령은 마지막 부분의 명령을 처리한다. ENTER 명령은 두 개의 즉시 피연산자를 가진다. C 호출 규약에선 두 번째 피연산자는 언제나 0이다. 첫 번째 인자는 지역 변수가 필요로 하는 바이트의 크기를

비록 ENTER 와 LEAVE 명령들이 서브프로그램의 시작 부분과 끝 부분을 간단하게 해주지만 잘 쓰이지는 않는다. 왜냐하면, 이들은 동일한 간단한 명령들의 모임보다 더 느리기 때문이다. 이는 우리가 언제나 하나의 명령이 여러개의 명령들보다 빠르다고 확신할 수 없음을 보여주는 하나의 예이다.

```

1    cal_sum:
2        push    ebp
3        mov     ebp, esp
4        sub     esp, 4          ; 지역 변수 sum 의 공간을 할당
5
6        mov     dword [ebp - 4], 0 ; sum = 0
7        mov     ebx, 1           ; ebx (i) = 1
8    for_loop:
9        cmp     ebx, [ebp+8]      ; i <= n 인가?
10       jnle   end_for
11
12       add     [ebp-4], ebx      ; sum += i
13       inc     ebx
14       jmp     short for_loop
15
16   end_for:
17       mov     ebx, [ebp+12]      ; ebx = sump
18       mov     eax, [ebp-4]      ; eax = sum
19       mov     [ebx], eax        ; *sump = sum;
20
21       mov     esp, ebp
22       pop     ebp
23       ret

```

Figure 4.8: 합 프로그램의 어셈블리 버전

나타낸다. LEAVE 명령은 피연산자를 갖지 않는다. 그림 4.10 은 이러한 명령들이 어떻게 쓰이는지 보여주고 있다. 이 때, 골격 프로그램 (그림 1.7) 또한 ENTER 와 LEAVE를 사용하고 있음을 주목해라.

## 4.6 다중 모듈 프로그램

다중 모듈 프로그램(*multi-module program*)은 한 개보다 많은 수의 목적 파일 (object file)들로 구성된 프로그램을 말한다. 여태까지 소개한 프로그램들은 모두 다중 모듈 프로그램이였다. 이들은 C 드라이버 목적 파일과 어셈블리 목적 파일 (C 라이브러리 목적 파일도)들로 이루어져 있다. 링커는 이러한 여러 개의 목적 파일들을 하나의 실행 가능한(executable) 프로그램으로 통합해 주는 것을 상기해라. 링커는 각 모듈(*i.e.* 목적 파일)들에서의 라벨들을 각각의 정의에 맞게 대응시켜 주어야만 한다. 모듈 B에서 정의된 라벨을 모듈 A에서 사용하기 위해선, `extern` 지시어가 사용되어야 한다.

ESP + 16	EBP + 12	sump
ESP + 12	EBP + 8	n
ESP + 8	EBP + 4	리턴 주소
ESP + 4	EBP	저장된 EBP
ESP	EBP - 4	sum

Figure 4.9:

```

1 subprogram_label:
2     enter LOCAL_BYTES, 0      ; 지역 변수에 의해 필요 한 = # 바이트
3 ; subprogram code
4     leave
5     ret

```

Figure 4.10: ENTER 와 LEAVE 명령, 그리고 지역 변수를 사용한 일반적인 서브프로그램의 모습

`extern` 지시어 옆에는 반점으로 구분은 라벨들의 목록들이 나열된다. 이 지시어는 어셈블러에 이 라벨들이 모듈이 외부(*external*)에서 정의되었음을 알려준다. 따라서, 이러한 라벨들은 다른 모듈에서 정의되었음에도 불구하고 사용될 수 있다. `asm_io.inc` 파일은 `read_int`, 루틴 등을 모두 외부로 선언한다.

어셈블리에서는 기본적으로 라벨들은 외부에서 접근이 불가능하도록 되어 있다. 만약 라벨이 다른 모듈에서 접근이 가능하다면 이 라벨은 정의된 모듈에서 반드시 전역으로 선언되어야 한다. `global` 지시어가 이 일을 한다. 아래 그림 1.7의 뼈대 프로그램의 13 행을 보면, `_asm_main` 라벨이 전역으로 정의되어 있음을 볼 수 있다. 이 정의 없이는 링커 오류가 발생할 것이다. 왜냐하면 C 코드는 내부(*internal*)로 선언된 `_asm_main` 라벨을 참조할 수 없기 때문이다.

다음 코드는 이전의 예제를 두 개의 모듈을 이용하여 사용한 것이다. 이 두 개의 서브프로그램 (`get_int` 와 `print_sum`) 은 `_asm_main` 루틴과는 달리 2 개의 개개의 소스 파일에 작성되어 있다.

---

main4.asm

```

1 %include "asm_io.inc"
2
3 segment .data
4 sum     dd    0
5
6 segment .bss
7 input   resd 1
8

```

```

9 segment .text
10      global _asm_main
11      extern get_int, print_sum
12 _asm_main:
13      enter 0,0          ; 셋업 투 트
14      pusha
15
16      mov edx, 1          ; edx 는 의사코드에서 'i' 와 같다.
17 while_loop:
18      push edx           ; i 를 스택에 저장한다.
19      push dword input    ; 입력값의 주소를 스택에 푸시
20      call get_int
21      add esp, 8          ; i 와 &input 을 스택에서 제거
22
23      mov eax, [input]
24      cmp eax, 0
25      je end_while
26
27      add [sum], eax       ; sum += input
28
29      inc edx
30      jmp short while_loop
31
32 end_while:
33      push dword [sum]     ; 합을 스택에 푸시
34      call print_sum
35      pop ecx             ; [sum] 을 스택에서 제거
36
37      popa
38      leave
39      ret

```

---

## sub4.asm

```

1 %include "asm_io.inc"
2
3 segment .data
4 prompt db      ") Enter an integer number (0 to quit): ", 0
5
6 segment .text
7      global get_int, print_sum
8 get_int:
9      enter 0,0
10

```

```

11      mov     eax, [ebp + 12]
12      call    print_int
13
14      mov     eax, prompt
15      call    print_string
16
17      call    read_int
18      mov     ebx, [ebp + 8]
19      mov     [ebx], eax           ; 입력 값을 메모리에 저장
20
21      leave
22      ret                 ; 호출 자료 분기
23
24 segment .data
25 result db      "The sum is ", 0
26
27 segment .text
28 print_sum:
29         enter   0,0
30
31         mov     eax, result
32         call    print_string
33
34         mov     eax, [ebp+8]
35         call    print_int
36         call    print_nl
37
38         leave
39         ret

```

---

sub4.asm

이전의 코드는 오직 전역 코드 라벨들만 가지고 있었다. 그러나, 전역 데이터 라벨들은 똑같은 방법으로 작동된다.

## 4.7 C 와 소통하기

오늘날, 매우 적은 수의 프로그램들만 완전히 어셈블리어로 작성된다. 컴파일러는 고급 레벨 코드를 효율적인 기계어 코드로 변환하는 일을 매우 잘한다. 고급 언어로 코드를 작성하는 것이 쉽기에, 이는 훨씬 대중적이다. 게다가, 고급 언어는 어셈블리어에 비해 훨씬 이식성이 높다!

만일 어셈블리어가 직접적으로 사용된다면 오직 코드의 작은 부분들에 서만 사용된다. 이는 두 가지 방법으로 가능하다: C에서 어셈블리 서브루틴을 호출하던지, 인라인(inline) 어셈블리를 이용하는 것이다. 인라인 어셈블

리는 프로그래머로 하여금 C 코드에 어셈블리 문장을 직접적으로 넣을 수 있게 해주는 것이다. 이는 매우 편리하다. 그러나, 인라인 어셈블리에도 몇 가지 단점이 있다. 어셈블리 코드는 반드시 컴파일러가 사용하는 형식으로 작성되어야만 한다. 그러나, 현재 NASM 의 형식을 지원하는 컴파일러가 없다. 다른 컴파일러들은 다른 형식을 사용한다. 볼랜드와 마이크로소프트사의 경우 MASM 형식을 이용한다. DJGPP 와 리눅스의 gcc 는 GAS<sup>3</sup> 형식을 이용한다. PC 에선 어셈블리 서브루틴을 호출하는 방법이 훨씬 표준화 되어 있다.

어셈블리 루틴들은 다음과 같은 이유로 C 코드와 함께 사용된다:

- C 로 하기애 어렵거나 매우 힘든 컴퓨터 하드웨어의 직접적인 접근
- 매우 빠르게 작동 되어야만 루틴, 컴파일러가 할 수 있는 것 보다도 빠르게 프로그래머가 직접 손으로 최적화를 해야 되는 부분

마지막 이유는 예전과는 달리 거의 사실이 아니게 되었다. 컴파일러 기술은 해를 거듭하여 진보하였고, 컴파일러들은 매우 효율적인 코드(특히, 컴파일러 최적화가 진행된다면)를 만들어 낼 수 있다. 어셈블리 루틴의 단점은 : 이식성이 낮고, 가독성이 떨어진다.

대부분의 C 호출 규약은 이미 이야기 하였다. 그러나, 몇 개의 설명이 필요한 부분들이 남아있다.

#### 4.7.1 레지스터 저장하기

먼저, C 는 서브루틴이 다음의 레지스터의 값을 보관하다고 생각한다: EBX, ESI, EDI, EBP, CS, DS, SS, ES. 이 말은, 서브루틴이 내부적으로 이들을 바꿀 수 없다는 것이 아니다. 그 대신에, 이것이 말은 이 레지스터들의 값을 바꾸더라도 서브루틴이 리턴할 경우 이전의 값을 복원할 수 있어야만 한다는 것이다. EBX, ESI, EDI 의 값을 반드시 변경되면 안되는데 왜냐하면 C 는 이 레지스터들을 레지스터 변수(*register variable*)로 사용하기 때문이다. 보통 스택은 위 레지스터들의 원래 값을 저장하는데 사용된다.

`register` 키워드는 컴파일러에게 C 변수를 메모리 공간에 저장하지 않고 레지스터에 저장하라고 명령하는 것이다. 현대의 컴파일러들은 이를 특별히 명시하지 않고도 자동적으로 해준다.

#### 4.7.2 함수들의 라벨

대부분의 C 컴파일러들은 함수나, 전역/정적 변수들의 이름 앞에 \_ 한 개를 붙인다. 예를 들어서 `f` 라는 이름의 함수는 `_f` 라는 이름의 라벨로 대응된다. 따라서, 이것이 어셈블리 루틴이 되기 위해선 반드시 `f` 가 아닌 `_f` 로 라벨이 붙어야만 한다. 리눅스 gcc 컴파일러는 어떠한 문자도 붙이지 않는다. 리눅스 ELF 실행 가능 파일들의 경우, C 함수 `f` 에 대해 그냥 `f` 라는 이름의 라벨을 사용한다. 그러나 DJGPP 의 gcc 는 `_` 를 붙인다. 어셈블리

---

<sup>3</sup>GAS 는 모든 GNU 컴파일러가 사용하는 어셈블러이다. 이는 AT&T 문법을 이용하는데, AT&T 의 문법은 문법이 대략 비슷한 MASM, TASM, NASM 들과는 달리 매우 다르다.

```

1 segment .data
2 x dd 0
3 format db "x = %d\n", 0
4
5 segment .text
6 ...
7 push dword [x] ; x의 값을 푸시
8 push dword format ; 형식 문자열의 주소를 푸시
9 call _printf ; _에 주목!
10 add esp, 8 ; 스택에서 인자들을 제거

```

Figure 4.11: printf로 호출

EBP + 12	x의 값
EBP + 8	형식 문자열의 주소
EBP + 4	주소를 리턴
EBP	EBP를 저장

Figure 4.12: printf 내부의 스택 모습

골격 프로그램(그림 1.7)을 보면 메인 루틴의 라벨은 `_asm_main`로 되어 있음을 주목해라.

### 4.7.3 인자 전달하기

C 호출 규약의 경우, 스택에 푸시된 함수들의 인자들은 함수 호출 시 역순으로 나타나게 된다.

예를 들어 다음과 같은 C 문장을 생각해 보자:`printf("x = %d\n", x);` 그림 4.11은 이것이 어떻게 컴파일 될지 보여준다. (동일한 NASM 형식으로 나타남) 그림 4.12은 `printf`의 처음 부분을 실행한 후의 스택의 모습을 보여준다. `printf` 함수는 C 라이브러리의 함수 중 하나로, 가변 개수의 인자들을 가질 수 있다. C 호출 규약의 경우 이러한 형식의 함수들을 작성할 수 있다.

C에서 어셈블리를 이용하여 임의의 개수의 인자를 받는 과정을 만들어 줄 필요는 없다. `stdarg.h` 헤더 파일은 위 과정을 손쉽게 할 수 있는 매크로를 제공해 준다. 다른 C 언어 책을 참조하면 된다.

형식 문자열의 주소가 마지막에 푸시되므로, 이것의 스택에서의 위치는 언제나 얼마나 많은 수의 인자들이 푸시되느냐에 상관 없이 EBP+8 이 된다. `printf` 코드는 형식 문자열을 보아서 얼마나 많은 수의 인자들이 전해졌는지 알아내어, 스택을 찾아 볼 수 있다.

당연하게도, `printf("x = %d\n")` 와 같이 실수를 한다면 `printf` 코드는 [EBP+12]에 위치한 더블워드 값을 출력할 것이다. 그러나, 이 값은 x의 값이 아니다!

#### 4.7.4 지역 변수의 주소 계산하기

`data` 나 `bss` 세그먼트에 정의된 라벨의 주소를 찾는 것은 간단하다. 기본적으로 링커가 이러한 작업을 한다. 그러나, 직관적으로 스택에 저장된 지역 변수(혹은 인자)의 주소를 계산하는 것은 어렵다. 그러나, 우리가 서브루틴들을 호출시에 이 작업을 자주 하게 된다. 아래와 같이 변수(x 라하자)의 주소를 함수(이를 `foo` 라 한다)에 전달하는 경우를 살펴보자. 만일 `x` 가 `EBP - 8` 의 스택에 위치해 있다면, 우리는 아래와 같이 그냥 사용할 수 없다.

```
mov     eax, ebp - 8
```

왜냐하면 `MOV` 명령을 통해 `EAX`로 저장되는 값이 계산되어야만 하기 때문이다. (이 명령의 피연산자는 반드시 상수여야 한다.) 그러나, 위와 같은 연산을 하는 명령이 있다. 이는 `LEA` (*Load Effective Address*의 약자)이다. 아래의 코드는 `EAX`로 `x`의 주소를 계산하여 집어 넣는다.

```
lea     eax, [ebp - 8]
```

이제 `EAX`는 `x`의 주소를 가지고, `foo`를 호출시에 스택에 푸시될 수 있다. 종종 위 명령을 `[EBP-8]`에 있는 데이터를 읽어 들이는 것으로 착각하는데, 이것은 사실이 아니다. `LEA` 명령은 절대로 메모리를 읽어들이지 않는다. 이는 오직 다른 명령이 읽어들일 주소값을 계산하고, 이를 첫 번째 레지스터 피연산자에 저장할 뿐이다. 이것이 메모리를 읽어들이지 않기 때문에 메모리 크기를 지정(*e.g. dword*) 할 필요가 없다.

#### 4.7.5 리턴값

`void` C 함수가 아닌 것들은 모두 값을 반환한다. C 호출 규약은 이를 어떻게 해야할지 정해 놓았다. 리턴값은 레지스터를 통해서 전달된다. 모든 정수형 (`char`, `int`, `enum`, *etc.*) 리턴값들은 `EAX` 레지스터에 저장되어 리턴된다. 만일 리턴값이 32비트보다 작을 경우 32비트로 확장되어 `EAX`에 저장된다. 64비트 값들은 `EDX:EAX` 레지스터 쌍에 저장된다. 포인터 값 또한 `EAX`에 저장된다. 부동 소수점 값들의 경우, 수치 부프로세서의 `ST0` 레지스터에 저장된다. (이 레지스터에 대해선, 부동 소수점 장에서 다루겠다.)

#### 4.7.6 다른 호출 규약

위에 설명한 것들은 모두 80x86 C 컴파일러에 의해 지원되는 표준 C 호출 규약이였다. 보통 컴파일러들은 다른 형식의 호출 규약들도 지원한다. 어셈블리 언어와 다른 언어를 같이 이용한다면, 컴파일러가 무슨 호출 규약을 이용하는지 아는 것이 매우 중요하다. 많은 경우 기본적으로는 표준

호출 규약을 이용한다. 그러나, 모든 컴파일러가 그러는 것은 아니다.<sup>4</sup> 여러 종류의 호출 규약을 사용할 수 있는 컴파일러들의 경우, 커맨드 라인에서 스위치를 이용하여 기본값으로 무슨 호출 규약을 이용 할지 설정 할 수 있다. 또한, C의 문법을 확장하여 개개의 함수를 다른 호출 규약을 이용하여 컴파일 할 수도 있다. 그러나, 이러한 확장은 표준화 되어 있지 않으며, 컴파일러마다 다를 수 있다.

GCC 컴파일러는 여러 종류의 호출 규약을 지원한다. 함수의 호출 규약은 `_attribute_`. 확장을 이용하여 개별로 지정할 수 있다. 예를 들어서, 표준 호출 규약을 이용하는 `f` 란 이름의 1 개의 `int` 인자를 가지는 `void` 함수를 선언하려면, 함수의 원형에 대해 아래와 같이 이용하면 된다.

```
void f( int ) __attribute__((cdecl));
```

GCC 는 또한 표준 호출(*standard call*) 규약 을 지원한다. `stdcall` 로 정의하려면 `cdecl` 을 `stdcall` 로 바꾸면 된다. `stdcall` 과 `cdecl` 의 차이 점은 `stdcall` 의 경우 서브루틴이 스택으로 부터 인자를 제거해야 된다는 점이다. (이는 Pascal 호출 규약과 유사하다) 따라서, `stdcall` 호출 규약은 오직 고정된 수의 인자를 가지는 함수들에게서만 사용될 수 있다. (*i.e* 이는 `printf` 나 `scanf` 와는 다르다)

GCC 는 또한 `regparm` 라는 attribute 를 지원하여 컴파일러로 하여금 스택을 이용하지 않고 레지스터를 통해 최대 3 개의 정수 인자를 전달하게 한다. 이러한 형식의 최적화는 많은 컴파일러들이 이용하고 있다.

볼랜드와 마이크로소프트는 호출 규약을 선언할 때 동일한 문법을 사용 한다. 이 들은 C 에 `_cdecl` 와 `_stdcall` 키워드를 추가하였다. 이 키워드들은 함수의 원형의 이름 바로 앞에 써 주면 된다. 예를 들어 위의 함수 `f` 는 볼랜드나 마이크로소프트사의 경우 아래와 같이 정의된다.

```
void __cdecl f( int );
```

각각의 호출 규약에는 장점과 단점이 있다. `cdecl` 의 가장 큰 장점으로는 이는 매우 단순하고 다루기 쉽다. 이는 어떠한 형식의 C 함수나 C 컴파일러에서 듣지 사용될 수 있다. 다른 형식의 호출 규약의 경우 서브루틴의 이식성을 떨어뜨린다. 하지만, 이 규약의 가장 큰 단점은 바로 다른 호출 규약들에 비해 느릴 수 있고, 더 많은 메모리를 차지한다는 것이다. (왜냐하면 함수가 호출 될 때마다, 코드에 저장된 인자들을 제거하는 명령을 수행해야 되기 때문이다)

`stdcall` 호출 규약의 장점은 `cdecl` 보다 메모리를 적게 소모한다는 점이다. CALL 명령 이후에 스택을 정리할 필요가 없다. 다만, 이 규약의 가장 큰 단점으로는 이 규약이 가변 개수의 인자들을 가지는 함수들에게 사용 될 수 없다는 점이다.

레지스터를 이용하는 호출 규약의 장점으로는 인자를 전달하는 속도가 매우 빠르다는 점이다. 그러나, 이러한 규약들은 좀 더 복잡하다. 인자들 중 일부는 레지스터에, 나머지는 스택에 보관되어야 하기 때문이다.

---

<sup>4</sup>Watcom C 컴파일러는 표준 호출 규약을 기본값으로 이용하지 않는 예이다. 자세한 내용은 Watcom 을 위한 예제 코드를 참조해라.

### 4.7.7 예제

다음 예제는 어떻게 어셈블리 루틴이 C 프로그램에서 작동할 수 있는지 보여준다. (이 프로그램은 그림 1.7과 같은 어셈블리 뼈대 프로그램이나, driver.c 모듈과 같은 것을 사용하지 않는다.)

---

#### main5.c

---

```

1 #include <stdio.h>
2 /* 어셈블리 루틴의 원형 */
3 void calc_sum( int, int * ) __attribute__((cdecl));
4
5 int main( void )
6 {
7     int n, sum;
8
9     printf ("Sum integers up to: ");
10    scanf("%d", &n);
11    calc_sum(n, &sum);
12    printf ("Sum is %d\n", sum);
13    return 0;
14 }
```

---

#### main5.c

---



---

#### sub5.asm

---

```

1 ; 서브 루틴 _calc_sum
2 ; 1 부터 n 까지 정수들의 합을 구한다.
3 ; 인자:
4 ;     n      - 어디까지 합할지 저장 ([ebp + 8]에 위치)
5 ;     sump   - sum 을 저장할 공간에 대한 int 포인트 ([ebp + 12]에 위치)
6 ; 의식 C 코드:
7 ; void calc_sum( int n, int * sump )
8 ;
9 ;     int i, sum = 0;
10 ;    for( i=1; i <= n; i++ )
11 ;        sum += i;
12 ;    *sump = sum;
13 ; }
14
15 segment .text
16         global _calc_sum
17 ;
```

```

Sum integers up to: 10
Stack Dump # 1
EBP = BFFFFB70 ESP = BFFFFB68
+16 BFFFFB80 080499EC
+12 BFFFFB7C BFFFFB80
+8 BFFFFB78 0000000A
+4 BFFFFB74 08048501
+0 BFFFFB70 BFFFFB88
-4 BFFFFB6C 00000000
-8 BFFFFB68 4010648C
Sum is 55

```

Figure 4.13: sub5 프로그램의 실행 예제

```

18 ; 지역 변수:
19 ; [ebp-4]에 위치한 합
20 _calc_sum:
21     enter 4,0           ; sum 을 위한 스택 공간을 할당
22     push   ebx          ; 매우 중요하다!
23
24     mov    dword [ebp-4],0 ; sum = 0
25     dump_stack 1, 2, 4   ; ebp-8에서 ebp+16 까지 스택을 출력
26     mov    ecx, 1         ; ecx 는 의사코드에서의 i
27 for_loop:
28     cmp    ecx, [ebp+8]   ; cmp i 와 n
29     jnle  end_for        ; if i <= n 가 아니면, 종료
30
31     add    [ebp-4], ecx   ; sum += i
32     inc    ecx
33     jmp    short for_loop
34
35 end_for:
36     mov    ebx, [ebp+12]   ; ebx = sum
37     mov    eax, [ebp-4]   ; eax = sum
38     mov    [ebx], eax
39
40     pop    ebx           ; restore ebx
41     leave
42     ret

```

---

왜 sub5.asm 의 22 행이 중요할까? 왜냐하면 C 호출 규약은 EBX 의

값이 바뀌면 안되기 때문이다. 만일, 이 처리를 해주지 않는다면 프로그램이 제대로 동작하지 않을 가능성이 매우 크다.

25 행은 `dump_stack` 매크로가 어떻게 작동하는지 보여주고 있다. 첫 번째 인자는 단지 정수 라벨, 두 번째와 세 번째 인자는 출력할 EBP의 범위를 설정하는 것임을 기억하자. 그럼 4.13은 프로그램을 실행한 모습을 보여준다. 이 덤프에서, 합을 저장한 곳의 주소가 BFFFFB80 (EBP + 12에 위치)이고, 어디까지 합할지 기억한 값은 0000000A (EBP + 8에 위치), 루틴의 리턴 주소는 08048501(EBP + 4에 위치), 저장된 EBP의 값은 BFFFFB88(EBP에 위치), 지역 변수의 값은 0 (EBP - 4에 위치), 마지막으로 저장된 EBX의 값은 4010648C(EBP - 8에 위치)임을 알 수 있다.

`calc_sum` 함수는 합을 저장하는 것을 포인터 인자로 받아들이는 대신에 합을 리턴 값으로 출력하게 수정할 수 있다. 합이 정수 값이기 때문에 그 합은 EAX에 저장되어야 한다. `main5.c`의 11 행은 아래와 같이 바뀐다:

```
sum = calc_sum(n);
```

또한, `calc_sum`의 원형 또한 변경되어야 한다. 아래는 수정된 어셈블리 코드이다:

---

sub6.asm

```

1 ; 서브 루틴 _calc_sum
2 ; 1 부터 n 까지 정수들의 합을 구한다.
3 ; 인자:
4 ;     n      - 어디까지 합할지 저장 ([ebp + 8]에 위치)
5 ; 리턴 값:
6 ;     합한 값
7 ; 의사 C 코드:
8 ; int calc_sum( int n )
9 ;
10 ;     int i, sum = 0;
11 ;     for( i=1; i <= n; i++ )
12 ;         sum += i;
13 ;     return sum;
14 ;
15 segment .text
16     global _calc_sum
17 ;
18 ; local variable:
19 ;     sum at [ebp-4]
20 _calc_sum:
21     enter 4,0           ; 합을 저장할 공간을 스택에 할당
22
23     mov    dword [ebp-4],0 ; sum = 0
24     mov    ecx, 1          ; ecx는 의사코드에서의 i

```

```

1 segment .data
2 format      db "%d", 0
3
4 segment .text
5 ...
6     lea    eax, [ebp-16]
7     push   eax
8     push   dword format
9     call   _scanf
10    add    esp, 8
11 ...

```

Figure 4.14: 어셈블리에서 scanf 호출하기

```

25 for_loop:
26     cmp    ecx, [ebp+8]      ; cmp i 와 n
27     jnle  end_for          ; if i <= n 가 아니면, 종료
28
29     add    [ebp-4], ecx      ; sum += i
30     inc    ecx
31     jmp    short for_loop
32
33 end_for:
34     mov    eax, [ebp-4]      ; eax = sum
35
36     leave
37     ret

```

---

sub6.asm

#### 4.7.8 어셈블리에서 C 함수를 호출하기

C 와 어셈블리를 같이 이용하는데 있어서 가장 큰 장점은 바로 어셈블리 코드를 통해서 C 라이브러리와 유저가 작성한 함수들에 접근이 가능하다는 점이다. 예를 들어, 만약 `scanf` 함수를 호출하여 키보드로부터 정수를 입력 받고 싶다고 하자. 그럼 4.14 는 위 작업을 하는 코드를 보여준다. `scanf` 는 C 호출 규약을 지킨다는 점을 잊지 말아야 한다. 이 말은 즉슨, 이는 EBX, ESI, EDI 레지스터의 값을 저장하지만, EAX, ECX, EDX 레지스터의 값은 변경 될 수 있다는 점이다. 사실, EAX 는 `scanf` 호출의 리턴 값을 저장하기 때문에 값이 반드시 바뀌게 된다. C 코드를 함께 이용하는 어셈블리 코드들의 예를 보고 싶다면 `asm_io.obj` 를 만들기 이용하였던 `asm_io.asm` 의 코드를 한 번 봄 보면 좋다.

## 4.8 재진입 및 재귀 서브프로그램

재진입 서브프로그램(reentrant)은 다음의 조건들을 반드시 만족해야 한다.

- 이는 어떠한 코드 명령도 수정하면 안된다. 고급 언어에선 조금 어렵겠지만, 어셈블리에서는 자기 자신의 코드를 수정하지 않는 일은 어렵지 않다. 예를 들어:

```
mov    word [cs:$+7], 5      ; 5 를 7 바이트 앞에 있는 워드에 복사
add    ax, 2                 ; 이전의 문장은 2에서 5로 바뀐다.
```

위 코드는 실제모드에서 작동한다. 그러나 보호모드에서는 코드 세그먼트가 오직 읽기 전용(Read-Only)이다. 따라서, 위 첫번째 행이 실행된다면 보호모드 시스템에선 프로그램이 중단될 것이다. 이러한 형식의 프로그래밍은 여러 이유에서 좋지 않다. 이는 혼란스러울 뿐더러, 유지, 보수가 어렵고, 코드 공유(code sharing)를 할 수 없다. (아래 참고)

- 이는 전역 데이터를 수정하면 안된다. (예를 들어 data나 bss 세그먼트에 들어있는 데이터) 모든 변수들은 스택에 보관된다.

재진입 코드를 쓰면 몇 가지 좋은 점들이 있다.

- 재진입 프로그램은 재귀적으로 호출 가능하다.
- 재진입 프로그램은 다수의 프로세스들에 의해 공유 가능하다. 많은 수의 멀티태스킹 운영체제에선, 하나의 프로그램에서 작동되는 여러 개의 인스턴스(instance)가 있을 때, 오직 코드의 한 부분만이 메모리에 상주한다. 공유되는 라이브러리와 DLL(동적 링크 라이브러리(Dynamic Link Libraries)) 또한 이 아이디어를 이용한다.
- 재진입 프로그램은 다중 쓰레드(multi-threaded)에서 더 잘 작동된다.<sup>5</sup> Windows 9x/NT 그리고 대부분의 UNIX 꿀의 운영체제(Solaris, Linux, etc.)가 다중 쓰레드 프로그램을 지원한다.

### 4.8.1 재귀 서브프로그램

이러한 형식의 서브프로그램들은 자기 자신을 호출한다. 재귀 호출은 직접(direct) 혹은 간접(indirect) 호출이 될 수 있다. 직접 재귀 호출은 foo라는 서브프로그램이 foo 내부에서 스스로 호출했을 때를 일컫는다. 간접

---

<sup>5</sup>다중 쓰레드 프로그램은 실행시 여러 개의 쓰레드를 가진다. 이 말은, 프로그램 자체가 멀티태스킹을 한다는 뜻이다.

```

1 ; finds n!
2 segment .text
3     global _fact
4 _fact:
5     enter 0,0
6
7     mov    eax, [ebp+8]      ; eax = n
8     cmp    eax, 1
9     jbe    term_cond        ; 만일 n <= 1 이면 종료
10    dec    eax
11    push   eax
12    call   _fact            ; eax = fact(n-1)
13    pop    ecx              ; eax 에 답 저장
14    mul    dword [ebp+8]    ; edx:eax = eax * [ebp+8]
15    jmp    short end_fact
16 term_cond:
17    mov    eax, 1
18 end_fact:
19    leave
20    ret

```

Figure 4.15: 재귀 팩토리얼 함수

재귀 호출은 서브프로그램이 자기 자신으로 부터 직접적으로 호출되지 않지만 다른 서브프로그램이 호출하기에 호출된 것을 의미한다. 예를 들면 `foo` 는 `bar` 을 호출 할 수 있고, `bar` 은 `foo` 를 호출 할 수 있는 경우 이다.

재귀 서브프로그램들은 반드시 종료 조건(*termination condition*) 이 있어야 한다. 만약 위 조건이 참이라면 더이상의 재귀 호출은 만들어 지지 않는다. 만약 재귀 루틴이 종료 조건이 없거나, 조건이 절대로 참이 될 수 없다면 이는 스스로 재귀를 무한히 반복할 것이다. (마치 무한 루프 처럼)

그림 4.15 은 재귀적으로 팩토리얼을 계산하는 함수를 보여준다. 이는 C 를 통해 아래와 같이 호출될 수 있다:

```
x = fact(3); /* find 3! */
```

그림 4.16 은 마지막으로 호출 될 때의 스택의 모습을 보여준다.

그림 4.17 와 4.18 은 각각 C 와 어셈블리를 이용한 좀 더 복잡한 예제를 보여준다. `f(3)` 의 결과는 무엇일까? ENTER 명령어 새로운 `i` 를 각 재귀 호출마다 스택에 생성한다는 점을 주목해라. 따라서, 각각의 `f` 의 재귀 인스턴스는 독립적인 변수 `i` 를 가지게 된다. 이는 `i` 를 레이터 세그먼트에 더블워드로 정의하는 것과 다르다.



Figure 4.16: 팩토리얼 함수의 스택 프레임

```

1 void f( int x )
2 {
3     int i;
4     for( i=0; i < x; i++ ) {
5         printf ("%d\n", i);
6         f(i);
7     }
8 }
```

Figure 4.17: 또 다른 예제 (C 버전)

#### 4.8.2 C 변수 저장 형식에 대해서

C 는 변수를 저장하기 위한 방법으로 몇 가지 형식들을 지원한다.

**전역(global)** 이러한 변수들은 모든 함수의 외부에서 정의되며, 고정된 메모리 위치에 저장되어 있다 (data 나 bss 세그먼트에) 또한, 프로그램 시작 부터 끝날 때 까지 계속 존재한다. 기본적으로 이들은 프로그램의 어떠한 함수에서든지 접근이 가능하다. 그러나, 이들이 static 으로 선언되어 있으면, 오직 같은 모듈에서의 함수들만이 이들에 접근 가능하다 (*i.e.* 어셈블리 형식으로 말하자면, 라벨이 외부(external) 이 아닌 내부(internal) 이다).

**정적(static)** 이들은 함수의 지역(local) 변수로, static 으로 선언된다. (불행이도, C 언어는 static 을 두 가지 용도로 사용한다!) 이러한 변수들 또한 고정된 메모리 위치에 상주한나, (data 나 bss) 오직 이들이 정의되어 있는 함수에서만 직접적으로 접근이 가능하다.

```
1 %define i ebp-4
2 %define x ebp+8           ; 유용한 매크로
3 segment .data
4 format     db "%d", 10, 0      ; 10 = '\n'
5 segment .text
6     global _f
7     extern _printf
8 _f:
9     enter 4,0          ; i 를 위한 스택 공간 할당
10
11    mov    dword [i], 0 ; i = 0
12 lp:
13    mov    eax, [i]      ; i < x 인가?
14    cmp    eax, [x]
15    jnl    quit
16
17    push   eax          ; printf 호출
18    push   format
19    call   _printf
20    add    esp, 8
21
22    push   dword [i]    ; f 호출
23    call   _f
24    pop    eax
25
26    inc    dword [i]    ; i++
27    jmp    short lp
28 quit:
29     leave
30     ret
```

Figure 4.18: 또다른 예제 (어셈블리 버전)

**자동(automatic)** 이는 C 변수가 함수 내에서 정의될 때 기본적으로 지정되는 형식이다. 이러한 변수들은 함수가 호출될 때 스택에 할당되고, 함수가 리턴될 때 스택에서 사라진다. 따라서 이들은 고정된 메모리 위치를 갖고 있지 않는다.

**레지스터(register)** 이 키워드는 컴파일러로 하여금 이 변수의 데이터를 위해 레지스터를 사용하도록 한다. 이는 단순히 요청이다. 즉, 컴파일러는 이를 반드시 지켜야 할 필요가 없다. 이 변수의 주소가 프로그램에서 사용된다면 이 요청은 무시된다 (왜냐하면 레지스터들은 주소가 없기 때문이다). 또한, 오직 단순한 정수 형식들만이 레지스터의 값이 될 수 있다. 구조형은 될 수 없다; 왜냐하면 이들은 레지스터 크기에 맞지 않기 때문이다! C 컴파일러는 종종 보통의 자동 변수들을 프로그래머에 알리지 않고 레지스터 변수로 바꾸어 주기도 한다.

**휘발성(volatile)** 이 키워드는 컴파일러에게 이 변수의 값이 어무 때나 바뀔 수 있음을 알려준다. 이 말은 컴파일러가 이 변수가 언제 수정될지에 대한 정보를 알 수 없다는 뜻이다. 종종 컴파일러는 변수의 값을 레지스터에 잠시 보관해 놓고, 변수가 사용되는 코드에서 이를 대신 사용한다. 그러나, `volatile` 형식의 변수들에겐 이러한 형태의 최적화를 할 수 없다. 가장 흔한 `volatile` 예제로는 다중 쓰레드 프로그램에서 두 개의 쓰레드에 의해 값이 변경 될 수 있는 변수이다. 아래의 코드를 고려하자

```
1 x = 10;  
2 y = 20;  
3 z = x;
```

`x` 는 다른 쓰레드에 의해 변경 될 수 있고, 1과 3 행에서 다른 쓰레드가 `x` 의 값을 변경하여 `z` 가 0이 안될 수 있다. 그러나, `x` 가 휘발성으로 선언되지 않으면 컴파일러는 `x` 가 변경되지 않았다고 생각하고 `z` 를 10 으로 바꾸어 준다.

`volatile` 은 컴파일러가 변수를 위해 레지스터를 사용하는 것을 막을 수 있다.



# Chapter 5

## 배열

### 5.1 소개

배열(array)은 메모리 상의 연속된 데이터들의 목록이라 볼 수 있다. 배열의 개개의 원소들은 같은 형이며, 메모리 상에서 같은 크기의 바이트를 사용해야 한다. 따라서 이러한 특징 때문에 배열에서의 번째 수(index)를 통해 원소에 효율적인 접근이 가능하다. 특정한 원소의 주소는 다음 세 가지 사실을 통해 계산 될 수 있다.

- 배열의 첫 번째 원소의 주소
- 원소의 바이트 크기
- 원소의 번째 수

배열의 첫 번째 원소의 인덱스를 0으로 생각하면 매우 편리하다 (C에서처럼). 첫 번째 원소의 위치를 다른 값으로도 정의할 수 있지만 이는 계산을 불편하게 한다.

#### 5.1.1 배열 정의하기

##### 데이터(data)와 bss 세그먼트에서 배열 정의하기

값들이 초기화 된 배열을 데이터(data) 세그먼트에 정의하기 위해선 db, dw, 등등 의 지시어를 사용하면 된다. NASM 은 또한 TIMES 라는 유용한 지시어를 제공하는데 이를 통해 같은 문장을 손으로 여러번 쓰는 번거러움 없이도 반복 할 수 있다.

그림 5.1 은 이러한 것들의 몇 가지 예를 보여주고 있다.

bss 세그먼트에 값들이 초기화 되지 않은 배열을 정의하기 위해선, resb, resw, 등등 의 지시어를 이용하면 된다. 이러한 지시어들은 얼마 만큼의 메모리를 지정할지 알려주는 피연산자가 있다. 그림 5.1 는 또한 이러한 형식의 정의의 예를 보여준다.

```

1 segment .data
2 ; 1,2,... 10 까지 10 개의 더블워드들로 구성된 배열을 정의
3 a1 dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
4 ; 10 개의 0 으로 워드들을로 구성된 배열 정의
5 a2 dw 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
6 ; TIMES 를 이용하여 위와 같은 작업 한다.
7 a3 times 10 dw 0
8 ; 200 개의 0 과 100 개의 1 을 구성된 바이트 배열 정의
9 a4 times 200 db 0
10 times 100 db 1
11
12 segment .bss
13 ; 10 개의 원소를 가지는 초기화 안된 배열 정의
14 a5 resd 10
15 ; 100 개의 원소를 가지는 초기화 안된 배열 정의
16 a6 resw 100

```

Figure 5.1: 배열 정의하기

### 배열을 스택에서의 지역 변수로 정의하기

스택에 직접적으로 지역 배열 변수를 정의할 방법은 없다. 이전처럼 우리는 모든 지역 변수들에 의해 필요한 메모리 바이트 수(배열 포함)을 계산하여 ESP로 부터 빼야 한다. (직접적으로 빼거나, ENTER 명령을 이용할 수 있다) 예를 들어 함수가, 1 개의 char 변수, 2 개의 더블워드 정수, 그리고 1 개의 50 개의 원소를 가지는 워드 배열을 필요로 하다면, 이는 총  $1+2\times4+50\times2=109$  바이트를 필요로하게 된다. 그러나, ESP에서 빼지는 더블워드 값은 반드시 4의 배수(위 경우 112)여야 한다. 왜냐하면 ESP를 항상 더블워드 경계(Double word boundary)에 놓이게 해야하기 때문이다. 우리는 109 바이트에 배열을 두 가지 방법으로 배열 할 수 있다. 그림 5.2는 그 방법들을 보여준다. 첫 번째 배열 방법은 더블워드가 더블워드 경계에 오도록 사용되지 않은 부분을 배열해 메모리 접근을 향상 시켰다.

#### 5.1.2 배열의 원소에 접근하기

C에서와는 달리 어셈블리에서는 [] 연산자가 없다. 배열의 원소에 접근하기 위해서는 원소의 주소가 계산되어야만 한다. 예를 들어 아래와 같은 두 개의 배열을 생각하자.

```

array1 db 5, 4, 3, 2, 1 ; 바이트 배열
array2 dw 5, 4, 3, 2, 1 ; 워드 배열

```



Figure 5.2: 스택에서의 배열

아래는 위 배열들을 사용하는 몇 가지 예 들이다.

```

1    mov    al, [array1]           ; al = array1[0]
2    mov    al, [array1 + 1]        ; al = array1[1]
3    mov    [array1 + 3], al        ; array1[3] = al
4    mov    ax, [array2]           ; ax = array2[0]
5    mov    ax, [array2 + 2]        ; ax = array2[1] (array2[2] 괄 아님!)
6    mov    [array2 + 6], ax        ; array2[3] = ax
7    mov    ax, [array2 + 1]        ; ax = ???

```

5 행에서 워드 배열의 두 번째 원소 (원소 1)에 접근하였다. 세 번째 원소 (원소 2)가 아니다. 왜냐하면, 워드는 2 바이트를 차지하기 때문에 워드 배열의 다음 원소를 지정하기 위해선 1이 아닌 2 바이트 앞으로 가야만 한다. 7 행은 첫 번째 원소의 1 바이트와 두 번째 원소의 1 바이트를 읽어 들일 것이다. C 에선, 컴파일러는 포인터의 형을 보고 얼마 만큼의 바이트를 앞으로 가야 할지 계산을 하기 때문에 프로그래머가 직접 계산 할 필요가 없다. 그러나 어셈블리에선 이 모든 것을 프로그래머가 고려 해야 한다.

그림 5.3은 array1의 원소들을 모두 더하는 코드를 보여준다. 7 행에서 AX는 DX에 더해진다. 왜 AL이 아니냐면, 먼저 ADD 명령의 두 개의 피연산자의 크기는 동일해야 한다. 두 번째로는 합이 1 바이트에 들어가기엔 너무 커지는 경우가 많다. 그러나 DX를 이용하여 최대 65,535 까지의 합을 저장할 수 있다. 그러나, AH 또한 더해지기 때문에 AH를 3 행에서 0으로 세트하였다.<sup>1</sup>

그림 5.4 와 5.5은 합을 구하는 또 다른 두 가지 방법들을 보여준다. 이탈릭체로 표시된 행은 그림 5.3의 6과 7행을 대체한 것이다.

<sup>1</sup>AH를 0으로 세팅하는 것은 AL이 부호 없는 수라고 암묵적으로 생각하는 것이다. 만일 부호가 있다면 6과 7행에 CBW 명령을 넣어야 한다.

```

1      mov    ebx, array1          ; ebx = array1 의 주소
2      mov    dx, 0               ; dx 에 합이 저장된다.
3      mov    ah, 0               ; ?
4      mov    ecx, 5
5 lp:
6      mov    al, [ebx]           ; al = *ebx
7      add    dx, ax             ; dx += ax (al 가 아님!)
8      inc    ebx               ; bx++
9      loop   lp

```

Figure 5.3: 배열의 원소들의 합 구하기 (버전 1)

```

1      mov    ebx, array1          ; ebx = array1 의 주소
2      mov    dx, 0               ; dx 에 합이 저장
3      mov    ecx, 5
4 lp:
5      add    dl, [ebx]           ; dl += *ebx
6      jnc    next              ; 캐리가 없다면 next로 분기
7      inc    dh                ; dh 1 증가
8 next:
9      inc    ebx               ; bx++
10     loop   lp

```

Figure 5.4: 배열의 원소들의 합 구하기(버전 2)

### 5.1.3 좀더 향상된 간접 주소 지정

놀랍지 않게도, 배열에선 간접 주소 지정이 종종 이용된다. 가장 널리 쓰이는 형태로는:

[ base reg + factor\*index reg + constant ]

이는

베이스 레지스터(base reg) 는 EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI 중 하나이다.

인수(factor) 는 1, 2, 4 or 8 중 하나이다. (1 이면 생략된다)

인덱스 레지스터(index reg) 는 EAX, EBX, ECX, EDX, EBP, ESI, EDI 중 하나이다. (ESP 가 없음을 유의해라)

```

1      mov     ebx, array1          ; ebx = array1 의 주소
2      mov     dx, 0               ; dx 에 합의 저장
3      mov     ecx, 5
4 lp:
5      add     dl, [ebx]          ; dl += *ebx
6      adc     dh, 0             ; dh += dl + 플래그 + 0
7      inc     ebx              ; bx++
8      loop    lp

```

Figure 5.5: 배열의 원소들의 합 구하기 (버전 3)

상수(constant)는 32비트 상수이다. 이 상수는 라벨 혹은 라벨 식(label expression)이 될 수 있다.

#### 5.1.4 예제

아래는 함수에 배열을 인자로 전달하는 예제이다. 이는 `array1c.c` 프로그램을 (아래에 나열됨) 드라이버로 사용한다. `driver.c`가 아니다.

---

array1.asm

```

1 %define ARRAY_SIZE 100
2 %define NEW_LINE 10
3
4 segment .data
5 FirstMsg      db      "First 10 elements of array", 0
6 Prompt        db      "Enter index of element to display: ", 0
7 SecondMsg     db      "Element %d is %d", NEW_LINE, 0
8 ThirdMsg      db      "Elements 20 through 29 of array", 0
9 InputFormat   db      "%d", 0
10
11 segment .bss
12 array         resd ARRAY_SIZE
13
14 segment .text
15     extern _puts, _printf, _scanf, _dump_line
16     global _asm_main
17 _asm_main:
18     enter 4,0           ; EBP - 4에 위치한 지역 더블 웨드 변수
19     push  ebx
20     push  esi
21

```

```

22 ; 배열을 100, 99, 98, 97, ... 토 정의
23
24     mov     ecx, ARRAY_SIZE
25     mov     ebx, array
26 init_loop:
27     mov     [ebx], ecx
28     add     ebx, 4
29     loop   init_loop
30
31     push    dword FirstMsg           ; FirstMsg 를 출력
32     call    _puts
33     pop    ecx
34
35     push    dword 10
36     push    dword array
37     call    _print_array          ; 배열의 첫 10 개 원소를 출력
38     add     esp, 8
39
40 ; prompt user for element index
41 Prompt_loop:
42     push    dword Prompt
43     call    _printf
44     pop    ecx
45
46     lea     eax, [ebp-4]           ; eax = 지역 더블워드의 주소
47     push    eax
48     push    dword InputFormat
49     call    _scanf
50     add     esp, 8
51     cmp     eax, 1               ; eax = scanf 의 리턴값
52     je     InputOK
53
54     call    _dump_line ; 나머지 행을 덤프 한 후 재시작 한다.
55     jmp    Prompt_loop         ; 입력값이 올바르지 않다면 분기
56
57 InputOK:
58     mov     esi, [ebp-4]
59     push    dword [array + 4*esi]
60     push    esi
61     push    dword SecondMsg      ; 원소의 값을 출력
62     call    _printf
63     add     esp, 12

```

```

64      push    dword ThirdMsg          ; 원소 20-29 의 값을 출력
65      call    _puts
66      pop     ecx
67
68      push    dword 10
69      push    dword array + 20*4      ; array[20] 의 주소
70      call    _print_array
71      add    esp, 8
72
73      pop     esi
74      pop     ebx
75      mov    eax, 0                 ; C로 리턴
76      leave
77      ret
78
79 ;
80 ;
81 ; _print_array 투 트
82 ; C호출 가능한 투 트으로, 더블워드 배열의 원소들을 부호 있는 정수로
83 ; 출력한다.
84 ; C 원형:
85 ; void print_array( const int * a, int n);
86 ; 인자:
87 ;   a - 출력할 배열에 대한 포인트(스택의 ebp+8에 위치)
88 ;   n - 출력할 정수의 개수 (스택의 ebp+12에 위치)
89
90 segment .data
91 OutputFormat db "%-5d %5d", NEW_LINE, 0
92
93 segment .text
94     global _print_array
95 _print_array:
96     enter 0,0
97     push  esi
98     push  ebx
99
100    xor   esi, esi           ; esi = 0
101    mov   ecx, [ebp+12]        ; ecx = n
102    mov   ebx, [ebp+8]         ; ebx = 배열의 주소
103 print_loop:
104     push  ecx               ; printf 는 ecx의 값을 바꾼다!
105

```

```

106      push    dword [ebx + 4*esi]      ; array[esi] 를 푸시
107      push    esi
108      push    dword OutputFormat
109      call    _printf
110      add     esp, 12                ; 인자를 제거 (ecx 만 남긴다!)
111
112      inc     esi
113      pop     ecx
114      loop   print_loop
115
116      pop     ebx
117      pop     esi
118      leave
119      ret

```

---



---

array1.asm

---

---

array1c.c

---

```

1 #include <stdio.h>
2
3 int asm_main( void );
4 void dump_line( void );
5
6 int main()
7 {
8     int ret_status ;
9     ret_status = asm_main();
10    return ret_status ;
11 }
12
13 /*
14 * 함수 dump_line
15 * 입력 버퍼로 부터 현재 행에 남은 모든 문자(char)를 덤프한다.
16 */
17 void dump_line()
18 {
19     int ch;
20
21     while( (ch = getchar()) != EOF && ch != '\n')
22         /* null body */;
23 }

```

---



---

array1c.c

---

### 다시보는 LEA 명령어

LEA 명령은 주소 계산 뿐만이 아니라 다른 목적으로도 사용될 수 있다. 대표적으로 빠른 계산을 위해서이다. 아래와 같은 경우를 보자.

```
lea    ebx, [4*eax + eax]
```

이는 효과적으로  $5 \times EAX$  를 EBX 에 저장한다. LEA 를 통해서 MUL 을 이용하는 것 보다 훨씬 빠르고 쉽게 사용할 수 있다. 그러나, 우리는 대괄호 속에 들어가 있는 식이 반드시 올바른 간접 주소 지정 형태여야 함을 명심해야 한다. 따라서, 위 명령을 통해 6 을 곱할 순 없다.

#### 5.1.5 다차원 배열

다차원 배열은 우리가 이전에 이야기 한 평범한 1 차원 배열과 크게 다를 바가 없다. 사실, 다차원 배열도 메모리 상에서는 1 차원 배열과 다를 바가 없다.

#### 2차원 배열

가장 단순한 다차원 배열은 2 차원 배열이다. 2 차원 배열은 원소들을 평면 위에 격자 형태로 그린다. 각각의 원소들은 두 개의 수를 이용하여 구별된다. 첫 번째 수는 원소가 위치한 행을 나타내고, 두 번째 수는 원소가 위치한 열을 나타낸다.

아래와 같이 3 행 2 열의 배열을 생각하자

```
int a [3][2];
```

C 컴파일러는  $6 (= 2 \times 3)$  개의 정수를 위한 공간을 할당하고, 각 원소들을 아래와 같이 나타낸다.

위치	0	1	2	3	4	5
원소	a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]

위 표가 시사하는 바는  $a[0][0]$  이라 나타낸 원소는 6 개의 원소를 가지는 1 차원 배열의 가장 처음의 원소이다.  $a[0][1]$  은 그 옆에 위치하고 나머지도 주르륵 배열된다. 2 차원 배열의 각 행은 메모리 상에 연속된 형태로 나타난다. 한 행의 마지막 원소 다음에는 그 다음 행의 첫 번째 원소가 이어서 나타나게 된다. 이러한 형식을 배열의 행 기준(rowwise) 표현이라 하며, 이는 C/C++ 컴파일러가 배열을 표현하는 방법 이기도 하다.

컴파일러는 행 기준표현에서  $a[i][j]$  가 어디에 나타날지 어떻게 결정할까? 이는  $i$  와  $j$  로 이루어진 간단한 공식을 계산하면 결정할 수 있다. 위의 경우  $2i + j$  를 계산하면 된다. 왜 이것을 계산해야 되는지 이해하는 것은 간단하다. 왜냐하면 각 행은 2 개의 원소로 구성되어 있기 때문이다.  $i$  행의 첫 번째 원소는  $2i$  위치에 된다. 또한  $j$  번째 열에 위치하게 되므로 결과적으로 원소의 위치는  $j$  를  $2i$  에 더하면 알 수 있게 된다. 이를 통해 N

---

```

1    mov    eax, [ebp - 44]      ; ebp - 44 는 i 의 위치
2    sal    eax, 1             ; i 에 2 를 곱한다.
3    add    eax, [ebp - 48]      ; j 를 더한다.
4    mov    eax, [ebp + 4*eax - 40]; ebp - 40 는 a[0][0] 의 주소 이다.
5    mov    [ebp - 52], eax      ; 결과를 x 에 저장 (ebp - 52 에 위치)

```

---

Figure 5.6:  $x = a[i][j]$  의 어셈블리 표현

개의 열을 가지는 배열의 경우에 이 공식이 어떻게 일반화 될 수 있는지를 유추할 수 있다.  $N \times i + j$ . 이 때 이 공식은 행의 개수에 상관이 없음을 주목하자.

예를 들어 *gcc* 가 아래의 코드를 어떻게 컴파일 하는지 보자. (위에서 정의한 *a* 배열을 이용하여):

$x = a[i][j];$

그림 5.6 는 이것이 어셈블리로 어떻게 나타났는지 보여준다. 따라서 컴파일러는 코드를 아래와 같이 바꾼다.

$x = *(&a[0][0] + 2*i + j);$

그리고 사실, 프로그래머가 위와 같이 써도 같은 결과가 나타난다.

행 기준 표현법에는 딱히 놀랄 만한 점은 없다. 열 기준(columnwise) 표현도 같은 방식으로 작동한다.

위치	0	1	2	3	4	5
원소	$a[0][0]$	$a[1][0]$	$a[2][0]$	$a[0][1]$	$a[1][1]$	$a[2][1]$

열 기준 표현법에선 각 열이 연속적으로 저장된다. 원소  $[i][j]$  는  $i+3j$ 에 저장된다. 다른 언어들 (FORTRAN 등) 은 이 열 기준 표현법을 이용한다. 이러한 사실을 아는 것은 다른 언어들과 함께 작업할 때 중요하다.

### 3 이상의 차원의 배열

2 보다 큰 크기의 차원을 갖는 배열들에 대해서도 같은 논리가 적용된다. 예를 들어 3 차원 배열을 생각해 보면:

**int b [4][3][2];**

이 배열은 아마 네 개의 [3][2] 배열이 연속적으로 메모리에 저장된 형태로 나타나게 될 것이다. 아래 테이블은 배열이 어떻게 나타나는지 보여준다.

위치	0	1	2	3	4	5
원소	$b[0][0][0]$	$b[0][0][1]$	$b[0][1][0]$	$b[0][1][1]$	$b[0][2][0]$	$b[0][2][1]$
위치	6	7	8	9	10	11
원소	$b[1][0][0]$	$b[1][0][1]$	$b[1][1][0]$	$b[1][1][1]$	$b[1][2][0]$	$b[1][2][1]$

$b[i][j][k]$  의 위치를 계산하는 공식은  $6i + 2j + k$  가 된다. 6 은 [3][2]

배열의 크기로 의해 결정된다. 보통  $a[L][M][N]$  으로 정의된 배열의 원소  $a[i][j][k]$  의 위치는  $M \times N \times i + N \times j + k$  이 된다. 이 때, 첫 번째 차원 ( $L$ ) 이 공식에 나타나지 않음을 주목해라.

더 높은 차원의 배열들의 경우에도 마찬가지로 위와 같은 방법을 적용할 수 있다. 예를 들어  $n$  차원 배열이  $D_1$  부터  $D_n$  까지의 차원을 가진다고 할 때,  $i_1$  부터  $i_n$  을 나타난 원소의 위치는 다음과 같은 공식으로 계산된다:

$$D_2 \times D_3 \cdots \times D_n \times i_1 + D_3 \times D_4 \cdots \times D_n \times i_2 + \cdots + D_n \times i_{n-1} + i_n$$

또는 간단히 표현하여,

$$\sum_{j=1}^n \left( \prod_{k=j+1}^n D_k \right) i_j$$

첫 번째 차원  $D_1$  은 공식에서 나타나지 않는다.

열 기준 표현에서는 일반화된 공식은 아래와 같이 나타난다.

이는 여러분들이 저자가 물리를 전공했다고 이야기 할 수 있는 이유이다.

$$i_1 + D_1 \times i_2 + \cdots + D_1 \times D_2 \times \cdots \times D_{n-2} \times i_{n-1} + D_1 \times D_2 \times \cdots \times D_{n-1} \times i_n$$

또는 간단히 표현하여

$$\sum_{j=1}^n \left( \prod_{k=1}^{j-1} D_k \right) i_j$$

이 경우 마지막 차원인  $D_n$  은 공식에서 나타나지 않는다.

### 다차원 배열을 C 에서 인자로 전달하기

다차원 배열의 행 기준 표현은 C 프로그래밍에서 직접적인 영향이 있다. 일 차원 배열의 경우 원소가 메모리 상에 어디에 있는지 계산 할 때 배열의 크기는 필요하지 않다. 그러나 다차원 배열들의 경우 이는 사실이 아니다. 이러한 배열들의 원소들에 접근하기 위해선 컴파일러는 첫 번째 차원을 제외한 나머지 모든 차원들을 알아야만 한다. 이는 다차원 배열을 인자로 가지는 함수의 원형을 만들 때 고려해야 할 부분이다. 아래의 코드는 컴파일 되지 않는다.

```
void f( int a[ ][ ] ); /* 차원에 대한 정보가 없다 */
```

그러나 아래의 코드는 컴파일 된다.

```
void f( int a[ ][2] );
```

또한 2 개의 열을 가지는 배열은 이 함수에 인자로 전달될 수 있다. 일 차원의 크기는 필요하지 않는다.<sup>2</sup>

아래와 같은 함수의 원형과 혼돈하면 안된다.

---

<sup>2</sup>물론 필요하다면 크기를 지정할 수는 있지만, 컴파일러에 의해 무시된다.

LODSB    AL = [DS:ESI] ESI = ESI ± 1	STOSB    [ES:EDI] = AL EDI = EDI ± 1
LODSW    AX = [DS:ESI] ESI = ESI ± 2	STOSW    [ES:EDI] = AX EDI = EDI ± 2
LODSD    EAX = [DS:ESI] ESI = ESI ± 4	STOSD    [ES:EDI] = EAX EDI = EDI ± 4

Figure 5.7: 문자열 읽기 쓰기 명령

```
void f( int * a[ ] );
```

이는 정수 포인터들에 대한 일차원 배열을 정의한다 (이는 이차원 배열처럼 행동하는 배열들을 만드는데 사용할 수 있다.).

좀 더 높은 차원의 배열들의 경우 나머지 차원들이 반드시 필요하나 첫 번째 차원만 예외적으로 써줄 필요가 없다. 예를 들어, 4 차원 배열 인자는 아래와 같다.

```
void f( int a[ ][4][3][2] );
```

## 5.2 배열/문자열 명령

80x86 계열의 프로세서들은 배열들에게만 적용되는 몇 가지 명령들을 지원한다. 이러한 명령들은 문자열 명령(string instruction)이라 부른다. 이들은 인덱스 레지스터(ESI, EDI)를 사용하여 작업을 실행한 후, 자동적으로 하나 또는 인덱스 레지스터 모두의 값을 1 증가 시키거나 감소시킨다. 플래그 레지스터의 방향 플래그(Direction Flag, DF)는 이 인덱스 레지스터의 값이 증가 되어야 할지 감소되어야 할지 결정한다. 이 방향 플래그의 값을 수정하는 명령은 두 개가 있다.

**CLD** 는 방향 플래그를 0 으로 한다. 이 상태에선 인덱스 레지스터들이 증가된다.

**STD** 는 방향 플래그를 1 로 한다. 이 상태에선 인덱스 레지스터들이 감소된다.

80x86 프로그래밍시 방향 플래그를 올바른 상태에 세팅해 놓지 않는 경우가 매우 자주 발생한다. 이는 코드가 대부분의 경우에 작동되나 (방향 플래그가 올바른 상태에 있을 때) 가끔씩 프로그램이 올바르게 작동되지 않는 경우가 발생하여 버그를 찾아내기 힘들게 한다.

### 5.2.1 메모리에 쓰고 읽기

단순한 문자열 명령은 메모리에 쓰거나 읽거나, 아니면 둘다 수행한다. 이는 바이트나 워드, 더블워드를 메모리에 쓰고 읽을 수 있다. 그림 5.7

```

1 segment .data
2 array1 dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
3
4 segment .bss
5 array2 resd 10
6
7 segment .text
8     cld           ; 이 부분을 잊지 말 것!
9     mov    esi, array1
10    mov    edi, array2
11    mov    ecx, 10
12 lp:
13     lodsd
14     stosd
15     loop  lp

```

Figure 5.8: 결과를 불러오고 저장한다.

은 이러한 명령을 수행하는 짧은 의사 코드가 나와있다. 이 코드에 몇 가지 주목할 부분들이 있다. 먼저 ESI 는 읽기에 사용되고, EDI 는 쓰기에 사용된다. 이는 SI 가 소스 인덱스(*Source Index*), DI 가 데스티네이션 인덱스(*Destination Index*) 의 약자로 사용되어 있음을 상기하면 편하다. 또한, 레지스터가 가지는 데이터는 고정되어 있다. (이는 AL, AX, EAX 중 하나) 마지막으로 저장 명령은 어떠한 세그먼트에 쓰기를 할지에 대해 DS 가 아닌 ES 에 의해 결정된다는 사실이다. 보호 모드 프로그래밍에서 이는 큰 문제가 아니다. 왜냐하면 오직 한 개의 데이터 세그먼트가 있고, ES 는 반드시 이를 참조하기 위해 자동적으로 초기화되어야 하기 때문이다. (DS 와 같이) 그러나, 실제 모드 프로그래밍에선 올바른 세그먼트 셀렉터 값<sup>3</sup> 으로 ES 를 초기화 하는 것이 매우 중요하다. 그림 5.8 는 이러한 명령을 사용하여 배열을 다른 것에 복사하는 예를 보여준다.

LODSx 명령과 STOSx 명령을 같이 (그림 5.8 의 13 과 14 행에서와 같이) 이용하는 경우는 매우 흔하다. 사실상, 이는 한 번의 MOVsx 문자열 명령을 통해 실행될 수 있다. 그림 5.9은 이러한 명령들이 실행하는 작업들을 설명한다. 그림 5.8 의 13 과 14 행은 같은 작업을 하는 단일 MOVSD 명령으로 바뀔 수도 있다. 유일한 차이점은 EAX 레지스터가 루프 전체에서 사용되느냐 사용되지 않느냐이다.

---

<sup>3</sup>또 다른 골치 아픈 문제로는 하나의 MOV 명령을 이용하여 DS 레지스터의 값을 ES 레지스터에 직접적으로 대입할 수 없기 때문이다. 그 대신에 DS 의 값을 범용 레지스터 (AX 등)에 대입된 후, 그 값을 다시 ES 에 대입해야만 한다. 따라서 두 번의 MOV 연산이 필요하다.

MOVSB	byte [ES:EDI] = byte [DS:ESI]
	ESI = ESI ± 1
	EDI = EDI ± 1
MOVSW	word [ES:EDI] = word [DS:ESI]
	ESI = ESI ± 2
	EDI = EDI ± 2
MOVSD	dword [ES:EDI] = dword [DS:ESI]
	ESI = ESI ± 4
	EDI = EDI ± 4

Figure 5.9: 메모리 이동 문자열 명령

```

1 segment .bss
2 array resd 10
3
4 segment .text
5     cld           ; 이를 잊지 마세요 !
6     mov    edi, array
7     mov    ecx, 10
8     xor    eax, eax
9     rep    stosd

```

Figure 5.10: 배열의 모든 원소를 0 으로 하는 예제

### 5.2.2 REP 명령 접두어

80x86 계열은 특별한 명령 접두어<sup>4</sup> REP 를 이용하여, 문자열 명령 위에 사용될 수 있다. 이 접두어는 CPU 로 하여금 특정한 횟수 동안 아래의 문자열 명령을 반복하게 한다. ECX 레지스터는 반복 횟수를 세는데 사용된다 (LOOP 명령에서처럼). REP 접두어를 이용하여 그림 5.8 ( 12 에서 15 행) 의 루프를 아래와 같이 하나의 문장으로 바꿀 수 있다.

rep movsd

그림 5.10 는 배열의 원소들을 모두 0 으로 바꾸는 또 다른 예제를 보여 준다.

---

<sup>4</sup> 명령 접두어는 명령이 아니다, 이는 문자열 명령 앞에 붙는 특별한 바이트로, 명령의 작업을 바꾸어 주는 역할을 한다. 다른 접두어들의 경우 메모리 접근에 대한 세그먼트 기본 설정들을 변경하는데 사용할 수 있다.

CMPSB	바이트 [DS:ESI] 와 바이트 [ES:EDI] 를 비교 ESI = ESI ± 1 EDI = EDI ± 1
CMPSW	워드 [DS:ESI] 와 워드 [ES:EDI] 를 비교 ESI = ESI ± 2 EDI = EDI ± 2
CMPSD	더블워드 [DS:ESI] 와 더블워드 [ES:EDI] 를 비교 ESI = ESI ± 4 EDI = EDI ± 4
SCASB	AL 과 [ES:EDI] 를 비교 EDI ± 1
SCASW	AX 와 [ES:EDI] 를 비교 EDI ± 2
SCASD	AX 와 [ES:EDI] 를 비교 EDI ± 4

Figure 5.11: 문자열 비교 명령들

### 5.2.3 비교 문자열 명령

그림 5.11 은 메모리와 다른 메모리 혹은 레지스터와 비교하는 몇 가지 새로운 문자열 명령들을 보여준다. 이들은 배열을 비교하거나 검색하는데 유용하게 사용할 수 있다. 이 명령들은 플래그 레지스터를 CMP 명령과 같이 세트할 수 있다. CMPSx 명령들은 대응되는 메모리와 비교를 수행하고 SCASx 는 특정한 값을 메모리에서 검색한다.

그림 5.12 는 더블워드 배열에서 12 를 검색하는 짧은 코드로 보여준다. 10 행에서의 SCASD 명령은 찾고자 하는 값이 찾아졌을 때에도 EDI 에 4 를 더한다. 따라서, 따라서 배열에서 12 가 보관된 주소를 구하고자 한다면 16 행에서 수행한 것처럼 EDI 에서 4 를 빼야 한다.

### 5.2.4 REPx 명령 접두어

REP- 처럼 문자열 비교 명령들과 함께 사용될 수 있는 명령 접두어들은 몇 가지 더 있다. 그림 5.13 는 두 개의 새로운 접두어와 이들의 역할을 나타냈다. REPE 와 REPZ 는 단순히 동의어 관계이다. (REPNE 와 REPNZ 의 관계도 마찬가지) 만일 반복된 문자열 비교 명령이 비교 결과 때문에 멈췄을 경우에도 인덱스 레지스터 혹은 레지스터들은 증가되고, ECX 는 감소된다. 그러나 플래그 레지스터는 반복이 종료되었을 때의 상태를 유지하고 있다. 따라서, Z 플래그를 이용하여 반복된 비교 명령이 비교 결과 때문에, 혹은 ECX 가 0 이되어서 종료되었는지 구분할 수 있게 된다.

그림 5.14 는 두 메모리 블록의 값이 같은지 확인하는 예제 코드이다. 7 행에서의 JE 는 이전의 명령에서 결과를 확인하는데 사용된다. 만일 반복

왜 반복되는 비교 문장 뒤에 ECX 의 값이 0 인지 확인하는 문장을 넣지 않았을까?

```

1 segment .bss
2 array      resd 100
3
4 segment .text
5     cld
6     mov    edi, array    ; 배열의 시작에 대한 포인터
7     mov    ecx, 100       ; 원소의 수
8     mov    eax, 12         ; 검색할 수
9 lp:
10    scasd
11    je     found
12    loop   lp
13    ; 찾지 못 할 경우 수행할 코드
14    jmp    onward
15 found:
16    sub    edi, 4          ; edi 는 이제 배열의 12 를 가리킨다.
17    ; code to perform if found
18 onward:

```

Figure 5.12: 검사 프로그램

REPE, REPZ	Z 플래그가 세트되어 있을 동안이나, 최대 ECX 만큼 시행할 때 까지 명령을 반복한다.
REPNE, REPNZ	Z 플래그가 클리어 되어 있을 동안이나, 최대 ECX 만큼 시행할 때 까지 명령을 반복한다.

Figure 5.13: REPx 명령 접두어

된 비교 명령이 두 개의 서로 다른 바이트들을 찾았기 때문에 비교 명령이 중단되었으면 Z 플래그는 계속 클리어 되어 있을 것이고 분기는 일어나지 않는다. 그러나 ECX 가 0 이 되었기에 비교가 중단되었다면 Z 플래그는 세트 되어 있으므로, equal 라벨로 분기하게 될 것이다.

### 5.2.5 예제

이 단원은 문자열 관련 작업들을 수행하는 몇 가지 함수들을 다룬다. 이들은 모두 어셈블리로 작성되어 있으며 이 함수들은 대표적인 C 라이브러리 함수들과 동일한 작업을 한다.

---

memory.asm

---

```

1 global _asm_copy, _asm_find, _asm_strlen, _asm_strcpy
2

```

```

1 segment .text
2     cld
3     mov    esi, block1      ; 첫 번째 블록의 주소
4     mov    edi, block2      ; 두 번째 블록의 주소
5     mov    ecx, size        ; 바이트로 나타낸 블록의 크기
6     repe   cmpsb          ; Z 플래그가 세트 될 동안 반복
7     je     equal           ; Z 가 세트 된다면 두 블록은 같다.
8     ; 두 블록이 같지 않다면 수행할 문장
9     jmp    onward
10    equal:
11    ; 두 블록이 같다면 수행할 문장
12    onward:

```

Figure 5.14: 메모리 블록을 비교하기

```

3 segment .text
4 ; _asm_copy 함수
5 ; 메모리 블록을 복사한다.
6 ; C원형
7 ; void asm_copy( void * dest, const void * src, unsigned sz);
8 ; 인자:
9 ;     dest - 복사될 것이 저장될 버퍼를 가리키는 포인터
10 ;    src  - 복사될 버퍼를 가리키는 포인터
11 ;    sz   - 복사 할 바이트 수
12
13 ; 다음으로 몇 가지 유용한 심볼들을 정의하였다.
14
15 %define dest [ebp+8]
16 %define src  [ebp+12]
17 %define sz   [ebp+16]
18 _asm_copy:
19     enter  0, 0
20     push    esi
21     push    edi
22
23     mov    esi, src          ; esi = 복사될 버퍼의 주소
24     mov    edi, dest         ; edi = 복사된 것이 저장될 버퍼의 주소
25     mov    ecx, sz           ; ecx = 복사 할 바이트 수
26
27     cld                      ; 방향 플래그를 초기화

```

```

28         rep     movsb          ; movsb 를 ECX 만큼 반복
29
30         pop     edi
31         pop     esi
32         leave
33         ret
34
35
36 ; _asm_find 함수
37 ; 메모리에서 특정한 바이트를 검색한다.
38 ; void * asm_find( const void * src, char target, unsigned sz);
39 ; 인자:
40 ;   src    - 검색할 버퍼를 가리키는 포인트
41 ;   target - 검색할 바이트 값
42 ;   sz     - 버퍼의 바이트 크기
43 ; 리턴값:
44 ;   만일 target 을 찾았다면 버퍼에서의 첫 번째 target 을 가리키는 포인트가
45 ;   리턴된다.
46 ;   그렇지 않으면 NULL 이 리턴된다.
47 ; 참고: target 은 바이트 값이지만 스택에 더블워드의 형태로 푸시된다.
48 ;         따라서 바이트 값은 하위 8 비트에 저장된다.
49 ;
50 %define src    [ebp+8]
51 %define target [ebp+12]
52 %define sz     [ebp+16]
53
54 _asm_find:
55     enter  0,0
56     push    edi
57
58     mov     eax, target      ; al 은 검색할 바이트 값을 보관
59     mov     edi, src
60     mov     ecx, sz
61     cld
62
63     repne  scasb          ; ECX == 0 혹은 [ES:EDI] == AL 때 까지 검색
64
65     je     found_it        ; 제로 플래그가 세트되었다면 검색에 성공
66     mov     eax, 0           ; 찾지 못했다면 NULL 포인트를 리턴
67     jmp     short quit
68 found_it:
69     mov     eax, edi

```

```

70      dec      eax          ; 찾았다면 (DI - 1) 을 리턴
71 quit:
72      pop      edi
73      leave
74      ret
75
76
77 ; _asm_strlen 함수
78 ; 문자열의 크기를 리턴
79 ; unsigned asm_strlen( const char * );
80 ; 인자:
81 ;   src - 문자열을 가리키는 포인터
82 ;   리턴값:
83 ;   문자열에서의 char 의 수 (마지막 0 은 세지 않는다) (EAX 를 리턴)
84
85 %define src [ebp + 8]
86 _asm_strlen:
87     enter    0,0
88     push     edi
89
90     mov      edi, src        ; edi = 문자열을 가리키는 포인터
91     mov      ecx, 0xFFFFFFFFh ; ECX 값으로 가능한 가장 큰 값
92     xor      al,al         ; al = 0
93     cld
94
95     repnz    scasb        ; 종료 0 을 찾는다.
96
97 ;
98 ; repnz 은 한 단계 더 실행되기 때문에, 그 값이는 FFFFFFFF - ECX,
99 ; 가 아닌 FFFFFFFF - ECX 이다.
100 ;
101    mov      eax,0xFFFFFFFh
102    sub      eax, ecx       ; 값이 = 0xFFFFFFFh - ecx
103
104    pop      edi
105    leave
106    ret
107
108 ; _asm_memcpy 함수
109 ; 문자열을 복사한다.
110 ; void asm_memcpy( char * dest, const char * src);
111 ; 인자:

```

```

112 ; dest - 복사 된 결과를 저장할 문자열을 가리키는 포인터
113 ; src - 복사 할 문자열을 가리키는 포인터
114 ;
115 %define dest [ebp + 8]
116 %define src [ebp + 12]
117 _asm_memcpy:
118     enter    0,0
119     push     esi
120     push     edi
121
122     mov      edi, dest
123     mov      esi, src
124     cld
125 cpy_loop:
126     lodsb          ; AL 을 불러오고 & si 증가
127     stosb          ; AL 을 저장하고 & di 증가
128     or     al, al      ; 조건 플래그를 세트 한다.
129     jnz   cpy_loop    ; 만일 종료 0 을 지나지 않았다면, 계속 진행
130
131     pop     edi
132     pop     esi
133     leave
134     ret

```

---

memory.asm

---



---

memex.c

---

```

1 #include <stdio.h>
2
3 #define STR_SIZE 30
4 /* prototypes */
5
6 void asm_copy( void *, const void *, unsigned ) __attribute__((cdecl));
7 void * asm_find( const void *,
8                  char target, unsigned ) __attribute__((cdecl));
9 unsigned asm_strlen( const char * ) __attribute__((cdecl));
10 void asm_memcpy( char *, const char * ) __attribute__((cdecl));
11
12 int main()
13 {
14     char st1[STR_SIZE] = "test string";
15     char st2[STR_SIZE];
16     char * st;

```

```
17  char ch;
18
19  asm_copy(st2, st1, STR_SIZE); /* 문자열의 30 문자를 복사 */
20  printf ("%s\n", st2);
21
22  printf ("Enter a char: "); /* 문자열의 바이트를 찾는다. */
23  scanf ("%c%*[^\n]", &ch);
24  st = asm_find(st2, ch, STR_SIZE);
25  if ( st )
26      printf ("Found it: %s\n", st);
27  else
28      printf ("Not found\n");
29
30  st1 [0] = 0;
31  printf ("Enter string :");
32  scanf ("%s", st1);
33  printf ("len = %u\n", asm_strlen(st1));
34
35  asm_strcpy( st2, st1); /* 문자열의 의미 있는 데이터를 복사 */
36  printf ("%s\n", st2 );
37
38  return 0;
39 }
```

---

memex.c

---



# Chapter 6

## 부동 소수점

### 6.1 부동 소수점 표현

#### 6.1.1 정수가 아닌 이진수

우리가 첫 번째 장에서 기수법에 대해 이야기 하였을 때, 오직 정수들에 대해서만 이야기 하였다. 그러나, 명백히도 십진법과 같이 다른 진법 체계에서도 정수가 아닌 수를 표현할 수 있다. 십진법에선 소수점 다음으로 나타나는 숫자들은 10에 음수승을 취하게 된다.

$$0.123 = 1 \times 10^{-1} + 2 \times 10^{-2} + 3 \times 10^{-3}$$

물론, 이진수들에 대해서도 동일하게 적용된다.

$$0.101_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0.625$$

1장에서 정수 이진수를 십진수로 바꾸는데 사용했던 방법을 동일하게 적용하여 아래와 같이 정수가 아닌 이진수를 십진수로 바꿀 수 있다.

$$110.011_2 = 4 + 2 + 0.25 + 0.125 = 6.375$$

십진수를 이진수로 바꾸는 것도 크게 어렵진 않다. 보통, 십진수를 두 개의 부분으로 나눈다: 정수 부분과 소수 부분. 정수 부분은 1장에서 다루었던 방법으로 이진수로 바꾼다. 소수 부분은 아래의 설명한 방법을 통해 이진수로 변경될 수 있다.

예를 들어서 각각의 비트를  $a, b, c, \dots$ 로 나타낸 이진수를 생각하자. 이는 아래와 같이 나타난다:

$$0.abcdef\dots$$

위 수에 2를 곱한 수는 아래와 같이 이진법으로 나타난다.

$$a.bcd\bar{e}f\dots$$

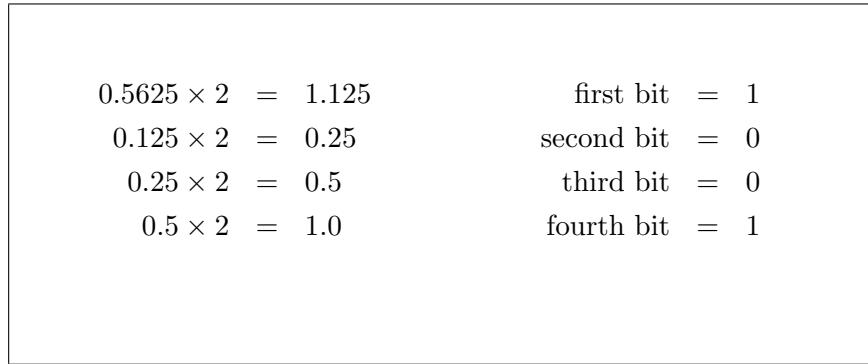


Figure 6.1: 0.5625 를 이진수로 바꾸기

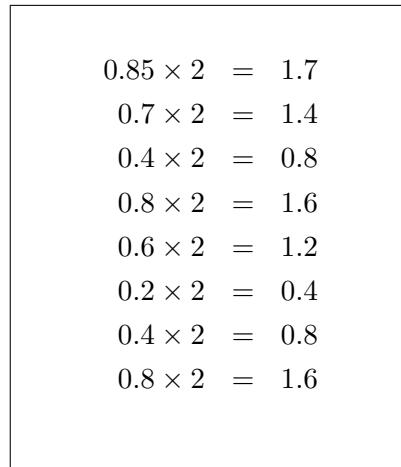


Figure 6.2: 0.85 를 이진수로 바꾸기

첫 번째 비트가 이제 1 의 자리에 있음을 유의해라.  $a$  를 0 으로 바꾼다면

$$0.bcd\ldots$$

그리고 2 를 다시 곱한다면

$$b.cdef\ldots$$

이제, 두 번째 비트 ( $b$ ) 가 1 의 자리에 오게 된다. 이 방법을 여러번 반복하여 필요한 비트 수 만큼까지 값을 찾을 수 있다. 그림 6.1 는 0.5625 를 이진수로 변환하는 예를 보여주고 있다. 이 방법은 소수 부분의 값이 0 이 될 때 중단을 한다.

또 다른 예제에선 23.85 를 이진수로 바꾸었다. 정수 부분은 이진수로 바꾸기가 매우 쉽다 ( $23 = 10111_2$ ). 하지만 소수 부분 ( $0.85$ )은 어떨까? 그림 6.2 는 소수 부분을 이진수로 바꾸는 처음 몇 부분의 과정을 보여주고 있다. 사실, 이 수를 완전히 이진수로 바꾸기 위해선 무한번의 연산이 필요하다.

이 말은 0.85 는 순환 이진 소수라는 것이다.<sup>1</sup> 위 계산에서 나타나는 숫자들은 특정한 규칙들이 있다. 이를 잘 살펴본다면  $0.85 = 0.11\overline{0110}_2$  임을 알 수 있다. 따라서,  $23.85 = 10111.11\overline{0110}_2$  이다.

따라서, 위 계산 결과를 통해 알 수 있는 점은 23.85 는 유한 개의 이진 비트를 사용하여 수를 정확히 나타낼 수 없다라는 점이다. (이는  $\frac{1}{3}$  이 십진법으로 유한개의 자리수를 가지는 소수로 정확하게 나타낼 수 없는 점과 같다) 이 장에서는 C 에서의 float 과 double 변수들이 이진수로 어떻게 저장되는지 살펴 볼 것이다. 따라서 23.85 와 같은 값들은 이 변수에 정확하게 저장될 수 없다. 오직 23.85 의 근사값 만이 변수에 저장될 것이다.

하드웨어를 단순화 하기 위해 부동 소수점 수들은 일정한 형식으로 저장된다. 이 형식은 과학적 기수법 표현 (다만 이진수에서는 10 의 막수를 사용하지 않고 2 의 막수를 사용한다) 을 사용한다. 예를 들어 23.85, 혹은  $10111.11011001100110\dots_2$  은 다음과 같이 저장된다.

$$1.01111011001100110\dots \times 2^{100}$$

(지수 (100) 도 이진수로 나타난다.). 정규화 된 부동 소수점 수는 아래의 같은 꼴을 가진다.

$$1.sssssssssssssssss \times 2^{eeeeeee}$$

이 때  $1.sssssssssssssss$  은 가수(significand) 라 하고  $eeeeeee$  은 지수(exponent) 라 한다.

### 6.1.2 IEEE 부동 소수점 표현

IEEE (전기 전자 기술자 협회, Institute of Electrical and Electronic Engineers) 는 부동 소수점 수들을 저장하기 위한 구체적인 이진 형식을 만든 국제적인 조직이다. 이 형식은 오늘날 만들어진 대부분 (모든 컴퓨터는 아니다!) 의 컴퓨터에서 사용된다. 많은 경우 이 형식은 컴퓨터 하드웨어 자체에서 지원된다. 예를 들어 인텔의 수치 보조처리기(numeric coprocessor) 이다. (이는 Pentium 부터 모든 인텔 CPU 들에게 추가되었다) IEEE 는 정밀도가 다른 두 개의 형식을 지원한다. 이는 단일 정밀도(single precision) 와 2배 정밀도(double precision) 이다. 단일 정밀도는 C 언어의 float 변수들이 사용하며 2배 정밀도는 double 변수들이 사용한다.

인텔의 수치 보조처리기는 또한 더 높은 정밀도를 가지는 확장 정밀도(extended precision)를 사용한다. 사실, 보조처리기에 있는 모든 데이터들은 이 정밀도로 표현된다. 보조처리기는 이를 메모리에 저장할 때에 단일 또는 2배 정밀도를 가지는 수들로 자동으로 변환한다.<sup>2</sup> 확장 정밀도는 IEEE

<sup>1</sup>이는 전혀 놀라운 일이 아닌데, 어떤 수는 특정한 진법을 통해 표현한다면 무한 소수가 될 수 있지만 다른 진법에서는 유한 소수가 될 수 있다. 예를 들어서  $\frac{1}{3}$  을 생각해 보면 십진법으로 표현시 3 이 무한히 반복 되지만 3 진법으로 표현한다면  $0.1_3$  과 같이 깔끔하게 나타날 수 있다.

<sup>2</sup> 일부 컴파일러 (예를 들어 Borland) 의 long double 형은 이 확장 정밀도를 이용한다. 그러나 대부분의 경우 double 과 long double 형 모두 2배 정밀도를 사용한다. (이는 ANSI C 에서 허용된다)

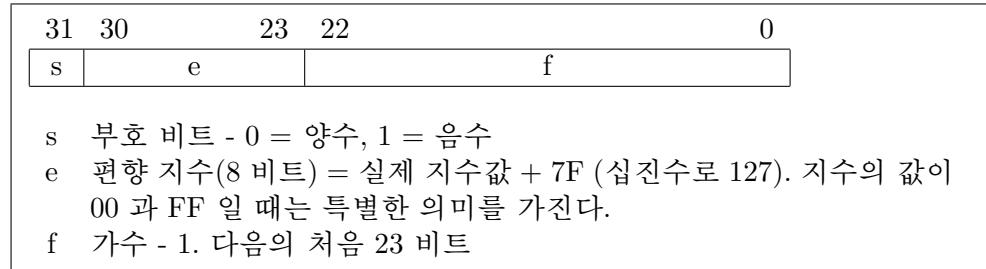


Figure 6.3: IEEE 단일 정밀도

의 float이나 double에 비해 다른 형식을 사용하며 여기서는 이야기 하지 않도록 하겠다.

### IEEE 단일 정밀도

단일 정밀도 부동 소수점 형식은 수를 32 비트로 표현한다. 이는 대체로 십진수로 7 자리 정도 정확하다. 부동 소수점 수들은 정수들에 비해 훨씬 복잡한 형식으로 저장된다. 그럼 6.3은 IEEE 단일 정밀도 수의 기본적인 형식을 보여준다. 이 형식에선 약간의 이상한 점이 있다. 부동 소수점 수들은 음수를 위해 2의 보수 표현법을 사용하지 않는다. 이들은 부호 있는 크기 표현(signed magnitude)을 사용한다. 31 째 비트는 위에 나와있듯이 수의 부호를 정한다.

이진 지수는 그 값 그대로 저장되지 않는다. 그 대신 7F를 더해서 23~30 번째 비트에 더해진다. 따라서 이 편향된 지수(biased exponent)는 언제나 음수가 아니다.

소수 부분은 언제나 정규화된 가수 부분이라 생각한다 ( $1.aaaaaaaa$  형식으로). 언제나 첫 번째 비트가 1이므로, 1은 저장되지 않는다. 이를 통해 비트 하나를 추가적으로 저장할 수 있게 되어 정확성을 향상 시킬 수 있다. 이 아이디어는 숨겨진 1 표현법(hidden one representation)이라 한다.

언제나 명심해 둘 것은 프로그램이 41 BE CC CD를 어떻게 해석하느냐에 따라 그 값이 달라질 수 있다는 사실이다! 예를 들어 단일 정밀도 부동 소수점 수라고 생각한다면 이 값은 23.850000381을 나타나게 된다. 하지만 더블 위드 정수라고 생각한다면 이는 1,103,023,309를 의미한다. CPU는 어는 것이 이 수에 대한 정확한 표현인지 알 수 없다.

그렇다면 23.85는 어떻게 저장될까? 먼저 이 수는 양수이므로 부호 비트는 0이다. 지수 값은 4이므로, 편향된 지수 값은  $7F + 4 = 83_{16}$ 가 된다. 마지막으로 가수 부분은 0111101100110011001100이다. (맨 처음의 1은 숨겨져 있다는 사실을 명심해라) 이를 모두 합치면 (참고로 이해를 돋기 위해서 부호 비트와 가수 부분은 밑줄이 쳐져 있고, 비트 들은 4비트 니브들로 나뉘어 나타냈다)

$$\underline{0} \underline{100} \underline{0001} \underline{1} \underline{011} \underline{1110} \underline{1100} \underline{1100} \underline{1100}_2 = 41BECCCC_{16}$$

이는 23.85를 정확하게 표현한 것은 아니다 (왜냐하면 이진법에서 순환 소수로 나타나기 때문). 만일 위 수를 십진수로 바꾼다면 그 값은 대략 23.849998474가 될 것이다. 이 값은 23.85와 매우 가깝지만 정확히 같지는 않다. 사실, C 에선 23.85를 위와 같이 나타내지 않는다. C 에선, 정확한 이진 표현에서 단일 정밀도 형식으로 변환 시, 마지막으로 잘라낸 비트가 1

$e = 0, f = 0$	이면 0 을 나타낸다. (이는 정규화 될 수 없다) +0 와 -0 가 있음을 유의해라.
$e = 0, f \neq 0$	는 비정규화 된 수 임을 나타낸다. 이것은 다음 단원에서 다룬다.
$e = FF, f = 0$	는 $(\infty)$ 을 나타낸다. 음의 무한과 양의 무한이 있을 수 있다.
$e = FF, f \neq 0$	$Nan$ (Not A Number) 라 부르는 정의되지 않은 값을 나타낸다.

Table 6.1:  $f$  와  $e$  의 특별한 값들

이므로 마지막 비트는 1로 반올림 된다. 따라서 23.85는 41 BE CC CD로 나타나게 된다. 이는 십진수로 바꾸면 23.850000381이므로 좀 더 23.85를 가깝게 표현할 수 있다.

-23.85는 어떻게 저장될까? 단순히 부호 비트만 바꾸면 된다. 따라서 C1 BE CC CD가 된다. 2의 보수 표현법을 취하지 않는다!

$e$ 와  $f$ 의 특정한 조합은 IEEE float에서 특별한 의미를 가진다. 표 6.1은 이러한 특별한 값을 보여준다. 무한대는 오버 플로우(over flow)나 0으로의 나눗셈에 의해 발생된다. 정의되지 않은 값(NaN)은 올바르지 못한 연산, 예를 들면 음수의 제곱근을 구한다면, 두 개의 무한대를 더할 때 등등 발생된다.

정규화된 단일 정밀도 수들은  $1.0 \times 2^{-126}$  ( $\approx 1.1755 \times 10^{-35}$ )에서  $1.11111\dots \times 2^{127}$  ( $\approx 3.4028 \times 10^{35}$ )까지의 값을 가질 수 있다.

### 비정규화 된 수

비정규화 된 수(Denormalized number)들은 너무 값이 작아서 정규화 할 수 없는 수들을 나타내기 위해 사용된다. (i.e.  $1.0 \times 2^{-126}$  미만) 예를 들어  $1.001_2 \times 2^{-129}$  ( $\approx 1.6530 \times 10^{-39}$ )라는 수를 생각하자. 이 수를 정규화하기에는 지수가 너무 작다. 그러나, 이 수는 비정규화 된 꼴:  $0.01001_2 \times 2^{-127}$ 로 나타낼 수 있다. 이 수를 저장하기 위해 편향된 지수는 0으로 맞춰 진다. (표 6.1 참조) 그리고  $2^{-127}$  와 곱해진 소수 부분은 맨 앞 자리의 비트를 포함한 완전한 수로 나타나게 된다. 따라서,  $1.001 \times 2^{-129}$ 는:

0 000 0000 0 001 0010 0000 0000 0000 0000

### IEEE 2배 정밀도

IEEE 2배 정밀도 표현은 64비트를 사용하며, 십진법으로 대략 15자리 정도 정확하게 나타낼 수 있다. 그림 6.4에서 나타나듯이 기본적인 형식은 단일 정밀도와 매우 유사하다. 다만 편향된 지수와 가수 부분이 각각 11비트와 52비트로 늘어났다.

63	62	52	51	0
s	e		f	

Figure 6.4: IEEE 2배 정밀도

편향된 지수 부분의 크기가 더 커졌으므로 두 가지 변화를 야기하게 된다. 먼저 편향된 지수 값을 구할 때 진짜 지수에 3FF(1023)을 더한다 (단일 정밀도의 7F가 아니다). 두 번째로 진짜 지수가 가질 수 있는 값의 범위가 넓어졌다. 2배 정밀도가 표현할 수 있는 수의 범위는 대략  $10^{-308}$  to  $10^{308}$  정도 된다.

또한 가수 부분이 더 커졌으므로 유효 숫자의 자리수가 더 늘어나게 되었다.

예를 들어 23.85를 다시 생각해 보자. 편향된 지수는 16진수로 4+3FF = 403가 될 것이다. 따라서 2배 정밀도 표현으로는 :

0100 0000 0011 0111 1101 1001 1001 1001 1001 1001 1001 1001 1010

혹은 40 37 D9 99 99 99 9A로 나타나게 된다. 이 수를 십진수로 변환한다면 그 값은 23.8500000000000014가 되어 (무려 12개의 0이 있다!) 23.85를 훨씬 정밀하게 표현할 수 있게 된다.

2배 정밀도도 단일 정밀도 표현과 같이 특별한 값들이 있다.<sup>3</sup> 비정규화된 수도 또한 매우 비슷하다. 유일한 차이점은 비정규화 된 수는  $2^{-127}$ 가 아닌  $2^{-1023}$ 가 된다.

## 6.2 부동 소수점 수들의 산술 연산

컴퓨터에서의 부동 소수점 수들의 산술 연산은 수들이 연속된 수학에서와는 다르다. 수학에서는 모든 수들이 정확한 값을 가졌다고 생각된다. 위에서 다루었듯이 컴퓨터가 다루는 많은 수들은 유한개의 비트를 통해 정확히 나타낼 수 없다. 모든 연산들은 제한된 정밀도를 가진다. 이 단원에서의 예제들은 단순화를 위해 8비트의 유효 숫자를 가진다고 생각한다.

### 6.2.1 덧셈

두 개의 부동 소수점 수들을 더하기 위해선 지수가 반드시 같아야 한다. 만일, 이 지수들이 같지 않다면 수의 가수를 쉬프트하여 큰 지수에 맞추어야 한다. 예를 들어  $10.375 + 6.34375 = 16.71875$ 를 이진법에서 수행한다고 하자.

$$\begin{array}{r} 1.0100110 \times 2^3 \\ + 1.1001011 \times 2^2 \\ \hline \end{array}$$

<sup>3</sup>유일한 차이점은 무한대와 NaN의 경우인데, 편향된 지수 값이 7FF가 아니라 FF가 된다.

이 두 수들은 지수값이 같지 않으므로 가수 부분을 쉬프트 하여 두 수의 지수가 같게 한 뒤 더한다.

$$\begin{array}{r} 1.0100110 \times 2^3 \\ + 0.1100110 \times 2^3 \\ \hline 10.0001100 \times 2^3 \end{array}$$

이 때  $1.1001011 \times 2^2$ 에 쉬프트 연산을 취하면서 마지막 비트를 없애고 반올림을 하여  $0.1100110 \times 2^3$ 가 되었음을 유의하자. 덧셈의 결과는  $10.0001100 \times 2^3$  (혹은  $1.00001100 \times 2^4$ )로  $10000.110_2$  혹은 16.75와 같이 같다. 이는 정확한 계산 결과인 16.71875와 같지 않다! 이는 단지 덧셈 과정에서 반올림에 의해 오차가 나타난 근사값에 불과하다.

컴퓨터 (혹은 계산기)에서의 부동 소수점 연산은 언제나 근사임을 명심해야 한다. 수학에서의 법칙들은 컴퓨터에서의 부동 소수점들에 적용될 수 없다. 수학은 언제나 컴퓨터가 절대로 만들어 낼 수 없는 무한대의 정밀도를 가정한다. 예를 들어서 수학에선  $(a+b)-b=a$ 라 말하지만 컴퓨터에서는 이 식이 성립하지 않을 수 있다!

### 6.2.2 뺄셈

뺄셈은 덧셈과 매우 유사하며, 같은 문제를 가지고 있다. 예를 들어서  $16.75 - 15.9375 = 0.8125$ 를 생각해보자.

$$\begin{array}{r} 1.0000110 \times 2^4 \\ - 1.1111111 \times 2^3 \\ \hline \end{array}$$

$1.1111111 \times 2^3$ 를 쉬프트 하면  $1.0000000 \times 2^4$  (반올림 됨)

$$\begin{array}{r} 1.0000110 \times 2^4 \\ - 1.0000000 \times 2^4 \\ \hline 0.0000110 \times 2^4 \end{array}$$

$0.0000110 \times 2^4 = 0.11_2 = 0.75$ 이므로 원 답과 일치하지 않는다.

### 6.2.3 곱셈과 나눗셈

곱셈의 경우 유효 숫자 부분은 곱해지고, 지수 부분은 더해 진다. 예를 들어  $10.375 \times 2.5 = 25.9375$ 을 생각해 보면

$$\begin{array}{r} 1.0100110 \times 2^3 \\ \times 1.0100000 \times 2^1 \\ \hline 10100110 \\ + 10100110 \\ \hline 1.10011111000000 \times 2^4 \end{array}$$

당연하게도 실제 결과는 8 비트로 반올림 되어

$$1.1010000 \times 2^4 = 11010.000_2 = 26$$

나눗셈은 좀 더 복잡하지만 반올림으로 인하여 같은 문제를 가진다.

#### 6.2.4 프로그래밍을 위한 조언

이 단원의 핵심적인 내용은 부동 소수점 연산이 정확하지 않다는 것이다. 프로그래머는 언제나 이를 고려해야 한다. 프로그래머들이 가장 많이 범하는 실수로는 연산이 정확하다고 가정 하에 부동 소수점 수들을 비교하는 것이다. 예를 들어  $f(x)$ 라는 이름의 함수를 고려하자. 이 함수는 복잡한 연산을 수행한다. 우리의 프로그램의 목적은 이 함수의 근을 찾는 것이다.<sup>4</sup> 여러분은 아마  $x$ 가 근인지 확인하기 위해 아래의 식을 이용할 것이다.

```
if (  $f(x) == 0.0$  )
```

그러나 만일  $f(x)$  가  $1 \times 10^{-30}$  을 리턴하면? 이는 진짜 근의 훌륭한 근사 값이다. 그러나, 위 식은 성립하지 않는다. 아마도  $f(x)$ 에서의 반올림으로 인한 오차로 인해 어떠한 IEEE 부동 소수점 형식으로 표현된  $x$ 값도 정확히 0을 리턴하지 않을 것이다.

위 보다 훨씬 낳은 방법은 아래와 같다.

```
if (  $\text{fabs}(f(x)) < \text{EPS}$  )
```

$\text{EPS}$  는 매우 작은 양수 값인 매크로로 정의되어 있다. (예를 들면  $1 \times 10^{-10}$ ) 이는  $f(x)$  가 0에 매우 가까울 때 성립이 된다. 통상적으로 두 개의 부동 소수점 값 ( $x$ ) 을 다른 ( $y$ ) 값에 비교 할 때 아래와 같이 사용한다:

```
if (  $\text{fabs}(x - y)/\text{fabs}(y) < \text{EPS}$  )
```

### 6.3 수치 부프로세서

#### 6.3.1 하드웨어

가장 초기의 인텔 프로세서들은 부동 소수점 연산을 지원하는 하드웨어가 없었다. 이 말은 부동 소수점 연산을 할 수 없다는 것이 아니다. 이는 단지 부동 소수점 수들의 연산을 위해 많은 수의 비-부동 소수점 명령들을 수행해야 했다는 뜻이다. 이러한 초기의 시스템에선 인텔은 부수적인 수치 부프로세서(*math coprocessor*)를 제공하였다. 이 수치 부프로세서는 부동 소수점 수들의 연산을 위한 소프트웨어 상의 프로시저의 비해 훨씬 빠른 속도로 명령을 수행하였다 (초기의 프로세서의 경우 대략 10 배 이상!). 8086/8088을 위한 부프로세서는 8087 이라 불리었다. 80286 에선 80287이, 80386 에선 80387 이 부프로세서 였다. 80486DX 프로세서에선 80486에 수치

---

<sup>4</sup>함수의 근이라 하면  $f(x) = 0$  을 만족하는 값  $x$  를 말한다.

부프로세서가 통합되었다.<sup>5</sup> 펜티엄 이후 80x86 세대의 모든 프로세서들에  
게는 수치 부프로세서가 통합되었다. 그러나, 독립적으로 떨어져 있었을 때  
처럼 프로그램 된다. 심지어 부프로세서가 없던 초기의 시스템들은 수학 부  
프로세서를 에뮬레이트 할 수 있는 소프트웨어가 있었다. 이 에뮬레이터는  
프로그램이 부프로세서 명령을 실행하면 자동적으로 활성화되어 소프트  
웨어 프로시저를 통해 부프로세서가 내놓았을 같은 값을 내놓았다. (비록  
매우 느렸지만)

수치 부프로세서는 8 개의 부동 소수점 레지스터가 있다. 각각의 레지  
스터는 80 비트의 데이터를 보관한다. 부동 소수점 수들은 이 레지스터들에  
언제나 80 비트 확장 정밀도로 저장된다. 이 레지스터들은 각각 ST0, ST1,  
ST2, ..., ST7 로 이름 붙여졌다. 부동 소수점 레지스터들은 주 CPU에서의  
정수 레지스터들과는 달리 다른 용도로 사용된다. 부동 소수점 레지스터들은  
스택(stack)으로 구성되어 있다. 스택이 후입 선출 (LIFO) 리스트임을  
상기하자. ST0 는 언제나 스택의 최상단의 값을 가리킨다. 모든 새로운 수  
들은 스택의 최상단에 들어간다. 기존의 수들은 새로운 수를 위한 공간을  
만들기 위해 스택 한 칸 내려간다.

수치 부프로세서에는 상태 레지스터(status register) 가 있다. 이는 몇  
가지의 플래그를 가진다. 비교 연산에서는 오직 4 개의 플래그들이 사용된  
다: C<sub>0</sub>, C<sub>1</sub>, C<sub>2</sub>, C<sub>3</sub>. 이것들의 사용은 나중에 다룰 것이다.

### 6.3.2 명령

보통의 CPU 명령들과 부프로세서의 명령들을 쉽게 구별하기 위해 모든  
부프로세서의 연상기호들은 F 로 시작한다.

#### 데이터를 불러오고 저장하기

부프로세서의 레지스터 스택의 최상위 데이터에 값을 저장하는 몇 가지  
연산들이 있다.

**FLD source** 는 메모리에서 부동 소수점 값을 불러와 스택 최상단에 저장  
한다. *source* 는 단일 혹은 2배 정밀도 값, 아니면 부프로세  
서의 레지스터 여야 한다.

**FILD source** 는 정수를 메모리로부터 읽어들어, 부동 소수점 형식으로 변  
환한 뒤 그 결과를 스택 최상단에 저장한다. *source* 는 워드,  
더블워드, 쿼드워드 중 하나여야 한다.

**FLD1** 1 을 스택 최상단에 저장한다.

**FLDZ** 0 을 스택 최상단에 저장한다.

스택으로 부터의 데이터를 메모리에 저장하는 명령들도 몇 가지 있다.  
이러한 명령들의 일부는 스택에서 수를 팝(pop) (*i.e.* 제거하다) 한 후에 그  
값을 저장한다.

---

<sup>5</sup>그러나 80486SX 에는 통합된 부프로세서가 없었다. 이를 위해 독립적인 80487SX 칩이  
있었다.

<b>FST dest</b>	스택의 최상단(ST0) 을 메모리에 저장한다. <i>dest</i> 은 단일 혹은 2배 정밀도 수 이거나 부프로세서 레지스터가 될 수 있다.
<b>FSTP dest</b>	은 스택의 최상단 값을 FST 와 같이 메모리에 저장한다. 그러나 수가 저장된 다음에는 그 값은 스택에서 팝 된다. <i>dest</i> 은 단일 혹은 2배 정밀도 수 이거나 부프로세서 레지스터가 될 수 있다.
<b>FIST dest</b>	는 스택의 최상단의 값을 정수로 변환한 뒤 메모리에 저장 한다. <i>dest</i> 는 워드 혹은 더블워드가 되어야만 한다. 스택 그 자체로는 바뀌지 않는다. 부동 소수점 수가 정수로 어떻게 변환되느냐에 따라 달라진다. 이는 특별한 (비-부동 소수점) 워드 레지스터로 부프로세서가 어떻게 작동할지 제어한다. 기본적으로 제어 워드는 초기화되어 있는데 이 때에는 수에 가장 가까운 정수로 변환되어 진다. 그러나, FSTCW (저장 제어 워드 Store Control Word) 와 FLDCW (불러오기 제어 워드, Load Control Word) 명령들을 통해 어떻게 변환 할지를 바꿀 수 있다.
<b>FISTP dest</b>	FIST 와 두 가지 빼고 동일하다. 하나는, 스택의 최상단이 팝 된다는 것이고, 다른 하나는 <i>dest</i> 가 큐드워드도 될 수 있다는 것이다.

스택 자체에서 데이터를 이동, 혹은 제거할 수 있는 명령이 두 개가 있다.

<b>FXCH ST<sub>n</sub></b>	스택에서의 ST0 와 ST <sub>n</sub> 的 값을 바꾼다. (이 때, <i>n</i> 은 1 부터 7 까지의 값을 가질 수 있다.)
<b>FFREE ST<sub>n</sub></b>	레지스터가 비었거나 미사용중임을 표시하여 스택에서의 레지스터의 값을 비운다.

### 덧셈과 뺄셈

각각의 덧셈 명령들은 ST0 과 다른 피연산자의 합을 계산한다. 그 결과는 언제나 부프로세서 레지스터에 저장된다.

<b>FADD src</b>	ST0 += src. src 는 어떤 부프로세서 레지스터이거나 메모리 상의 단일 혹은 2배 정밀도 수여도 상관 없다.
<b>FADD dest, ST0</b>	dest += ST0. dest는 임의의 부프로세서 레지스터이다.
<b>FADDP dest 또는 FADDP dest, ST0</b>	dest += ST0 하고 팝 한다. dest 는 임의의 부프로세서 레지스터이다.
<b>FIADD src</b>	ST0 += (float) src. 정수를 ST0 에 더한다. src 는 메모리 상의 워드 혹은 더블워드 값이다.

두 개의 피연산자에 대해 덧셈 명령은 한 가지 이지만, 뺄셈 명령은 2 가지의 서로 다른 뺄셈 명령이 존재할 수 있다. 왜냐하면 피연산자의 순서가 뺄셈에서는 중요하기 때문이다. (*i.e.*  $a + b = b + a$ , 하지만  $a - b \neq b - a$ )

```

1 segment .bss
2     array      resq SIZE
3     sum       resq 1
4
5 segment .text
6     mov      ecx, SIZE
7     mov      esi, array
8     fldz          ; ST0 = 0
9 lp:
10    fadd     qword [esi]   ; ST0 += *(esi)
11    add      esi, 8       ; 다음 double 투 이동
12    loop     lp
13    fstp     qword sum    ; sum 에 결과를 저장

```

Figure 6.5: 배열의 원소 합 구하는 예제

은 아니다!) 각 명령에 대해 뺄셈을 역으로 수행하는 명령이 있다. 이 역명령들은 끝에 언제나 R이나 RP가 붙는다. 그럼 6.5는 배열의 double형 원소들을 모두 더하는 코드를 보여주고 있다. 10에서 13행까지 우리는 메모리 피연산의 크기를 확실히 지정해야 한다. 그렇지 않는다면 어셈블러는 메모리 피연산자가 float(더블워드)인지 double(쿼드워드)인지를 알 수 없기 때문이다.

FSUB <i>src</i>	<i>ST0 -= src</i> . <i>src</i> 는 임의의 부프로세서 레지스터거나 메모리 상의 단일 혹은 2배 정밀도 수 이면 된다.
FSUBR <i>src</i>	<i>ST0 = src - ST0</i> . <i>src</i> 는 임의의 부프로세서 레지스터거나 메모리 상의 단일 혹은 2배 정밀도 수 이면 된다.
FSUB <i>dest</i> , <i>ST0</i>	<i>dest -= ST0</i> . <i>dest</i> 는 임의의 부프로세서 레지스터이다.
FSUBR <i>dest</i> , <i>ST0</i>	<i>dest = ST0 - dest</i> . <i>dest</i> 는 임의의 부프로세서 레지스터이다.
FSUBP <i>dest</i> 혹은 FSUBP <i>dest</i> , <i>ST0</i>	<i>dest -= ST0</i> 한 후 팝 한다. <i>dest</i> 는 임의의 부프로세서 레지스터이다.
FSUBRP <i>dest</i> or FSUBRP <i>dest</i> , <i>ST0</i>	<i>dest = ST0 - dest</i> 한 후 팝 한다. <i>dest</i> 는 임의의 부프로세서 레지스터이다.
FISUB <i>src</i>	<i>ST0 -= (float) src</i> . <i>ST0</i> 에서 정수를 뺀다. <i>src</i> 는 반드시 메모리 상의 워드 혹은 더블워드여야 한다.
FISUBR <i>src</i>	<i>ST0 = (float) src - ST0</i> . <i>ST0</i> 에서 정수를 뺀다. <i>src</i> 는 반드시 메모리 상의 워드 혹은 더블워드여야 한다.

## 곱셈과 나눗셈

곱셉 명령은 덧셈 명령과 완벽히 동일하다.

FMUL <i>src</i>	<i>ST0 *= src. src</i> 는 임의의 부프로세서 레지스터거나 메모리 상의 단일 혹은 2배 정밀도 수 이면 된다.
FMUL <i>dest</i> , ST0	<i>dest *= ST0.dest</i> 는 임의의 부프로세서 레지스터이다.
FMULP <i>dest</i> 혹은 FMULP <i>dest</i> , ST0	<i>dest *= ST0</i> 한 후 팝 한다. <i>dest</i> 는 임의의 부프로세서 레지스터이다.
FIMUL <i>src</i>	<i>ST0 *= (float) src. ST0</i> 에 정수를 곱한다. <i>src</i> 는 메모리 상의 워드 혹은 더블워드 여야 한다.

놀랍지 않게도 나눗셈 명령은 뺄셈 명령과 거의 동일하다. 다만 0 으로 나눈다면 무한대가 된다.

FDIV <i>src</i>	<i>ST0 /= src. src</i> 는 임의의 부프로세서 레지스터거나 메모리 상의 단일 혹은 2배 정밀도 수 이면 된다.
FDIVR <i>src</i>	<i>ST0 = src / ST0. src</i> 는 임의의 부프로세서 레지스터거나 메모리 상의 단일 혹은 2배 정밀도 수 이면 된다.
FDIV <i>dest</i> , ST0	<i>dest /= ST0. dest</i> 는 임의의 부프로세서 레지스터이다.
FDIVR <i>dest</i> , ST0	<i>dest = ST0 / dest. dest</i> 는 임의의 부프로세서 레지스터이다.
FDIVP <i>dest</i> 혹은 FDIVP <i>dest</i> , ST0	<i>dest /= ST0</i> 한 후 팝 한다. <i>dest</i> 는 임의의 부프로세서 레지스터이다.
FDIVRP <i>dest</i> 혹은 FDIVRP <i>dest</i> , ST0	<i>dest = ST0 / dest</i> 한 후 팝한다. <i>dest</i> 는 임의의 부프로세서 레지스터이다.
FIDIV <i>src</i>	<i>ST0 /= (float) src. ST0</i> 를 정수로 나눈다. <i>src</i> 는 메모리 상의 워드 혹은 더블워드 여야 한다.
FIDIVR <i>src</i>	<i>ST0 = (float) src / ST0. 정수를 ST0 로 나눈다.</i> <i>src</i> 는 메모리 상의 워드 혹은 더블워드 여야 한다.

## 비교

부프로세서는 부동 소수점 수들에 대한 비교 명령도 수행한다. FCOM 계열의 명령들이 이 작업을 한다.

```

1 ;      if ( x > y )
2 ;
3     fld    qword [x]          ; ST0 = x
4     fcomp qword [y]          ; ST0 와 y 를 비교
5     fstsw ax                ; 플래그 레지스터에 C 비트를 이동
6     sahf
7     jna    else_part         ; x>y 가 아니면 else_part 로 분기
then_part:
8     ; then 을 위한 부분
9     jmp    end_if
else_part:
10    ; else 를 위한 부분
11
12
13 end_if:

```

Figure 6.6: 비교 예제

**FCOM src** ST0 와 src 를 비교. src 는 임의의 부프로세서 레지스터거나 메모리 상의 단일 혹은 2배 정밀도 수 이면 된다.

**FCOMP src** ST0 와 src 를 비교한 후 팝 한다. src 는 임의의 부프로세서 레지스터거나 메모리 상의 단일 혹은 2배 정밀도 수 이면 된다.

**FCOMPP** ST0 와 ST1 를 비교 한 후 팝을 두 번 한다.

**FICOM src** ST0 와 (float) src 를 비교한다. src 는 메모리 상의 워드 혹은 더블워드 정수이다.

**FICOMP src** ST0 와 (float) src 를 비교 한 후 팝 한다. src 는 메모리 상의 워드 혹은 더블워드 정수이다.

**FTST** ST0 와 0 을 비교한다.

이 명령들은 부프로세서 상태 레지스터의  $C_0, C_1, C_2, C_3$  비트들을 바꾼다. 불행이도 CPU 가 직접 이러한 비트에 접근하는 것은 불가능 하다. 조건 분기 명령은 부프로세서의 상태 레지스터가 아닌 플래그 레지스터를 사용한다. 그러나 상태 워드의 비트들을 이에 대응되는 플래그 레지스터의 비트에 옮기는 것은 아래의 몇 가지 새로운 명령을 이용한다면 쉽다

**FSTSW dest** 부프로세서의 상태 워드를 메모리의 워드나 AX 레지스터에 저장한다.

**SAHF** AH 레지스터를 플래그 레지스터에 저장한다.

**LAHF** 플래그 레지스터의 비트들과 함께 AH 를 불러온다.

그림 6.6 는 코드 예제를 보여준다. 5 행과 6 행은 부프로세서의 상태 워드의  $C_0, C_1, C_2, C_3$  비트들을 플래그 레지스터에 대입한다. 전달한 비트들은 두 개의 부호가 없는 정수의 비교 결과와 동일한다. 이는 우리가 7 행에서 JNA 를 사용한 이유이다.

```
1 global _dmax
2
3 segment .text
4 ; function _dmax
5 ; 두 개의 double 인자 들 중 큰 것을 리턴한다.
6 ; C 원형
7 ; double dmax( double d1, double d2 )
8 ; 인자들 :
9 ;   d1 - 첫 번째 double
10 ;   d2 - 두 번째 double
11 ; 리턴 값 :
12 ;   d1 과 d2 중 큰 것 (ST0에 대입하여)
13 %define d1    ebp+8
14 %define d2    ebp+16
15 _dmax:
16     enter 0, 0
17
18     fld    qword [d2]
19     fld    qword [d1]          ; ST0 = d1, ST1 = d2
20     fcomip st1               ; ST0 = d2
21     jna    short d2_bigger
22     fcomp   st0              ; d2 를 팝 한다.
23     fld    qword [d1]          ; ST0 = d1
24     jmp    short exit
25 d2_bigger:                   ; d2 가 더 크면 아무 것도 안 함
26 exit:
27     leave
28     ret
```

Figure 6.7: FCOMIP 예제

```

1 segment .data
2     x          dq  2.75           ; double 형식으로 변환
3     five       dw   5
4
5 segment .text
6     fild    dword [five]        ; ST0 = 5
7     fld     qword [x]          ; ST0 = 2.75, ST1 = 5
8     fscale                         ; ST0 = 2.75 * 32, ST1 = 5

```

Figure 6.8: FSCALE 예제

펜티엄 프로(그리고 그 이후의 프로세서들 (펜티엄 II, III)) 는 두 개의 새로운 비교 명령을 지원하는데 이는 CPU 의 플래그 레지스터의 값을 직접적으로 변경할 수 있다.

**FCOMI** *src* ST0 와 *src* 를 비교한다. *src* 는 반드시 부프로세서 레지스터 여야 한다.

**FCOMIP** *src* ST0 와 *src* 를 비교한 후 팝 한다. *src* 는 반드시 부프로세서 레지스터 여야 한다.

그림 6.7 은 두 개의 double 값 중 큰 것을 찾는 FCOMIP 를 사용한 예제 서브루틴을 보여주고 있다. 이 명령들을 정수 비교 함수들과 혼동하지 말라 (FICOM 과 FICOMP).

### 잡다한 명령들

이 단원은 부프로세서가 제공하는 여러 잡다한 명령들에 대해 다루어 보도록 한다.

**FCHS** ST0 = - ST0, ST0 의 부호를 바꾼다.

**FABS** ST0 = |ST0|, ST0 의 절대값을 취한다.

**FSQRT** ST0 =  $\sqrt{ST0}$ , ST0 의 제곱근을 구한다.

**FSCALE** ST0 =  $ST0 \times 2^{[ST1]}$ , ST0 에 2 의 몇수를 빠르게 곱한다. ST1 는 스택에서 제거되지 않는다. 그림 6.8 는 이 명령을 어떻게 사용하는지에 대한 예제를 보여준다.

### 6.3.3 예제

#### 6.3.4 2차 방정식의 근의 공식

우리의 첫 번째 예제는 2차 방정식의 근의 공식을 어셈블리로 나타낸 것이다.

$$ax^2 + bx + c = 0$$

근의 공식을 통해 두 개의 근을 구할 수 있다.  $x: x_1$  와  $x_2$ .

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

제곱근 안의 식 ( $b^2 - 4ac$ ) 은 판별식(discriminant) 라 부른다. 이 값은 근이 어떤 성질을 띠는지 판별하는데 유용하게 쓰인다.

1. 실수인 중근을 가진다.  $b^2 - 4ac = 0$
2. 두 개의 실수근을 가진다.  $b^2 - 4ac > 0$
3. 두 개의 복소근을 가진다.  $b^2 - 4ac < 0$

아래 어셈블리 서브루틴을 사용하는 작은 C 프로그램을 나타냈다.

---

quadt.c

---

```

1 #include <stdio.h>
2
3 int quadratic( double, double, double, double *, double * );
4
5 int main()
6 {
7     double a,b,c, root1, root2;
8
9     printf ("Enter a, b, c: " );
10    scanf ("%lf %lf %lf", &a, &b, &c);
11    if ( quadratic( a, b, c, &root1, &root2 ) )
12        printf (" roots: %.10g %.10g\n", root1, root2 );
13    else
14        printf (" No real roots\n");
15    return 0;
16 }
```

---

quadt.c

---

여기 어셈블리 서브루틴이 있다:

---

quad.asm

---

```

1 ; 함수 quadratic
2 ; 2차 방정식의 근을 찾는다.
3 ;      a*x^2 + b*x + c = 0
4 ; C 원형:
5 ;      int quadratic( double a, double b, double c,
```

```

6 ;           double * root1, double *root2 )
7 ; 인자:
8 ;   a, b, c - 2차 방정식의 각 계수들 (위를 참조)
9 ;   root1 - 첫 번째 근을 저장할 double 을 가리키는 포인터
10;  root2 - 두 번째 근을 저장할 double 을 가리키는 포인터
11; 리턴값:
12;  실근을 찾으면 1 을 리턴, 아니면 0 리턴
13
14 %define a          qword [ebp+8]
15 %define b          qword [ebp+16]
16 %define c          qword [ebp+24]
17 %define root1       dword [ebp+32]
18 %define root2       dword [ebp+36]
19 %define disc        qword [ebp-8]
20 %define one_over_2a qword [ebp-16]
21
22 segment .data
23 MinusFour      dw     -4
24
25 segment .text
26     global _quadratic
27 _quadratic:
28     push    ebp
29     mov     ebp, esp
30     sub     esp, 16      ; 2 개의 더블을 할당한다 (disc 와 one_over_2a)
31     push    ebx         ; 원래의 ebx 값을 저장
32
33     fild   word [MinusFour]; 스택 -4
34     fld    a             ; 스택: a, -4
35     fld    c             ; 스택: c, a, -4
36     fmulp st1            ; 스택: a*c, -4
37     fmulp st1            ; 스택: -4*a*c
38     fld    b             ; 스택: b, b, -4*a*c
39     fld    b             ; 스택: b*b, -4*a*c
40     fmulp st1            ; 스택: b*b - 4*a*c
41     faddp st1            ; 스택: 0 과 비교
42     ftst
43     fstsw ax
44     sahf
45     jb    no_real_solutions ; 만일 disc < 0, 이면 해가 없다.
46     fsqrt
47     fstp    disc          ; 저장 후 스택을 팝 한다.

```

```

48      fld1          ; 스택: 1.0
49      fld    a        ; 스택: a, 1.0
50      fscale         ; 스택: a * 2^(1.0) = 2*a, 1
51      fdivp st1       ; 스택: 1/(2*a)
52      fst    one_over_2a ; 스택: 1/(2*a)
53      fld    b        ; 스택: b, 1/(2*a)
54      fld    disc      ; 스택: disc, b, 1/(2*a)
55      fsubrp st1       ; 스택: disc - b, 1/(2*a)
56      fmulp st1       ; 스택: (-b + disc)/(2*a)
57      mov    ebx, root1
58      fstp   qword [ebx] ; *root1에 저장
59      fld    b        ; 스택: b
60      fld    disc      ; 스택: disc, b
61      fchs           ; 스택: -disc, b
62      fsubrp st1       ; 스택: -disc - b
63      fmul   one_over_2a ; 스택: (-b - disc)/(2*a)
64      mov    ebx, root2
65      fstp   qword [ebx] ; *root2에 저장
66      mov    eax, 1       ; 리턴값은 1
67      jmp    short quit
68
69 no_real_solutions:
70      mov    eax, 0       ; 리턴값은 0
71
72 quit:
73      pop    ebx
74      mov    esp, ebp
75      pop    ebp
76      ret

```

---

### 6.3.5 파일로 부터 배열을 읽기

이번 예제에서는 어셈블리가 파일로 부터 double 을 읽어들인다. 아래는 짧은 C 테스트 프로그램이다.

---

readt.c

---

```

1  /*
2  * 이 프로그램은 32비트 read_doubles() 어셈블리 프로시저를 테스트 한다.
3  * 이는 stdin 으로 부터 double 을 읽어 들인다. (파일로 부터 읽기 위해
4  * 리다이렉션(redirection)을 사용한다.)
5  */

```

```

6 #include <stdio.h>
7 extern int read_doubles( FILE *, double *, int );
8 #define MAX 100
9
10 int main()
11 {
12     int i, n;
13     double a[MAX];
14
15     n = read_doubles( stdin, a, MAX );
16
17     for( i=0; i < n; i++ )
18         printf( "%3d %g\n", i, a[i] );
19     return 0;
20 }
```

---

readt.c

---

아래는 어셈블리 루틴이다.

---

read.asm

---

```

1 segment .data
2 format db      "%lf", 0           ; format for fscanf()
3
4 segment .text
5     global _read_doubles
6     extern _fscanf
7
8 %define SIZEOF_DOUBLE    8
9 %define FP                dword [ebp + 8]
10 %define ARRAYP            dword [ebp + 12]
11 %define ARRAY_SIZE        dword [ebp + 16]
12 %define TEMP_DOUBLE       [ebp - 8]
13
14 ;
15 ; _read_doubles 함수
16 ; C 원형:
17 ;     int read_doubles( FILE * fp, double * arrayp, int array_size );
18 ; 이 함수는 텍스트 파일로 부터 double 을 읽어 들여서 EOF 가 나올 때 까지 나
19 ; 배열이 꽉 찰 때 까지 배열에 저장한다.
20 ; 인자:
21 ;     fp          - 읽어 들일 FILE 포인터 (읽기 가능해야 한다)
22 ;     arrayp      - 읽어 들인 값을 저장할 double 배열을 가리키는 포인터
```

```

23 ; array_size - 배열의 원소의 개수
24 ; 리턴:
25 ; 배열에 저장된 double 의 개수 (EAX 에 저장됨)
26
27 _read_doubles:
28     push    ebp
29     mov     ebp,esp
30     sub     esp, SIZEOF_DOUBLE      ; 스택에 하나의 double 을 정의
31
32     push    esi                  ; esi 저장
33     mov     esi, ARRAYP          ; esi = ARRAYP
34     xor     edx, edx            ; edx = 배열 원소 위치 (처음이 0)
35
36 while_loop:
37     cmp     edx, ARRAY_SIZE      ; edx < ARRAY_SIZE 인가?
38     jnl     short quit          ; 아니면 루프 루프 종료
39 ;
40 ; TEMP_DOUBLE 를 double 값을 읽어 들이기 위해 fscanf() 를 호출
41 ; fscanf() 는 edx 값을 박끌 수 있으므로 저장해 놓는다.
42 ;
43     push    edx                  ; edx 저장
44     lea     eax, TEMP_DOUBLE
45     push    eax                  ; &TEMP_DOUBLE 푸시
46     push    dword format        ; &format 푸시
47     push    FP                  ; 파일 포인터 푸시
48     call    _fscanf
49     add    esp, 12
50     pop    edx                  ; edx 복원
51     cmp    eax, 1                ; fscanf 가 1 을 리턴했는가?
52     jne    short quit          ; 아니면 루프 종료
53 ;
54 ; TEMP_DOUBLE 를 ARRAYP[edx] 에 대입
55 ; (8 바이트의 double 은 두 개의 4 바이트 테이스터를 통해 대입된다.)
56 ;
57     mov    eax, [ebp - 8]
58     mov    [esi + 8*edx], eax      ; 하위 4 바이트를 복사
59     mov    eax, [ebp - 4]
60     mov    [esi + 8*edx + 4], eax ; 상위 4 바이트를 복사
61
62     inc    edx
63     jmp    while_loop
64

```

```

65 quit:
66     pop    esi          ; esi 를 복원한다.
67
68     mov    eax, edx      ; 리턴값을 eax 에 저장
69
70     mov    esp, ebp
71     pop    ebp
72     ret
----- read.asm -----

```

### 6.3.6 소수 찾기

마지막 예제는 소수를 찾는 프로그램이다. 이미 앞에서 한 번 다루어 보았지만 이번 것은 좀더 속도가 향상되었다. 이번 프로그램은 앞에서 찾은 소수를 배열에 저장해 놓고 특정한 수가 소수 인지를 판별하기 위해 특정한 수 이전의 모든 짝수로 나누어 보는 것이 아니라 배열에 저장해 놓은 소수들만으로 나누어 본다.

또다른 차이점은 특정한 수가 소수 인지를 판별하기 위해 그 수의 제곱근 이하의 소수들만으로 나누어 본다는 것이다. 이 때문에 부프로세서의 제어 워드를 변경하여 제곱근을 정수로 저장할 때 반올림 하기 보단 내림을 해야한다. 이는 제어 워드의 10과 11 번째 비트를 변경하면 된다. 이 비트들은 RC (반올림 제어, Rounding Control) 비트로 불린다. 만일 두 비트 모두 0이라면 (기본값) 부프로세서는 정수로 변환시에 반올림을 한다. 두 비트가 모두 1이라면 부프로세서는 정수로 변환시 내림을 한다. 유의해야 할 점은 기존의 제어 워드를 저장하고 리턴하기 전에 복원해야 한다는 것이다.

아래는 C 드라이버 프로그램이다

---

### fprime.c

---

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 /*
4  * find_primes 함수
5  * 지정된 개수의 소수들을 찾는다
6  * 인자들:
7  *   a - 배열을 보관할 배열
8  *   n - 찾을 소수의 개수
9  */
10 extern void find_primes( int * a, unsigned n );
11
12 int main()
13 {
14     int status;

```

```

15     unsigned i;
16     unsigned max;
17     int * a;
18
19     printf ("How many primes do you wish to find? ");
20     scanf("%u", &max);
21
22     a = calloc( sizeof(int), max);
23
24     if ( a ) {
25
26         find_primes (a,max);
27
28         /*찾은 마지막 20 개의 소수를 출력 */
29         for(i = ( max > 20 ) ? max - 20 : 0; i < max; i++ )
30             printf ("%3d %d\n", i+1, a[i]);
31
32         free(a);
33         status = 0;
34     }
35     else {
36         fprintf ( stderr , "Can not create array of %u ints\n", max);
37         status = 1;
38     }
39
40     return status ;
41 }
```

---

fprime.c

---

아래는 어셈블리 루틴이다.

---

```

1     segment .text
2         global _find_primes
3 ;
4 ; find_primes 함수
5 ; 짜정한 수 만큼의 소수를 찾는다
6 ; 인자:
7 ;   array - 소수를 보관할 배열
8 ;   n_find - 찾을 소수의 개수
9 ; C 원형:
10 ;extern void find_primes( int * array, unsigned n_find )
```

---

```

11   ;
12 %define array          ebp + 8
13 %define n_find         ebp + 12
14 %define n              ebp - 4           ; 현재까지 찾은 소수의 개수
15 %define isqrt          ebp - 8           ; guess의 제곱근의 내림한 값
16 %define orig_cntl_wd  ebp - 10          ; 원래 제어 워드 값
17 %define new_cntl_wd   ebp - 12          ; 새로운 제어 워드 값
18
19 _find_primes:
20     enter 12,0           ; 지역 변수를 위한 공간을 할당
21
22     push  ebx            ; 가능한 테이스터 변수들을 저장
23     push  esi
24
25     fstcw   word [orig_cntl_wd]      ; 현재의 제어 워드를 저장
26     mov      ax, [orig_cntl_wd]
27     or       ax, 0C00h             ; 반올림 비트를 11로 한다. (내림)
28     mov      [new_cntl_wd], ax
29     fldcw   word [new_cntl_wd]
30
31     mov      esi, [array]        ; esi는 배열을 가리킨다.
32     mov      dword [esi], 2      ; array[0] = 2
33     mov      dword [esi + 4], 3    ; array[1] = 3
34     mov      ebx, 5             ; ebx = guess = 5
35     mov      dword [n], 2        ; n = 2
36 ;
37 ; 이 박깥 루프는 각 루프마다 새로운 소수를 하나씩 찾는다. 이 새로운 소수는
38 ; 배열의 끝 부분에 추가된다. 기존의 소수 찾는 루프 그램들과는 달리 이 함수는
39 ; 소수 판별을 위해 그 수 미만의 모든 홀수들로 나누어 보지 않는다.
40 ; 오직 이미 찾은 소수들 만으로 나누어 본다. (이 때문에 이 소수들을 배열에
41 ; 저장한 것이다.)
42 ;
43 while_limit:
44     mov      eax, [n]
45     cmp      eax, [n_find]        ; while ( n < n_find )
46     jnb      short quit_limit
47
48     mov      ecx, 1             ; ecx는 원소의 위치를 가리키는데 사용
49     push    ebx               ; guess를 스택에 저장
50     fild    dword [esp]        ; guess를 부프로세서 스택에 불러온다
51     pop      ebx               ; 스택에서 guess를 뺀다.
52     fsqrt   guess             ; sqrt(guess) 값을 구한다.

```

```

53         fistp    dword [isqrt]           ; isqrt = floor(sqrt(guess))
54 ;
55 ; 이 내부의 루프는 guess (ebx)를 이전에 찾은 소수들로 나누어서
56 ; guess 의 소수 인수를 찾을 때 까지 혹은
57 ; 소수 인수가 floor(sqrt(guess)) 보다 클 때 까지 나눈다.
58 ;(floor 은 내림 함수)
59 while_factor:
60     mov      eax, dword [esi + 4*ecx]       ; eax = array[ecx]
61     cmp      eax, [isqrt]                   ; while ( isqrt < array[ecx]
62     jnbe    short quit_factor_prime
63     mov      eax, ebx
64     xor      edx, edx
65     div      dword [esi + 4*ecx]
66     or       edx, edx                    ; && guess % array[ecx] != 0 )
67     jz       short quit_factor_not_prime
68     inc      ecx                      ; 다음 소수로 시도 한다
69     jmp      short while_factor
70
71 ;
72 ; 새 소수를 찾음 !
73 ;
74 quit_factor_prime:
75     mov      eax, [n]
76     mov      dword [esi + 4*eax], ebx      ; guess 를 배열의 끝 부분에 추가
77     inc      eax
78     mov      [n], eax                     ; inc n
79
80 quit_factor_not_prime:
81     add      ebx, 2                      ; 다음 풀 수를 시도해 본다
82     jmp      short while_limit
83
84 quit_limit:
85
86     fldcw   word [orig_cntl_wd]          ; 제어 워드를 복원
87     pop     esi
88     pop     ebx                         ; 레지스터 변수들을 복원
89
90     leave
91     ret

```

---

prime2.asm

# Chapter 7

## 구조체와 C++

### 7.1 구조체

#### 7.1.1 서론

C에서 사용되는 구조체는 연관된 데이터를 하나의 변수에 보관하는 것이라 말할 수 있다. 이는 몇 가지 장점이 있다.

1. 구조체에 정의된 데이터가 연관되어 있음을 보임으로써 코드를 명료하게 할 수 있다.
2. 이를 통해 함수로의 데이터 전달을 단순하게 할 수 있다. 여러 개의 변수를 독립적으로 전달하는 대신에 이를 이용해 하나의 단위만 전달하면 된다.
3. 이는 코드의 지역성(*locality*)<sup>1</sup>을 향상시킨다.

어셈블리의 관점에서 볼 때 구조체는 원소들의 크기가 제각각인 배열로 볼 수 있다. 실제 배열의 원소들의 크기는 언제나 같은 형이자, 같은 크기여야 한다. 이를 통해 실제 배열에서는 배열의 시작 주소와 원소의 크기, 원소의 번째 수만 알면 원소의 주소를 계산할 수 있게 된다.

그러나 구조체의 원소들은 동일한 크기를 가질 필요가 없다. (그리고 대부분 동일한 크기가 아니다) 이 때문에 구조체의 각각의 원소들은 뚜렷이 지정되어 있어야 하고 수치적인 위치 대신에 태그(tag) (또는 이름)를 가진다.

어셈블리에서 구조체의 원소에 접근하는 것은 배열의 원소의 접근하는 것과 비슷하다. 원소에 접근하기 위해서는 먼저 구조체의 시작 주소를 알아야 하고, 구조체의 시작 부분으로부터의 각 원소의 상대 오프셋(relative offset)을 알아야 한다. 그러나, 배열의 경우 이 오프셋이 원소의 번째 수만 알고도 계산될 수 있었지만, 구조체의 원소들은 컴파일러에 의해 오프셋이 지정된다.

---

<sup>1</sup>운영체제에 관련한 책의 가장 메모리 관리 단원을 살펴보라.

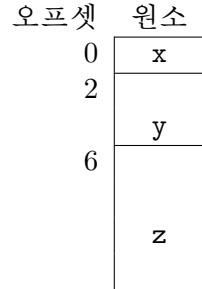


Figure 7.1: 구조체 S

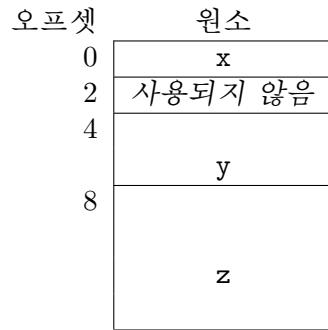


Figure 7.2: 구조체 S

예를 들어 아래의 구조체를 보자.

```
struct S {
    short int x;      /* 2 바이트 정수 */
    int y;          /* 4 바이트 정수 */
    double z;        /* 8 바이트 float */
};
```

그림 7.1 은 S 형의 변수가 컴퓨터 메모리 상에 어떻게 나타나는지 보여준다. ANSI C 표준에 따르면 메모리 상에 배열된 구조체의 원소들의 순서는 **struct** 정의에서 나타난 순서와 동일해야 한다고 하였다. 또한 첫 번째 원소는 구조체의 최상단에 위치해야 한다고 하였다. (*i.e* 오프셋이 0) 이는 또한 stddef.h 헤더 파일에 **offsetof()** 라는 매우 유용한 매크로를 정의하였는데 이 매크로는 구조체의 특정한 원소의 오프셋 값을 계산한 후 리턴한다. 이 매크로는 두 개의 인자를 가지는데 첫 번째는 구조체의 이름이고 두 번째는 오프셋 값을 계산할 원소의 이름이다. 따라서 그림 7.1 에서의 **offsetof(S,y)** 의 결과는 2 가 된다.

```
struct S {
    short int x;      /* 2 바이트 정수 */
    int y;          /* 4 바이트 정수 */
    double z;        /* 8 바이트 float */
} __attribute__((packed));
```

Figure 7.3: 압축된 *gcc*에서의 구조체 사용

### 7.1.2 메모리 배열하기

*gcc* 컴파일러를 사용시, `offsetof` 매크로를 이용하여 `y`의 오프셋을 구한다면 2가 아닌 4를 리턴하게 된다. 왜냐하면 *gcc*(그리고 다수의 컴파일러들이)는 변수들을 기본적으로 더블워드 경계에 놓기 때문이다. 32비트보호 모드에서 CPU는 메모리 상의 데이터가 더블워드 경계에 놓여 있을 때 더 빨리 읽어들일 수 있다. 그림 7.2는 `S` 구조체가 *gcc*를 사용했을 때 메모리 상에 어떻게 나타날지 보여준다. 컴파일러는 구조체에 사용되지 않는 2바이트를 끼워 넣어서 `y`(그리고 `z`도)를 더블워드 경계에 놓이게 하였다. 이는 `offsetof`를 사용하여 오프셋을 계산하는 것이 정의된 구조체의 형태만을 보고 값을 계산하는 것 보다 훨씬 좋은 것인지를 알 수 있다.

주소가 4로 나누어 진다면 주소는 더블워드 경계에 놓여 있다는 점을 상기해라

당연하게도 구조체가 오직 어셈블리에서만 사용된다면 프로그래머는 스스로 오프셋을 지정할 수 있다. 그러나 어셈블리와 C를 함께 사용한다면 어셈블리 코드와 C 코드의 구조체의 원소들의 오프셋이 반드시 일치해야 한다. 한 가지 문제가 되는 점은 C 컴파일러마다 원소에 대해 다른 오프셋 값을 줄 수 있다는 것이다. 예를 들어 위에서 보았듯이 *gcc* 컴파일러는 `S` 구조체를 그림 7.2와 같은 모습으로 만들지만, 볼랜드 사의 컴파일러는 구조체를 그림 7.1과 같은 모습으로 만든다. C 컴파일러들은 데이터를 정렬하기 위해 여러가지 방법을 사용한다. 그러나 ANSI C 표준은 그 어떤 방법도 구체화 하지 않았으므로 컴파일러들이 저마다 다른 방법들을 사용하게 되었다.

*gcc* 컴파일러는 데이터의 정렬 방법을 구체화 하기 위해 유연하고도 복잡한 방법을 지원한다. 이 컴파일러에선 특별한 문법을 사용함으로써 어떠한 형이라도 데이터를 정렬하는 방법을 지정할 수 있게 하였다. 예를 들어 아래의 행을 보면

```
typedef short int unaligned_int __attribute__((aligned(1)));
```

이는 `unaligned_int`라는 새로운 형을 정의하는데 이는 바이트 경계에 맞추어져 있다. (`__attribute__` 다음의 괄호들이 꼭 필요하다) `aligned` 인자 1은 2의 몇수들로 바뀔 수 있으며 이는 다른 정렬 조건을 나타낸다. (2는 워드 정렬, 4는 더블워드 정렬 등등) 만일 구조체의 `y` 원소가 `unaligned_int` 형이라면 *gcc*는 `y`를 오프셋 2에 놓을 것이다. 그러나, `z`는 오프셋 8에 놓이게 되는데 왜냐하면 `double` 형들은 기본값이 더블워드 정렬이기 때문이다. 물론 `z`를 오프셋 6에 놓이게 할 수 있도록 변경할 수 있다.

```
#pragma pack(push) /* 이전 정렬 상태를 저장한다*/
#pragma pack(1)    /* 바이트 기준 정렬로 정의 */

struct S {
    short int x;    /* 2 바이트 정수 */
    int      y;    /* 4 바이트 정수 */
    double   z;    /* 8 바이트 float */
};

#pragma pack(pop) /* 이전의 정렬 상태를 복원한다*/
```

Figure 7.4: 마이크로소프트나 볼랜드 사의 압축된 구조체

*gcc* 컴파일러는 또한 구조체를 압축(*pack*) 하는 것을 지원한다. 이는 컴파일러로 하여금 구조체 생성시 최소한의 공간만을 사용하게 한다. 그림 7.3은 S 가 이 방법으로 어떻게 다시 쓰일 수 있는지 보여준다. 이러한 형태의 S 는 최소한의 바이트를 사용하여, 이 경우 14 바이트를 사용한다.

마이크로소프트와 볼랜드 사의 컴파일러는 `#pragma` 지시어를 통해 위와 동일한 작업을 할 수 있다.

### `#pragma pack(1)`

이 지시어는 컴파일러가 바이트 경계에 맞추어 구조체의 원소들을 압축하게 한다. (*i.e* 추가적인 사용 없이) 1 은 2, 4, 8, 16 등으로 변경될 수 있는데 이는 각각 워드, 더블워드, 쿼드워드, 패리그래프 경계에 맞추라는 뜻이다. 이 지시어는 다른 지시어의 의해 변경되기 전 까지 계속 효력을 발휘한다. 이 때문에 이와 같은 지시어가 많이 사용되는 헤더 파일에서는 문제가 발생하기도 한다. 만일 이 헤더 파일이 다른 구조체를 포함한 헤더파일 보다 먼저 포함(include) 되었다면 이 구조체는 기본값과 다른 방식으로 메모리 상에 배열 될 수 있다. 이와 같은 오류는 찾기가 매우 힘들다. 프로그램의 각기 다른 모듈들이 구조체의 원소들을 각기 다른 장소에 배열하게 된다.

이와 같은 문제를 피하는 방법이 있다. 마이크로소프트와 볼랜드 사는 현재의 정렬 방법을 저장한 후 나중에 복원할 수 있게 한다. 그림 7.4 는 이를 나타낸다.

### 7.1.3 비트필드

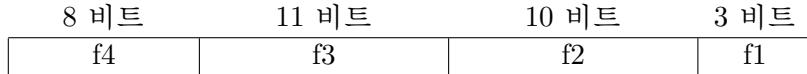
비트필드는 구조체의 멤버들로 하여금 특정한 크기의 비트만 사용할 수 있게 한다. 비트의 크기가 꼭 8 의 배수가 되야할 필요는 없다. 비트필드 멤버는 `unsigned int` 나 `int` 멤버에 콜론(:)와 비트 크기를 적음으로써 정의된다. 그림 7.5 는 이 예를 보여준다. 이는 32 비트 변수를 다음과 같은 부분으로 나뉘어 정의한다.

```
struct S {
    unsigned f1 : 3; /*3 비트필드*/
    unsigned f2 : 10; /*10 비트필드*/
    unsigned f3 : 11; /* 11 비트필드 */
    unsigned f4 : 8; /* 8 비트필드 */
};
```

Figure 7.5: 비트필드 예제

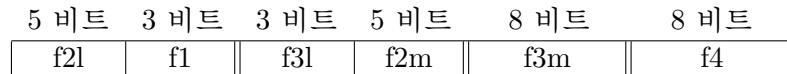
바이트 \ 비트	7	6	5	4	3	2	1	0							
0	연산 부호 (08h)														
1	논리 단위 #	논리 블록 주소의 최상위 비트													
2	논리 블록 주소의 중간														
3	논리 블록 주소의 최하위 바이트														
4	전달 크기														
5	제어														

Figure 7.6: SCSI 읽기 명령 형식



첫 번째 비트필드는 더블워드의 최하위 비트들을 가리킨다.<sup>2</sup>

그러나 우리가 이 비트들이 실제로 메모리 상에 어떻게 나타나는지 본다면 이 형식은 그리 간단한 것이 아니다. 이 문제는 비트필드가 바이트 경계에 걸쳐 있을 때 나타난다. 왜냐하면 리틀 엔디안 형식에서 바이트는 메모리에 거꾸로 저장되기 때문이다. 예를 들어 S 구조체의 비트필드는 메모리에 아래와 같이 나타난다.



$f2l$  라벨은  $f2$  비트필드의 하위 5 비트를 의미한다 (*i.e* LSB).  $f2m$  라벨은  $f2$  비트필드의 상위 5 비트를 가리킨다. 수직선은 바이트 경계를 표시한다. 만일 모든 바이트들을 거꾸로 배치한다면  $f2$  와  $f3$  의 조각들이 올바른 위치에 합쳐질 것이다.

메모리에서의 배열 형태는 데이터가 프로그램 바깥으로 전송되지 않는 한 중요하지 않다. (사실 이러한 경우는 비트필드를 다룰 때 잘 발생한다) 하드웨어 장치 인터페이스를 만들 때, 홀수의 비트를 가지는 비트필드를 사용하는 경우가 잦은데 이는 비트필드를 사용하면 매우 유용하기 때문이다.

<sup>2</sup> 사실 ANSI/ISO C 표준은 컴파일러로 하여금 비트들이 어떻게 나열되는지는 정확히 정하지 않았다. 그러나 대부분의 C 컴파일러들 (*gcc*, 마이크로소프트, 블랜드 사의 컴파일러)들은 이와 같은 모습으로 배열한다.

```

1 #define MS_OR_BORLAND (defined(__BORLANDC__) \
2                                || defined(_MSC_VER))
3
4 #if MS_OR_BORLAND
5 # pragma pack(push)
6 # pragma pack(1)
7 #endif
8
9 struct SCSI_read_cmd {
10     unsigned opcode : 8;
11     unsigned lba_msb : 5;
12     unsigned logical_unit : 3;
13     unsigned lba_mid : 8; /* 중간 비트들*/
14     unsigned lba_lsb : 8;
15     unsigned transfer_length : 8;
16     unsigned control : 8;
17 }
18 #if defined(__GNUC__)
19     __attribute__((packed))
20 #endif
21 ;
22
23 #if MS_OR_BORLAND
24 # pragma pack(pop)
25 #endif

```

Figure 7.7: SCSI 읽기 명령 형식 구조체

그 예로는 SCSI<sup>3</sup> 를 들 수 있다. SCSI 장치에 직접적으로 읽기 명령을 내리는 방법은 그림 7.6 에 나타난 형식대로 6 바이트의 메세지를 장치에 전달하면 된다. 이를 비트 필드로 이용하여 표현하는데 어려움은 논리 블록 주소(*logical block address*)에 있는데, 왜냐하면 이것이 3 개의 다른 바이트에 걸쳐 있기 때문이다. 그림 7.6 에는 빅 엔디안 형식으로 저장된 데이터를 볼 수 있다. 그림 7.7 에는 모든 컴파일러에 대해서 사용 가능하게 한 정의를 보여준다. 첫 2 행은 매크로를 정의하는데 이는 마이크로소프트나 블랜드 사의 컴파일러로 컴파일 될 때 참이 된다. 아마 혼동되는 부분은 11에서 14 행 까지 일 것이다. 먼저 우리는 lba\_mid 와 lba\_lsb 필드가 하나의 16 비트 필드로 정의되지 않고 따로 정의됐는지 의문을 가질 것이다. 이는 데이터가 빅 엔디안 순서로 되어 있기 때문이다. 16 비트 필드는 컴파일러에 의해 리틀

---

<sup>3</sup>소형 컴퓨터 시스템 인터페이스 (Small Computer Systems Interface) 의 약어로 하드 디스크 등등의 장치에 대한 규격이다.

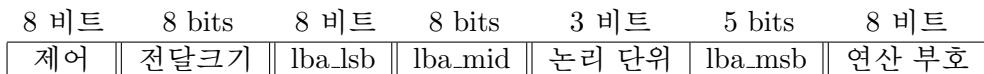


Figure 7.8: SCSI\_read\_cmd 필드의 모습

```

1 struct SCSI_read_cmd {
2     unsigned char opcode;
3     unsigned char lba_msb : 5;
4     unsigned char logical_unit : 3;
5     unsigned char lba_mid; /*중간 비트들 */
6     unsigned char lba_lsb;
7     unsigned char transfer_length;
8     unsigned char control;
9 }
10 #if defined(__GNUC__)
11     __attribute__((packed))
12 #endif
13 ;

```

Figure 7.9: 다른 형태의 SCSI 읽기 명령 형식 구조체

엔디안 순서로 저장될 것이다. 두 번째로 lba\_msb 와 logical\_unit 는 거꾸로 된 것처럼 보인다. 그러나 이 경우엔 적용되지 않는다. 이들은 이 순서대로 집어 네져야 한다. 그림 7.8 은 48 비트필드가 메모리에 어떻게 표현되는지 보여준다. (바이트 경계는 두 개의 선으로 표현되었다.) 이것이 메모리에 리틀 엔디안 순서로 저장된다면 비트들은 원하는 형식으로 배열될 것이다. (그림 7.6).

일을 더 복잡하게 하는 것은, SCSI\_read\_cmd 은 마이크로소프트 C 에서 완벽하게 작동하지 않는다는 점이다. **sizeof(SCSI\_read\_cmd)** 라는 식을 계산하였을 때, 마이크로소프트 C 에서는 6 이 아닌 8 을 리턴한다. 왜냐하면 마이크로소프트 컴파일러는 비트를 메모리 나타낼 때 비트필드의 형을 기준으로 나타내기 때문이다. 모든 비트필드들이 **unsigned** 로 정의되어 있으므로 컴파일러는 구조체 맨 마지막에 2 바이트를 덧붙여서 더블워드의 배수로 만들어 준다. 이와 같은 현상은 모든 필드를 **unsigned short** 로 정의해 줌으로써 해결할 수 있다. 그렇게 한다면 마이크로소프트는 6 바이트가 2 바이트 워드의 정수배 이므로 부수적인 바이트를 추가할 필요가 없어지게 된다.<sup>4</sup> 다른 컴파일러는 이와 같이 바꾸어도 정확하게 동작한다. 그림 7.9 는 3 개 컴파일러 모두에서 잘 작동하는 정의를 보여준다. **unsigned char** 를 이용하여 오직 두 개만 비트필드를 이용했다.

이전에 우리가 이야기 하였던 것이 혼란스러워도 걱정할 필요가 없다.

<sup>4</sup>여러가지 다른 형의 비트필드를 사용하는 것은 매우 혼란스러운 .

왜냐하면 정말로 혼란스러운 내용이니까 말이다. 저자는 덜 혼란스럽기 위해서 비트필드를 사용하지 않고 비트를 각각 다루기 위해서 비트 연산을 사용한다.

#### 7.1.4 어셈블리에서 구조체 사용하기

위에서 이야기 하였듯이 어셈블리에서 구조체를 사용하는 것은 배열을 사용하는 것과 비슷한다. 간단한 예로 구조체 S의 원소 y를 0으로 만드는 루틴을 어떻게 만들지 생각해 보자. 이 루틴의 원형은 아래와 같다.

```
void zero_y( S * s_p );
```

어셈블리 루틴은

---

```

1 %define      y_offset 4
2 _zero_y:
3     enter 0,0
4     mov    eax, [ebp + 8]      ; 스택에서 s_p(구조체 포인터)를 불러온다
5     mov    dword [eax + y_offset], 0
6     leave
7     ret

```

---

C는 함수에게 구조체를 값으로 전달하는 것을 허용한다. 그러나 이는 좋지 않은 생각이다. 값으로 전달을 할 경우, 구조체의 모든 데이터들이 스택에 복사된 후 루틴에게 전달될 것이다. 단순히 구조체를 가리키는 포인터를 전달하는 것이 더 효과적이라 할 수 있다.

C는 또한 구조체 타입이 함수의 리턴값으로도 사용할 수 있게 한다. 하지만 명백히도, 구조체는 EAX 레지스터에 저장되어 리턴될 수 없다. 이 상황을 여러 컴파일러들이 각기 다른 방법으로 해결 한다. 가장 많이 쓰이는 방법은 컴파일러가 내부적으로 함수를 구조체를 가리키는 포인터를 인자로 갖게 변경하는 것이다. 이 포인터는 루틴이 호출된 곳 밖에서 정의된 구조체에 반환값을 대입하는데 사용된다.

대부분의 어셈블러 (NASM 포함)은 여러분의 어셈블리 코드에서 구조체를 정의하는 것을 지원하는 자체적인 방법이 있다. 자세한 내용은 이에 관련한 문서들을 참조하여라.

## 7.2 어셈블리와 C++

C++ 프로그래밍 언어는 C 언어의 확장이라 볼 수 있다. 어셈블리와 C가 서로 작업하는 방법 대부분이 C++에서도 동일하게 적용된다. 그러나, 몇 가지 규칙들은 수정되어야 한다. 또한, C++의 몇 가지 확장된 부분들은 어셈블리로 이해하기가 편하다. 이 단원은 독자 여러분이 C++의 기본적인 이해가 되어있다고 생각한다.

```

1 #include <stdio.h>
2
3 void f( int x )
4 {
5     printf ("%d\n", x);
6 }
7
8 void f( double x )
9 {
10    printf ("%g\n", x);
11 }
```

Figure 7.10: 두 개의 f() 함수

### 7.2.1 오버로딩과 이름 맹글링(Name Mangling)

C++은 같은 이름으로 서로 다른 함수 (그리고 클래스 멤버 함수)를 정의하는 것을 허용한다. 2개 이상의 함수가 동일한 이름을 사용한다면, 이 함수들을 오버로드(*overloaded*)되었다라고 한다. 만일 이 두 함수들이 C에서 같은 이름으로 정의되었다면 C 링커는 오류를 출력할 것이다. 왜냐하면 링킹시 오브젝트 파일에서 동일한 심볼에 두 개의 정의를 찾을 것이기 때문이다. 예를 들어 그림 7.10의 코드를 고려해 보자. 어셈블리에서 두 개의 `f` 라 이름 붙여진 라벨은 명백히 오류를 출력한다.

C++은 C와 동일한 링킹 과정을 거치지만, 함수에 이름 붙여진 라벨을 이름 맹글링(*name mangling*)이라는 과정을 통해 오류를 피해갈 수 있다. 사실 C도 이름 맹글링을 사용한다. C에서 함수에 대한 라벨을 정의하기 위해서 함수에 이름에 `_`를 붙인다. 그러나 C는 그림 7.10의 두 함수의 이름을 동일한 라벨로 변경해 오류를 출력할 것이다. C++은 이름 맹글링 시 조금 더 복잡한 방법을 사용하여 이 두 함수에 대한 서로 다른 라벨을 만들어 준다. 예를 들어 그림 7.10의 첫 번째 함수는 DJGPP에 의해 `_f_Fi`라는 라벨을 가지고 두 번째 함수는 `_f_Fd`라는 라벨을 부여받게 된다. 이를 통해 링크 오류를 피해나갈 수 있다.

불행히도 C++에서 라벨을 어떻게 만드는지에 대한 기준이 없기 때문에 각기 다른 컴파일러들이 자신들만의 방법으로 이름 맹글링을 한다. 예를 들어 볼랜드 C++은 그림 7.10의 두 함수에 대해 `@f$qi`와 `@f$qd`라는 라벨을 부여하게 된다. 그러나 이러한 규칙들은 완전히 무작위적이지는 않다. 맹글링 된 이름은 함수의 특징을 나타낸다. 여기서 이 특징은 함수의 인자들의 배열 순서와 형에 의해 정해진다. 예를 들어 `int` 인자만을 가지는 함수는 맹글링 된 이름 끝 부분에 `i`가 붙게 된다. (이는 볼랜드와 DJGPP 모두 동일) 또한 `double` 인자를 가지는 함수는 끝에 `d`가 붙게 된다. 만일 아래와 같은 원형을 가지는 함수 `f` 가 있다면,

```
void f( int x, int y, double z);
```

DJGPP 는 이 함수의 이름을 `_f_Fiid`로 맹글링 하고 볼랜드는 `@f$qiid`로 한다.

함수의 리턴 형식은 함수의 특징에 포함되지 않으며, 맹글링 된 이름에 나타나지도 않는다. 이를 통해 우리는 C++의 오버로딩 규칙에 대해 알 수 있다. 오직 고유의 특징을 가지는 함수 많이 오버로드 될 수 있다. 만일 C++에서 동일한 이름과 특징을 가지는 함수가 정의된다면, 이는 동일한 맹글링된 이름을 가지게 되므로 링커 애러를 발생하게 된다. 기본적으로 C++ 함수들은 오버로드 되지 않아도 이름이 맹글링 된다. 왜냐하면 컴파일러는 컴파일 시에 이 함수가 오버로드 됐는지 안됐는지 알 수 있는 방법이 없기 때문이다. 사실 함수의 이름을 맹글링하는 방법과 비슷한 방법으로 전역변수의 이름을 맹글링한다. 따라서, 만일 어떤 파일에 특정한 형을 가지는 전역 변수를 정의하였는데 다른 파일에서 다른 형으로 사용한다면 링커 오류가 발생하게 된다. 이러한 C++의 특징은 형이 안전한 링킹(*typesafe linking*)이라 불린다. 이를 통해 일정하지 않은 원형에 대한 오류까지도 잡아낼 수 있다. 이는 특정한 모듈에서의 함수의 정의가 다른 모듈에서의 원형과 일치하지 않을 때 발생한다. C에서는 이를 디버그하기 매우 어렵다. 왜냐하면 C는 이 오류를 잡아내지 못하기 때문이다. 프로그램은 오류 없이 컴파일되고 링크되지만 함수를 호출시 함수가 예상했던 값들과 다른 값을 스택에 넣어 프로그램이 동작을 하지 못한다. 하지만 C++에서는 이는 링커 오류를 발생해 미연에 이러한 일을 방지할 수 있다.

C++ 컴파일러가 함수 호출을 분석(parsing)할 때, 함수에 전달되는 인자들의 형을 살펴 보아서 일치하는 함수가 존재하는지 살펴본다.<sup>5</sup> 만일 일치하는 함수를 찾으면 컴파일러는 이름 맹글링 규칙을 사용하여 올바른 함수에 대한 CALL 명령을 수행한다.

각기 다른 컴파일러가 각기 다른 맹글링 규칙을 사용하므로 서로 다른 컴파일러에서 컴파일된 C++ 코드는 링크가 되지 않을 수 있다. 이 사실은 우리가 미리 컴파일된 C++ 라이브러리를 사용할 때 중요하게 작용할 수 있다. 만일 우리가 C++ 코드와 함께 사용될 수 있는 함수를 어셈블리에서 작성한다고 한다면 C++ 코드를 컴파일하는데 사용되는 컴파일러를 보고 이름 맹글링 규칙을 적용시켜야 한다. (또는 아래에 나온 기술을 이용한다)

아마 눈치가 빠른 학생들은 그림 7.10의 코드가 예상했던 대로 작동하는 것인지 물어볼 것이다. C++는 모든 함수의 이름을 맹글링하기에 `printf` 함수도 맹글링 되므로 컴파일러는 라벨 `_printf`에 올바른 CALL 을 할 수 없다는 것이다. 이는 올바른 생각이다. `printf`의 원형이 단순히 파일 상단에 놓여 있다면 이와 같은 일이 발생할 것이다. 그 원형은:

```
int printf( const char *, ...);
```

DJGPP 는 이를 `_printf_FPCce`로 맹글링할 것이다. (F 는 함수(function)를, P 는 포인터(pointer), C 는 상수(const), c 는 char, e 는 생략(ellipsis)을

---

<sup>5</sup> 이 형은 정확히 일치할 필요가 없다. 컴파일러는 인자들을 형변환 것들도 고려해 보기 때문이다. 이 과정에 대한 규칙은 이 책의 범위를 훨씬 뛰어 넘는다. 자세한 내용은 C++ 책들을 참조해라.

각각 뜻한다) 이는 보통의 C 라이브러리의 `printf` 함수를 호출하지 않을 것이다! 당연하게도 C++ 코드에서 C 함수를 호출하는 방법이 있다. 이는 매우 유용한데 왜냐하면 엄청난 수의 유용한 C 코드들이 있기 때문이다. 게다가 C 코드를 호출하는 것을 할 수 있다면 C++에서 또한 보통의 C 맹글링 규약을 이용한 어셈블리 코드도 호출할 수 있게 된다.

C++에선 `extern` 키워드를 이용하여 특정한 함수와 전역 변수에 C 맹글링 규칙을 적용할 수 있다. C++ 용어로 이렇게 선언된 함수와 변수는 *C 연계(C linkage)*를 사용한다고 말한다. 예를 들어, `printf`를 C 연계시키려면 아래와 같이 원형을 쓰면 된다.

```
extern "C" int printf( const char *, ... );
```

이 명령은 컴파일러로 하여금 이 함수에 대해 C++ 이름 맹글링을 적용하지 않으며, 그 대신에 C 규칙에 따라 맹글링을 하게 한다. 그러나, 위와 같이 정의된 `printf`는 오버로드 되지 않는다. 이와 같은 방법을 통해 C++와 어셈블리가 손쉽게 소통할 수 있다. 함수를 C 연계를 사용하게 정의한 뒤, C 호출 규약을 사용하는 것이다.

편의를 위해, C++은 또한 여러개의 함수들과 전역 변수를 한꺼번에 연계시킬 수 있게 해준다. 이는 중괄호를 사용하면 된다.

```
extern "C" {
    /* C 연계 전역 변수들과 함수 원형들*/
}
```

만일, 여러분이 C/C++ 컴파일러에서 모두 작동되는 ANSI C 헤더를 보게 된다면, 헤더 파일 상단에 다음과 같은 코드가 있음을 알 수 있다.

```
#ifdef __cplusplus
extern "C" {
#endif
```

그리고 하단에 중괄호로 닫는 부분이 있음을 알 수 있다. C++ 컴파일러들은 `__cplusplus`를 매크로 (`_` 가 두 개)를 정의한다. 위 코드는 헤더파일이 C++에서 컴파일 될 경우 헤더파일 전체를 `extern "C"`로 감싸고, C에서 컴파일 된다면 어떠한 짓도 하지 않는다. (이는 C 컴파일러를 `extern "C"`를 문법상의 오류로 처리하기 때문에 필요하다). 위와 같은 방법으로 C와 C++ 모두에서 돌아가는 어셈블리 루틴을 위한 헤더파일도 작성할 수 있다.

### 7.2.2 레퍼런스

레퍼런스(*Reference*)는 C++의 또 다른 새 기능이다. 이는 함수의 인자들을 포인터를 직접적으로 사용하지 않고도 전달될 수 있게 한다. 예를 들어 그림 7.11의 코드를 고려해 보자. 사실, 레퍼런스 인자들은 정말로 단순하다. 이들은 정말 포인터 같다. 컴파일러는 단순히 이를 프로그래머에게 숨기는 것이다. (이는 파스칼 컴파일러가 `var` 인자들을 포인터로 실행하는 것처럼) 컴파일러가 함수를 호출하는 7행에서의 어셈블리 코드를 생성할

```

1 void f( int & x )      // &는 레퍼런스 인자를 의미한다
2 { x++; }
3
4 int main()
5 {
6     int y = 5;
7     f(y);      // y에 대한 레퍼런스가 전달되었다. & 가 없음을 유의!
8     printf ("%d\n", y); // 6을 출력
9     return 0;
10}

```

Figure 7.11: 레퍼런스(Reference) 예제

때, 이는 y의 주소를 전달한다. 만일 f를 어셈블리로 작성한다면 그것이 함수의 원형이었던 것처럼<sup>6</sup>

```
void f( int * xp);
```

레퍼런스는 단순히 편의를 위해서 만들어 졌는데, 특히 오버로딩 연산자에 편리하다. 이는 C++의 또 다른 기능으로 구조체나 클래스 형에 대한 평범한 연산자에 의미를 부여하는 것이다. 예를 들어 더하기 (+) 기호는 보통 문자열 객체를 합칠 때 사용한다. 따라서, 만일 a와 b가 문자열이라면 a + b는 문자열 a와 b가 합친 결과가 리턴될 것이다. C++은 사실 이 작업을 하기 위해 함수를 호출한다. (사실, 이러한 식들은 operator +(a,b)와 같은 함수를 이용해 나타낼 수 있다) 편의를 위해, 문자열 객체의 값을 전달하기 보단 주소를 전달하는 것이 좋을 것이다. 레퍼런스를 사용하지 않는다면 이는 operator +(&a, &b)로 가능하지만, 연산자 사용시 &a + &b와 같이 나타내야 하는 불편함이 있다. 그러나 레퍼런스를 이용하면 이를 단순히 a + b만으로 가능하고, 자연스러워 보인다.

### 7.2.3 인라인 함수

인라인 함수(*Inline functions*)는 C++의 또 다른 기능이다.<sup>7</sup> 인라인 함수들은 오류가 잦은, 인자를 갖는 전처리기 기반 매크로들을 대체하기 위하여 만들어 졌다. C에서 숫자를 제곱하는 매크로는 아래와 같이 생겼다.

```
#define SQR(x) ((x)*(x))
```

전처리기는 C를 이해하지 못하고, 단순한 치환만을 하기 때문에 대부분의 경우에 올바른 값을 얻기 위해서 소괄호로 묶어야 한다. 그러나, 이와 같이 함에도 불구하고 SQR(x++)와 같은 식에서는 올바른 값을 얻을 수 없다.

<sup>6</sup> 물론, 7.2.1에서 이야기했던 것처럼 이름 맹글링을 피하기 위해 C 연계로 작성해야 한다.

<sup>7</sup> C 컴파일러들은 종종 이 기능을 ANSI C의 확장 형태로 지원한다

```

1 inline int inline_f( int x )
2 { return x*x; }
3
4 int f( int x )
5 { return x*x; }
6
7 int main()
8 {
9     int y, x = 5;
10    y = f(x);
11    y = inline_f(x);
12    return 0;
13 }
```

Figure 7.12: 인라인 예제

매크로들은 단순한 함수의 호출에서 복잡한 함수 호출과정을 만드는 것을 생략하기 위해 만들어진다. 서브프로그램에 관한 장에서 다루었듯이, 함수를 호출하는 과정은 몇 개의 단계를 거친다. 매우 단순한 함수를 호출 할 때에는 함수를 호출하는 과정이 함수를 실행하는 과정보다 더 오래 걸릴 수도 있다! 인라인 함수는 보통의 함수를 작성하는 것과 매우 비슷하지만, 코드에서 이 함수를 CALL 하는 명령은 이루어 지지 않는다. 그 대신에, 인라인 함수의 호출에 관한 부분은 함수가 수행하는 코드로 대체된다. C++ 에선 함수의 정의 앞에 **inline** 키워드를 붙임으로써 이 함수를 인라인으로 선언할 수 있다. 예를 들어 그림 7.12에서 선언된 함수를 살펴보자. 10 행에서의 **f** 함수의 호출은 보통의 함수 호출을 수행한다. (어셈블리에서 **x** 가 **ebp-8**에, **y** 가 **ebp-4**에 있다고 생각하자)

---

```

1     push    dword [ebp-8]
2     call    _f
3     pop     ecx
4     mov     [ebp-4], eax
```

---

그러나 11 행에서 나타난 **inline\_f** 함수의 호출은 아래와 같다.

---

```

1     mov     eax, [ebp-8]
2     imul   eax, eax
3     mov     [ebp-4], eax
```

---

위 경우 인라인시 두 가지 장점이 있다. 먼저, 인라인 함수는 더 빠르다. 어떠한 인자도 스택에 푸시되지 않으며, 어떠한 스택 프레임도 생성되거나

```

1  class Simple {
2  public:
3      Simple();           // 디폴트 생성자
4      ~Simple();          // 소멸자
5      int get_data() const; // 멤버 함수들
6      void set_data( int );
7  private:
8      int data;           // 멤버 데이터
9  };
10
11 Simple::Simple()
12 { data = 0; }
13
14 Simple::~Simple()
15 { /*아무 것도 안함*/ }
16
17 int Simple::get_data() const
18 { return data; }
19
20 void Simple::set_data( int x )
21 { data = x; }

```

Figure 7.13: 단순한 C++ 클래스

파괴되지 않고, 어떠한 분기도 이루어 지지 않는다. 두 번째로 인라인 함수는 코드를 적게 사용한다. 물론, 위 경우 이것이 사실이였지만 모든 경우에서 사실인 것은 아니다.

인라인 함수의 가장 큰 단점은 인라인 코드는 링크가 되지 않기 때문에 인라인 함수의 코드는 이를 사용하는 모든 파일에서 있어야 한다. 이전의 예제의 어셈블리 코드는 이를 보여준다. 인라인이 아닌 함수를 호출하는데 필요한 정보는 오직 인자, 리턴 값의 형, 호출 규약과 함수에 대한 라벨의 이름이다. 이 모든 정보는 함수의 원형에서 얻을 수 있다. 그러나, 인라인 함수는 함수의 모든 코드에 대한 정보를 필요로 하다. 이 말은 인라인 함수의 어떠한 부분이라도 바뀐다면 이 함수를 사용하는 모든 소스 파일들은 반드시 다시 컴파일 되야만 한다. 인라인이 아닌 함수들의 경우 함수의 원형이 바뀌지 않는 한, 이 함수를 사용하는 파일들은 굳이 다시 컴파일이 될 필요가 없다. 이러한 이유 때문에 인라인 함수에 대한 코드는 보통 헤더파일에 위치한다. 이는 C에서 실행 가능한 코드는 절대로 헤더파일에 위치하면 안된다는 규칙과 상충된다.

```
void set_data( Simple * object, int x )
{
    object->data = x;
}
```

Figure 7.14: C 버전의 Simple::set\_data()

---

```
1 _set_data__6Simplei:           ; 맹글링 된 이름
2     push    ebp
3     mov     ebp, esp
4
5     mov     eax, [ebp + 8]      ; eax = 객체를 가리키는 포인터 (this)
6     mov     edx, [ebp + 12]      ; edx = 정수 인자
7     mov     [eax], edx         ; 데이터는 오프셋 0에 위치
8
9     leave
10    ret
```

---

Figure 7.15: Simple::set\_data( int ) 가 컴파일 된 모습

#### 7.2.4 클래스

C++ 클래스는 객체(*object*)의 타입을 설명한다. C++에서 객체는 데이터 멤버와 함수 멤버<sup>8</sup>, 혹은 일반적으로 메서드(*method*)를 포함할 수 있다. 다시 말해, 이는 데이터와 함수가 연관이 되어 있는 구조체이다. 그림 7.13에 정의된 단순한 클래스를 살펴보자. Simple 타입의 변수는 단순히 C에서의 하나의 int를 멤버로 가지는 구조체와 같다. 이 함수들은 구조체에 해당하는 메모리에 저장되지 않는다. 그러나 멤버 함수들은 다른 함수들과 다르다. 이들은 숨겨진 인자를 주고 받는다. 그 인자는 멤버 함수가 작업하는 객체를 가리키는 포인터이다.

예를 들어 그림 7.13의 Simple 클래스의 set\_data 메서드를 보자. 이것이 C에서 작성된다면 그림 7.14과 같이 함수의 코드에서 작업하는 객체에 대한 포인터를 인자로 받아들이는 함수가 될 것이다. DJGPP 컴파일러 (그리고 gcc 와 볼랜드 컴파일러들도 동일)에 -S 스위치를 사용하면 컴파일러가 코드와 동일한 어셈블리 코드를 만들어 낸다. DJGPP 와 gcc가 생성한 어셈블리 파일에는 .s 확장자가 붙는데, 불행이도 NASM 과 MASM 문법과 다른 AT&T 어셈블리를 사용한다.<sup>9</sup> (Borland 와 MS 컴파일러들은

사실 C++은 this 키워드를 이용하여 멤버 함수 내부에서 객체를 가리키는 포인터에 접근한다.

<sup>8</sup>보통 멤버 함수(*member function*)이라 부른다.

<sup>9</sup>gcc 컴파일러 시스템은 자체적인 어셈블러인 *gas*를 가지고 있다. *gas* 어셈블러는 AT&T 문법을 사용하므로 컴파일러에 의해 생성된 어셈블리 코드는 *gas* 형식으로 되어 있다. 인터넷 상에서 INTEL과 AT&T 문법의 차이점을 서술한 페이지가 몇 개 있다. 또

.asm 확장자를 가지는 MASM 문법의 어셈블리 파일을 만든다) 그림 7.15은 DJGPP 의 어셈블리 코드를 NASM 문법으로 변환한 것이다. 문장의 의미를 파악하기 쉽게 하기 위해 일부 주석을 달았다. 가장 첫 번째 행에서 set\_data 메서드에 클래스의 이름과 인자에 대한 정보를 맹글링한 것으로 이루어진 라벨이 붙었음을 알 수 있다. 라벨에 클래스에 이름에 관한 정보도 포함된 이유는 다른 클래스가 set\_data 라는 이름의 메서드를 가질 수 있기 때문이다. 또한 인자들에 관한 정보도 평범한 C++ 함수들처럼 오버로드 할 수 있도록 포함되었다. 그러나 이전과 같이 다른 컴파일러들은 이 정보를 다른 식으로 맹글링 할 것이다.

그리고 2 와 3 행에서 우리에게 친숙한 부분이 나타난다. 5 행에서 스택에 저장된 인자는 EAX로 저장된다. 이는 x 인자가 아니다. 대신에 이는 숨겨진 인자<sup>10</sup>로 작업하는 객체를 가리키는 포인터이다. 6 행에서, x 인자를 EDX 에 저장하고 7 행에서는 EDX 를 EAX 가 가리키는 더블워드에 저장한다. 이는 우리가 현재 작업하는 Simple 의 객체의 data 멤버이다. 이 멤버는 클래스의 유일한 데이터로 Simple 의 오프셋 0 에 위치해 있다.

### 예제

이 단원은 이 장의 내용을 바탕으로 임의의 크기의 부호가 없는 정수를 나타낸는 C++ 클래스를 만든다. 정수의 크기가 임의로 정해지기 때문에 이는 부호가 없는 정수(더블워드)들의 배열에 저장될 것이다. 이 배열은 동적 할당을 이용하면 아무 크기로나 만들 수 있다. 더블워드는 역순으로 저장된다.<sup>11</sup> (*i.e* 최하위 더블워드는 0 의 위치에 있다) 그림 7.16 는 Big\_int 클래스의 정의를 보여준다.<sup>12</sup> Big\_int 의 크기는 데이터를 저장하는데 사용되는 부호 없는 배열의 크기로 계산된다. 클래스의 size\_ 데이터 멤버는 오프셋이 0 이고, number\_ 멤버는 오프셋이 4 이다.

이 예제들을 단순화 하기 위해 같은 크기의 배열을 가지는 객체 인스턴스 (instance) 만이 더해지거나 빼질 수 있다.

이 클래스는 3 개의 생성자를 가진다. 하나는 ( 9 행) 보통의 부호 없는 정수를 이용하여 클래스 인스턴스를 초기화 하는 것이고 다른 하나는 ( 18 행) 16 진수의 값을 보관하는 문자열을 이용하여 인스턴스를 초기화 한다. 마지막으로는 ( 21 행) 복사 생성자(*copy constructor*)이다.

여기에서는 덧셈과 뺄셈 연산자들이 어떻게 작동하는지에 대해 초점을 맞출 것인데, 왜냐하면 여기서 어셈블리가 사용되기 때문이다. 그림 7.17 은 이 연산자들과 관계있는 헤더파일의 일부분을 보여주고 있다. 이는 어셈블리 루틴을 호출하기 위해 연산자들이 어떻게 설정되어 있는지 보여준다. 서로 다른 컴파일러가 연산자 함수를 위해 다른 종류의 맹글링 규칙을

---

한 a2i 라는 이름의 무료 프로그램 (<http://www.multimania.com/placr/a2i.html>) 을 통해 AT&T 형식을 NASM 형식으로 변환할 수 있다.

<sup>10</sup>어셈블리에서는 숨겨지는 것이 없다!

<sup>11</sup>왜냐하면 덧셈 연산이 배열의 처음 부분부터 진행되기 때문이다.

<sup>12</sup>이 예제의 완전한 코드에 대한 예제 소스 코드를 보라. 본문은 오직 코드의 일부분만 이야기 할 것이다.

```

1  class Big_int {
2  public:
3      /*
4      * 인자:
5      *   size          – 정수의 크기로, unsigned int 가
6      *                   사용된 수로 나타낸다
7      *   initial_value – Big_int 가 보통의 unsigned int 였을 때 초기값
8      */
9      explicit Big_int( size_t size,
10                      unsigned initial_value = 0);
11     /*
12     * 인자:
13     *   size          – 정수의 크기로, unsigned int 가
14     *                   사용된 수로 나타낸다
15     *   initial_value – Big_int 의 초기값의 16 진수의 문자열 형태
16     */
17     Big_int( size_t size,
18             const char * initial_value );
19
20     Big_int( const Big_int & big_int_to_copy );
21     ~Big_int();
22
23     // Big_int 의 크기를 반환(unsigned int 의 수로)
24     size_t size() const;
25
26     const Big_int & operator = ( const Big_int & big_int_to_copy );
27     friend Big_int operator + ( const Big_int & op1,
28                                 const Big_int & op2 );
29     friend Big_int operator - ( const Big_int & op1,
30                                 const Big_int & op2 );
31     friend bool operator == ( const Big_int & op1,
32                               const Big_int & op2 );
33     friend bool operator < ( const Big_int & op1,
34                               const Big_int & op2 );
35     friend ostream & operator << ( ostream & os,
36                                     const Big_int & op );
37
38     private:
39         size_t size_;    // 부호 없는 배열의 크기
40         unsigned * number_; // 값을 가지는 부호 없는 배열을 가리키는 포인터
41     };

```

Figure 7.16: Big\_int 클래스의 정의

```

1 //어셈블리 루틴의 원형
2 extern "C" {
3     int add_big_ints( Big_int & res,
4                         const Big_int & op1,
5                         const Big_int & op2);
6     int sub_big_ints( Big_int & res,
7                         const Big_int & op1,
8                         const Big_int & op2);
9 }
10
11 inline Big_int operator + ( const Big_int & op1, const Big_int & op2)
12 {
13     Big_int result (op1.size ());
14     int res = add_big_ints( result , op1, op2);
15     if (res == 1)
16         throw Big_int :: Overflow();
17     if (res == 2)
18         throw Big_int :: Size_mismatch();
19     return result ;
20 }
21
22 inline Big_int operator - ( const Big_int & op1, const Big_int & op2)
23 {
24     Big_int result (op1.size ());
25     int res = sub_big_ints( result , op1, op2);
26     if (res == 1)
27         throw Big_int :: Overflow();
28     if (res == 2)
29         throw Big_int :: Size_mismatch();
30     return result ;
31 }
```

Figure 7.17: **Big\_int** 클래스 산술 연산 코드

사용하기 때문에 인라인 연산자 함수들은 C 연계 어셈블리 루틴을 호출하게 설정되어 있다. 이는 보통의 직접적인 호출과 속도가 비슷하며, 다른 컴파일러에 의해 컴파일 되어도 잘 작동하게 할 수 있다. 이 기술은 또한 어셈블리에서 예외를 던지는(throw) 상황을 막을 수 있다.

왜 여기에 어셈블리가 사용될까? 다중 정밀도 연산을 사용하기 위해, 캐리가 하나의 더블워드로 부터 다음 더블워드로 전달되어야 한다는 사실을 상기해라. C++(그리고 C)는 프로그래머로 하여금 CPU의 캐리 플래스에 접근하는 것을 막고 있다. 이 덧셈을 수행하는 유일한 방법은 C++이 독립적으로 캐리 플래그를 다시 계산한 뒤에 그 다음 더블워드에 더하는 수밖에 없다. 하지만 어셈블리에선 캐리 플래그에 마음대로 접근할 수 있으며, ADC 명령을 통해 자동적으로 캐리 플래그를 더할 수 있으므로 효과적이다.

너무 글이 길어지는 것을 막기 위해 오직 add\_big\_ints 어셈블리 루틴만 여기에서 다룰 것이다. 아래는 이 루틴을 위한 코드이다. (big\_math.asm에서 가져옴)

```
----- big_math.asm -----
1 segment .text
2     global add_big_ints, sub_big_ints
3 %define size_offset 0
4 %define number_offset 4
5
6 %define EXIT_OK 0
7 %define EXIT_OVERFLOW 1
8 %define EXIT_SIZE_MISMATCH 2
9
10 ; 덧셈과 뺄셈 서브 루틴들을 위한 인자들
11 %define res ebp+8
12 %define op1 ebp+12
13 %define op2 ebp+16
14
15 add_big_ints:
16     push    ebp
17     mov     ebp, esp
18     push    ebx
19     push    esi
20     push    edi
21     ;
22     ; op1 을 가리키기 위해 esi 를 설정
23     ; op2 를 가리키기 위해 edi
24     ; res 를 가리키기 위한 ebx
25     mov     esi, [op1]
26     mov     edi, [op2]
27     mov     ebx, [res]
```

```

28      ;
29      ; 모든 Big_int 들이 같은 크기여야 함을 명심해라
30      ;
31      mov     eax, [esi + size_offset]
32      cmp     eax, [edi + size_offset]
33      jne     sizes_not_equal           ; op1.size_ != op2.size_
34      cmp     eax, [ebx + size_offset]
35      jne     sizes_not_equal           ; op1.size_ != res.size_
36
37      mov     ecx, eax                 ; ecx = Big_int 의 크기
38      ;
39      ; 레지스터를 이에 대응하는 배열을 가리키게 맞춘다.
40      ;     esi = op1.number_
41      ;     edi = op2.number_
42      ;     ebx = res.number_
43      ;
44      mov     ebx, [ebx + number_offset]
45      mov     esi, [esi + number_offset]
46      mov     edi, [edi + number_offset]
47
48      clc                           ; 깨끗 플래그 초기화
49      xor     edx, edx               ; edx = 0
50      ;
51      ; addition loop
52 add_loop:
53      mov     eax, [edi+4*edx]
54      adc     eax, [esi+4*edx]
55      mov     [ebx + 4*edx], eax
56      inc     edx                  ; 깨끗 플래그를 변경하지 말라
57      loop   add_loop
58
59      jc    overflow
60 ok_done:
61      xor     eax, eax               ; 리턴값 = EXIT_OK
62      jmp     done
63 overflow:
64      mov     eax, EXIT_OVERFLOW
65      jmp     done
66 sizes_not_equal:
67      mov     eax, EXIT_SIZE_MISMATCH
68 done:
69      pop     edi

```

```

70      pop      esi
71      pop      ebx
72      leave
73      ret


---


          big_math.asm

```

위의 대부분의 코드는 독자 여러분이 이해하기에 쉽다. 25에서 27 행에서 함수를 통해 인자로 전달된 `Big_int` 객체를 가리키는 포인터가 저장된 레지스터를 저장한다. 레퍼런스는 단지 포인터에 불과하다는 것을 상기해라. 31에서 35 행은 세 객체 배열들의 크기가 같은지 확인한다. (데이터 멤버에 접근하기 위해 `size_`의 오프셋이 포인터에 더해졌다.) 44에서 46 행에선 레지스터 값들을 조정하여, 객체 자기 자신이 아닌 각 객체의 배열을 가리키게 하였다. (여기서도 객체 포인터에 `number_` 멤버의 오프셋이 더해졌다)

52에서 57 행의 루프는 최하위 더블워드 순으로 배열에 저장된 정수들을 더한다. 이런 순서로 더하는 이유는 확장 정밀도 산술 연산(2.1.5 참조) 때문이다. 59 행은 오버플로우가 발생하였는지 확인한다. 오버플로우 시, 최상위 더블워드의 마지막 덧셈에 의해 캐리 플래그가 세트된다. 배열에 저장된 더블워드들이 리틀 엔디안 형식으로 되어 있기 때문에 루프는 배열의 처음 부분부터 시작되어 배열의 끝에서 끝난다.

그림 7.18는 `Big_int`를 사용하는 짧은 예제 클래스를 보여준다. `Big_int` 상수는 16 행과 같이 반드시 명확하게 정의되어야 한다. 이는 두 가지 연유에서 중요하다. 먼저, 부호가 없는 정수를 `Big_int`로 변환하는 변환 생성자가 없다. 두 번째로는 오직 같은 크기의 `Big_int` 들만이 더해질 수 있다. 이는 변환을 복잡하게 하는데 왜냐하면 어떠한 크기로 변환해야 할지 모르기 때문이다. 클래스를 좀 더 복잡하게 만든다면 크기가 서로 다른 것들도 더해질 수 있게 할 수 있다. 그러나 이를 여기에서 보임으로써 이 예제를 복잡하게 만들고 싶은 생각은 없다. (그러나, 독자 여러분이 이를 한 번 해보기를 권한다)

### 7.2.5 상속과 다형성

상속(*Inheritance*)을 통해 하나의 클래스를 다른 클래스에 데이터와 메서드를 상속 할 수 있다. 예를 들어 그림 7.19의 코드를 보자. 이는 두 개의 클래스 A와 B가 나오는데, B 클래스가 A로 부터 상속 받는다. 프로그램의 출력 결과는

```

Size of a: 4 Offset of ad: 0
Size of b: 8 Offset of ad: 0 Offset of bd: 4
A::m()
A::m()

```

두 개의 클래스(A와 상속받은 B)의 ad 데이터 멤버가 동일한 오프셋을 가짐을 주목해라. 이는 f 함수에 A 객체 혹은 A를 물려받은(*i.e.* 상속받은)

```
1 #include "big_int.hpp"
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     try {
8         Big_int b(5,"800000000000000a00b");
9         Big_int a(5,"800000000000010230");
10        Big_int c = a + b;
11        cout << a << " + " << b << " = " << c << endl;
12        for( int i=0; i < 2; i++ ) {
13            c = c + a;
14            cout << "c = " << c << endl;
15        }
16        cout << "c-1 = " << c - Big_int(5,1) << endl;
17        Big_int d(5, "12345678");
18        cout << "d = " << d << endl;
19        cout << "c == d " << (c == d) << endl;
20        cout << "c > d " << (c > d) << endl;
21    }
22    catch( const char * str ) {
23        cerr << "Caught: " << str << endl;
24    }
25    catch( Big_int :: Overflow ) {
26        cerr << "Overflow" << endl;
27    }
28    catch( Big_int :: Size_mismatch ) {
29        cerr << "Size mismatch" << endl;
30    }
31    return 0;
32 }
```

Figure 7.18: Big\_int 의 단순한 사용

```
1 #include <cstddef>
2 #include <iostream>
3 using namespace std;
4
5 class A {
6 public:
7     void __cdecl m() { cout << "A::m()" << endl; }
8     int ad;
9 };
10
11 class B : public A {
12 public:
13     void __cdecl m() { cout << "B::m()" << endl; }
14     int bd;
15 };
16
17 void f( A * p )
18 {
19     p->ad = 5;
20     p->m();
21 }
22
23 int main()
24 {
25     A a;
26     B b;
27     cout << "Size of a: " << sizeof(a)
28         << " Offset of ad: " << offsetof(A,ad) << endl;
29     cout << "Size of b: " << sizeof(b)
30         << " Offset of ad: " << offsetof(B,ad)
31         << " Offset of bd: " << offsetof(B,bd) << endl;
32     f(&a);
33     f(&b);
34     return 0;
35 }
```

Figure 7.19: 상속 예제

---

```

1 _f__FP1A:          ; 맹글링 된 함수 이름
2     push    ebp
3     mov     ebp, esp
4     mov     eax, [ebp+8]      ; eax 는 객체를 가리킨다.
5     mov     dword [eax], 5    ; ad 를 위해 오프셋 0 을 사용
6     mov     eax, [ebp+8]      ; 객체의 주소를 A::m() 에 전달
7     push    eax
8     call    _m__1A          ; A::m() 의 맹글링 된 메서드 이름
9     add     esp, 4
10    leave
11    ret

```

---

Figure 7.20: 간단한 상속 예제를 위한 어셈블리 코드

```

1 class A {
2 public:
3     virtual void __cdecl m() { cout << "A::m()" << endl; }
4     int ad;
5 };
6
7 class B : public A {
8 public:
9     virtual void __cdecl m() { cout << "B::m()" << endl; }
10    int bd;
11 };

```

Figure 7.21: 다형적 상속(Polymorphic Inheritance)

클래스의 객체를 가리키는 포인터를 전달했을 수도 있기 때문이다. (*i.e* 상속된) 그림 7.20 은 함수에 대한 어셈블리 코드를 보여준다. (*gcc* 에 의해 생성되었고, NASM 형식으로 변환됨)

출력 결과에서 *A* 의 *m* 메서드가 *a* 와 *b* 객체로 부터 호출 되었을 때 모두 호출되었다는 것을 주목하자. 어셈블리에서 *A::m()* 으로 호출하는 것은 어셈블리에서 코딩하기가 어렵다. 완전한 객체 지향 프로그래밍에서 호출되는 메서드는 어떠한 객체가 함수에 전달되느냐에 따라 달려있다. 이를 다형성(*polymorphism*) 이라 한다. C++ 은 기본적으로 이 기능을 사용하지 않는다. 우리는 *virtual* 키워드를 통해 이 기능을 사용할 수 있다. 그림 7.21 는 이 두 클래스가 어떻게 바뀌는지 보여준다. 다른 코드들은 바뀔 필요가 없다. 다형성은 여러가지 형태로 구현 될 수 있다. 불행이도 현재 *gcc* 에서의 구현은 이전의 구현 되었던 것보다 훨씬 복잡하게 되었다. 여기에 대해선 깊이 들어가지는 않고 윈도우즈 기반의 마이크로소프트와 볼랜드 컴파일러

---

```

1 ?f@@YAXPAVA@@@Z:
2     push    ebp
3     mov     ebp, esp
4
5     mov     eax, [ebp+8]
6     mov     dword [eax+4], 5 ; p->ad = 5;
7
8     mov     ecx, [ebp + 8]    ; ecx = p
9     mov     edx, [ecx]        ; edx = vtable 을 가리키는 포인터
10    mov    eax, [ebp + 8]    ; eax = p
11    push    eax             ; "this" 포인터를 푸시
12    call    dword [edx]      ; vtable 의 첫 번째 함수 호출
13    add    esp, 4            ; 스택을 정리한다
14
15    pop    ebp
16    ret

```

---

Figure 7.22: f() 함수를 위한 어셈블리 코드

들이 사용하는 형태에 대해서만 이야기 하고자 한다. 이 컴파일러들에서의 구현은 몇 년 동안 바뀌지 않았을 뿐더러 앞으로도 바뀌지 않을 것이다.

*virtual* 키워드를 붙임으로써 프로그램의 출력은 아래와 같이 바뀐다:

```

Size of a: 8 Offset of ad: 4
Size of b: 12 Offset of ad: 4 Offset of bd: 8
A::m()
B::m()

```

이제, f의 두번째 호출에선 B 객체를 전달했기 때문에 B::m() 메서드를 호출하게 된다. 하지만 달라진 것은 이것이 다가 아니다. A의 크기는 8이 된다. (그리고 B는 12) 또한 ad의 오프셋은 0이 아닌 4가 된다. 그렇다면 오프셋 0에는 무엇이 있을까? 이것에 대한 대답은 어떻게 다형성이 구현되었는지에 따라 달라진다.

가상 메서드를 한 개라도 가지는 C++ 클래스들은 메서드 포인터들의 배열을 가리키는 포인터의 추가적인 숨겨진 영역이 있다.<sup>13</sup> 이 테이블은 종종 가상테이블(*vtable*)이라 불린다. A와 B 클래스의 가상테이블을 가리키는 포인터는 오프셋 0에 저장되어 있다. 윈도우즈 컴파일러는 언제나 이 포인터들을 클래스 시작 부분인 상속 트리의 최상단에 위치해 놓는다. 함수 f(그림 7.19)의 가상 메서드 버전의 어셈블리 코드(그림 7.22)를 보면 m

<sup>13</sup>가상 메서드가 없는 클래스들의 경우 C++ 컴파일러들은 언제나 동일한 데이터 멤버를 가지는 보통의 C 구조체와 호환 가능한 클래스를 만든다

```

1  class A {
2  public:
3      virtual void __cdecl m1() { cout << "A::m1()" << endl; }
4      virtual void __cdecl m2() { cout << "A::m2()" << endl; }
5      int ad;
6  };
7
8  class B : public A { // B inherits A's m2()
9  public:
10     virtual void __cdecl m1() { cout << "B::m1()" << endl; }
11     int bd;
12 };
13 /*주어진 객체의 vtable 을 출력 */
14 void print_vtable( A * pa )
15 {
16     // p 는 pa 를 더블워드의 배열로 생각한다
17     unsigned * p = reinterpret_cast<unsigned *>(pa);
18     // vt 는 가상테이블을 포인터들의 배열로 생각한다
19     void ** vt = reinterpret_cast<void **>(p[0]);
20     cout << hex << "vtable address = " << vt << endl;
21     for( int i=0; i < 2; i++ )
22         cout << "dword " << i << ":" << vt[i] << endl;
23
24     // 가상 함수를 이식성이 매우 떨어지는 방법으로 호출한다!
25     void (*m1func_pointer)(A *); // 함수 포인터 변수
26     m1func_pointer = reinterpret_cast<void (*)(A*)>(vt[0]);
27     m1func_pointer(pa); // 함수 포인터를 통해 메서드 m1 을 호출
28
29     void (*m2func_pointer)(A *); // 함수 포인터 변수
30     m2func_pointer = reinterpret_cast<void (*)(A*)>(vt[1]);
31     m2func_pointer(pa); // 함수 포인터를 통해 메서드 m2 을 호출
32 }
33
34 int main()
35 {
36     A a; B b1; B b2;
37     cout << "a: " << endl; print_vtable (&a);
38     cout << "b1: " << endl; print_vtable (&b);
39     cout << "b2: " << endl; print_vtable (&b2);
40     return 0;
41 }
```

Figure 7.23: 좀 더 복잡한 예제

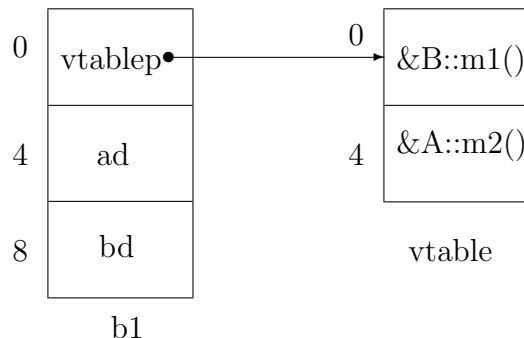


Figure 7.24: b1 의 내부적 표현

메서드 호출은 라벨이 아님을 알 수 있다. 9 행은 객체로 부터 가상테이블의 주소값을 찾는다. 11 행에서 객체의 주소값은 스택에 푸시된다. 12 행에서 가상테이블의 첫 번째 주소<sup>14</sup>로 분기함으로써 가상 메서드를 호출한다. 이 호출은 라벨을 사용하지 않고, EDX 가 가리키는 코드 주소로 분기한다. 이러한 형식의 호출은 나중 바인딩(*late binding*) 라 한다. . 나중 바인딩은 코드가 실행될 때 까지 어떠한 메서드를 호출할지 결정할 수 없게 한다. 이를 통해 객체의 올바른 메서드를 호출하게 한다. 보통의 경우 (그림 7.20)는 호출되는 형태는 빠른 바인딩(*early binding*) 이라 한다. (컴파일 타임(compile time)에 무슨 메서드로 분기해야 할지 알 수 있다.)

세심한 독자들의 경우 왜 그림 7.21의 클래스 메서드들을 \_\_cdecl 키워드를 사용함으로써 명백하게 C 호출 규약으로 정의하였는지에 대해 궁금해 할 것이다. 기본적으로 마이크로소프트는 표준 C 규약과 다른 호출 규약을 C++ 클래스 메서드에 사용한다. 이는 대신에 메서드에서 작업하는 객체를 가리키는 포인터를 스택을 사용하는 대신에 ECX 레지스터에 넣어 전달한다. 스택은 메서드의 다른 인자들을 전달하는데 사용된다. \_\_cdecl 변경자는 표준 C 호출 규약을 사용하게끔 한다. 볼랜드 C++ 은 C 호출 규약을 기본으로 사용한다.

이번에는 약간 더 복잡한 예제를 살펴보자 (그림 7.23). 여기서, 클래스 A 와 B 는 각각 2 개의 메서드를 가지고 있다: m1 과 m2. B 클래스가 자신의 m2 메서드를 정의하는 것이 아니라 A 클래스의 메서드를 상속 받는다는 것을 기억해라. 그림 7.24 은 b 객체가 메모리 상에 어떻게 나타나는지 보여준다. 그림 7.25 은 프로그램의 출력 결과를 보여준다. 먼저, 각 객체의 가상테이블 주소를 보자. 두 개의 B 객체들의 가상 테이블 주소들은 같으므로 이들은 같은 가상테이블을 공유한다. 가상테이블은 클래스의 특징이지 객체가 아니다 (정적 데이터 멤버 처럼). 다음으로 가상테이블에 있는 주소값들을 보라. 출력 결과를 보면 우리는 m1 메서드 포인터가 오프셋 0 (혹은

<sup>14</sup>당연하게도 이 값은 이미 ECX 레지스터에 들어 있다. 그 값은 8 행에서 들어갔다. 또한 10 행을 제거하는 대신에 그 다음 행에 ECX 를 푸시하게 바꾸면 된다. 이 코드는 별로 효율적이지 않는데 왜냐하면 컴파일러 최적화 옵션이 켜지지 않은 채 생성된 코드 이기 때문이다.

```

a:
vtable address = 004120E8
dword 0: 00401320
dword 1: 00401350
A::m1()
A::m2()
b1:
vtable address = 004120F0
dword 0: 004013A0
dword 1: 00401350
B::m1()
A::m2()
b2:
vtable address = 004120F0
dword 0: 004013A0
dword 1: 00401350
B::m1()
A::m2()

```

Figure 7.25: 그림 7.23 의 프로그램의 출력 값

더블워드 0)에 있고 m2 는 오프셋 4 (더블워드 1)에 있음을 알 수 있다. m2 메서드 포인터는 A 와 B 의 가상테이블들에서 동일한데, 왜냐하면 B 는 m2 메서드를 A 클래스로 부터 상속 받았기 때문이다.

25에서 32 행은 객체에 대한 객체의 가상테이블의 주소값을 읽어서 가상 함수를 호출할 수 있는지를 보여주고 있다.<sup>15</sup>. 메서드 주소는 C 유형의 함수 포인터와 this 포인터가 저장된다. 그림 7.25에서의 출력값에서 우리는 이것이 실제로 작동한다는 것을 알 수 있다. 그러나, 코드를 이렇게 쓰지는 말라. 이는 오직 가상 메서드들이 가상테이블을 어떻게 사용하는지 보여주기 위해서 쓴 것이다.

여기서 몇 가지 교훈을 얻을 수 있다. 하나는 우리가 이진 파일에 클래스 변수들을 읽고 쓸 때 매우 주의해야 한다는 점이다. 우리는 단순히 전체 객체를 읽거나 쓸 수 없는데, 왜냐하면 이는 파일에 가상테이블 포인터를 읽거나 쓸 것이기 때문이다. 이는 프로그램의 메모리 상에 가상테이블이 있는 곳을 가리키는 포인터이고, 아는 프로그램마다 다를 것이기 때문이다. 이같은 문제가 C 의 구조체에서도 발생 할 수 있지만, C 구조체의 경우 프로그래머가 특별히 지정해야만 포인터를 가지게 된다. A 나 B 클래스에선 명백히 정의된 포인터들이 없다.

재차 강조하지만 컴파일러들은 서로 다른 방식으로 가상 메서드를 구현했다는 점을 잊지 말아야 한다. 윈도우즈 에서는 COM (Component Object

<sup>15</sup> 이 코드는 오직 MS 와 블랜드 컴파일러에서만 작동된다. gcc 에서는 작동하지 않는다

Model) 클래스 객체들이 COM 인터페이스를 구현하기 위해 가상테이블을 사용한다.<sup>16</sup> 마이크로소프트와 같은 방법으로 가상 메서드를 가상테이블을 이용하여 구현한 컴파일러만이 COM 클래스를 생성할 수 있다. 이는 볼랜드가 마이크로소프트와 같은 구현을 사용하였고, 왜 *gcc*에서는 COM 클래스를 만들 수 없는지 알 수 있다.

가상 메서드를 위한 코드는 비-가상 메서드 것과 정확히 같다. 오직 이를 호출하는 코드만이 다를 뿐이다. 만일 컴파일러가 어떤 가상 메서드가 호출되는지 정확히 알 수 있다면, 이는 가상테이블을 무시하고 직접적으로 메서드를 호출 할 수 있다. (*e.g* 빠른 바인딩)

### 7.2.6 다른 C++ 기능들

다른 C++ 기능(*e.g* 런타임 형식의 정보, 예외 처리 및 다중 상속)에 대해 이야기 하는 것은 이 책의 범위를 훌쩍 뛰어 넘는다. 만일 독자 여러분이 더 자세히 알고 싶다면, Ellis 와 Stroustrup 이 쓴 *The Annotated C++ Reference Manual* 과 Stroustrup 이 쓴 *The Design and Evolution of C++* 을 읽어 보기를 권한다.

---

<sup>16</sup> COM 클래스는 또한 `_stdcall` 호출 규약을 사용한다. 표준 C 가 아니다



## Appendix A

# 80x86 명령어 모음

### A.1 비-부동 소수점 명령들

여기서는 인텔 80x86 계열 CPU 들의 비-부동 소수점 명령들에 대한 정보와 사용 형식들에 대해 설명하고자 한다.

명령들의 사용 형식을 나타내기 위해 아래와 같은 약어들을 이용했다.

R	범용 레지스터
R8	8 비트 레지스터
R16	16 비트 레지스터
R32	32 비트 레지스터
SR	세그먼트 레지스터
M	메모리
M8	바이트
M16	워드
M32	더블워드
I	즉시 데이터

이 약어들은 여러 개의 피연산자를 가지는 명령들을 위해 조합되어 사용될 수 있다. 예를 들어,  $R, R$  라 한다면 이 명령이 두 개의 레지스터를 피연산자로 갖는다는 것이다. 두 개의 피연산자를 가지는 많은 수의 명령들은 동일한 피연산자를 가질 수 있다.  $O2$  는 다음과 같은 피연산자를 표현하는데 사용한다:  $R, R, R, M, R, I, M, R, M, I$ . 만일 8 비트 레지스터 혹은 메모리가 피연산자로 사용될 수 있다면  $R/M8$  과 같이 나타낸다.

아래의 표는 또한 각 명령에 따라 플래그 레지스터들의 비트들이 어떻게 영향을 받는지 보여주고 있다. 만일 특정한 칸이 빈 칸이라면, 이에 대응하는 비트는 영향을 받지 않는다는 것이다. 만일 비트가 특정한 값으로만 바뀐다면 1 또는 0 을 나타내었다. 만일 명령의 피연산자에 의해 비트의 값이 결정된다면 C 라고 나타낸다. 만일 비트가 특정한 규칙 없이 바뀐다면 ? 로 나타낸다. 또한, 방향 플래그의 값을 바꾸는 명령은 CLD 와 STD 밖에 없기 때문에 표에는 나타내지 않았다.

이름	설명	형식	플래그					
			O	S	Z	A	P	C
ADC	캐리와 더한다	O2	C	C	C	C	C	C
ADD	정수를 더한다	O2	C	C	C	C	C	C
AND	And 비트 연산	O2	0	C	C	?	C	0
BSWAP	바이트를 스왑(swap)	R32						
CALL	루틴을 호출	R M I						
CBW	바이트를 워드로 변환							
CDQ	더블워드를 쿼드워드로 변환							
CLC	캐리 초기화							0
CLD	방향 플래그 초기화							C
CMC	캐리에 보수							
CMP	정수를 비교	O2	C	C	C	C	C	C
CMPSB	바이트를 비교		C	C	C	C	C	C
CMPSW	워드를 비교		C	C	C	C	C	C
CMPSD	더블워드를 비교		C	C	C	C	C	C
CWD	워드를 더블워드로 변환 후 DX:AX에 저장							
CWDE	워드를 더블워드로 변환 후 EAX에 저장							
DEC	정수를 감소	R M	C	C	C	C	C	
DIV	부호 없는 나눗셈	R M	?	?	?	?	?	?
ENTER	스택 프레임을 만듦	I,0						
IDIV	부호 있는 나눗셈	R M	?	?	?	?	?	?
IMUL	부호 있는 곱셈	R M	C	?	?	?	?	C
			R16,R/M16					
			R32,R/M32					
			R16,I					
			R32,I					
			R16,R/M16,I					
			R32,R/M32,I					
INC	정수를 증가	R M	C	C	C	C	C	
INT	인터럽트 발생	I						
JA	초과 시 분기	I						
JAE	이상이면 분기	I						
JB	미만 시 분기	I						
JBE	이하 시 분기	I						
JC	캐리 있으면 분기	I						
JCXZ	CX = 0 이면 분기	I						
JE	같으면 분기	I						
JG	초과 시 분기	I						

이름	설명	형식	플래그					
			O	S	Z	A	P	C
JGE	이상 시 분기	I						
JL	미만 시 분기	I						
JLE	이하 시 분기	I						
JMP	무조건 분기	R M I						
JNA	초과 아니면 분기	I						
JNAE	미만이면 분기	I						
JNB	미만 아니면 분기	I						
JNBE	초과면 분기	I						
JNC	캐리 없으면 분기	I						
JNE	같지 않으면 분기	I						
JNG	초과 아니면 분기	I						
JNGE	미만이면 분기	I						
JNL	미만 아니면 분기	I						
JNLE	초과면 분기	I						
JNO	오버 플로우 없으면 분기	I						
JNS	부호 없으면 분기	I						
JNZ	0 이 아니면 분기	I						
JO	오버 플로우 시 분기	I						
JPE	짝수 패리티 시 분기	I						
JPO	홀수 패리티 시 분기	I						
JS	부호 있으면 분기	I						
JZ	0 이면 분기	I						
LAHF	AH 에 플래그 레지스터 저장							
LEA	주소 값을 계산	R32,M						
LEAVE	스택 프레임을 떠남.							
LODSB	바이트를 불러옴							
LODSW	워드를 불러옴							
LODSD	더블워드를 불러옴							
LOOP	루프	I						
LOOP/LOOPZ	같다면 루프	I						
LOOPNE/LOOPNZ	다르면 루프	I						
MOV	데이터를 이동	O2 SR,R/M16 R/M16,SR						
MOVSB	바이트를 이동							
MOVSW	워드를 이동							
MOVSD	더블워드를 이동							

이름	설명	형식	플래그					
			O	S	Z	A	P	C
MOVZX	부호 있는 것을 이동	R16,R/M8 R32,R/M8 R32,R/M16						
MOVZX	부호 없는 것을 이동	R16,R/M8 R32,R/M8 R32,R/M16						
MUL	부호 없는 곱셈	R M	C	?	?	?	?	C
NEG	음을 취함	R M	C	C	C	C	C	C
NOP	명령 없음							
NOT	1 의 보수를 취함	R M						
OR	OR 연산	O2	0	C	C	?	C	0
POP	스택에서 팝	R/M16 R/M32						
POPA	전체를 팝							
POPF	플래그 레지스터를 팝		C	C	C	C	C	C
PUSH	스택에 푸시	R/M16 R/M32 I						
PUSHA	전체를 푸시							
PUSHF	플래그 레지스터를 푸시							
RCL	캐리와 함께 왼쪽 회전	R/M,I R/M,CL	C					C
RCR	캐리와 함께 오른쪽 회전	R/M,I R/M,CL	C					C
REP	반복							
REPE/REPZ	같다면 반복							
REPNE/REPNZ	다르면 반복							
RET	리턴							
ROL	왼쪽으로 회전	R/M,I R/M,CL	C					C
ROR	오른쪽으로 회전	R/M,I R/M,CL	C					C
SAHF	AH 를 플래그 레지스터에 복사		C	C	C	C	C	C
SAL	왼쪽으로 쉬프트	R/M,I R/M, CL						C
SBB	받아 내림과 함께 뺄셈	O2	C	C	C	C	C	C
SCASB	바이트를 찾기		C	C	C	C	C	C
SCASW	워드를 찾기		C	C	C	C	C	C
SCASD	더블워드를 찾기		C	C	C	C	C	C

이름	설명	형식	플래그				
			O	S	Z	A	P
SETA	초과 시 세트	R/M8					
SETAE	이상 시 세트	R/M8					
SETB	미만 시 세트	R/M8					
SETBE	이하 시 세트	R/M8					
SETC	캐리를 세트	R/M8					
SETE	같으면 세트	R/M8					
SETG	초과 시 세트	R/M8					
SETGE	이상 시 세트	R/M8					
SETL	미만 시 세트	R/M8					
SETLE	이하 시 세트	R/M8					
SETNA	초과가 아니면 세트	R/M8					
SETNAE	미만이면 세트	R/M8					
SETNB	미만이 아니면 세트	R/M8					
SETNBE	초과면 세트	R/M8					
SETNC	캐리가 없으면 세트	R/M8					
SETNE	같지 않으면 세트	R/M8					
SETNG	초과가 아니면 세트	R/M8					
SETNGE	미만이면 세트	R/M8					
SETNL	미만이 아니면 세트	R/M8					
SETNLE	초과면 세트	R/M8					
SETNO	오버 플로우 없으면 세트	R/M8					
SETNS	부호가 없으면 세트	R/M8					
SETNZ	0 이 아니면 세트	R/M8					
SETO	오버 플로우면 세트	R/M8					
SETPE	짝수 패리티면 세트	R/M8					
SETPO	홀수 패리티면 세트	R/M8					
SETS	부호가 있으면 세트	R/M8					
SETZ	0 이면 세트	R/M8					
SAR	오른쪽으로 산술 쉬프트	R/M,I					C
SHR	오른쪽으로 논리 쉬프트	R/M, CL					C
SHL	왼쪽으로 논리 쉬프트	R/M,I					C
STC	캐리 플래그를 세트						1
STD	방향 플래그를 세트						
STOSB	바이트를 저장						
STOSW	워드를 저장						
STOSD	더블워드를 저장						

이름	설명	형식	플래그					
			O	S	Z	A	P	C
SUB	뺄셈	O2	C	C	C	C	C	C
TEST	논리 비교	R/M,R	0	C	C	?	C	0
XCHG	서로 교환	R/M,I						
		R/M,R						
		R,R/M						
XOR	XOR 비트 연산	O2	0	C	C	?	C	0

## A.2 부동 소수점 명령들

여기에서는 80x86 수치 부프로세서 명령들에 대해 살펴 보겠다. 아래 표에서는 명령에 대해 간단하게 설명한다. 공간을 절약하기 위해 명령이 스택을 팝하는지에 대해서는 설명에 포함시키지 않았다.

명령에 대해 무슨 피연산자가 사용될 수 있는지 알려주기 위해 아래와 같은 약어를 사용하였다.

$STn$	부프로세서 레지스터
F	메모리 상의 단일 정밀도 수
D	메모리 상의 2배 정밀도 수
E	메모리 상의 확장 정밀도 수
I16	메모리 상의 정수 워드
I32	메모리 상의 정수 더블워드
I64	메모리 상의 정수 쿼드워드

별표(\*)로 표기된 명령들은 펜티엄 프로 이상의 프로세서들에서만 지원된다.

명령	설명	형식
FABS	$ST0 =  ST0 $	
FADD <i>src</i>	$ST0 += src$	$STn F D$
FADD <i>dest</i> , <i>ST0</i>	$dest += ST0$	$STn$
FADDP <i>dest</i> [, <i>ST0</i> ]	$dest += ST0$	$STn$
FCHS	$ST0 = -ST0$	
FCOM <i>src</i>	$ST0$ 와 <i>src</i> 를 비교	$STn F D$
FCOMP <i>src</i>	$ST0$ 와 <i>src</i> 를 비교	$STn F D$
FCOMPP <i>src</i>	$ST0$ 와 <i>ST1</i> 을 비교	
FCOMI* <i>src</i>	비교 후 결과를 플래그 레지스터에 저장	$STn$
FCOMIP* <i>src</i>	비교 후 결과를 플래그 레지스터에 저장	$STn$
FDIV <i>src</i>	$ST0 /= src$	$STn F D$
FDIV <i>dest</i> , <i>ST0</i>	$dest /= ST0$	$STn$
FDIVP <i>dest</i> [, <i>ST0</i> ]	$dest /= ST0$	$STn$
FDIVR <i>src</i>	$ST0 = src / ST0$	$STn F D$
FDIVR <i>dest</i> , <i>ST0</i>	$dest = ST0 / dest$	$STn$
FDIVRP <i>dest</i> [, <i>ST0</i> ]	$dest = ST0 / dest$	$STn$
FFREE <i>dest</i>	비었다고 표시	$STn$
FIADD <i>src</i>	$ST0 += src$	$I16 I32$
FICOM <i>src</i>	$ST0$ 와 <i>src</i> 를 비교	$I16 I32$
FICOMP <i>src</i>	$ST0$ 와 <i>src</i> 를 비교	$I16 I32$
FIDIV <i>src</i>	$ST0 /= src$	$I16 I32$
FIDIVR <i>src</i>	$ST0 = src / ST0$	$I16 I32$
FILD <i>src</i>	<i>src</i> 를 스택에 푸시	$I16 I32 I64$

명령	설명	형식
FIMUL <i>src</i>	ST0 *= <i>src</i>	I16 I32
FINIT	부프로세서를 초기화	
FIST <i>dest</i>	ST0 저장	I16 I32
FISTP <i>dest</i>	ST0 저장	I16 I32 I64
FISUB <i>src</i>	ST0 -= <i>src</i>	I16 I32
FISUBR <i>src</i>	ST0 = <i>src</i> - ST0	I16 I32
FLD <i>src</i>	<i>src</i> 를 스택에 푸시	STn F D E
FLD1	1.0 을 스택에 푸시	
FLDCW <i>src</i>	제어 워드 레지스터를 불러옴	I16
FLDPI	$\pi$ 를 스택에 푸시	
FLDZ	0.0 을 스택에 푸시	
FMUL <i>src</i>	ST0 *= <i>src</i>	STn F D
FMUL <i>dest</i> , ST0	<i>dest</i> *= ST0	STn
FMULP <i>dest</i> [,ST0]	<i>dest</i> *= ST0	STn
FRNDINT	ST0 를 반올림	
FSCALE	$ST0 = ST0 \times 2^{[ST1]}$	
FSQRT	$ST0 = \sqrt{ST0}$	
FST <i>dest</i>	ST0 저장	STn F D
FSTP <i>dest</i>	ST0 저장	STn F D E
FSTCW <i>dest</i>	제어 워드 레지스터를 저장	I16
FSTSW <i>dest</i>	상태 워드 레지스터를 저장	I16 AX
FSUB <i>src</i>	ST0 -= <i>src</i>	STn F D
FSUB <i>dest</i> , ST0	<i>dest</i> -= ST0	STn
FSUBP <i>dest</i> [,ST0]	<i>dest</i> -= ST0	STn
FSUBR <i>src</i>	$ST0 = src - ST0$	STn F D
FSUBR <i>dest</i> , ST0	$dest = ST0 - dest$	STn
FSUBP <i>dest</i> [,ST0]	$dest = ST0 - dest$	STn
FTST	ST0 를 0.0 과 비교	
FXCH <i>dest</i>	ST0 와 <i>dest</i> 의 값을 교환	STn

# Index

## 【 기호 】

16진법 ..... 3-4  
2의 보수 ..... 30-31  
    산술 연산 ..... 35-39  
80x86  
    CPU ..... 6

## 【 A 】

ADC ..... 39, 56  
ADD ..... 13, 38  
AND ..... 52  
array1.asm ..... 101-104

## 【 B 】

BSWAP ..... 61  
BYTE ..... 16

## 【 C 】

C driver ..... 19  
C 와 소통하기 ..... 82  
C 와 함께 작업하기 ..... 90  
C++ ..... 150-171  
    Big\_int 예제 ..... 158-163  
    name mangling ..... 151  
    references ..... 154  
    virtual ..... 166  
    vtable ..... 167, 171  
    나중 바인딩 ..... 169  
    다형성 ..... 166-171  
    레퍼런스 ..... 153  
    멤버 함수 ..... *see* 메서드  
    복사 생성자 ..... 158  
    빼른 바인딩 ..... 169  
    상속 ..... 163-171  
    이름 맹글링 ..... 153

    인라인 함수 ..... 154-156

    클래스 ..... 157-171

    형이 안전한 링킹 ..... 152

    CALL ..... 71-72

    CBW ..... 33

    CLC ..... 39

    CLD ..... 108

    clock ..... 6

    CMP ..... 40

    CMPSB ..... 111

    CMPSD ..... 111

    CMPSW ..... 111

    COM ..... 171

    CPU ..... 5-7

    CWD ..... 33

    CWDE ..... 33

## 【 D 】

DEC ..... 13  
DIV ..... 36, 50  
do while 루프 ..... 45  
DWORD ..... 16

## 【 F 】

FABS ..... 133  
FADD ..... 128  
FADDP ..... 128  
FCHS ..... 133  
FCOM ..... 131  
FCOMI ..... 133  
FCOMIP ..... 132, 133  
FCOMP ..... 131  
FCOMPP ..... 131  
FDIV ..... 130  
FDIVP ..... 130

FDIVR .....	130	print_nl .....	17
FDIVRP .....	130	print_string .....	17
FFREE .....	128	read_char .....	17
FIADD .....	128	read_int .....	17
FICOM .....	131	어셈블리 I/O 라이브러리 .....	17
FICOMP .....	131	IDIV .....	36
FIDIV .....	130	if 문 .....	44
FIDIVR .....	130	IMUL .....	35–36
FILD .....	127	INC .....	13
FIST .....	128	<b>【 J 】</b>	
FISUB .....	129	JC .....	41
FISUBR .....	129	JE .....	43
FLD .....	127	JG .....	43
FLD1 .....	127	JGE .....	43
FLDCW .....	128	JL .....	43
FLDZ .....	127	JLE .....	43
FMUL .....	130	JMP .....	40–41
FMULP .....	130	JNC .....	41
FSCALE .....	133	JNE .....	43
FSQRT .....	133	JNG .....	43
FST .....	128	JNGE .....	43
FSTCW .....	128	JNL .....	43
FSTP .....	128	JNLE .....	43
FSTSW .....	131	JNO .....	41
FSUB .....	129	JNP .....	41
FSUBP .....	129	JNS .....	41
FSUBR .....	129	JNZ .....	41
FSUBRP .....	129	JO .....	41
FTST .....	131	JP .....	41
FXCH .....	128	JS .....	41
<b>【 G 】</b>		JZ .....	41
gas .....	157	<b>【 L 】</b>	
<b>【 I 】</b>		LAHF .....	131
I/O .....	16–18	LEA .....	85, 105
asm_io library .....	18	listing file .....	24
dump_math .....	18	LODSB .....	108
dump_mem .....	18	LOSDS .....	108
dump_regs .....	18	LODSW .....	108
dump_stack .....	18	LOOP .....	43
print_char .....	17	LOOPE .....	43
print_int .....	17	LOOPNE .....	44

- LOOPNZ ..... 44     **【 S 】**  
 LOOPZ ..... 43     SAHF ..... 131  
                        SAL ..... 50  
                        SAR ..... 50  
**【 M 】**  
 MASM ..... 12     SBB ..... 39  
 math.asm ..... 37–38     SCASB ..... 111  
 memory.asm ..... 112–117     SCASD ..... 111  
 MOV ..... 13     SCASW ..... 111  
 MOVS ..... 110     SCSI ..... 147–149  
 MOVSD ..... 110     SET $xx$  ..... 56  
 MOVSW ..... 110     SETG ..... 57  
 MOVSX ..... 33     SHL ..... 49  
 MOVZX ..... 33     SHR ..... 49  
 MUL ..... 35–36, 50, 105     STD ..... 108  
                        STOSB ..... 108  
                        STOSD ..... 108  
                        SUB ..... 13, 38
- 【 N 】**  
 NASM ..... 12     **【 T 】**  
 NEG ..... 36, 57     TASM ..... 12  
 NOT ..... 54     TCP/IP ..... 61  
**【 O 】**  
 OR ..... 53     TEST ..... 53  
                        TWORD ..... 16
- 【 P 】**  
 prime.asm ..... 45–47     **【 U 】**  
 prime2.asm ..... 139–142     UNICODE ..... 61
- 【 Q 】**  
 quad.asm ..... 133–136     **【 W 】**  
 QWORD ..... 16     while 루프 ..... 45  
                        WORD ..... 16
- 【 R 】**  
 RCL ..... 51     **【 X 】**  
 RCR ..... 51     XCHG ..... 61  
 read.asm ..... 136–139     XOR ..... 53
- REP ..... 110     **【 ㄱ 】**  
 REPE ..... 111, 112     간접 주소 지정 ..... 67–68  
 REPNE ..... 111, 112     배열 ..... 100–104  
 REPNZ ..... *see* REPNE     개시 코드 ..... 24  
 REPZ ..... *see* REPE     구조체 ..... 143–150  
 RET ..... 71–72, 74     offsetof() ..... 144  
 ROL ..... 51     배열 하기 ..... 145–146  
 ROR ..... 51     비트 필드 ..... 146–150  
                        기계어 ..... 5, 11

- 【 ㄴ 】**
- 니블 ..... 4
- 【 ㄷ 】**
- 다중 모듈 프로그램 ..... 79–82
  - 데이터 세그먼트 ..... 21
  - 디버깅 ..... 18
- 【 ㄹ 】**
- 라벨 ..... 15–16
  - 레지스터 ..... 5, 7–8
    - 32비트 ..... 8
    - EDI ..... 109
    - EDX:EAX ..... 33, 36, 39, 85
    - EFLAGS ..... 8
    - EIP ..... 8
    - ESI ..... 109
    - IP ..... 8
    - 베이스 포인터 ..... 8
    - 색인 ..... 7
    - 세그먼트 ..... 7, 8, 109
    - 스택 포인터 ..... 7, 8
    - 위치 레지스터 ..... 7
  - 플래그 ..... 8, 40
    - CF ..... 40
    - DF ..... 108
    - OF ..... 40
    - PF ..... 42
    - SF ..... 40
    - ZF ..... 40
  - 리스팅 파일 ..... 25
  - 링킹 ..... 24
- 【 ㅁ 】**
- 메모리 ..... 4–5
    - 가상 ..... 10
    - 세그먼트 ..... 9, 10
    - 페이지 ..... 10
  - 메모리:세그먼트 ..... 9
  - 메서드 ..... 157
  - 문자열 명령 ..... 108–117
- 【 ㅂ 】**
- 바이트 ..... 4
- 배열 ..... 97–117
- 다차원 ..... 105–108
- 2 차원 ..... 105–106
- 인자 ..... 107–108
- 접근 ..... 98–104
- 정의 ..... 97–98
- 정적 ..... 97
- 지역 변수 ..... 98
- 보호 모드
- 16비트 ..... 9
  - 32비트 ..... 10
- 부동 소수점 ..... 119–142
- 산술 연산 ..... 124–126
  - 표현 ..... 119–124
    - 2배 정밀도 ..... 123–124
    - IEEE ..... 121–124
    - 단일 정밀도 ..... 122–123
    - 비정규화 ..... 123
    - 숨겨진 1 ..... 122
- 부동 소수점 부프로세서 ..... 126–142
- 곱셈과 나눗셈 ..... 130
  - 덧셈과 뺄셈 ..... 128–129
  - 불러오고 저장하기 ..... 127–128
  - 비교 ..... 130–133
  - 하드웨어 ..... 126–127
- 분기 예측 ..... 55–56
- 비트 수 세기 ..... 62–66
- 두 번째 방법 ..... 63–64
  - 세 번째 방법 ..... 64–66
  - 첫 번째 방법 ..... 62–63
- 비트 연산
- AND ..... 52
  - C ..... 58–59
  - NOT ..... 53
  - OR ..... 52
  - XOR ..... 53
  - 쉬프트 ..... 49–51
    - 논리 쉬프트 ..... 49–50
    - 산술 쉬프트 ..... 50
    - 회전 ..... 51
    - 회전 쉬프트 ..... 51
  - 어셈블리 ..... 54–55

- 빼대 파일 ..... 26
- 【 ㅅ 】**
- 서브 루틴 ..... *see* 서브프로그램
  - 서브프로그램 ..... 68-95
    - 재진입 ..... 91
    - 호출 ..... 71-79
  - 스택 ..... 70-79
    - 인자 ..... 72-75
    - 지역 변수 ..... 79, 85
    - 지역변수 ..... 77
  - 실제 모드 ..... 8
  - 십진법 ..... 1
- 【 ㅇ 】**
- 어셈블러 ..... 12
  - 어셈블리어 ..... 12-13
  - 엔디안 ..... 25-26, 59-62
    - 엔디안 변환 ..... 61
  - 연상 기호 ..... 12
  - 워드 ..... 8
  - 이진법 ..... 1-3
  - 이진수의 덧셈 ..... 2
  - 인터럽트 ..... 10
- 【 ㅈ 】**
- 재귀 ..... 91-92
  - 저장 형식
    - 레지스터 ..... 95
    - 자동 ..... 93
    - 전역 ..... 93
    - 정적 ..... 93
    - 휘발성 ..... 95
  - 정수 ..... 29-40
    - 곱셈 ..... 35-36
    - 나눗셈 ..... 36
    - 부호 비트 ..... 29, 32
    - 부호 없는 정수 ..... 29, 40
    - 부호 있는 정수 ..... 29-31, 40
    - 부호 확장 ..... 32-35
    - 비교 ..... 40
    - 표현 ..... 29-35
    - 1의 보수 ..... 30
- 2의 보수 ..... 30-31
- 부호있는 크기 ..... 29
- 확장 정밀도 ..... 38-39
- 조건 분기 ..... 41-43
- 주석 ..... 13
- 즉시 피연산자 ..... 13
- 지시어 ..... 14-16
  - %정의 ..... 14
  - DX ..... 15, 97
  - DD ..... 15
  - DQ ..... 15
  - equ ..... 14
  - extern ..... 80
  - RESX ..... 14, 97
  - TIMES ..... 15, 97
  - 데이터 ..... 14-16
  - 전역 ..... 22, 80, 82
- 지역성 ..... 143
- 【 ㅊ 】**
- 추론적 실행 ..... 55
- 【 ㅋ 】**
- 컴파일러 ..... 6, 12
    - DJGPP ..... 22, 24
    - gcc ..... 22
      - \_attribute ..... 86, 145, 148, 149
    - Watcom ..... 86
    - 마이크로소프트 ..... 23
      - pragma pack ..... 146, 148, 149
    - 볼랜드 ..... 23, 24
  - 코드 세그먼트 ..... 22
- 【 ㅌ 】**
- 텍스트 세그먼트. *see* 코드 세그먼트
- 【 ㅎ 】**
- 호출 규약 ..... 72-79, 85-86
    - \_cdecl ..... 86
    - \_stdcall ..... 86
    - C ..... 22, 74, 82-86
      - 라벨 ..... 83-84
      - 레지스터 ..... 83
      - 리턴값 ..... 85

인자.....	84
stdcall .....	74, 86, 171
레지스터 .....	86
파스칼.....	74
표준 호출.....	86